O'REILLY®

Soccer Analytics with Machine Learning

Learning Predictive Modeling Techniques with Sports Data



Haipeng Gao, Ari Joury,
Weining Shen & Guanyu Hu

"An engaging, practical guide that combines soccer passion with modern machine learning. This book will help readers turn real match data into meaningful insights and predictive models."

Alexander Aue
Professor of statistics, University of California, Davis

Soccer Analytics with Machine Learning

Struggling to grasp machine learning concepts or unsure how to apply them in the real world? This book aims to change that by using the world's most popular game—soccer—to illuminate key concepts in predictive modeling and data science. You'll develop a solid foundation in machine learning through engaging examples that bridge academic principles with practical applications.

Written by experts in both machine learning and sports analytics, this practical Python-focused guide introduces fundamental data science techniques using real soccer data. Ideal for students, analysts, and soccer fans alike, it offers instructions on models and techniques such as logistic regression, random forests, deep learning, simulations, and feature engineering. But instead of memorizing algorithms, you'll learn by building predictive models to analyze match outcomes, test betting strategies, run simulated game scenarios, and more.

- **Understand machine learning concepts by working with real sports data**
- **Develop, refine, and evaluate machine learning models, using Python for data analysis**
- **Carry out detailed analyses and research on soccer game predictions and betting strategies to surface valuable insights**
- **Apply the skills you learn to predictive modeling scenarios in other industries**

Haipeng Gao is a data science and machine learning expert with extensive industry experience building and optimizing large-scale machine learning solutions at leading technology companies, including LinkedIn, TikTok, and PayPal.

Ari Joury is a data scientist and entrepreneur working at the intersection of machine learning, causal inference, and applied analytics. He is founder and CEO of Wangari Global, where he builds AI systems for decision support in finance, insurance, and sustainability.

Weining Shen is an associate professor of statistics at the University of California, Irvine. His research focuses on machine learning, Bayesian statistics, sports analytics, and large language models.

Guanyu Hu is an associate professor at Michigan State University. His research focuses on spatial statistics, Bayesian nonparametric methods, and sports analytics.

DATA SCIENCE

US $59.99 CAN $74.99
ISBN: 978-1-098-18111-6



O'REILLY®

Praise for *Soccer Analytics with Machine Learning*

Data is everywhere, and this book proves it. While the beautiful game is loved the world over, few have dared to apply the rigorous scalpel of data analysis to such a complex environment, until now. This is a fantastic resource. The insights are accessible, the examples are clear, and the methods are immediately applicable across the field. A must-read for anyone looking to bridge the gap between soccer and machine learning.

—Otu Ekanem, CTO, Kanopy AM

Few books manage to be genuinely useful to both the soccer obsessive and the machine learning practitioner. This one is. From expected goals to ensemble methods, it covers the field with clarity, honesty about uncertainty, and a deep appreciation for the game itself.

—Matteus Tanha, AI consultant

Finally a book that teaches machine learning the way science actually works: on real, messy data, building the judgment to know when and how models work, not just the skills to build them.

—Louis Delloye, senior R&D strategy analyst, Sanofi

An engaging, practical guide that combines soccer passion with modern machine learning. This book will help readers turn real match data into meaningful insights and predictive models.

—Alexander Aue, professor of statistics, University of California, Davis

Soccer Analytics with Machine Learning

Learning Predictive Modeling Techniques with Sports Data

Haipeng Gao, Ari Joury, Weining Shen, and Guanyu Hu

O'REILLY®

Soccer Analytics with Machine Learning
by Haipeng Gao, Ari Joury, Weining Shen, and Guanyu Hu

Copyright © 2026 Haipeng Gao, Ari Joury, Weining Shen, and Guanyu Hu. All rights reserved.

Published by O'Reilly Media, Inc., 141 Stony Circle, Suite 195, Santa Rosa, CA 95401.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Michelle Smith
Development Editor: Jeff Bleiel
Production Editor: Clare Laylock
Copyeditor: Sharon Wilkey
Proofreader: Laura K. Miller
Indexer: Krsta Technology Solutions
Cover Designer: Susan Brown
Cover Illustrator: Monica Kamsvaag
Interior Designer: David Futato
Interior Illustrator: Kate Dullea

June 2026: First Edition

Revision History for the First Edition
2026-06-15: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098181116* for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Soccer Analytics with Machine Learning*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights. This book is not intended as financial advice. Please consult a qualified professional if you require financial advice.

978-1-098-18111-6

[LSI]

Table of Contents

Preface xiii

1. The Soccer Analytics Landscape 1
The Problems That Soccer Analytics Helps Solve 2
Soccer Clubs: Making Decisions Across the Organization 2
Betting Markets: Modeling Uncertainty and Pricing Outcomes 4
Fans and Analysts: Reading the Game Beyond the Scoreline 5
The Data-Driven Evolution in Soccer 6
From Outcomes to Process: The Limits of Traditional Analysis 6
The Data Revolution: From Event Logs to Tracking Systems 7
Analytics in Practice: Metrics, Applications, and Scope 7
Why Soccer Resists Easy Analysis 8
A Framework for Soccer Analysis 9
The Three Types of Analysis 9
Data Sources for Modern Analysis 10
From Player to League: Levels of Aggregation 11
Principles That Run Through Everything 12
Looking Ahead: A Guided Tour of the Book 13
Conclusion 13

2. Python Fundamentals: Building Your Analytics Toolkit 15
Python: The Language Behind Modern Analytics 15
Your Analytics Workspace: Setting Up the Environment 16
Installation: Getting Python Ready 17
Virtual Environments: Isolating Your Projects 17
Jupyter Notebook: Creating a Workspace for Interactive Analysis and More 20
Repositories: Storing Your Project Files 22
Python Basics: The Core Moves 23

Primitive Data Types: Representing Basic Information 24
Basic Operations: Working with Values and Comparisons 25
Variable Assignment: Storing and Reusing Information 26
Data Structures: Organizing Information 27
Conditional Statements: Writing Decision Rules 32
Loops: Repeating Computation Efficiently 33
List Comprehensions: Performing Compact and Pythonic Iteration 35
Functions: Providing Reusable Logic for Analysis 36
Looking Ahead: From Python Basics to Real Analysis 36
Conclusion 37

3. Exploratory Data Analysis in Soccer. 39
Understanding Soccer Data Levels 40
Match-Level Data 40
Player-Level Data 40
Event-Level Data 41
What Is EDA in Soccer Analytics? 41
Loading StatsBomb Soccer Data 42
Getting the Data 43
Loading the Data 43
Taking a First Look at the Data 44
Structured EDA by Data Types 45
Numerical Features: The Quantifiable Game 45
Categorical Features: The Qualitative Aspects 49
Ordinal Features: The Ordered Categories 51
Preliminary Exploration: Getting Your Bearings 52
Building Tournament DataFrames 53
Counting the Frequency of Event Types 54
Filtering: Passes by Player or Team 54
Grouping: Per-Match Pass Summaries 54
Aggregating: Shots, Goals, and xG by Team 55
Basic Data Cleaning for Soccer 57
Handling Missing Values 57
Standardizing Names 58
Visualizing Soccer Data 59
A Quick Note on Matplotlib and Seaborn 59
Bar Charts and Rankings 60
Histograms and Distributions 61
Boxplots and Variability 63
Time Series and Cumulative Plots 65
Scatterplots: Shot Volume Versus Goals 66
Soccer-Specific Visualizations with mplsoccer 68

Pass Location Heatmaps 68
Advanced Pitch Visualizations: Passing Networks 70
Basic Pitch Visualizations 71
Advanced Heatmap Visualizations 73
Case Study: Japan Women's at the 2019 World Cup 75
Team-Level Overview 75
Match Flow 77
Spatial Patterns 78
Player-Level Insights 78
Wrap-up of the Case Study 80
From EDA to Machine Learning 80
Features as Building Blocks 81
The Standard Pipeline 81
Looking Ahead: From Exploration to Regression 81
Conclusion 82

4. Regression Techniques for Soccer Analytics. 85
From Lines on a Field to Lines of Best Fit: Linear Regression 86
Predicting Market Value: A Player's Worth 86
Building the Model: The Line of Best Fit 88
Visualizing the Model 89
Evaluating Performance: How Good Is Our Model? 90
Counting Goals: Poisson Regression 92
Predicting Goals in a Match 93
Visualizing the Poisson Model 94
Finding Similar Players: K-Nearest Neighbors Regression 95
Predicting Player Value with KNN 96
Choosing the Right k 97
Evaluating and Interpreting Regression Models in Soccer Contexts 97
Using Key Performance Metrics 97
Understanding the Importance of a Validation Set 98
Interpreting Coefficients 99
Visualizing KNN: Decision Boundaries 99
Comparing Models 102
Advanced Topics in Poisson Regression: Overdispersion and Zero-Inflation 103
Overdispersion: When Variance Exceeds the Mean 103
Negative Binomial Regression 104
Zero Inflation: Too Many Zeros 104
Hurdle Models: An Alternative Approach 105
Model Diagnostics: Ensuring That Your Model Is Sound 106
Residual Analysis: Looking for Patterns 106
Goodness-of-Fit Tests 107

Influential Observations: Outliers that Matter 108
Model Selection: Choosing the Best Model 108
Practical Case Study: Predicting Match Outcomes 109
Generating the Dataset 109
Performing Exploratory Analysis 110
Building and Comparing Models 112
Evaluating on Test Data 113
Interpreting the Results 115
Feature Engineering for Better Predictions 116
Interaction Features 116
Polynomial Features 117
Categorical Features 117
Lagged Features 118
Domain-Specific Features 119
Common Pitfalls and Practical Tips 120
Pitfall 1: Overfitting to the Training Data 120
Pitfall 2: Ignoring Multicollinearity 120
Pitfall 3: Using the Wrong Model for the Data 121
Pitfall 4: Forgetting About Data Quality 121
Tip 1: Start Simple, Then Add Complexity 122
Tip 2: Visualize, Visualize, Visualize 122
Tip 3: Document Your Process 122
Tip 4: Validate with Domain Experts 122
Looking Ahead: Where Regression Meets the Real World 122
From Regression to Classification 122
Ensemble Methods: Combining Models 123
Deep Learning: The Frontier 123
Time Series and Sequential Data 124
Causal Inference: Beyond Prediction 124
The Human Element 124
Conclusion 125

5. Predicting Soccer Outcomes with Classification. 127
From Coin Flips to Goal Predictions: Understanding Classification 128
Logistic Regression: Your First Classifier 129
The Intuition: From Numbers to Probabilities 129
Hands-On: Building an Expected Goals Model 130
Measuring Success: How Good Are Your Predictions? 133
The Confusion Matrix: A Report Card for Your Model 133
Precision and Recall: The Two Sides of Performance 135
The ROC Curve: A Complete Performance Picture 136
Evaluation of Probabilistic Models with Cost Functions 137

Beyond Binary: Multiclass Classification 139
K-Nearest Neighbors: Finding Similar Situations 141
The Intuition: Birds of a Feather 141
Hands-On: KNN for Shot Prediction 142
Choosing the Right K 142
Case Study: Building a Complete Match Outcome Predictor 144
Step 1: Define the Problem 144
Step 2: Gather and Prepare the Data 145
Step 3: Train and Evaluate the Model 146
Step 4: Interpret the Results 148
Key Takeaways from the Case Study 148
Practical Tips and Common Pitfalls 149
Handling Imbalanced Data 149
Selecting and Engineering Features 149
Avoiding Overfitting 150
Choosing When to Use Which Algorithm 151
Tuning the Decision Threshold: Finding the Sweet Spot 151
Understanding the Threshold Trade-Off 151
Finding the Optimal Threshold 152
Deciding When to Adjust the Threshold 153
Comparing Models: Which Classifier Is Best? 153
The Model Comparison Framework 154
Hands-On: Model Comparison for xG 154
Visualization of Model Performance 155
Advanced Topics: Going Deeper 156
Scaling and Normalizing Features 156
Handling Missing Data 158
Combining Classifiers with Ensemble Methods 159
Real-World Considerations 160
Data Quality and Availability 160
Model Maintenance and Updates 160
Ethical Considerations 160
Looking Ahead: From Linear Boundaries to Decision Trees 160
Conclusion 161

6. Advanced Classification Methods. 163
The Classification Tree: Soccer's Decision Flowchart 164
The Intuition: Learning the If-Then Rules of Soccer 164
Hands-On: Building a Tree to Predict Pass Outcome 164
Interpreting the Soccer Intelligence 166
Understanding Pros and Cons: When to Use a Single Tree 166
Tuning the Tree: Preventing Overfitting 167

Random Forest: The Wisdom of the Crowd 169
The Intuition: Ensemble Learning 170
Hands-On: Building a Random Forest for to Predict Match Outcome 170
Interpreting Feature Importance 171
Tuning Random Forests 172
Using Out-of-Bag Error 172
Understanding When Random Forests Struggle 173
XGBoost: Learning from Mistakes 173
The Intuition: Sequential Learning 173
Hands-On: XGBoost for xG Prediction 173
Performing SHAP Analysis: Explaining the Opaque Model 174
Tuning XGBoost: The Art of Hyperparameter Optimization 176
Knowing When to Use XGBoost Versus Random Forest 176
Practical Applications: From Theory to Practice 177
Example 1: Building a Comprehensive xG Model 177
Example 2: Predicting Player Actions 178
Example 3: Recognizing Tactical Patterns 178
Handling Imbalanced Data in Soccer 179
Using Feature Engineering for Tree-Based Models 180
Looking Ahead: From Handcrafted Rules to Learned Representations 180
Conclusion 181

7. Deep Learning for Soccer Analytics: From Intuition to Implementation. 183
Why Deep Learning for Soccer? Beyond Traditional Models 185
Looking Inside the Model: How Neural Networks Capture Patterns 186
Learning from Data: Training, Loss, and Optimization 190
Building Your First Soccer Prediction Model with PyTorch 193
Binary Outcomes: Will the Home Team Win? 194
Match Results: Win, Draw, or Loss 197
Goal Prediction: Modeling Scoring Output 200
Full Match Simulation: From Goals to Outcomes 204
A Second Lens: Implementing the Model in TensorFlow 209
Binary Outcomes: Will the Home Team Win? 210
Match Results: Win, Draw, or Loss 212
Goal Prediction: Modeling Scoring Output 214
Full Match Simulation: From Goals to Outcomes 216
Looking Ahead: From Flexible Models to Smarter Inputs 219
Conclusion 220

8. Feature Engineering for Soccer Analytics. 221
Feature Engineering: From Raw Data to Predictive Signals 222
Performance Features: The Fundamentals 222

Goals For and Goals Against: Measuring Scoring Performance 222
Venue Splits: Considering Home Versus Away 224
Shot Quality: Evaluating Chance Creation 226
Tactical Features: Describing How Teams Play 228
Passing and Possession: Measuring Ball Control 228
Tackles, Interceptions, and Recoveries: Tracking Defensive Actions 231
Pressing Intensity: Applying Defensive Pressure 232
Set Pieces: Capturing Dead-Ball Situations 234
Contextual Features: Incorporating Match Conditions 236
Rest and Scheduling: Modeling Fatigue and Congestion 236
Form and Momentum: Capturing Recent Performance Trends 237
Ranking Features and Advanced Ranking Algorithms: Estimating Team Strength 239
The Colley Matrix: A Linear Algebra Approach 240
PageRank: A Network-Based Perspective 242
Elo Ratings: A Dynamic Rating System 245
A Comparison of Methods: Strengths and Trade-Offs 248
Player-Level Features 249
Feature Engineering Pipeline: From Data to Model Input 251
Case Study: Predicting the 2019 Women's World Cup Final 255
Looking Ahead: From Better Inputs to Harder Decisions 256
Conclusion 257

9. The Art and Science of Soccer Betting. 259
Soccer Betting Basics 260
Types of Soccer Bets 260
Odds 262
Arbitrage Opportunities 266
The Real Dataset Used in This Chapter 270
Choosing a Side 272
Always Pick the Home Team 272
Always Pick the Lowest Odds 272
Use a Simple Machine Learning Baseline 273
Betting Strategies 274
Fixed-Stake Betting 275
Kelly Criterion 278
From Predictions to Potential Profit 282
The Reality of Profitability 282
Practical Considerations 282
The Importance of Discipline 283
The Role of Technology 284
Ethical Considerations 284

Looking Ahead: Building on Top of the Fundamentals 284
Conclusion 285

10. What's Next for Soccer Analytics?..... 287
What This Book Was Really About 287
From Questions to Data 288
Data Preparation Is Part of the Analysis 288
Exploration Before Prediction 288
Feature Engineering Is Where Domain Knowledge Enters 288
Models Support Judgment, They Do Not Replace It 289
The Next Stage in Soccer Analytics 289
From Static Datasets to Living Data Pipelines 289
From Team-Level Models to Player-Level and Lineup-Level Models 290
Richer Data: Tracking, Video, and Multimodal Analysis 291
Stronger Modeling Directions 293
From Notebooks to Analytical Products 294
AI and Large Language Models in the Analytics Workflow 294
Challenges and Limits 295
Data Quality and Access 296
Interpretability and Trust 296
Uncertainty Will Not Disappear 296
Ethics, Privacy, and Responsible Use 296
Beyond Soccer 297
A Practical Roadmap After This Book 298
Final Thoughts 299

Appendix: References and Further Reading..... 301

Glossary..... 309

Index..... 317

Preface

Soccer has always been a game driven by passion, intuition, and endless debate. For decades, it has been the world's most watched and discussed sport, supported by the largest global fan base on the planet. Fans love it not only for its results, but for the stories, emotions, memories, and moments of uncertainty—or what we sometimes call miracles—that it creates.

In recent years, this traditional way of enjoying soccer has been joined by something new: data. Advances in artificial intelligence (AI) and machine learning, combined with the growing availability of high-quality sports data and modern computing infrastructure, have made rigorous analysis accessible far beyond professional clubs and research labs. Today, fans and practitioners alike can explore tactics, evaluate performance, and make predictions—sometimes out of curiosity, sometimes for strategic insight, and sometimes as part of casual, responsible betting. With these tools, independent analysis is no longer limited to professionals.

What excites us most about this shift is not only the technology itself but also what it enables. Data-driven methods encourage clearer thinking, more disciplined reasoning, and a deeper appreciation of soccer's complexity. They do not replace passion or intuition; they sharpen them.

This book sits at the intersection of two fast-moving worlds: soccer and machine learning. Soccer provides rich, imperfect, and fascinating data; machine learning offers principled ways to engage with it. Together, they form an ideal setting for learning data analytics and predictive modeling in a concrete, motivating, and immediately applicable way.

The timing of this book is particularly meaningful. Its publication in 2026 comes during the FIFA World Cup, a moment when global attention, curiosity, and debate around soccer peak. As fans analyze matches, tactics, and outcomes with renewed intensity, this book offers an opportunity to engage more deeply with the game—to learn, experiment, and think critically about soccer through data and machine learning when interest is at its highest.

Who Should Read This Book

This book is written for soccer fans who want to explore the analytical side of the game. Readers should be willing to work with code and data, but no specialized technical background is required beyond basic programming familiarity and a general understanding of soccer. No prior experience with sports analytics is assumed.

More broadly, the book is ideal for anyone who wants to learn machine learning through real, meaningful problems rather than abstract exercises. This includes the following:

- Students of data science, statistics, optimization, computer science, or sports analytics
- Professionals seeking to strengthen applied machine learning skills
- Educators looking for engaging, hands-on material to teach predictive modeling

Why We Wrote This Book

Throughout our careers—in academia, industry, and teaching—we noticed a gap: no resource combined methodological rigor with hands-on practice, grounded in a single, coherent domain. Many machine learning books are abstract; many sports analytics resources are narrow or assume prior expertise.

This book attempts to fill that gap. It reflects a collaboration between industry and academic authors, blending hands-on experience with large-scale, real-world systems and deep expertise in statistics and sports analytics. It is the book we would have wanted when first applying machine learning to real data: practical, honest about limitations, and focused on building intuition as much as building models.

Our goal is that readers finish this book not only knowing how to implement predictive models, but also understanding why certain approaches work, when they fail, and how the same ideas can be applied beyond soccer to other real-world problems.

Navigating This Book

The book is organized to guide readers from context and fundamentals to advanced modeling techniques and practical applications:

- Chapter 1 introduces the field, motivates the book's focus, and frames the main analytical questions.
- Chapter 2 establishes the Python-based workflow used throughout the book.
- Chapter 3 covers core statistics, visualizations, and habits for understanding data in soccer datasets.

- Chapter 4 develops regression-based prediction for continuous and count outcomes.
- Chapter 5 introduces binary and multiclass classification tasks such as expected goals and match-result prediction.
- Chapter 6 extends the classification toolkit with tree-based and ensemble methods.
- Chapter 7 introduces neural networks and hands-on deep learning workflows.
- Chapter 8 shows how domain knowledge and data representation can significantly improve model performance.
- Chapter 9 examines betting, optimization, risk, and decision making under uncertainty.
- Chapter 10 draws the main ideas together and points toward future applications and further study.
- The Appendix and Glossary provide references and terminology to support ongoing learning.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.



This element signifies a tip or suggestion.



This element signifies a general note.



This element indicates a warning or caution.

Using Code Examples

For the latest book updates, additional resources, and related information, visit *https://SoccerAnalyticsML.com*. Companion code, exercises, and supplemental material are available at *https://oreil.ly/supp_SoccerAnalytics*.

If you have a technical question or a problem using the code examples, please send an email to *support@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: *Soccer Analytics with Machine Learning* by Haipeng Gao, Ari Joury, Weining Shen, and Guanyu Hu (O'Reilly). Copyright 2026 Haipeng Gao, Ari Joury, Weining Shen, and Guanyu Hu, 978-1-098-18111-6.

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning



For more than 40 years, O'Reilly Media has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *https://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
141 Stony Circle, Suite 195
Santa Rosa, CA 95401
800-889-8969 (in the United States or Canada)
707-827-7019 (international or local)
707-829-0104 (fax)
support@oreilly.com
https://oreilly.com/about/contact.html

We have a web page for this book, where we list errata and any additional information. You can access this page at *https://oreil.ly/soccer-analytics_ML*.

For news and information about our books and courses, visit *https://oreilly.com*.

Find us on LinkedIn: *https://linkedin.com/company/oreilly*.

Watch us on YouTube: *https://youtube.com/oreillymedia*.

Acknowledgments

This book would not have been possible without the support of our families, who patiently accepted the time and energy this project required. We are grateful to colleagues and collaborators who challenged our thinking, tested our ideas, and offered valuable feedback.

We also want to express our sincere thanks to the reviewers who read the manuscript so carefully and helped strengthen it in important ways. Their comments, corrections, questions, and suggestions pushed us to make this book clearer, more accurate, and more useful. We are especially grateful to Laurie Shaw, Poli Garcia, and Ijlal Amir, as well as to the reviewers who chose to remain anonymous, for the time, care, and thoughtful feedback they brought to the review process.

Our thanks also go to the editorial team at O'Reilly. We are grateful to senior acquisitions editor Michelle Smith, development editor Jeff Bleiel, and production editor Clare Laylock for their support and guidance throughout the development and publication of this book. We greatly appreciated the opportunity to work with all of you.

Last but not least, we want to thank the broader soccer analytics and open source communities. The shared datasets, tools, discussions, and collective curiosity within these communities made this work possible and remind us that progress in both soccer and data science is a collective effort.

CHAPTER 1

The Soccer Analytics Landscape

It is Saturday evening. You are at home watching Liverpool play. Before kickoff, you placed a small, casual bet on them to win—not as a professional strategy, but simply to make the match more engaging. As the game unfolds, your phone rests nearby with a live betting app open. The win probability updates continuously.

Liverpool strings together several passes in the opponent's half. The probability shifts from 54% to 57%. A sudden counterattack follows. A dangerous shot. The probability drops to 49%. The score has not changed. Yet your confidence moves with the numbers.

This experience captures a defining feature of modern soccer: the game is no longer interpreted solely through narrative and intuition. Increasingly, it is accompanied—and often understood—through probabilities.

But when a model assigns a 49% chance of victory, several natural questions immediately arise. What determines those probabilities? What assumptions underlie that number? How stable are these estimates? How should we interpret them? Which signals in the data are meaningful, and which are merely noise?

Answering these questions requires understanding the unique analytical challenges of soccer. The sport is low scoring and highly variable. Outcomes often hinge on a small number of events. Apparent dominance may not translate into goals, while short-term performance swings may reflect randomness rather than structural change. Human perception gravitates toward memorable moments; statistical reasoning, in contrast, requires consistency across many moments.

Soccer analytics exists within this tension. This chapter provides an overview of the modern soccer analytics landscape. We discuss the types of questions soccer analytics seeks to answer, the kinds of data available, the perspectives through which analysis can be conducted, the modeling approaches commonly used, and the conceptual

principles that guide disciplined evaluation. Using the preceding example, we introduce the broader scope of what soccer analytics is and how it is applied.

Equally important, this chapter also clarifies the scope of this book. Soccer analytics is a broad field, and covering every aspect would be impossible. Instead, we deliberately focus on a narrower domain: constructing predictive models that are rigorous enough to illustrate fundamental analytical tools while remaining accessible to readers.

By attempting to answer several classic yet interesting questions in soccer analysis, we hope the topics explored in this book will draw you into the field of soccer analytics and equip you with the modern analytical toolkit needed to further explore and advance the discipline.

The Problems That Soccer Analytics Helps Solve

Soccer analytics is not pursued simply for intellectual curiosity. In practice, it exists to support decisions.

Across professional clubs, leagues, media organizations, and betting markets, data-driven analysis helps answer a wide range of operational and strategic questions. The resulting decisions vary in scale—from real-time tactical adjustments during a match to long-term roster construction and financial planning. Understanding these decision contexts clarifies why analytics has become an integral part of modern soccer organizations.

Soccer Clubs: Making Decisions Across the Organization

Within professional soccer clubs, analytics supports a wide range of decisions across coaching, player management, and organizational operations.

Match preparation and tactical decisions

One of the most visible applications of analytics lies in match preparation and tactical planning. Coaching staffs must evaluate opponents, determine formations, select starting lineups, and plan substitution strategies. These decisions often rely on a combination of experience, video review, and statistical analysis.

Data analysts contribute by examining patterns across historical matches and identifying measurable tendencies. For example, analysts may evaluate how a team performs under different pressing intensities, how certain formations influence shot creation, or how substitution timing affects late-game scoring probabilities.

Typical questions analysts help address include the following:

- Which tactical setups historically perform best against a specific opponent?
- How does pressing intensity influence defensive stability?
- When does fatigue begin to meaningfully affect defensive performance?
- Which lineup combinations produce the most stable attacking structure?

Analytics does not replace coaching judgment. Instead, it provides structured evidence that complements tactical expertise and observational insight.

Player recruitment and roster management

Another major application of soccer analytics lies in player recruitment and long-term roster construction.

Professional clubs operate within financial constraints and competitive pressures. Decisions about signing, selling, or extending players often involve large financial commitments. Analytics helps reduce uncertainty in these decisions by identifying measurable performance indicators and projecting future contributions.

Recruitment departments frequently combine scouting reports with statistical profiles derived from historical event data. Analysts may evaluate how well a player's production translates across leagues, whether their style fits a specific tactical system, and how performance evolves across age ranges.

In practice, analysts often help answer questions such as these:

- Which players produce the highest expected goal contribution relative to their role?
- How stable are a player's performance metrics across seasons?
- How likely is a player's performance to translate into a different league or tactical system?
- What is a player's projected performance trajectory over the next several seasons?

These insights support transfer strategy, contract negotiations, and long-term roster planning.

Player performance and development

Modern clubs also rely on analytics to support athlete development and health management. Advances in sports science and wearable technology have enabled teams to collect detailed physiological and workload data. GPS tracking, heart-rate monitors, and training metrics allow performance staff to monitor player fatigue, recovery patterns, and injury risk.

Data analysts working alongside sports scientists and medical staff help translate this information into actionable insights.

For example, analytics can help answer questions such as these:

- How does cumulative training load influence injury risk?
- Which workload patterns improve performance without increasing fatigue?
- How does match congestion affect sprint output and recovery?
- What training interventions best support long-term player development?

These analyses support evidence-based training programs and help ensure that players remain healthy and available throughout the season.

Club operations and fan engagement

Professional soccer organizations are also complex businesses. Beyond the pitch, teams must manage ticket sales, marketing campaigns, sponsorship partnerships, and fan engagement. Analytics plays an increasingly important role in these operations.

Teams may develop predictive models to forecast ticket demand, optimize dynamic pricing strategies, or identify segments of fans most likely to respond to specific promotions. Digital engagement data—from mobile apps, streaming platforms, and social media—also provides insight into fan behavior.

Typical analytical questions in this domain include the following:

- How should ticket prices adjust based on opponent strength and demand?
- Which marketing campaigns generate the highest fan engagement?
- How can clubs improve fan retention across seasons?
- Which digital platforms drive the most interaction with supporters?

Although these business applications fall outside the technical scope of this book, they illustrate the broader organizational role of analytics within professional sports.

Betting Markets: Modeling Uncertainty and Pricing Outcomes

Soccer analytics also plays a central role in sports betting markets.

Bookmakers such as bet365, DraftKings, and other betting platforms must set odds that reflect the probability of multiple match outcomes. These odds act as market prices that allow betting activity to occur while also managing the financial exposure of the bookmaker. In practice, odds must strike a careful balance: they must be accurate enough to reflect the underlying probabilities of outcomes while also accounting for market behavior and betting volume.

To accomplish this, bookmakers rely on statistical models that estimate probabilities for events such as match outcomes, total goals, scorelines, and other in-game occurrences. These models typically incorporate large historical datasets that capture team performance, scoring patterns, and contextual factors. Variables may include measures of team strength, recent form, home-field advantage, injuries, travel schedules, and historical head-to-head matchups.

Many betting models build upon classical statistical frameworks such as Poisson goal models, which estimate the expected number of goals scored by each team. More advanced systems may incorporate machine learning methods that combine hundreds of predictive features derived from event data, rankings, and market information.

The objective of these models is not perfect prediction. Soccer remains a low-scoring and high-variance sport, where rare events and randomness can strongly influence outcomes. Instead, bookmakers aim to produce well-calibrated probabilities that allow markets to function efficiently while maintaining balanced risk exposure.

In addition to internal modeling, bookmakers must also react to the behavior of the betting market itself. When large volumes of wagers arrive on one side of a market, odds may adjust to rebalance risk. As a result, betting markets represent an interaction between statistical forecasting and market dynamics.

Because betting markets update continuously as new information arrives—lineups, injuries, weather conditions, or match events—probabilistic modeling and uncertainty estimation are central to their operation. Live betting systems in particular must update probabilities in real time as the match state evolves.

For analysts and researchers, betting markets also provide a useful benchmark. Because odds reflect the aggregated expectations of sophisticated models and informed bettors, they are often treated as a reference point when evaluating predictive models in soccer analytics.

Fans and Analysts: Reading the Game Beyond the Scoreline

Finally, soccer analytics is not limited to professional organizations. Fans and independent analysts increasingly use data and statistical models to better understand the game.

In many cases, this engagement begins with simple curiosity: estimating the probability that a team will win, comparing performance trends over time, or exploring which factors most strongly influence match outcomes. For some fans, building predictive models becomes an engaging way to interact with the sport, transforming passive viewing into an active analytical exercise.

This perspective connects directly to the scenario that opened this chapter. A fan watching a match may notice live win probabilities shifting from 54% to 57%, then falling again after a counterattack. Behind those numbers are statistical models trained on large collections of historical match data, continuously updating as the state of the game changes.

For independent analysts and enthusiasts, building such models can be both educational and entertaining. It provides a hands-on way to explore statistical reasoning, machine learning methods, and the dynamics of a complex sport. Through experimentation with data, fans can test hypotheses, evaluate predictive signals, and develop a deeper understanding of how performance translates into outcomes. This book is written largely for this audience: readers who want to understand how predictive models in soccer are constructed, evaluated, and interpreted using real data.

Soccer analytics therefore sits at the intersection of sport, data science, and decision making. From tactical preparation to roster construction, from player development to betting markets—and even fan curiosity—analytical tools help structure reasoning under uncertainty.

The Data-Driven Evolution in Soccer

Soccer's transformation from an intuition-driven craft to a data-informed discipline has reshaped the way the game is understood, analyzed, and managed at every level. This section traces that evolution—from the limits of traditional analysis, through the data infrastructure that made new approaches possible, to the metrics and applications that define the field today.

From Outcomes to Process: The Limits of Traditional Analysis

Soccer has long been described as the *beautiful game*—fluid, creative, and often unpredictable. For decades, match analysis relied primarily on observational expertise. Coaches and analysts interpreted performance through video review, accumulated experience, and a limited set of summary statistics such as goals scored, goals conceded, and league standings. These tools remain valuable, but they provide only a relatively coarse and often subjective view of performance.

Traditional evaluation also depended heavily on final outcomes. A league table, for example, awards the same three points for a 1–0 win against a title contender as for a 1–0 win against a relegation candidate. This approach is effective for determining standings, but it captures little about the contextual difficulty or underlying quality of each result. This limitation is especially consequential in soccer because the sport is relatively low scoring, and outcomes are often shaped by a small number of events—a deflection, a refereeing decision, or a single moment of individual brilliance. As a result, match outcomes alone can be noisy indicators of team quality.

Modern soccer analytics developed in part as a response to this problem. Instead of focusing only on whether a team won or lost, analysts increasingly examine the processes that generate outcomes: how chances were created, how defensive pressure was applied, how possession moved through space, and how players contributed both on and off the ball. These intermediate signals are often more informative than raw results for evaluating performance and, in many settings, more useful for predicting future outcomes.

The Data Revolution: From Event Logs to Tracking Systems

This shift has been enabled by a dramatic increase in the granularity of available data. Over the past two decades, several developments have transformed the informational environment of the sport. Event data records on-ball actions such as passes, shots, tackles, dribbles, interceptions, and fouls, typically with timestamps and spatial coordinates. This makes it possible to reconstruct sequences of play and quantify contributions such as progressive passing, entries into dangerous areas, and possession patterns.

Tracking data adds another layer of detail. Optical tracking systems and other electronic performance and tracking systems record the positions of players and the ball multiple times per second, making it possible to study off-ball movement, team shape, spacing, pressing behavior, and spatial control in ways that were previously difficult to measure. FIFA's electronic performance and tracking systems framework explicitly covers both camera-based and wearable technologies for measuring positional and physical information.

Training and physiological data further extend the analytical scope beyond match play. Wearable systems can help monitor workload, movement intensity, and recovery-related signals, supporting player development, load management, and injury-risk reduction. FIFA likewise notes that these systems may be combined with accelerometers, gyroscopes, and heart-rate monitoring to measure load and physiological parameters.

Underpinning all this is modern computational infrastructure. Cloud computing, large-scale data pipelines, and machine learning frameworks allow analysts to process large match databases, train predictive models, and simulate many possible scenarios efficiently. This has helped move soccer analysis beyond descriptive summaries and toward probabilistic modeling and decision support.

Analytics in Practice: Metrics, Applications, and Scope

Together, these developments have enabled forms of analysis that were previously difficult or impossible at scale. Expected goals, or xG, estimates the likelihood that a shot will result in a goal based on contextual information such as shot location and chance characteristics. Because it models shot quality rather than simply counting goals, it

is often treated as a more stable indicator of chance creation than raw scoring totals. Related ideas, such as expected assists (xA) and expected threat (xT), extend this logic to earlier phases of possession by estimating the value of actions that contribute to creating danger. StatsBomb, for example, describes xG as a model-based estimate derived from past similar shots and notes that models use contextual features like location, body part, and pass type.

At the team level, analysts also use metrics to describe style and defensive behavior. One widely used example is passes per defensive action, or PPDA, which indicates the number of opposition passes a team allows relative to its defensive actions in a defined area of the pitch; lower values are generally associated with more aggressive pressing. This kind of metric helps analysts compare team behavior systematically rather than relying only on qualitative impressions. These capabilities have reshaped decision making across the sport.

Clubs now integrate analytics into recruitment, performance analysis, and match preparation. Statistical profiles can help identify players who may be undervalued in the transfer market, while opponent-specific data can inform tactical planning. Media organizations increasingly use advanced metrics and visualizations to provide richer explanations of matches, and betting markets continuously update probabilities as new information becomes available. At the same time, analytics does not replace traditional expertise. Video analysis remains essential for interpreting tactical intent, and coaching judgment provides context that numbers alone cannot fully capture. In practice, analytics is most useful when it complements rather than substitutes for scouting, tactical observation, and domain knowledge.

At its core, soccer analytics still addresses familiar questions: Why did a particular outcome occur? What is likely to happen next? What decisions can improve the chances of success? What has changed is not the curiosity itself, but the quality, scale, and structure of the data used to investigate those questions.

This book operates within that framework—but within clearly defined boundaries. Soccer is not only a complex sport but also a difficult statistical environment: events are sparse, outcomes are highly variable, and contextual factors often influence performance in subtle ways. The next section directly examines these challenges before introducing the analytical framework that guides the rest of the book.

Why Soccer Resists Easy Analysis

Soccer presents several characteristics that make predictive modeling particularly challenging. First, it is a low-scoring sport. Matches often contain only a small number of goals, which increases variance and reduces the signal-to-noise ratio. A single event can disproportionately influence the outcome of a match. Second, soccer is continuous and state-dependent. Events occur sequentially and interact with one another. A defensive mistake early in the match may change tactical behavior for

the remaining 90 minutes. Third, direct experimentation is limited. Unlike many other sports, substitution opportunities are restricted, and tactical adjustments occur within tight constraints. This limits the ability to observe controlled comparisons. Fourth, context matters greatly. Opponent strength, match state, fatigue, travel schedules, and player availability all influence performance.

These contextual variables often interact in complex ways. Some influential factors are also rare or partially unobservable. Individual defensive positioning, communication among players, or tactical discipline may shape match outcomes without being fully captured in event data. These characteristics mean that soccer models must be carefully evaluated. Predictive systems require calibrated probabilities, robust validation procedures, and cautious interpretation. In this book, we rely on several guiding principles while acknowledging these challenges. Together, they shape the way models are designed, evaluated, and interpreted.

A Framework for Soccer Analysis

Soccer analytics is ultimately about improving the way we reason about a sport that contains substantial uncertainty. Analytical methods attempt to distinguish short-term fluctuations from the underlying signals that reflect team quality and performance.

To make sense of the field, this section organizes soccer analytics along three complementary axes: the *types of analysis* that describe the kinds of questions analysts attempt to answer, the *data sources* that enable modern analysis, and the *levels of aggregation* at which soccer performance can be studied. Together, these dimensions form the framework for most modern soccer analysis.

The Three Types of Analysis

Soccer analytics can be understood as a progression of three layers that move from understanding the past to guiding future decisions. Table 1-1 shows an overview of these layers.

Descriptive analytics focuses on summarizing what has already occurred. It transforms raw match data into interpretable measurements that describe team behavior and performance. Examples include pass completion rates, shot maps, pressing efficiency, and a team's performance shifts depending on match state—whether it is winning, losing, or drawing. These metrics establish a factual baseline that allows analysts to understand how a team has played.

Predictive analytics moves from description to estimation. Instead of asking what happened, it asks what is likely to happen next. Historical data is used to estimate probabilities for future events—models such as xG, probabilistic match outcome

forecasts, or ranking systems designed to estimate a team's underlying strength beyond its current position in the league table.

Prescriptive analytics connects predictions to decision making. Once a model produces a probabilistic forecast, the next question becomes how that information should influence strategy. Professional clubs may use simulation models to evaluate substitution timing, identify optimal transfer targets under budget constraints, or explore tactical adjustments against specific opponents.

Table 1-1. Common types of analytics and how they map to soccer decisions

Analytics type	Core question	Soccer example	Typical outputs
Descriptive	What happened?	Shot maps and xG summaries for a match	Summaries, dashboards, distributions
Predictive	What is likely to happen?	Win/draw/loss forecasts; shot-level goal probability	Probabilities, risk estimates, forecasts
Prescriptive	What should we do?	Substitution timing based on fatigue and match state	Recommendations, optimized actions, scenario analysis

This book focuses primarily on the descriptive and predictive layers. We begin with disciplined descriptive analysis and then develop models designed to forecast match outcomes.

Data Sources for Modern Analysis

The growth of soccer analytics has been driven largely by the increasing richness and resolution of available data. Several categories now form the foundation of modern analysis. A summary is shown in Table 1-2.

Event data records discrete on-ball actions during a match. Passes, shots, tackles, interceptions, dribbles, and fouls are logged with timestamps and spatial coordinates. Because it captures the sequence of play, event data allows analysts to reconstruct attacking buildup, measure ball progression, and evaluate how teams create scoring opportunities. Event data is also the most widely available data source and forms the primary dataset used in this book.

Tracking data provides much finer spatial resolution. Optical camera systems or GPS sensors capture the positions of all players and the ball many times per second—often around 25 frames per second. This data reveals the spatial context behind events, allowing analysts to study defensive shape, spacing between players, pressing intensity, and off-ball movement.

Physiological and performance data records metrics such as sprint counts, high-intensity running distance, heart rate, and recovery indicators. These measurements are typically used by sports science departments to monitor player workload, reduce injury risk, and support long-term player development.

Betting market data offers a distinct and informative perspective. Bookmakers aggregate vast amounts of information to produce odds that reflect market expectations about match outcomes. Converting these odds into implied probabilities allows analysts to benchmark predictive models against a highly competitive forecasting environment.

Table 1-2. Core soccer data sources and practical considerations

Data type	Content	Strengths	Common cautions
Event data	Timestamped on-ball actions (passes, shots, tackles, etc.)	Widely available; supports most analytical workflows	Limited off-ball context; vendor definitions may differ.
Tracking data	High-frequency player and ball coordinates	Captures spacing, shape, and off-ball movement	Large volume; requires careful alignment and standardization.
Physiological and performance data	Player health data such as sprint counts and heart rate	Reduces injury risk and supports long-term player development	Such data is usually not public and will not be used in this book.
Betting market data	Bookmaker odds and implied probabilities	Reflects aggregated forecasting expertise	Margins and liquidity vary; odds reflect public information.

From Player to League: Levels of Aggregation

Soccer performance can be studied at different levels of aggregation, and the choice of level shapes both what questions can be asked and how reliably they can be answered. Table 1-3 provides a summary.

At the *player level*, analysts attempt to isolate the contribution of individual actions. Modern metrics such as on-ball value (OBV) and valuing actions by estimating probabilities (VAEP) estimate how much a single pass, shot, or defensive action changes a team's probability of scoring or conceding. These approaches are particularly valuable in scouting and recruitment.

At the *team level*, individual actions are aggregated into collective performance indicators. Analysts examine patterns such as shot quality, defensive efficiency, possession structure, and chance creation. Because team-level metrics tend to be more stable over time than individual statistics, they provide a stronger foundation for predictive modeling.

At the *match level*, analysts focus on the dynamics of a single game. Contextual factors such as opponent strength, scoreline, red cards, and game state strongly influence team behavior. Adjusting for these conditions helps separate underlying team quality from situational effects.

At the *league level*, analysis provides the broader competitive context. Teams compete within environments that differ in playing style, pace, and scoring frequency. League-level adjustments allow analysts to account for differences in competition quality and strength of schedule when comparing performance across competitions.

Table 1-3. The soccer analytics hierarchy and what each level contributes

Level	Unit	Relevance	Example adjustment
Player	Individual	Skill, role, and playing time shape observed performance	Apply per-90 normalization and position-based modeling.
Team	Club or national team	Style and teammates influence player opportunities	Control for possession, tempo, and tactical context.
Match	Single game	Opponent strength and match state affect behavior	Include opponent quality, scoreline, and game context.
League	League or tournament	Baseline difficulty varies across countries and divisions	Adjust for league strength when comparing player output.

This book focuses primarily on the team level. By treating the team as the central unit of analysis, we can construct predictive models that capture persistent performance signals while remaining robust to short-term fluctuations in individual player performance.

Principles That Run Through Everything

Beyond layers, data, and levels, several conceptual ideas appear repeatedly throughout modern soccer analytics. Rather than isolated rules, they form a connected way of thinking about evidence and inference in a noisy, complex sport.

The first is the distinction between *association and causation*. Statistical relationships do not automatically imply causal explanations. A variable may improve the predictive accuracy of a model without representing a true causal mechanism. Certain passing patterns may correlate with winning, for instance, yet the underlying driver might be broader tactical structure rather than the passes themselves. Analysts must therefore interpret predictive variables with care, resisting the temptation to read causation into correlation.

The second is *heterogeneity*. Teams differ substantially in playing style, tactical philosophy, financial resources, and competitive environment. Models that assume uniform behavior across all teams often perform poorly precisely because they ignore these structural differences. Recognizing heterogeneity helps analysts design models that remain robust across leagues, seasons, and tactical contexts.

The third is the importance of *contextual comparison*. Performance metrics are meaningful only when interpreted relative to context. Raw totals—goals scored, possession percentage—do not automatically indicate team strength. A team that scores frequently against weaker opposition may not perform as well against stronger competition. Ranking systems and strength-of-schedule adjustments provide a structured way to make comparisons that are fair and informative.

These three principles—causal discipline, structural awareness, and contextual interpretation—guide both model design and the way results are read throughout the chapters that follow.

Looking Ahead: A Guided Tour of the Book

Soccer analytics spans a wide range of modeling approaches, from simple statistical summaries to modern machine learning systems. Rather than focusing on any single method, this book presents a structured progression of techniques, with each layer of complexity building on the foundations established before it.

The journey begins with Chapter 2, which establishes the computational workflow used throughout the rest of the book. In Chapter 3, we then examine data structure, summarize performance, and visualize play.

From there, the modeling chapters proceed in sequence. Chapter 4 develops continuous-outcome and count-based prediction. Chapter 5 introduces binary and multiclass classification. Chapter 6 extends that work with tree-based and ensemble models.

Chapter 7 introduces neural-network-based methods. Chapter 8 shows how to turn soccer knowledge into stronger model inputs. Chapter 9 connects forecasting to valuation, staking, and decision making under uncertainty. Finally, Chapter 10 draws the main ideas together and points to the next steps after the book.

Across all these approaches, three methodological principles remain central: validation, calibration, and interpretability. The goal is not algorithmic novelty, but disciplined predictive modeling within a clear and practical analytical framework.

Conclusion

Soccer analytics is no longer a niche pursuit; it has become a fundamental lens through which the game is understood, managed, and enjoyed. The shift from intuition-driven observation to data-informed analysis has reshaped decision making across professional clubs, betting markets, and the broader fan community. By moving beyond raw outcomes to examine the underlying processes, analysts can now quantify performance, evaluate risk, and uncover patterns that traditional methods often miss. Yet the sport's inherent low-scoring nature, continuous flow, and complex contextual dependencies ensure that modeling soccer remains a challenging endeavor that requires both statistical rigor and deep domain knowledge.

This chapter has mapped the territory, establishing the descriptive, predictive, and prescriptive layers that define modern analysis, while outlining the data sources and levels of aggregation that make such work possible. The principles of causal discipline, structural awareness, and contextual interpretation introduced here will

serve as our compass. As we move forward into the practical application of these concepts, starting with the computational foundations in the next chapter, the goal is not simply to build models but to develop a disciplined, probabilistic approach to reasoning about the beautiful game.

CHAPTER 2

Python Fundamentals: Building Your Analytics Toolkit

In Chapter 1, we mapped the analytical landscape of soccer and clarified the focus of this book: using data to describe matches, explain patterns, and build predictions. To do that well, we need a reliable computational environment in which analysis can be written, tested, reproduced, and extended.

Python is that environment for the rest of the book. It gives us a readable language, a mature ecosystem of data tools, and a practical workspace for moving from raw data to analytical output. Before we analyze matches, engineer features, or train predictive models, we first need to set up that workspace and become comfortable with its core habits.

This chapter therefore builds the foundation for everything that follows. You will install Python, create an isolated working environment, use Jupyter Notebook, and learn the syntax and data structures that later chapters rely on. The material is introductory, but it underpins every analysis, model, and conclusion in the book.

Even when the examples in this chapter are intentionally simple, we will try to keep them close to the kinds of objects that soccer analysts actually work with: team names, scores, match outcomes, shot counts, and basic summaries. The point is to learn the programming move in a context that will still feel familiar when we return to full match and event datasets.

Python: The Language Behind Modern Analytics

Python is well suited to soccer analytics for several practical reasons. It is readable, expressive, and approachable, even for those with limited programming experience. Its syntax is designed to be clear and concise, allowing complex ideas to be

implemented with relatively little code. This emphasis on clarity—often described as being *Pythonic*—reduces friction between thinking and implementation.

Beyond its accessibility, Python has become one of the dominant languages in data science and machine learning. Its widespread adoption has led to a mature ecosystem of libraries, tools, and contributors across many domains. Sports analytics is no exception. A growing community of analysts relies on Python for data processing, modeling, visualization, and predictive work in soccer and other sports.

The strength of the ecosystem also provides practical advantages. Extensive documentation, community support, and open source collaboration make it easier to troubleshoot issues, learn new techniques, and build on existing work. This collective infrastructure lowers barriers while maintaining methodological rigor.

Although this book focuses on soccer, the skills developed here extend well beyond a single sport. The workflows, libraries, and modeling patterns introduced in this chapter are widely applicable across machine learning tasks in many domains.

Let's now start the practical work: installing Python, creating a virtual environment, and launching Jupyter Notebook.

Your Analytics Workspace: Setting Up the Environment

A good analytical workspace should be organized, reproducible, and easy to return to. Before we write serious code, it helps to understand the four main pieces of that workspace: Python itself, a virtual environment, Jupyter Notebook, and repositories.

Python is the main environment where the analysis takes place. A virtual environment keeps each project's software dependencies separate, so one project does not accidentally interfere with another. You can think of this virtual environment as a dedicated training room inside a gym: each project has its own room and equipment, with the tools it needs arranged in one place. This separation also protects the rest of your computer from project-specific changes and makes your work easier to reproduce later.

Jupyter Notebook gives you an interactive way to work through analysis step by step. Instead of running one large script all at once, you can organize your work into code cells for loading data, cleaning it, exploring patterns, training models, and evaluating results. That makes it especially useful for learning, experimentation, and iterative analysis.

Your first step is to install Python. After that, we will set up a virtual environment and a Jupyter notebook so you can begin running and experimenting with Python code.



For a quicker start, readers with a Google account can create a *.ipynb* Colaboratory file in Google Drive to begin using Jupyter Notebook immediately. If you choose Google Colab instead of working locally, you can skip the next two sections on virtual environments and how to create them!

Installation: Getting Python Ready

You can download and install Python directly from its download page. You can just install the most recent version for your operating system (typically Mac, Linux, or Windows) for now.

Once Python is downloaded, run the installer and follow the prompts. To verify the installation, you'll need to open a terminal or command prompt window. If you've never used one before, search your applications for *Terminal* or *Command prompt* (almost every computer comes with one preinstalled). Then, type the following command on the first line:

```
python3 --version
```

After pressing Enter, you should see the Python version number just below the prompt. In that case, the installation was successful.

If you're encountering errors, try specifying your exact problem (including the Python distribution number), your machine and exact operating system (such as macOS High Sierra or Windows 10), and the error you encountered in ChatGPT, Claude, or another chatbot of your choice. These chatbots can help guide you further if you're stuck on errors.

Virtual Environments: Isolating Your Projects

A Python *virtual environment* is an isolated, self-contained directory that includes a specific Python installation and a set of additional packages. In practice, this virtual environment gives each project its own controlled software setup, so one project does not accidentally interfere with another.

At first, creating a virtual environment for every project may feel like an extra step. In practice, that separation matters because Python projects often depend on specific library versions. A model you build today might rely on package versions that differ from the ones required by another project a few months later. If everything is installed into one shared environment, those dependencies can easily conflict.

Virtual environments solve this by giving each project its own local combination of Python and libraries. That makes your work easier to reproduce, easier to debug, and less likely to break when you install new packages elsewhere on your machine. For exactly this reason, using virtual environments is a widely accepted best practice in software development. We will talk more about libraries in Chapter 3.

Now, imagine that you're working on two projects:

- Project A requires Python 3.7 and TensorFlow 1.15.
- Project B requires Python 3.8 and TensorFlow 2.3.

The numbers 3.7 and 3.8 refer to specific releases of Python. TensorFlow is a commonly used package in machine learning that comes with its own releases. Such specific requirements typically arise because the project relies on particular functionalities that are available in only certain versions of Python and/or any package.

In addition, certain packages and Python versions cannot always be mixed. If a project depends on an older release of TensorFlow, for example, it may not work with every newer Python version. When that happens, a virtual environment gives you a safe place to install exactly the versions the project needs without disrupting anything else on your machine. If you are unsure about compatibility, the package documentation is usually the best place to check.

A typical analyst will work on multiple projects over time, each with its own software requirements. Virtual environments keep those requirements separate and reduce the risk of version conflicts.

To reiterate, here are the key features of virtual environments:

Isolation
: Each virtual environment contains its own Python installation and packages, which helps prevent conflicts between projects.

Project specificity
: You can create a setup for each project with exactly the dependencies it needs.

Reproducibility
: Virtual environments make it easier to re-create the same software setup on another machine and obtain consistent behavior.

To create and manage virtual environments, Python comes with a built-in tool called `venv`. It lets you build isolated spaces where your packages and dependencies can live—separate from the rest of your computer.

Let's create a virtual environment for a new project called `soccer_analytics`. The following command will use whichever Python version your `python3` command points to, so make sure it refers to a recent installation.

First, make sure you've installed Python (see "Installation: Getting Python Ready" on page 17). Then follow these steps:

1. Create the environment:

   ```
   python3 -m venv soccer_analytics
   ```

   The flag `-m` tells Python to run a specific module rather than a plain script—in this case, `venv`, which creates the virtual environment.

2. Activate the environment. Once you've created the environment, you need to activate it to work within it. How exactly to do that depends on your operating system.

   For macOS or Linux, you'll type this in your terminal:

   ```
   source soccer_analytics/bin/activate
   ```

   For Windows, the command depends on whether you're using Command Prompt (CMD) or PowerShell:

   ```
   soccer_analytics\Scripts\activate.bat  # for CMD
   .\soccer_analytics\Scripts\Activate.ps1  # for PowerShell
   ```

   The # symbol denotes a comment, so everything after it is only for humans and ignored by computers.

   After activating a virtual environment, the terminal prompt should change to show the name of the active environment (e.g., `soccer_analytics` appears at the start of the prompt).

3. After you're done working in the virtual environment, deactivate it. To deactivate the current virtual environment, you simply type in the terminal or command prompt:

   ```
   deactivate
   ```

   You will then return to the base environment on your main operating system, or to whichever state it was in before any environments had ever been activated.



Multiple ways to create virtual environments exist, and new methods may still emerge. However, this approach is a tried-and-tested way to get you started fast.

Jupyter Notebook: Creating a Workspace for Interactive Analysis and More

Having a dedicated training room is not enough on its own. You also need a structured training routine—one that breaks your session into clear, ordered steps, and records what happened at each one. Think of a serious weightlifting program: you don't just walk into the gym and lift randomly. You follow a sequence—biceps first, then legs, then core—and after each set, you log the weight, the reps, and how it felt. That record is invaluable. Without it, your training is essentially binary: either the session went well overall, or it didn't—but if your training progress slows or you stop meeting your goals, you have no way of reviewing your specific progress or knowing what to tweak. Jupyter Notebook works precisely this way.

The virtual environment you just created will allow you to run both Python scripts and Jupyter notebooks. Files that are Python scripts end with *.py* and are good for quick execution and production-grade models. A *.py* script is like completing your entire workout and checking the logbook only at the very end. You can see the final result, but if you pulled a muscle midway through the workout, tracing the exact move that caused it is difficult.

However, when you're still building and exploring, Jupyter notebooks are better. These files, which end with *.ipynb*, have transformed the way data scientists, researchers, and educators interact with code and data. Jupyter notebooks are great because they allow you to create and share documents that integrate not only live code but also equations, visualizations, and narrative text.

Features and benefits of Jupyter notebooks

Think of a Jupyter notebook as your structured training log come to life. Each *cell* is one step in your routine—one exercise, one set—and you can run it, immediately see the result, add a note explaining your reasoning, and only then move on to the next. If something goes wrong at the leg press, you know it was the leg press—not a mystery buried somewhere in an hour-long session. That kind of step-by-step visibility is exactly what makes Jupyter notebooks so powerful for exploration and analysis.

Here is a more detailed overview of the features and benefits of Jupyter notebooks:

Interactive computing
: They execute code in small, manageable chunks (as noted earlier, these are often known as *cells*). This capability supports rapid experimentation and iteration, simplifies debugging, and allows you to easily check out intermediate results. This is particularly helpful when you want to view outputs such as results or visualizations (e.g., plots), especially when working with data. Think of each cell as a single exercise set—you perform it, check your form in the mirror, adjust,

and move on to the next one. You don't have to complete the entire workout before finding out whether your technique was off.

Visualization
: Notebooks enable inline display of a wide range of visualizations, which enhances data exploration and presentation. This makes data analysis more dynamic and informative. It's like having a performance screen mounted on the wall of your training room, showing your stats in real time as you work through each set.

Blended content
: Notebooks seamlessly integrate executable code with explanatory text, mathematics, and rich media, resulting in comprehensive and self-documenting analyses that are excellent for presentations. Imagine a training log that doesn't record just your lifts but also includes coaching notes, diagrams of proper form, and video clips—all in one place, ready to share with your team.

Extensibility
: Many plug-ins and extensions are available to enhance the functionality of Jupyter notebooks, such as tools for code formatting, magic functions, and more. Just as you might add a foam roller station, a heart-rate monitor, or a resistance band rack to customize your training room, you can extend Jupyter to suit exactly the way you like to work.

In addition to all this, Jupyter notebooks support a wide range of programming languages, including R, which is another popular and powerful tool for statistical analysis. This makes them even more suitable for various data science and analytical tasks. Think of them as a multipurpose training room that accommodates not just one sport but several; whether you're doing strength work, cardio, or mobility training, the room adapts to your needs.

Your first notebook: Getting up and running

You're ready to set up your training room and get your first session underway. To launch a Jupyter notebook (which we sometimes just call Jupyter or a notebook for short), open a terminal and enter the following commands:

```
pip install jupyter
jupyter notebook
```

The first command produces a lot of output; it gives you the ability to create and play around with Jupyter notebooks. Think of it as the delivery truck arriving with all your equipment—a lot is coming off the truck, but once it's unloaded, your room is stocked. The second command opens a new browser window or tab with the Jupyter interface. Now, you can create a new notebook in Jupyter via the graphical user interface in your browser.

Once you have a notebook up and running, click the first cell and run the following code. To execute a notebook cell, you can click the arrow in the top menu bar, or you can hit Shift-Enter—like snapping straight into your next exercise without losing your rhythm or breaking a sweat between sets:

```
import sys
import matplotlib
import numpy as np
import pandas as pd
print(f"Python version: {sys.version}")
print(f"NumPy version: {np.__version__}")
print(f"Pandas version: {pd.__version__}")
print(f"Matplotlib version: {matplotlib.__version__}")
```

This code confirms that essential data science libraries are correctly installed. It's your presession equipment check—making sure the barbells are loaded, the machines are functioning, and nothing is missing before you begin. We'll talk more about libraries in Chapter 3, so don't worry if you don't understand what the preceding commands do just yet!

If you encounter any errors at this point, create a separate cell with the + button, and use the up or down arrow to move the cell if needed. Then, you'll need to run the following snippet in that cell (again, click the side arrow symbol or press Shift-Enter to execute it):

```
%pip install matplotlib
```

Note that you might need to change `matplotlib` to whichever library was throwing an error for you previously. This is like discovering that a piece of equipment is missing from your room and quickly ordering a replacement—a quick fix before the real training begins. Once all libraries are loaded, you have set up your first fully equipped training room—the right space, the right gear, all zones clearly marked, and everything in its place. Now you need to learn how to train.

Repositories: Storing Your Project Files

As you begin working with Python for analytics, you'll quickly encounter the concept of a repository. A *repository* is a home for a project's files. It keeps code, notebooks, datasets, and documentation together in one place so that you can return to them, organize them, and share them with others.

Many repositories are hosted on platforms like GitHub. You may also come across similar platforms such as GitLab or Bitbucket. You do not need to become an expert in Git or GitHub right now to follow this book. For our purposes, it is enough to understand that a repository is the place where a project lives. In later chapters, you may download data files, notebooks, or example code from a repository that accompanies the book.

Behind the scenes, repositories are usually managed with a version control system called *Git*. *Version control* keeps track of changes to files over time, which makes it easier to update a project, compare older and newer versions, and collaborate with other people without losing work. You do not need to learn Git in detail yet, but it helps to know that Git is the tool that makes repositories so useful.

In practice, beginners usually interact with repositories in one of two ways:

Download a ZIP file
: This is the simplest option if you just want a copy of the files on your computer.

Clone the repository
: This creates a local working copy that is connected to the online version, which makes it easier to update your files later.

If you're new to this, think of a repository as your project's locker room: everything related to the project is stored there in an organized way. Git then acts like the staff logbook that records what changed, when it changed, and who changed it, so the project is easier to manage over time. As we move into Chapter 3, this idea will become useful when we start working with real data and code files outside a single notebook.

Python Basics: The Core Moves

In soccer, you don't need to know how to do a full Ronaldo chop to play a decent game (though we'd like to see you do that in due time as well). The ability to dribble, pass, and shoot is already going to get you from zero to being a respectable force to be reckoned with on the field.

As a soccer data analyst, you'll need to know some basic concepts. At this stage, a Jupyter notebook may feel unfamiliar and mostly empty—like stepping onto a soccer field for the first time without knowing how the game is played.

Starting as a beginner is expected. We are here to help you grow your core Python skills step by step so you can navigate this environment effectively. As an analyst, you'll work with data types, variables, loops, and other essential fundamentals.

If that sounds a bit intimidating, don't worry. We'll go through all this step by step, and you'll have plenty of time to revisit these concepts as you proceed through the book. If, on the other hand, you're already somewhat familiar with these concepts, consider the remainder of this chapter like a warm-up exercise: revisit regularly but remember to take it easy!

Primitive Data Types: Representing Basic Information

As an analyst, your primary basis of work is data. Think of it as the soccer moves that players perform every day. Data comes in many forms. Various *primitive* data types, which are built-in in Python, represent single values. More complex data structures also exist and are discussed further in this chapter.

The four fundamental primitive data types in Python are *integers*, *floats*, *strings*, and *Booleans*:

- Integers are whole numbers like `2` goals, `38` league matches, or `90` minutes.
- Floats are numbers with decimal places like `0.73` xG or `61.5` possession.
- Strings are, well, strings of characters put together, like `"Liverpool"` or `"Quarter final"`; they are always found inside single or double quotation marks.
- Booleans can take only two possible values: `True` or `False`. They might sound simplistic at first glance, but they're quite handy for performing sanity checks, executing `if` statements, and representing details such as whether a pass was completed.

A soccer player intuitively knows whether an excellent pass or a complete curveball is coming their way. As an analyst, you have a tool if your intuition isn't enough: the `type()` function. (You'll learn more about functions in "Functions: Providing Reusable Logic for Analysis" on page 36.) A function returns the type or class of whichever object you give it as an argument.

Here are some examples using the `type()` function to determine the data type:

```
print(type(True))
print(type(2))
print(type(0.73))
print(type("Liverpool"))
```

Note that you can execute these examples in your Jupyter notebook, and you'll see that you get the same values. In this chapter, code examples are written as plain runnable snippets so you can copy them directly into a notebook cell. When it is helpful to show the result, we place it in a separate output block right after the code.

If you run this code, you should see the following:

```
<class 'bool'>
<class 'int'>
<class 'float'>
<class 'str'>
```

You can see that the data types are somewhat abbreviated. As you might have expected, the values `True`, `2`, `0.73`, and `"Liverpool"` are `boolean`, `int`, `float`, and `string` types, respectively.

Consider these primitive data types your basic passes and dribbles, which you will use constantly to represent soccer-related data. If you want to combine different moves (like dribbling and then passing to get the ball over to your striker), you'll need some operations to glue those moves together.

Basic Operations: Working with Values and Comparisons

Sometimes you'll need to combine a dribble and a pass. In your world as an analyst, you have operations to combine data types. They fall into two groups: *arithmetic* operations and *comparisons*.

Let's look at these basic operations. We'll use the same conventions as in the example in the preceding section. In this chapter, comments are kept brief and used only when they add clarity. Here are some examples:

```
print(1 and True)
print(0 or 1)
print(5 > 1)
print(1 > True)
print(1 + 5)
print(True * 5)
print(5 / 2)
print(7 ** 2)
print("Team A" + ":" + "Win")
```

This produces the following:

```
True
1
True
False
6
5
2.5
49
'Team A:Win'
```

You'll notice that `1` is regarded as equivalent to `True` (see the first line). Conversely, `0` is equivalent to `False`. This can come in handy when you don't want to type out the words `True` and `False`.

The second expression, `0 or 1`, returns `1` because it's true that anything is either `False` (`0`) or `True` (`1`). Conversely, if you typed `0 and 1` into your notebook cell and executed it, it would return `0` or `False`, because nothing can be `True` and `False` at the same time.

The symbols `<` and `>` are basic comparisons. It's true that `5` is bigger than `1`. But it's not true that `True` (i.e., `1` as we now know) is bigger than `1`.

Other standard math operations like `+` work too. They even hold up for `True` and `False`, which are then just interpreted as `1` and `0`, respectively. The symbol `**` denotes exponentiation. This means that `7 ** 2` is the same as the multiplication `7 * 7`. Analogously, `7 ** 3` is the same as `7 * 7 * 7`, and so on.

Finally, strings of characters can be joined using the `+` operator. So here we're not talking about an addition in the mathematical sense but rather about gluing several strings together. Hence, the three strings `"Team A"`, `":"`, and `"Win"` can be glued together to form the new string `"Team A:Win"`.

Variable Assignment: Storing and Reusing Information

Sometimes you know the objective is to move the ball from your own penalty area up to the halfway line, but you are not yet sure how to execute it. Do you dribble forward under pressure, or pass to a teammate to help progress the play?

In soccer analytics, this reflects a more general problem: you often need to represent something before it is fully known. For example, when analyzing your favorite team's performance, you do not know in advance how many goals it will score in a match. The outcome depends on many factors, including opponent strength, match context, and a degree of randomness.

In Python, we handle this kind of uncertainty using variables. A *variable* is a named container for a value that can be stored, updated, and reused throughout your analysis. For example, you might create a variable called `home_team_goals` and assign it a value as your analysis develops. This process is called *variable assignment*, and it is one of the most fundamental concepts in Python and in programming more broadly.

Variables are powerful because they allow you to store and reuse information efficiently. You can update a variable at any time and automatically use its most recent value throughout your code. For instance, if you initially assume your team will score three goals but you later update that expectation to two, you do not need to change every instance in your code. You simply update `home_team_goals`, and all downstream calculations will reflect the change when the code is rerun. This makes code easier to maintain, less error-prone, and more flexible when working with real-world data.

Here is an example of two variables being assigned and then accessed:

```
home_team_goals = 2
away_team_goals = 1
print(home_team_goals)
print(away_team_goals)
```

This prints the following:

```
2
1
```

So here you have that your home team scored two goals, and the away team scored one goal. This example uses *snake case* for naming: words are separated with underscores, and all letters remain lowercase. Examples include `home_team_goals` and `away_team_goals`. You will often see snake case in Python code, libraries, and documentation, and in this book we'll keep our own examples consistent by mainly using it in our variable names.

Another naming convention you may encounter is *camel case*: the first letter of each word after the first is capitalized, as in `homeTeamGoals`. Recognizing both styles is useful, but for consistency we will stick to snake case in our examples.

Whichever naming style you choose in your own work, the important point is to pick one convention and stick to it as much as you can. When you follow a consistent convention, you'll find it easier to keep an overview of all your variables and keep the code clean if you add more functionalities later.

Data Structures: Organizing Information

Sometimes counting goals in a single match is not enough. You may want to store data across multiple games or track all results for a specific team over a season. At this point, simple data types are no longer sufficient.

To move to the next level, you need more structured ways to organize data. In Python, lists, tuples, dictionaries, and sets provide that foundation. These data structures allow you to store and manage groups of values.

Each structure has a different level of flexibility and organization. Table 2-1 provides a high-level overview of how these data structures compare.

Table 2-1. Overview of the four data structures

Data structure	Ordered?	Changeable?	Duplicate values allowed?	Typical use
List	Yes	Yes	Yes	Storing a sequence of values you may want to update or slice
Tuple	Yes	No	Yes	Storing a fixed record that should not change
Set	No	Yes	No	Storing unique values and comparing collections
Dictionary	Yes	Yes	Keys must be unique	Storing values by name, using key-value pairs

Lists: Ordered and flexible collections

A *list* in Python is just an array of several data entries. You can assign a list to a variable so you can play around with it later. That is basically like giving the list a name so you can call it when you need it.

Like a regular to-do list, a list in Python has specific properties. It usually has a length, indicating the number of items in it. You can find out the length with the built-in function `len()`. You can also find out the maximum and minimum values in a list with `max()` and `min()`, respectively (as long all items are integers or floats, this will work).

And you can *slice* lists, or select a subsection of them. Note that in Python, the counter starts with 0 and not with 1, so the first item on a list is called with the index `0`.

In the following example, we're counting the goals a team has scored over multiple games. We can figure out the number of games they played, the most goals scored in a single game, and the number of goals scored in the first three games:

```
goals_per_game = [0, 1, 1, 0, 2]
print(len(goals_per_game))
print(max(goals_per_game))
print(goals_per_game[0:3])
```

This prints the following:

```
5
2
[0, 1, 1]
```

We can see that the team played five games and scored a maximum of two goals in any of those games. To get a list of goals scored per game in the first three games, we use `goals_per_game`, with the expression `[0:3]`, which is a slice that extracts a subset of the list.

When we slice a list, `0` is the start index. The start index is always inclusive, which means that the element at that index is taken into account. In the preceding case, `3` is the end index. The end index is always exclusive, which means that the element at that index is *not* taken into account. This is why you find only the elements at positions 0, 1, and 2 in the returned output, but not the element at position 3 (the fourth element). Therefore, `[0:3]` returns the elements from index 0 up to, but not including, index 3. In this example, the team scored no goals in the first game, one in the second, and one in the third.

Lists are particularly well suited for representing sequences such as goals or other statistics, and they make it easy to access specific entries through slicing. Although it's possible to mix data types within lists, maintaining consistency—such as using only integers—is cleaner and generally leads to fewer unexpected errors.

Tuples: Fixed records and immutable structure

Tuples are immutable sequences, meaning they cannot be changed after creation. This makes them different from lists, because you can modify lists however you like. To create tuples, you use parentheses instead of square brackets.

For example, let's say that we have a tuple for the result of a match, indicating the number of goals each team scored. We might also want to store one team's past points records in the league, just for good measure. If we try to tamper with either tuple, we'll run into trouble:

```
match_result = (2, 1)
points_record = (3, 0, 1)
try:
    match_result[0] = 4
except TypeError as error:
    print(f"Error: {error}")
```

This prints the following:

```
Error: 'tuple' object does not support item assignment
```

You can see that the tuple is immutable because trying to manipulate the first entry (or any entry, for that matter) results in an error message. Tuples are therefore very useful when you need to make sure you're not making mistakes, or when you're relying on historical records that cannot change because they happened in the past.

Sets: Collections of unique elements

Sets are useful for storing unique data. Unlike tuples, order doesn't matter, and no duplicates are allowed. Unlike tuples, sets are also changeable; you can add or remove elements after creating them. Sets are thus useful for maintaining unique collections such as team IDs or names.

What's great about sets is that you can compare them. For example, if you want to find out which players both scored a goal and received a red card in the last game, you could check the intersection between the set that contains the players who have scored a goal and the set that contains the names of the players who got a red card. To do this, you'd use the `&` operator. If, on the other hand, you want to find all the players who either scored a goal or got a red card, or both, you should use the `|` operator. And if you want to obtain the names of the players who scored but did *not* get a red card, you should use the `-` operator. Here's how it works in detail:

```
players_scored_last_game = {"Lionel Messi", "Cristiano Ronaldo"}
players_received_red_card = {"Cristiano Ronaldo", "Luis Suárez"}
all_unique_players = sorted(
    players_scored_last_game | players_received_red_card
)
players_scored_not_red_card = sorted(
    players_scored_last_game - players_received_red_card
)
players_scored_and_red_card = sorted(
    players_scored_last_game & players_received_red_card
)
print(f"All unique players: {all_unique_players}")
```

```
print(
    f"Players who scored but did not receive a red card: "
    f"{players_scored_not_red_card}"
)
print(
    f"Players who scored and received a red card: "
    f"{players_scored_and_red_card}"
)
```

Here are the results:

```
All unique players:
['Cristiano Ronaldo', 'Lionel Messi', 'Luis Suárez']
Players who scored but did not receive a red card:
['Lionel Messi']
Players who scored and received a red card:
['Cristiano Ronaldo']
```

You can do many more things with sets, but generally they help you play around with various data points and their properties. These comparative operations are unique to sets and can't be used for lists, tuples, or dictionaries.

Dictionaries: Key-value organization

Lists, tuples, and sets are great, but sometimes you need to move beyond indexes in order to find specific values. For example, if you want to find everything out about a team (let's call it team A), storing all kinds of data points about overall goals, current standings, and points in its league in one list won't be useful. You'll lose track of which entry is which!

This is where a *dictionary* comes in handy. You can describe each value with a *key*, such as `average_goals_made` or `current_standing` to keep track of various entries. What's more, you can then access those entries just by using the keys. In the next example, you'll create a dictionary for the achievements of team A and then call and print one of the elements of that dictionary. We'll use this naming convention:

- We define `average_goals_made` as the average number of goals a team scores per game.
- The average number its opponents score is `average_goals_conceded`.
- The `average_winning_rate` is the proportion of games the team typically wins.
- The `current_standing` is the team's standing in the league right now.
- The number of points the team has collected is `points`.

Here's how we retrieve a value from a dictionary:

```
team_a_stats = {
    "average_goals_made": 2.5,
    "average_goals_conceded": 1.2,
    "average_winning_rate": 0.75,
    "current_standing": 1,
    "points": 45,
}
print(
    "Team A's average goals made: "
    f"{team_a_stats['average_goals_made']}"
)
```

This prints the following:

```
Team A's average goals made: 2.5
```

The last piece of code takes advantage of Python's `print()` function, which helps you print anything you want onto the screen. The `f` within the parentheses of the `print` statement helps Python understand that you want it to print out the contents of a variable. That variable comes in curly braces, and in this case is `team_a_stats['average_goals_made']`, whose value tells you the number of goals team A made on average, which is stored under the key `"average_goals_made"` in the dictionary `team_a_stats`.

Dictionaries go beyond just storing simple values. You can store any data type in dictionaries, such as lists or tuples, or even other dictionaries:

```
league_stats = {
    "Team A": {
        "average_goals_made": 2.5,
        "average_goals_conceded": 1.2,
        "average_winning_rate": 0.75,
        "current_standing": 1,
        "points": 45,
    },
    "Team B": {
        "average_goals_made": 1.8,
        "average_goals_conceded": 1.5,
        "average_winning_rate": 0.60,
        "current_standing": 2,
        "points": 38,
    },
    "Team C": {
        "average_goals_made": 1.6,
        "average_goals_conceded": 1.7,
        "average_winning_rate": 0.55,
        "current_standing": 3,
        "points": 33,
    },
}
```

```
print(
    f"Team A's average goals made: "
    f"{league_stats['Team A']['average_goals_made']}"
)
for team_name, team_stats in league_stats.items():
    goals_per_match = team_stats["average_goals_made"]
    print(
        f"{team_name}: {goals_per_match:.2f} "
        f"average goals made per match"
    )
```

This prints the following:

```
Team A's average goals made: 2.5
Team A: 2.50 average goals made per match
Team B: 1.80 average goals made per match
Team C: 1.60 average goals made per match
```

If you are typing multiline code into Jupyter yourself, each indented line must keep its leading spaces. Python uses indentation to define blocks, so if copied code picks up extra spaces or loses them, Jupyter may raise an `IndentationError`.

The last bit of the code example is more advanced. It includes not only a `print()` statement, but also a new variable, `goals_per_match`, that we created from the dictionary entry with the key `"average_goals_made"`, and all this runs over all the dictionaries within the main dictionary by means of a `for` loop. If you feel a bit confused by this, don't worry. Revisit this example after you've read "Loops: Repeating Computation Efficiently" on page 33.

Conditional Statements: Writing Decision Rules

Once you're playing around with sets, you'll be getting into the habit of comparing and combining things. To make this easier, `if` statements can be very useful. Also called *conditional statements*, they are an essential concept for decision making in programming.

In most soccer leagues, the standings are determined using a points system. We're likely not telling you news here: a team gets 3 points for a win, 1 point for a draw, and 0 points for a loss.

We can implement these rules programmatically with conditional statements. But first, here's one small but important detail for beginners: the symbol = assigns a value to a variable, while == checks whether two values are equal. For example, `points = 3` stores the value 3 in `points`, whereas `team_goals == opponent_goals` asks whether the two scores are the same. Here's an example:

```
team_goals = 2
opponent_goals = 1
if team_goals > opponent_goals:
```

```
    points = 3
    result = "Win"
elif team_goals == opponent_goals:
    points = 1
    result = "Draw"
else:
    points = 0
    result = "Loss"
print(f"Result: {result}, Points: {points}")
```

This prints the following:

```
Result: Win, Points: 3
```

In this example:

- If `team_goals` is greater than `opponent_goals`, the team wins and earns 3 points.
- If `team_goals` is equal to `opponent_goals`, the match is a draw, and the team earns 1 point.
- If `team_goals` is less than `opponent_goals`, the team loses and earns 0 points.

You can see that conditionals typically come with a comparison of sorts. More generally speaking, the expression is *if [something is true]: [do this], else [do that]*—where the expressions in square brackets should be replaced by something computer-readable.

Loops: Repeating Computation Efficiently

We encountered a `for` loop in "Dictionaries: Key-value organization" on page 30. *Loops* allow you to process data in batches so you don't have to type out many operations that would happen again and again anyway. Thus, by iterating through (large) datasets, you can compute metrics such as goals per game and average possession percentages without having to write repetitive code.

Our first example iterates over a list we call `points_list`, which contains the points earned over three matches. We now are summing the points from `points_list` in order to compute the total points a team accumulated in the past three games:

```
points_list = [3, 1, 0]
total_points = 0
for points in points_list:
    total_points += points
print(f"Total points accumulated: {total_points}")
```

This prints the following:

```
Total points accumulated: 4
```

You can see here a `for` loop that iterates over each element in `points_list`. Each element in that list is temporarily called `points`, just so we can refer to it. There's a total points count, `total_points`, which is initially 0 and to which we add each element from inside `points_list`. The operator `+=` is a common shortcut. Another, longer expression for that same line would be `total_points = total_points + points`.

Our second example uses the function `zip()` to iterate over `team_goals` and `opponent_goals` simultaneously, thus calculating both standing points at the same time. This `zip()` function is extremely useful for unpacking elements in multiple data structures (e.g., lists). If the data structures are not the same length, `zip()` stops as soon as the shortest one runs out of elements.

In this example, three games have been played, and now we want to know how many points the home team has gathered in the league. Here's how that looks in code:

```
team_goals = [2, 1, 3]
opponent_goals = [1, 2, 1]
total_points = 0
for goals_scored, goals_conceded in zip(
    team_goals, opponent_goals
):
    if goals_scored > goals_conceded:
        total_points += 3
    elif goals_scored == goals_conceded:
        total_points += 1
print(f"Total standing points: {total_points}")
```

Here's the output:

```
Total standing points: 6
```

Aside from `for` loops, Python has `while` loops. Their mode of functioning is along the lines of *while [something is true]: [do something]*, and the loop stops when the base condition becomes false. These loops are fairly rare because they carry the danger of going on forever or even crashing your computer; nevertheless, they can occur sometimes.

You can also place loops inside other loops. For example, if you have a list of lists, and you want to do something with each element, you might make a `for` loop to loop through the outer loop and then another `for` loop to find the elements in the sublists. These techniques are somewhat advanced, and we'll explain in detail what's happening when examples of this come up in this book.

List Comprehensions: Performing Compact and Pythonic Iteration

Often in data analysis, you want to build a new list from data you already have. You might transform values, filter them, or compare one sequence with another. *List comprehensions* are a concise way to directly define such a list.

You enclose them in square brackets, just like lists, and the output is indeed a list. Python still loops through the underlying data for you, but the main idea is not *here is a shorter loop.* The main idea is *here is the list I want to create.*

The following example shows the same result written first as a traditional loop and then as a list comprehension:

```
squares = []
for number in range(10):
    squares.append(number ** 2)
print(squares)
squares = [number ** 2 for number in range(10)]
print(squares)
```

The output is as follows:

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

You can also use `zip()` within a list comprehension. We'll reuse our example where three games have been played and we have the goals scored by each team. Now we're trying to find out the number of games won by the home team:

```
home_goals = [2, 1, 3]
away_goals = [1, 2, 1]
home_wins = [
    home > away for home, away in zip(home_goals, away_goals)
]
print(f"Home wins: {home_wins}")
```

This prints the following:

```
Home wins: [True, False, True]
```

Here, `home_wins` returns `True` if home goals exceed away goals (i.e., if the home team won). In this case, this is not a one-liner because we still had to define `home_goals` and `away_goals` beforehand. Nevertheless, the list comprehension gives us a clear way to describe the list we want: one Boolean value for each match, indicating whether it was a home win.

Sometimes, however, loops won't get you where you want to be. What if there is no specific data structure to iterate through, yet you keep accomplishing the same moves nevertheless? In a game, you might perform the same type of pass over and over again, but it's not like you'd be called to perform it whenever the clock strikes an hour.

You'll have to pass whenever you're asked to. This calls for a function—in this case, a pass function.

Functions: Providing Reusable Logic for Analysis

Functions are essential in computer programming for encapsulating reusable pieces of code. For instance, a function can be used to calculate the points accumulated by a soccer team. Here is an example of this in action:

```
def calculate_points(games: list) -> int:
    """Return total points from goal pairs."""
    total_points = 0
    for goals_scored, goals_conceded in games:
        if goals_scored > goals_conceded:
            total_points += 3
        elif goals_scored == goals_conceded:
            total_points += 1
    return total_points
games = [(2, 1), (1, 1), (3, 0), (0, 2)]
total_points = calculate_points(games)
print(f"Total points accumulated: {total_points}")
```

This prints the following:

```
Total points accumulated: 7
```

This function loops through past results, adds 3 points for each win and 1 point for each draw, and returns the total. It takes as an argument `games`, which is a list of tuples specifying the outcome of past games.

The expression in triple quotation marks is a *docstring*. Like a long expression after a # symbol, it helps humans understand what a function or code block does without having to read the code itself. The docstring in this example is intentionally short, but it still tells us what the function expects and what it returns.

What's great about this function (and any function) is that you can feed it any argument you want, and it will automatically re-execute the whole code you wrote on a new argument. You'll be encountering functions again and again in this book because they're that essential. To avoid having trouble with calling functions, make sure that the argument has the data type and structure that is expected for the function to work!

Looking Ahead: From Python Basics to Real Analysis

At this point, the most important part of the setup is in place. You now have a working analytical environment, a sense of how notebooks and repositories fit into your workflow, and a first practical toolkit for working with Python itself. That toolkit includes the building blocks you will rely on throughout the rest of the book: data

types, operations, variables, data structures, conditionals, loops, list comprehensions, and functions.

This is a substantial amount of material, so don't worry if it does not yet feel fully natural. Like technical training on the pitch, these skills become more fluent through repetition, not through a single reading. If something still feels abstract, the best next step is simple: run the examples, modify them, and see what changes. That kind of hands-on experimentation is where confidence begins to grow.

You can also return to this chapter whenever you need to refresh a concept or double-check a piece of syntax. Its purpose is not just to introduce Python once but to serve as a stable reference point as the analysis becomes more complex.

In Chapter 3, we will start putting this foundation to work on real soccer data. That means moving beyond the training ground and into exploratory data analysis: loading data, inspecting it, visualizing it, and beginning to ask meaningful analytical questions. The tools from this chapter will now start becoming part of a full analytical workflow.

Conclusion

Establishing a robust computational environment is the crucial first step in transitioning from conceptual soccer analytics to practical implementation. In this chapter, we have laid the groundwork by setting up Python, configuring isolated virtual environments, and introducing Jupyter Notebook as an interactive workspace. By mastering the core syntax and data structures—such as lists, tuples, sets, and dictionaries—along with control flow mechanisms like loops and conditional statements, you now possess the fundamental tools required to manipulate and analyze data programmatically. These foundational skills are not just abstract exercises; they are the building blocks used to process match events, calculate performance metrics, and structure the datasets that drive modern soccer insights.

With this technical foundation in place, the focus now shifts from learning the language to applying it to real-world scenarios. The programming habits developed here—writing reusable functions, managing dependencies, and organizing code logically—will serve as a reliable anchor as the complexity of our analysis increases. In the next chapter, we will move beyond these basic constructs and dive into exploratory data analysis, where we will load actual soccer datasets, visualize patterns on the pitch, and begin extracting the meaningful signals that inform predictive modeling.

CHAPTER 3

Exploratory Data Analysis in Soccer

In Chapter 2, you built a toolkit of essential Python skills. Now, it's time to put those skills to work on real soccer data. This chapter introduces *exploratory data analysis* (EDA), the initial, open-ended process of inspecting a dataset to understand its structure, find patterns, and identify anomalies. Just as a coach studies match footage before deciding on tactics, EDA is a structured investigation of the data. It's how analysts gain their first understanding of what has been recorded, how often, and where the gaps or oddities lie. Instead of jumping straight to predictive models, we start by asking simple questions to build intuition. This crucial first step prevents us from drawing conclusions on shaky ground and lays the foundation for all subsequent analysis.

As we perform EDA, we'll see how to adapt general-purpose data science tools into soccer-specific analyses. Rather than just counting rows or calculating averages, we'll explore passing networks, shot distance distributions, and team-level tactical patterns. These summaries will appear again in later chapters—for example, when we evaluate classification models in Chapter 5 or engineer features in Chapter 8.

By the end of this chapter, you will have a repeatable workflow for exploring any soccer dataset. You will be able to do the following:

- Load, inspect, and clean raw match data from formats like JavaScript Object Notation (JSON) and comma-separated values (CSV).
- Structure your analysis by data type (numerical, categorical, and ordinal)
- Aggregate event-level data into meaningful team and player summaries
- Create publication-ready visualizations, from standard bar charts to soccer-specific heatmaps and passing networks.
- Translate raw numbers into tactical insights about team and player styles

The goal is to establish a rhythm: explore, clean, visualize, and interpret. Once you are comfortable with this process, you will be ready to move from describing what happened to predicting what might happen next.

Understanding Soccer Data Levels

Before we dive into the data, let's talk about how it's organized. Think of it like watching a game from different perspectives. You could be in a helicopter seeing the whole stadium, on the sideline watching the teams, or right on the pitch following a single player. Soccer data has a similar layered structure, and understanding these levels is crucial because the kinds of questions you can answer depend on the level of data you have.

Match-Level Data

Match-level data is the 30,000-foot view. It gives you the final score, the teams that played, the competition name, and the date. It's the kind of information you'd find on a scoreboard or fixture list. Match-level data can describe a single match or be aggregated across many matches—for example, all games in the group stage of a tournament. Match-level data is perfect for answering high-level questions like these:

- Who won the most games in the group stage?
- Which teams scored the most goals in the tournament?
- How many matches went to extra time?

You can't analyze a team's passing patterns with only match-level data, but you can identify trends across competitions and seasons. It's the foundation for tournament analysis and league standings.

Player-Level Data

Now we're on the sideline. *Player-level data* tracks the performance of individual players across a match or tournament. How many goals did Megan Rapinoe score? How many kilometers did Sam Kerr run? How accurate were Alex Morgan's passes?

Player-level data is often aggregated from finer-grained event data and helps us compare players across roles and teams. It's essential for the following:

- Player recruitment and scouting
- Performance evaluation and ratings
- Fantasy soccer and player awards
- Assessment of individual contributions to team success

Event-Level Data

Event-level data provides the most granular view, putting you right on the pitch with the players. It records every individual *event* in a match—each pass, shot, tackle, dribble, or foul—along with who did it, where it happened on the pitch, and when. Event data does not, however, tell you where all the other players were on the pitch at that moment; it tracks only on-ball actions.

In soccer analytics, the term *event* refers specifically to any discrete on-ball action that changes or continues possession. It doesn't mean *tournament* or *competition event*, as in everyday English.

Event-level data allows analysts to study these elements:

- Tactical patterns and team formations
- Player actions and decision making
- The flow and momentum of matches
- Spatial control and territorial dominance

The beauty of event-level data is that you can aggregate it upward to create player-level or match-level summaries—but not the other way around. Once you have event data, you can answer questions at any level of detail.

Understanding these levels is crucial because each connects to a different class of analytical questions. You can't determine a tournament winner from a single pass, just as you can't analyze a team's passing structure with only match-level data. In this book, we'll work with all three levels, often combining them to build a complete analytical picture.

What Is EDA in Soccer Analytics?

To carry out EDA, we need the right equipment. Just as boots, shin guards, and jerseys prepare a player for the pitch, Python libraries equip an analyst to handle data effectively.

From Chapter 2, you already have the Python basics: variables, loops, and functions. Now we add a few essential tools that you will see throughout the rest of the book:

NumPy
: The training ground where raw numbers become arrays and matrices.

pandas
: Your match sheet and playbook. It organizes messy JSON or CSV files into tidy DataFrames, like putting players in positions on the pitch.

Matplotlib and Seaborn
: Your chalkboard. They let you sketch charts and diagrams that reveal tactics hidden in the numbers.

mplsoccer
: Your tactical board. It lets you visualize data directly on a soccer pitch, bringing analysis to life.



The mplsoccer library is not part of the standard Anaconda distribution. You will likely need to install it yourself. From your command line, you can install it with `pip`:

```
pip install mplsoccer
```

All these libraries are free to use. Once they're installed, you can import them into your notebook with a few lines:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from mplsoccer import Pitch, VerticalPitch
```

These will cover most of the work in this chapter. Later, we will bring in scikit-learn and deep learning libraries as the analysis grows more ambitious.

Like players showing up to a Champions League match, you don't want to take the field under-equipped. NumPy, pandas, and your plotting tools are the core kit that lets you start exploring soccer data with confidence.

Loading StatsBomb Soccer Data

Before we can analyze any match, we need data. For this chapter (and some later ones, such as Chapter 8), we will use the open dataset provided by StatsBomb Open Data. StatsBomb is a well-regarded provider in the soccer analytics community, and its open release is one of the most detailed event datasets available; every pass, shot, dribble, foul, and player position is carefully logged. Alongside these events, the dataset also contains match metadata and team lineups.

In this chapter, we will focus on the 2019 FIFA Women's World Cup (WWC). The data is published under a permissive license, free to download, and widely used in both research and practice. Working with it here prepares you for many of the formats you will encounter in real-world soccer analytics.

Getting the Data

The StatsBomb data lives on GitHub, a popular repository for code and data of all types. To download it, open your command line editor and type the following:

```
mkdir soccer-data && cd soccer-data
git clone https://github.com/statsbomb/open-data.git
```



The StatsBomb open-data repository is quite large (around 15 GB). Before cloning, it is a good idea to create a dedicated directory for it to keep your projects organized.

This command fetches the full history of StatsBomb's open data. Depending on your connection, it can take 10 minutes or more—after all, you are pulling down years of professional match data.

Once the command finishes, you will see a folder called *open-data* with several subfolders. The most important one is *data/*, which contains the actual JSON files. Each match, lineup, and event log is stored as a separate JSON file, organized by competition and season.

For the 2019 WWC, the matches live in the following location:

```
open-data/data/matches/72/30.json
```

Here, `72` is the competition ID for the Women's World Cup, and `30` is the season ID for 2019. The *.json* extension indicates JSON, a format that represents structured data as nested lists and dictionaries. JSON is widely used in sports analytics because it can capture complex, hierarchical information—such as a pass with its start location, outcome, and assisting player—while still being machine-readable.

Loading the Data

With the files on disk, the next step is to load them into a structure that is easy to analyze. In Python, that structure is the pandas DataFrame. A *DataFrame* is essentially a smart table. It combines the familiarity of a spreadsheet with the power of a database query, and it is optimized for fast computation. Once your data is in a DataFrame, you can filter, group, and summarize it with just a few lines of code.

Let's load the 2019 WWC match data into a DataFrame called `matches_df`:

```
import json
import numpy as np
import pandas as pd
from pathlib import Path
BASE = Path("open-data/data")  # Adjust if needed
# Load all WWC 2019 matches
```

```
matches = json.load(open(BASE / "matches/72/30.json", "r"))
matches_df = pd.json_normalize(matches, sep="_")
print("Number of matches:", len(matches_df))
matches_df.head()
```

If you run this code in a Jupyter notebook, the output confirms that 52 matches were loaded. The `pd.json_normalize()` function is particularly useful here; it takes the nested JSON structure and flattens it into a clean DataFrame with columns like `match_id`, `competition_stage_name`, and `season_name`. Note that after normalization, nested fields such as `home_team.home_team_name` become `home_team_home_team_name` (using the `sep="_"` argument).

At this point, you are no longer looking at abstract JSON text. You have real soccer matches, organized in a way that lets you start asking questions.

Taking a First Look at the Data

Before setting up tactics, a coach scans the team sheet to see who is available. Our equivalent is a quick inspection of the DataFrame: what columns do we have, what types of values, and where are the gaps? These checks tell us whether the data is ready for analysis and highlight areas to revisit later.

The pandas library provides two handy methods for this first look:

```
# Basic structure of the matches DataFrame
matches_df.info()
# Quick descriptive stat
matches_df.describe()
```

The `info()` method reports the number of rows, column names, and data types (integer, string, float). This is like glancing at the lineup: who is on the field and in which position. The `describe()` method summarizes numeric columns with counts, averages, and ranges. This is like scanning player stats to see how experienced or balanced the squad is.

From these summaries, we can already answer basic but useful questions: how many matches were played, which tournament stages are included, and which teams participated. It is the analytical equivalent of the kickoff—you have the ball, you know the shape of the field, and you are ready to move it forward.

Later in this chapter, we will go deeper by loading the event data (passes, shots, tackles) and the lineups (players and positions) for each match. For now, we have taken our first touch: the dataset is on the pitch, and we can start to play with it.

Structured EDA by Data Types

Now that we have the ball, it's time to get a feel for it. A structured approach to exploratory analysis means handling each type of data with techniques suited to its nature. In our soccer dataset, we'll encounter three main types of features—numerical, categorical, and ordinal—each requiring different analytical and visualization tools.

Numerical Features: The Quantifiable Game

Numerical features are the measurable quantities in our dataset. They can be *continuous*, like the length of a pass or the distance of a shot, or *discrete*, like the number of goals scored or shots taken. Together, they form the backbone of soccer analytics.

Continuous variables

Continuous numerical variables can take any value within a range, rather than fixed steps. In soccer data, examples include the following:

- Pass length (measured in meters)
- Shot distance from goal
- Player speed and distance covered
- Expected goals (xG) values

Let's explore one simple continuous feature: the distribution of xG values across all shots in the tournament. Since xG is a probability from 0 to 1, it can take any value in that range and is a genuine continuous variable. This code plots how often every xG value occurs:

```
import matplotlib.pyplot as plt
import seaborn as sns
from soccerml_figures.plotting import (
    apply_margins,
    create_figure,
    output_path,
    save_figure,
    use_publication_style,
)
use_publication_style()
# Histogram of xG values across all shots
fig, ax = create_figure()
sns.histplot(
    shots["shot_statsbomb_xg"], bins=30, color="salmon", ax=ax,
)
ax.set_xlabel("Expected Goals (xG)")
ax.set_ylabel("Frequency")
ax.set_title("Distribution of Shot xG Values (WWC 2019)")
```

```
apply_margins(fig)
save_figure(fig, output_path(3, "shot-xg-distribution"))
```

Figure 3-1 shows the result of this code.

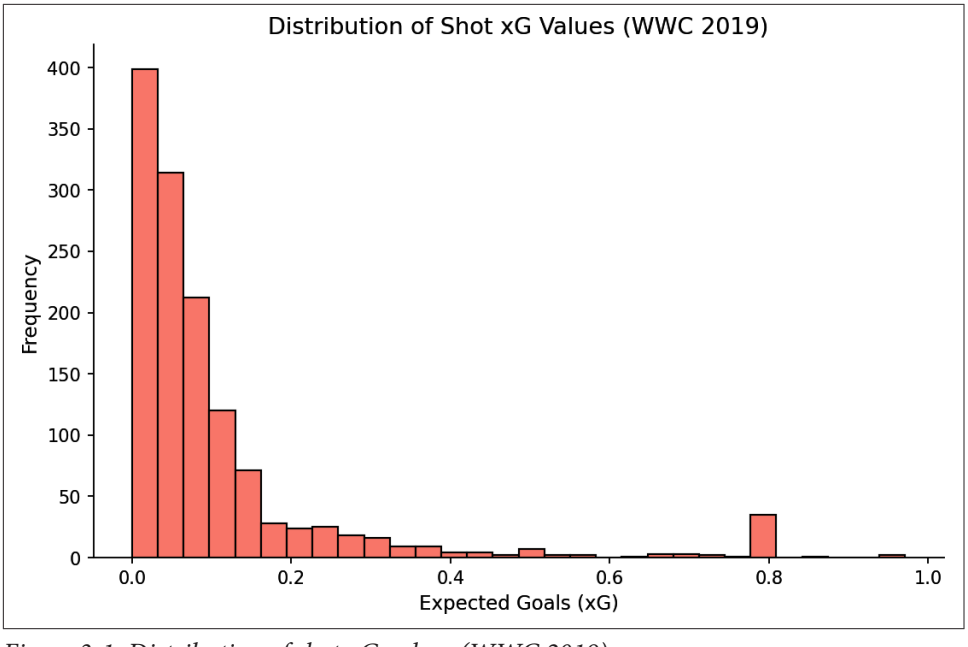


Figure 3-1. Distribution of shot xG values (WWC 2019)

One note on plots: To keep the figures publication-ready, we use a shared plotting helper from the project. It standardizes figure size, dots per inch (DPI), and text sizing so we do not repeat plotting boilerplate code in every example going forward. If you think you're missing a few lines, refer back to the preceding code to help you.

From here onward, we reuse this shared plotting helper. Its implementation details are omitted for brevity.

This histogram reveals that the vast majority of shots have very low xG values; most are long-range efforts or attempts from difficult angles. The small cluster near 0.8 corresponds to penalties, which carry a high probability of being scored. The distribution is heavily right-skewed, confirming that high-quality chances are rare and precious.

What this means for an analyst: Understanding the distribution of xG helps set expectations for what constitutes a high-quality chance. This baseline is essential when evaluating team performance and building predictive models.

Discrete variables

Discrete numerical variables can take only specific integer values—counts of things, rather than continuous measurements. In soccer, these include the following:

- Number of shots taken
- Number of passes completed
- Number of fouls committed
- Number of corners awarded
- Number of goals scored (a count, not a measurement)

Let's look at the distribution of goals scored per team per match, and also at match duration. Note that `match_duration` is not a direct field in the StatsBomb JSON; we compute it from the event data by taking the maximum minute recorded per match:

```
# Distribution of goals scored per team per match
import numpy as np

bins = list(range(0, 10))
home_vals, _ = np.histogram(
    matches_df["home_score"][matches_df["home_score"] <= 8].values,
    bins=bins,
)
away_vals, _ = np.histogram(
    matches_df["away_score"][matches_df["away_score"] <= 8].values,
    bins=bins,
)
fig, ax = create_figure()
ax.step(bins, list(home_vals) + [0], where="post",
        linewidth=2.5, linestyle="-", color="black", label="Home")
ax.step(bins, list(away_vals) + [0], where="post",
        linewidth=2.5, linestyle="--", color="black", label="Away")
ax.set_xlabel("Goals Scored")
ax.set_ylabel("Frequency")
ax.set_title(
    "Distribution of Goals Scored per Match (WWC 2019)"
)
ax.legend()
apply_margins(fig)
save_figure(fig, output_path(3, "goals-per-match-distribution"))
```

Figure 3-2 shows the result of goals scored per team per match.

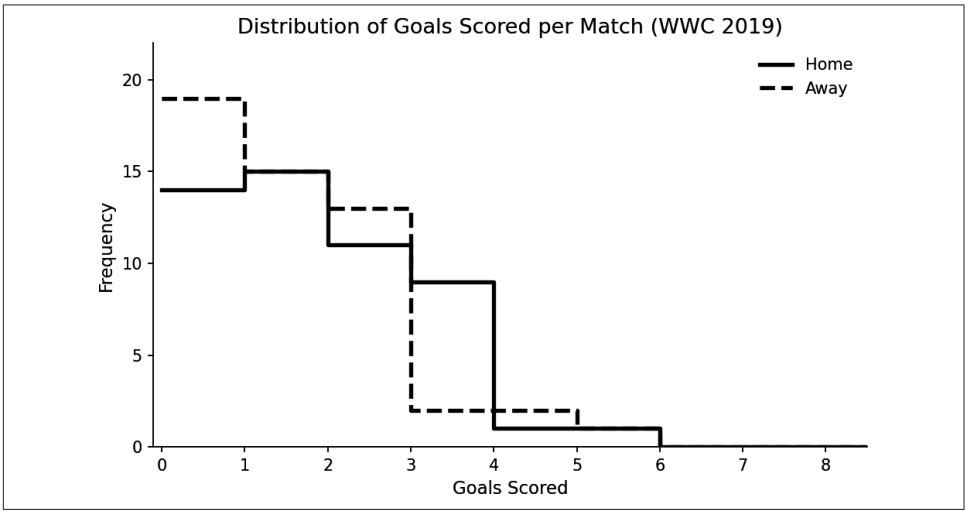


Figure 3-2. Distribution of goals scored per match (WWC 2019)

Figure 3-3 shows the distribution of match lengths.

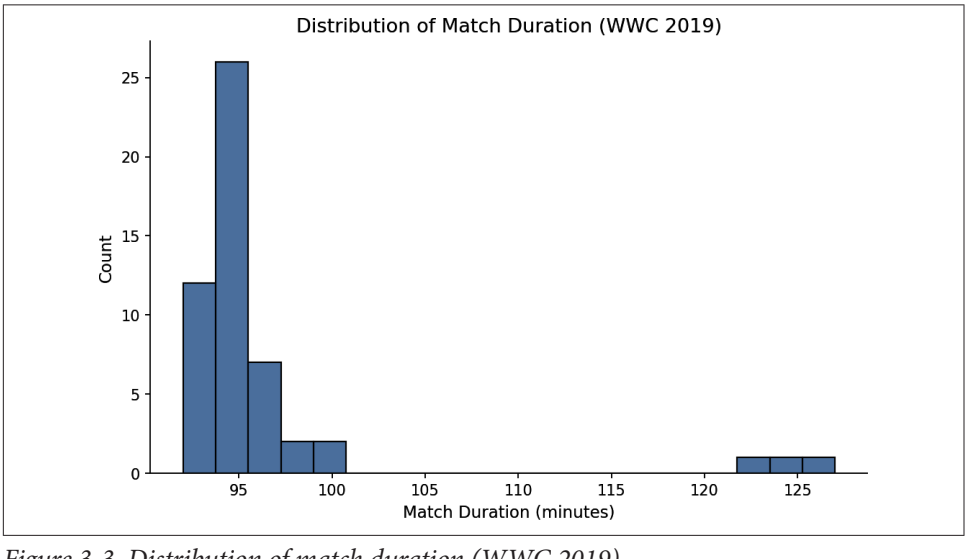


Figure 3-3. Distribution of match duration (WWC 2019)

Most matches see one or two goals per team, confirming soccer's reputation as a low-scoring sport. Match durations cluster around 90–95 minutes, with some extending longer because of stoppage time or extra time in knockout rounds. Visualizations like these give us a quick sense of what is typical—and make anomalies stand out immediately.

Categorical Features: The Qualitative Aspects

Categorical features represent qualitative data without a mathematical ordering. They answer *what* and *who* questions rather than *how much*. In soccer data, here are some examples:

- Team names
- Player names
- Event types (pass, shot, foul)
- Match outcomes (win, draw, loss)

Let's explore team participation by seeing which teams played in the tournament and how often:

```
# Count appearances for each team
team_counts = pd.concat(
    [
        matches_df["home_team_home_team_name"],
        matches_df["away_team_away_team_name"],
    ]
).value_counts()
# Create a horizontal bar plot
fig, ax = create_figure()
sns.barplot(
    y=team_counts.index[:10],
    x=team_counts.values[:10],
    orient="h",
    color="#72B7B2",
    ax=ax,
)
ax.set_xlabel("Number of Matches")
ax.set_ylabel("")
ax.set_title("Top 10 Teams by Number of Matches Played")
apply_margins(fig, left=0.2)
save_figure(fig, output_path(3, "top-teams-by-matches"))
```

Figure 3-4 shows the result.

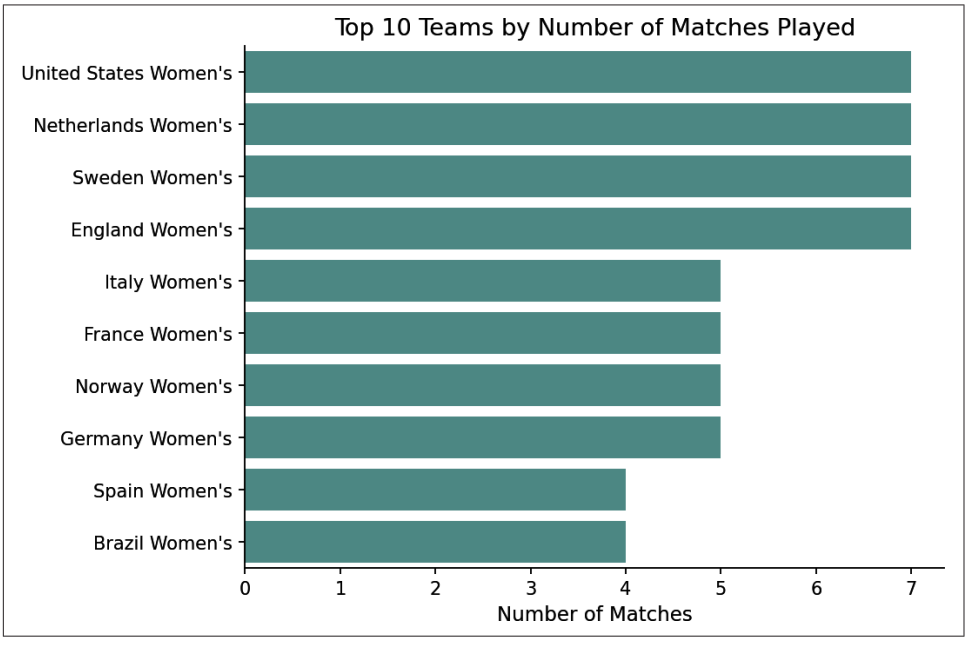


Figure 3-4. Top 10 teams by number of matches played

This type of visualization immediately tells us which teams progressed furthest in the tournament. Teams with more matches reached later stages, while those with fewer matches were eliminated earlier. You can reuse such code and execute it to produce a corresponding image when you want to gain clarity about a tournament you don't know about yet.

What this means for an analyst: This simple count reveals tournament progression without needing to look at detailed match results. It's a quick way to identify the most successful teams and understand the tournament structure.

Ordinal Features: The Ordered Categories

Ordinal features are categorical but have a clear, meaningful order. They're like categorical data with a ranking system built in. Here are some examples in soccer:

- Competition stages (group stage < round of 16 < quarterfinals < semifinals < final)
- Player ratings (poor < average < good < excellent)
- Match importance levels

Let's examine how matches are distributed across tournament stages:

```
# Define the correct order of tournament stages
stage_order = [
    "Group Stage",
    "Round of 16",
    "Quarter-finals",
    "Semi-finals",
    "Third Place Play-off",
    "Final",
]
# Convert to ordered categorical
matches_df["competition_stage_name"] = pd.Categorical(
    matches_df["competition_stage_name"],
    categories=stage_order,
    ordered=True,
)
# Count matches per stage
stage_counts = (
    matches_df["competition_stage_name"]
    .value_counts()
    .sort_index()
)
# Create the visualization
fig, ax = create_figure()
sns.barplot(
    x=stage_counts.values,
    y=stage_counts.index,
    orient="h",
    color="#54A24B",
    ax=ax,
)
ax.set_xlabel("Number of Matches")
ax.set_ylabel("")
ax.set_title("Number of Matches per Tournament Stage")
apply_margins(fig, left=0.2)
save_figure(fig, output_path(3, "matches-by-stage"))
```

Figure 3-5 shows the result.

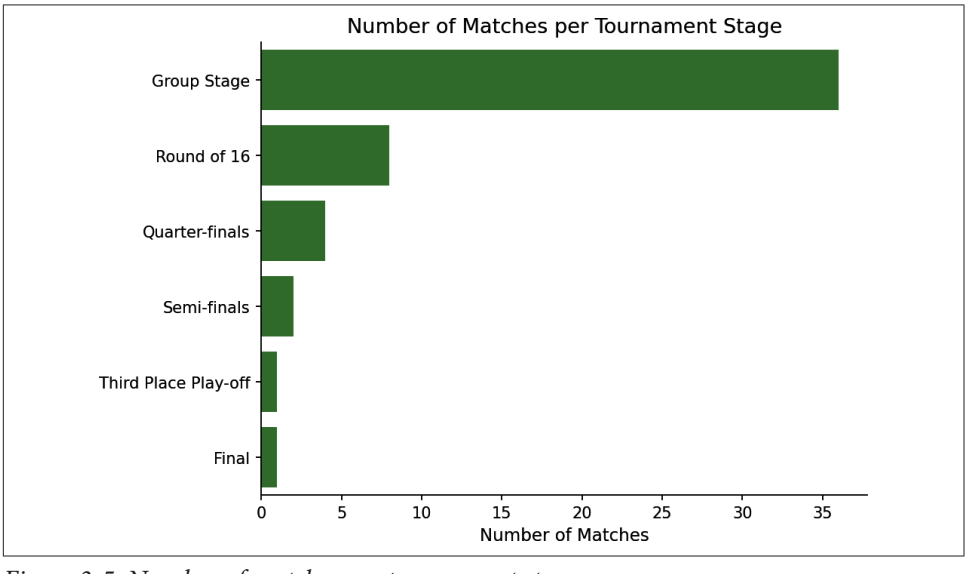


Figure 3-5. Number of matches per tournament stage



The StatsBomb open data uses `competition_stage.name` as the field name in the raw JSON. After calling `pd.json_normalize(matches, sep="_")`, this becomes `competition_stage_name`. If you see a `KeyError`, check the exact column name with `matches_df.columns`.

This ordered visualization follows the natural progression of the tournament. The group stage has the most matches (36), followed by the round of 16 (8 matches), and so on. This respects the tournament's knockout structure and makes the data much easier to interpret.

What this means for an analyst: Respecting the natural order in ordinal data makes visualizations more intuitive and helps viewers understand the underlying structure. This approach also ensures that any statistical analysis properly accounts for the ordering relationship.

Preliminary Exploration: Getting Your Bearings

In exploratory analysis, *controlling the midfield* means doing a few simple tasks consistently well: filtering to the events you care about; grouping by soccer entities such as team, player, or match; and aggregating those results into compact summaries. The pandas library gives us the tools to do all three.

The kinds of questions we will start with are deliberately simple:

- How many events of each type do we have (passes, shots, tackles, …)?
- How does pass completion and accuracy vary by team?
- What does each team's shot profile look like—total shots, goals scored, and cumulative xG?

The xG value estimates the quality of a shot by assigning a probability that it becomes a goal. Cumulative xG adds up these chances across a game or tournament, allowing us to judge chance creation independent of finishing luck. Together, summaries such as pass accuracy and xG act like a compass; they point us toward who controls possession, who creates danger, and where results might flatter or undersell a team.

With that compass in hand, our next task is to put the dataset into a format that lets us ask these questions easily.

Building Tournament DataFrames

StatsBomb's open dataset is organized as one JSON file per match. To work effectively across the entire tournament, collecting those files into two central tables is helpful:

`events_df`
: Every pass, shot, tackle, or foul in the tournament

`lineups_df`
: The players and their teams for each match

Instead of juggling dozens of individual JSON files, we can now filter, group, and summarize from just these two DataFrames. Here is the pattern for extracting events from a single match:

```
BASE = Path("open-data/data")
mid = 69321  # Example match id: 2019 WWC Final
events = json.load(open(BASE / f"events/{mid}.json"))
events_df = pd.json_normalize(events, sep="_")
events_df["match_id"] = mid
```

The same principle applies to lineups. Looping through all match IDs repeats this pattern until you have stacked the full tournament together. Once prepared, `events_df` and `lineups_df` will act as your analytical *midfield* for the rest of this chapter.



If you are running this code in Google Colab, you can use a tool such as Gemini to automate the loop across all match IDs. Alternatively, you can generate the boilerplate code once with any chatbot and then reuse it. The key point is that this preparation is done only once; after that, you can work directly with the consolidated DataFrames.

Counting the Frequency of Event Types

Once the event table is ready, the first step is simply to see the kinds of actions it contains. Each row in `events_df` has a `type_name`, such as *pass*, *shot*, or *duel*. Counting the frequency of these event types gives us an immediate sense of the dataset's shape:

```
events_df["type_name"].value_counts().head(10)
```

The output shows passes dominating, followed by ball receipts, duels, and goalkeeper actions further down the list. This distribution is exactly what you would expect in a typical soccer tournament: passes occur constantly, while shots and goals are much rarer but more decisive events.

This simple frequency count serves as a sanity check. If we saw an unusual pattern—say, more shots than passes—it would signal a data quality issue that needs investigation before proceeding with analysis.

Filtering: Passes by Player or Team

The simplest aggregation is filtering. Instead of looking at all events, we focus on a specific subset. For example, we can see all passes by the United States Women's team:

```
us_passes = events_df.query("type_name == 'Pass'").query(
    'team_name == "United States Women\'s"'
)
print(
    f"US Women made {len(us_passes)} "
    "passes in the tournament"
)
```

This basic filtering is like asking, "Show me only the events I care about." It's the foundation for more complex analysis.

Grouping: Per-Match Pass Summaries

More interesting is to group events and calculate summaries. For example, we can calculate passing statistics for each team in each match:

```
passes = events_df.query("type_name == 'Pass'").copy()
passes["completed"] = passes["pass_outcome_name"].isna()
per_match_pass = (
```

```
    passes.groupby(["match_id", "team_name"])
    .agg(
        attempted=("type_name", "count"),
        completed=("completed", "sum"),
    )
    .reset_index()
)
per_match_pass["pass_accuracy"] = (
    per_match_pass["completed"] / per_match_pass["attempted"]
)
```

This creates a summary table (Table 3-1) where each match is displayed alongside its passing statistics.

Table 3-1. Pass accuracy per match and team

match_id	team_name	attempted	completed	pass_accuracy
22921	France Women's	641	562	0.876755
22921	Korea Republic Women's	328	249	0.759146
22924	Nigeria Women's	332	209	0.629518
22924	Norway Women's	470	363	0.772340
22926	China PR Women's	344	224	0.651163

This table shows that France attempted 641 passes against Korea in match 22921. That number may seem high at first glance, but it is realistic in the context of international tournament play. It also tells a tactical story: France dominated possession in that game, circulating the ball patiently to control tempo. By contrast, a team with far fewer passes may have relied on direct play, quick transitions, or defending deep.

Per-match summaries like these are the bridge between raw event logs and higher-level analysis. They allow us to compare styles across games, track consistency, and ultimately engineer features that can feed into predictive models later in the book.

Aggregating: Shots, Goals, and xG by Team

Passing tells us who controls the ball, but goals decide matches. A natural next step in EDA is to roll up shots and goals to the team level—and, if available, include xG to measure the quality of those chances. This code shows you how:

```
shots = events_df.query("type_name == 'Shot'").copy()
shots["is_goal"] = shots["shot_outcome_name"].eq("Goal")
# If StatsBomb xG is present, include it
xg_col = (
    "shot_statsbomb_xg"
    if "shot_statsbomb_xg" in shots.columns
    else None
)
per_match_shot = shots.groupby(["match_id", "team_name"]).agg(
```

```
        shots=("type_name", "count"),
        goals=("is_goal", "sum"),
        **({"xg": (xg_col, "sum")} if xg_col else {})
    )
```

This produces a table of match-level summaries: how many shots each team took, how many became goals, and how many goals the quality of those shots would predict on average.

Rolling up one level higher, we can build tidy tournament summaries:

```
agg_ops = {"shots": "sum", "goals": "sum"}
if "xg" in per_match_shot.columns:
    agg_ops["xg"] = "sum"
team_summary = per_match_shot.groupby("team_name").agg(agg_ops)
if "xg" in team_summary.columns:
    team_summary["xg_per_shot"] = team_summary[
        "xg"
    ] / team_summary["shots"].clip(lower=1)
```

The resulting Table 3-2 shows total shots, total goals, cumulative xG, and average xG per shot (a measure of shot quality). From the first entries, displayed here, we can see that while China's team had a fair number of shots, it converted very few into goals. On the other hand, Australia and Brazil didn't have many more shots than China but converted many more.

Table 3-2. Shots and goals per team

team_name	shots	goals	xg	xg_per_shot
Argentina Women's	18	2	1.754022	0.097446
Australia Women's	57	9	9.541980	0.167403
Brazil Women's	51	7	6.629535	0.129991
Cameroon Women's	43	3	3.852797	0.089600
Canada Women's	61	4	6.528409	0.107023
Chile Women's	34	1	2.857332	0.084039
China PR Women's	40	1	2.482836	0.062071
England Women's	89	13	12.912852	0.145088
France Women's	91	10	8.466149	0.093035
Germany Women's	77	10	10.020947	0.130142

From here, we can already start telling tactical stories:

- Teams with high shot volume but low xG per shot may be settling for long-range efforts, taking too many pointless shots from long distances.
- Teams that score fewer goals than their cumulative xG might simply be unlucky or lacking clinical finishers.

- Teams outperforming xG often rely on hot streaks or exceptional individual skill, patterns that are difficult to sustain over time.

This contrast between goals and expected goals is one of the most powerful lenses in modern soccer analytics. Goals tell us what happened. xG suggests what *should* have happened given the chances created. Combining the two allows us to separate signal from noise, and it prepares the ground for visualizations such as cumulative xG timelines and scatterplots that we will explore in the next section.

Basic Data Cleaning for Soccer

Real-world data is rarely perfect. It's like a pitch after a rainy match—a bit messy. Before we can do any serious analysis, we need to do some basic cleaning. More advanced cleaning and feature engineering techniques are covered in Chapter 8.

Analysts must prepare their datasets before analysis. This work is not glamorous, but it is vital. Raw event logs contain quirks and inconsistencies that, if left untreated, can lead to inaccurate summaries and even cause later models to fail.

Handling Missing Values

Real soccer data is rarely spotless. Passes may be logged without outcomes, coordinates may be missing, or player IDs may be absent for substitutions. If we do not carefully handle these gaps, later summaries and models can become skewed or unstable.

The following line produces a quick summary of where in our `events_df` dataset data gaps might be hiding:

```
events_df.isnull().sum()
```

This command produces a list of the number of missing values of each table column. Fields such as `pass_outcome_name` or `location` often appear with gaps. (Spoiler alert: StatsBomb data is, in fact, very clean. This will likely not be the case, however, for other data that you might deal with later in your analyst career!)

What should we do with missing or messy values? The answer depends on context:

- *Drop values* when they are rare and nonessential.
- *Impute (fill in)* when the missing value has a natural default.
- *Interpret carefully* when the missing value itself carries information.

For example, in StatsBomb data, a pass is marked as *completed* by leaving `pass_outcome_name` blank. In this case, the missing value is actually meaningful:

```
# Treat NaN in pass_outcome_name as "completed"
passes = events_df.query("type_name == 'Pass'").copy()
passes["completed"] = passes["pass_outcome_name"].isna()
```

Here, the not a number (NaN) is not a problem but the solution; it signals success that the pass was completed. In other contexts, you might replace missing numbers with the column mean or drop rows entirely if too many values are absent.



Handling missing data has no one-size-fits-all rule. In soccer data, a blank cell might mean a logging error, or it might encode a deliberate design choice (such as StatsBomb's convention for completed passes). Always check what the data provider intended before deciding how to clean.

The key lesson is that data cleaning is not about making the data look neat. Rather, it's about making sure your analysis reflects what happened on the pitch. A careless assumption here can cascade through the rest of the workflow, undermining both exploratory insights and predictive models later.

Standardizing Names

Another source of potential trouble is inconsistent naming. A team might appear as *USA, United States,* or *United States Women's.* Players may have full names in one match and nicknames in another. Without standardization, `groupby()` operations can double-count, and leaderboards can become misleading.

A simple way to handle this is with a mapping dictionary:

```
# Example: standardize team names
name_map = {
    "USA": "United States Women's",
    "United States": "United States Women's",
    "Brasil": "Brazil Women's",
}
matches_df["home_team_home_team_name"] = matches_df[
    "home_team_home_team_name"
].replace(name_map)
matches_df["away_team_away_team_name"] = matches_df[
    "away_team_away_team_name"
].replace(name_map)
```

For players, the same principle applies. You might decide that *Marta Vieira da Silva* should always be shortened to *Marta,* or you may prefer to keep the full name consistently. Either choice is fine, as long as it is applied uniformly.

As noted previously, StatsBomb's open data is usually very clean, so you may not need to do much of this. Still, it is good practice to be aware of the issue and to keep

a record of any mappings you apply. Consistency matters more than the particular convention you choose.

Visualizing Soccer Data

Numbers in a table can tell us a lot, but patterns often emerge much more clearly when we draw them. Visualization is a central step in exploratory data analysis because it connects raw numbers back to the flow of the game. Coaches sketch tactics on a whiteboard; analysts do the same with charts and figures.

In this section, we will build a small visualization toolkit, moving from basic bar charts to more advanced soccer-specific graphics such as cumulative timelines and heatmaps. Each plot type has strengths and weaknesses. The goal is not to master every option but to learn how to pick the right one for the question at hand.

A Quick Note on Matplotlib and Seaborn

All the figures in this chapter use Matplotlib and Seaborn. *Matplotlib* is the core Python library for plotting; it provides the low-level building blocks for charts. *Seaborn* sits on top of Matplotlib, offering higher-level functions and prettier defaults. Most analysts use them together: Seaborn for quick plots, Matplotlib for fine-tuning.

To use them, just make sure to place the following lines in the beginning of your notebook:

```
import matplotlib.pyplot as plt
import seaborn as sns
from soccerml_figures.plotting import (
    apply_margins,
    create_figure,
    output_path,
    save_figure,
)
```

Then, Seaborn can draw a histogram in one line:

```
sns.histplot(data=shots, x="shot_statsbomb_xg")
```

Under the hood, this is just a convenient wrapper around Matplotlib. If you need to adjust details such as colors, labels, or axes, you can always fall back on Matplotlib. Then you'd add below that one line something like this:

```
plt.xlabel("Shot xG")
plt.ylabel("Count")
plt.title("Distribution of shot quality (xG)")
```

Together, these two libraries give you both speed and flexibility: Seaborn for the quick sketch, Matplotlib for the finishing touches.

Bar Charts and Rankings

Bar charts are the simplest way to compare teams or players. They are especially useful for rankings: top scorers, most passes, or best passing accuracy. Seaborn and pandas make this straightforward.

We already saw a table of pass accuracy by team. Turning it into a bar chart makes the comparison much easier to grasp. Let's create a visualization of the top teams by pass accuracy:

```
# Calculate pass accuracy by team
team_pass_accuracy = (
    per_match_pass.groupby("team_name")["pass_accuracy"]
    .mean()
    .sort_values(ascending=False)
    .head(8)
)
fig, ax = create_figure()
sns.barplot(
    x=team_pass_accuracy.values,
    y=team_pass_accuracy.index,
    orient="h",
    color="#4C78A8",
    ax=ax,
)
ax.set_xlabel("Pass Accuracy")
ax.set_ylabel("")
ax.set_title("Top 8 Teams by Pass Accuracy (WWC 2019)")
apply_margins(fig, left=0.2)
save_figure(fig, output_path(3, "pass-accuracy"))
```

Figure 3-6 shows the output.

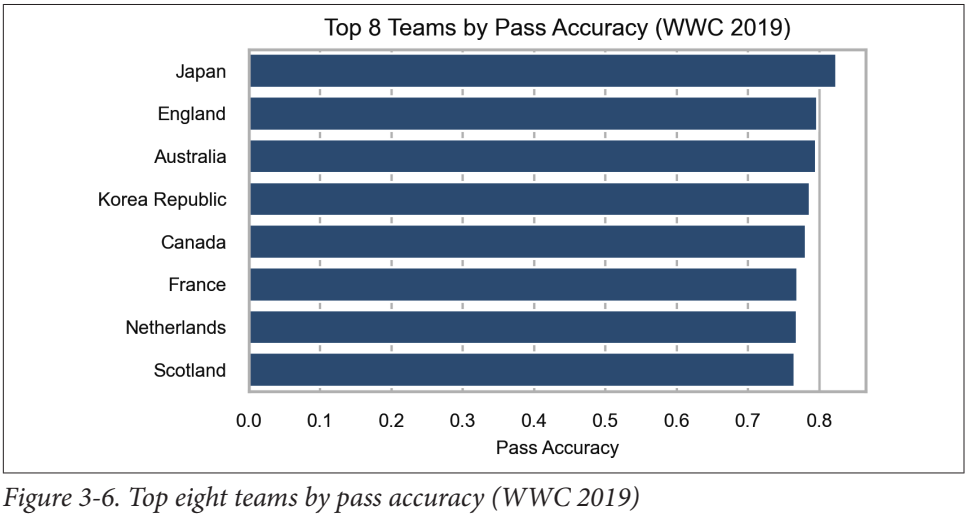


Figure 3-6. Top eight teams by pass accuracy (WWC 2019)

Bar charts like this let us compare teams at a glance. Japan led the women's tournament in pass accuracy, edging out England and Australia. High accuracy suggests a possession-oriented style, while lower values often reflect teams that take more risks with long balls or fast transitions.

What this means for an analyst: Pass accuracy is a fundamental indicator of a team's playing style. Teams with consistently high pass accuracy typically favor possession-based approaches, building attacks patiently through short, safe passes. Lower accuracy might indicate more direct play, where teams are willing to attempt riskier passes to create quick scoring opportunities.

Histograms and Distributions

Bar charts compare teams or players, but sometimes we want to see the overall distribution of a statistic: are most values clustered in one range, or are they spread out? Histograms answer this by showing how often different values occur.

Take pass length as an example. Some teams build patiently with short passes; others rely on longer balls to move the game forward quickly. We can plot a histogram of pass lengths over the whole tournament, aggregated over all teams. This works as follows, using the table `passes` we created in "Grouping: Per-Match Pass Summaries" on page 54:

```
fig, ax = create_figure()
sns.histplot(
    data=passes,
    x="pass_length",
    bins=30,
    color="skyblue",
    ax=ax,
)
ax.set_xlabel("Pass length (meters)")
ax.set_ylabel("Count")
ax.set_title("Distribution of pass lengths")
apply_margins(fig)
save_figure(fig, output_path(3, "pass-length-histogram"))
```

Figure 3-7 shows the result.

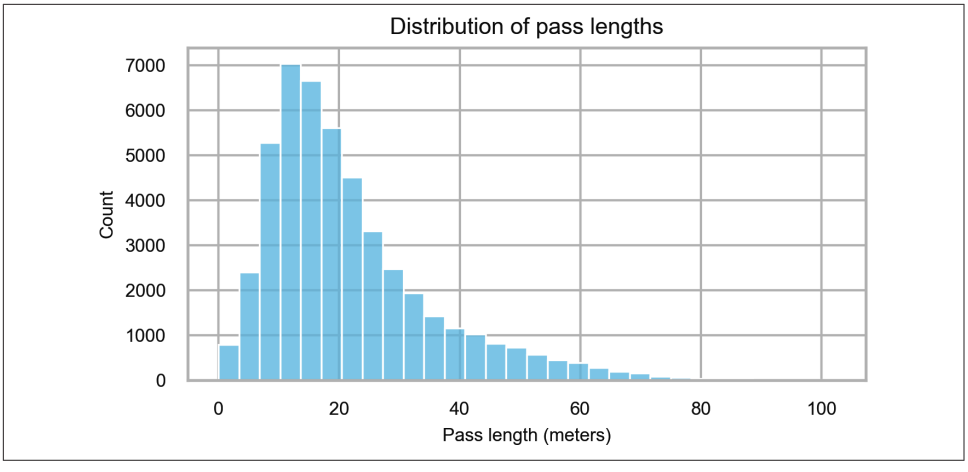


Figure 3-7. Distribution of pass lengths (WWC 2019)

The sharp peak under 20 meters reflects short, safe passes typical of possession play. The high bars indicate that, in the 2019 WWC, many short and safe passes were played. The long tail stretching toward 40 meters and beyond suggests teams mixing in more direct balls—but this is a rarer occurrence.

What this means for an analyst: This distribution reveals the tactical preferences across the tournament. The dominance of short passes (under 20 meters) indicates that most teams favored patient buildup play over direct, long-ball tactics. The long tail shows that while long passes are less common, they're still an important part of the game, often used to switch play or launch counterattacks.

We can apply the same idea to shot quality. Instead of raw distance, StatsBomb data provides xG, a probability that each shot will result in a goal. Plotting these values reveals how often teams create high- versus low-quality chances:

```
fig, ax = create_figure()
sns.histplot(
    data=shots,
    x="shot_statsbomb_xg",
    bins=25,
    color="salmon",
    ax=ax,
)
ax.set_xlabel("Shot xG")
ax.set_ylabel("Count")
ax.set_title("Distribution of shot quality (xG)")
apply_margins(fig)
save_figure(fig, output_path(3, "shot-quality"))
```

Figure 3-8 shows the result.

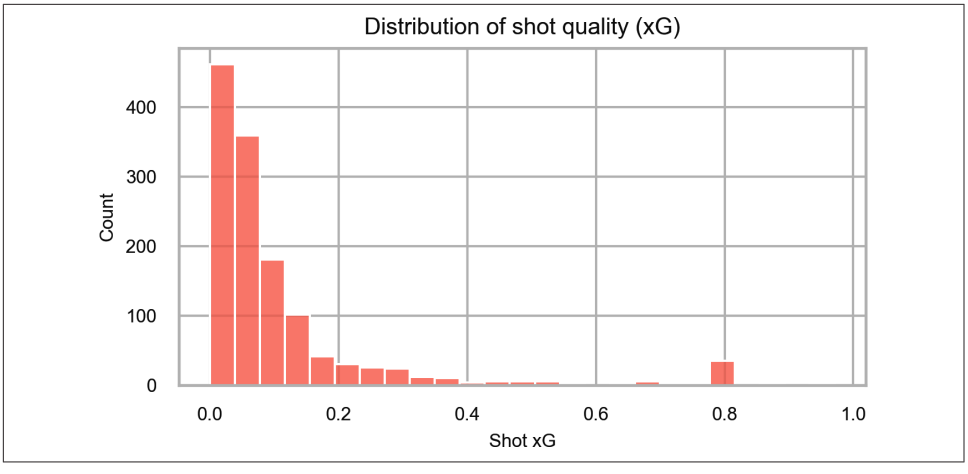


Figure 3-8. Distribution of shot quality (xG, WWC 2019)

This histogram skews heavily toward low xG values, with most shots carrying less than a 10% chance of scoring. That is normal: long-range efforts and tight angles are more common than clean looks inside the box. The rare right-tail values represent high-quality chances—often the difference-makers in tight matches. The small peak around an xG of 0.8 corresponds to penalties, which have a high probability of being scored.

What this means for an analyst: This distribution confirms that high-quality scoring chances are rare and precious in soccer. The vast majority of shots have low xG values, emphasizing the importance of shot selection and chance creation. Teams that can consistently create shots with higher xG values will typically be more successful than those taking many low-quality attempts.

Boxplots and Variability

Totals and averages are useful, but they can hide important variation. Some teams are consistent from match to match, while others swing between brilliant and poor performances. Boxplots are a good way to visualize this variability.

As an example, let's look at passing accuracy across matches. Instead of showing a single average per team, we can plot the distribution of pass accuracy values across all matches:

```
team_order = (
    per_match_pass.groupby("team_name")["pass_accuracy"]
    .mean()
    .sort_values()
    .index
)
fig, ax = create_figure()
```

```
sns.boxplot(
    data=per_match_pass.reset_index(),
    x="pass_accuracy",
    y="team_name",
    orient="h",
    order=team_order,
    color="#72B7B2",
    ax=ax,
)
ax.set_xlabel("Pass accuracy")
ax.set_ylabel("Team")
ax.set_title("Per-match pass accuracy distribution")
apply_margins(fig, left=0.2)
save_figure(fig, output_path(3, "pass-boxplot"))
```

Figure 3-9 shows the result.

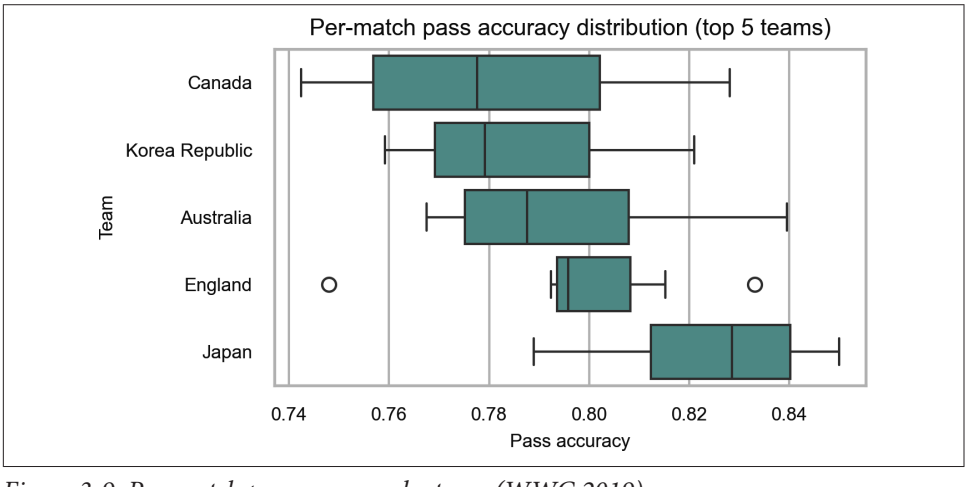


Figure 3-9. Per-match pass accuracy by team (WWC 2019)

This boxplot reveals several patterns:

- Teams with narrow boxes (small interquartile range) were very consistent in their passing accuracy every game (e.g., England, Australia, and Spain).
- Teams with wide boxes (high variability) alternated between strong passing performances and weaker ones (e.g., France and US).
- Outliers highlight exceptional matches—either unusually crisp passing or games where possession broke down.

What this means for an analyst: Boxplots like these help us identify not just how good a team is on average, but its stability across a tournament. In knockout competitions, consistency is often just as valuable as peak performance. Teams with narrow

boxes are more predictable and reliable, while those with wide boxes might be more dangerous but also more vulnerable to off-days.

Time Series and Cumulative Plots

Totals tell us how much happened, but not when. A within-match timeline shows momentum: who created chances early, who dominated later, and whether goals reflected the balance of play. A simple and informative visualization is the cumulative shots timeline.

The following code builds such a timeline for a single match (replace `match_id` with any game you want to explore):

```
mid = 22921  # Example match id
# Select shots from this match and compute exact minute
shots_m = events_df.query(
    "type_name == 'Shot' and match_id == @mid"
).copy()
shots_m["minute_exact"] = (
    shots_m["minute"].fillna(0)
    + shots_m["second"].fillna(0) / 60.0
)
# Sort and create cumulative shot count
shots_m = shots_m.sort_values("minute_exact")
shots_m["cum_shots"] = (
    shots_m.groupby("team_name").cumcount() + 1
)
# Plot timeline
fig, ax = create_figure()
for team in shots_m["team_name"].unique():
    team_shots = shots_m[shots_m["team_name"] == team]
    ax.step(
        team_shots["minute_exact"],
        team_shots["cum_shots"],
        where="post",
        label=team,
        linewidth=2,
    )
ax.set_xlabel("Match minute")
ax.set_ylabel("Cumulative shots")
ax.set_title("Cumulative shots timeline")
ax.legend()
ax.grid(True, alpha=0.3)
apply_margins(fig)
save_figure(fig, output_path(3, "cumulative-shots-timeline"))
```

Figure 3-10 shows the output.

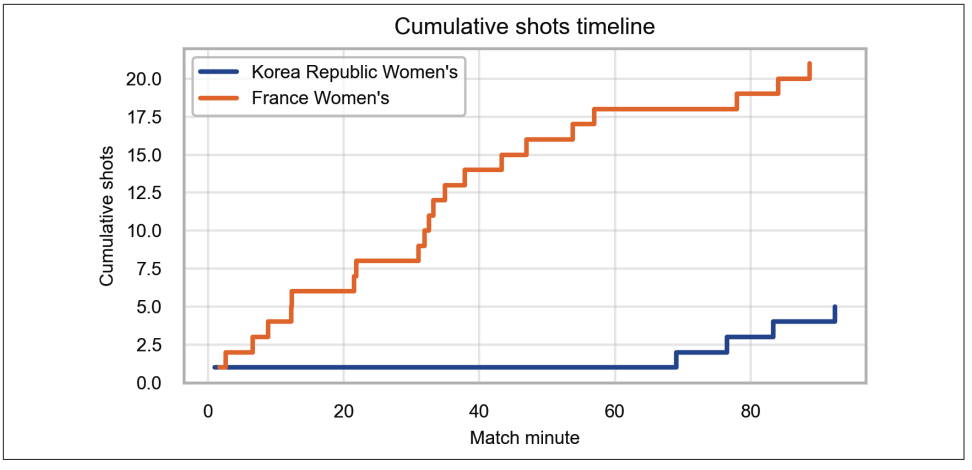


Figure 3-10. Cumulative shots timeline for a single match

This timeline shows how attacking pressure built up over 90 minutes. Here's how we'd interpret the timeline:

- France's line rises steadily but never steeply, reflecting gradual buildup and patient chance creation.
- Korea Republic's line stays flatter, showing fewer attempts overall.
- The absence of sharp bursts indicates that France rarely unleashed waves of shots, instead probing with consistent but moderate pressure.

What this means for an analyst: Timelines like this highlight when momentum was with each side, not just the final shot count. They give coaches and analysts a sense of whether a team's style was sustained throughout the match or concentrated in short bursts. Sharp increases indicate periods of intense pressure, while flat sections suggest defensive phases or loss of attacking rhythm.

Scatterplots: Shot Volume Versus Goals

Sometimes we want to see the relationship between two variables. Scatterplots are perfect for this. Let's examine the relationship between shot volume and goals scored:

```
# Calculate per-match averages for teams
team_averages = (
    per_match_shot.groupby("team_name")
    .agg(
        shots_per_match=("shots", "mean"),
        goals_per_match=("goals", "mean"),
    )
    .reset_index()
)
```

```
trend_x = team_averages["shots_per_match"]
trend_y = team_averages["goals_per_match"]
slope, intercept = np.polyfit(trend_x, trend_y, 1)
x_line = np.linspace(trend_x.min(), trend_x.max(), 100)
fig, ax = create_figure()
sns.scatterplot(
    data=team_averages,
    x="shots_per_match",
    y="goals_per_match",
    s=90,
    ax=ax,
)
ax.plot(
    x_line,
    slope * x_line + intercept,
    color="blue",
    linestyle="--",
    alpha=0.8,
)
ax.set_xlabel("Shots per Match")
ax.set_ylabel("Goals per Match")
ax.set_title("Teams: Shot Volume vs. Goals (WWC 2019)")
apply_margins(fig)
save_figure(fig, output_path(3, "shots-vs-goals-team"))
```

Figure 3-11 shows the output.

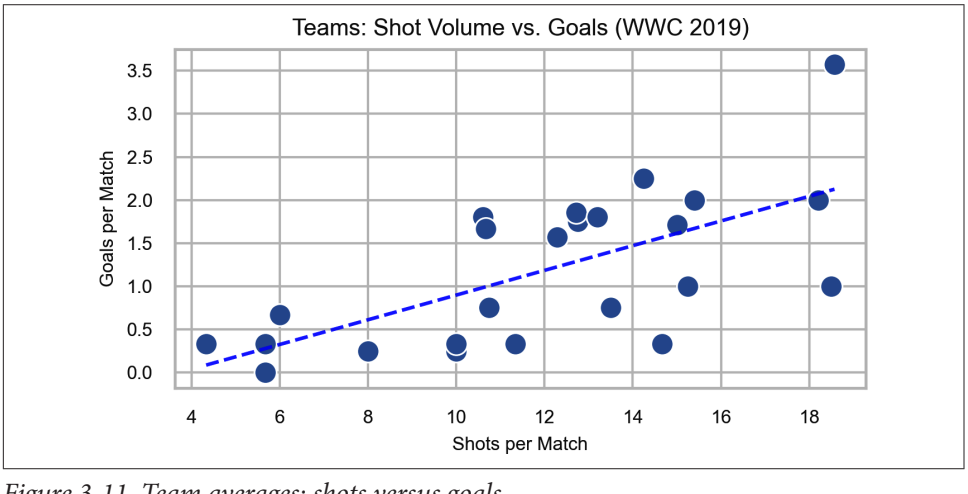


Figure 3-11. Team averages: shots versus goals

The scatter shows a clear positive relationship: teams that generate more shots tend to score more goals. This is not surprising, but it is a useful reminder that volume matters. The dashed line is a trend line (a simple linear regression), showing the average relationship. At the same time, quality also plays a role. Some teams converted fewer goals from many attempts, while others relied on fewer, higher-quality chances. This balance between quantity and quality is one of the central questions in modern soccer analytics.

What this means for an analyst: The positive correlation confirms that shot volume is important, but the scatter around the trend line shows that conversion efficiency varies significantly among teams. Teams above the line are more clinical finishers, while those below might need to improve their shot selection or finishing quality. The trend line helps identify over- and under-performers relative to their shot volume.

Soccer-Specific Visualizations with mplsoccer

While standard plots are useful, nothing beats seeing the data on a proper soccer pitch. The mplsoccer library is a fantastic tool that lets us do just that. It helps us translate coordinates from our data into tactical pictures that are immediately understandable to coaches, players, and fans.

Think of mplsoccer as your tactical whiteboard. Just as coaches draw formations and player movements on a board, we can plot actual match data directly onto a pitch representation. This bridges the gap between abstract numbers and the spatial reality of soccer.

Pass Location Heatmaps

We now know how far (and how well) teams shoot. But where on the pitch do they build play? A heatmap of pass start locations shows whether a team prefers wide overloads, central progression, or deep recycling near its own box.

StatsBomb events store coordinates either as a nested list column (`location`) or as flattened columns (for example, after `pd.json_normalize`). The following extracts pass start coordinates into a reusable `passes_xy` table:

```
def build_passes_xy(events_df):
    p = events_df.query("type_name == 'Pass'").copy()
    # Handle both nested and flattened coordinate schemas
    if "location" in p.columns:
        p[["x", "y"]] = p["location"].apply(pd.Series)
    else:
        # Common flattened variants from json_normalize
        for a, b in (
            ("location_0", "location_1"),
            ("location.x", "location.y"),
            ("location_x", "location_y"),
```

```
        ):
            if a in p.columns and b in p.columns:
                p["x"] = p[a]
                p["y"] = p[b]
                break
    # Light cleanup: drop missing coords and throw-ins
    p = p.dropna(subset=["x", "y"])
    if "pass_type_name" in p.columns:
        p = p[p["pass_type_name"] != "Throw-in"]
    return p
passes_xy = build_passes_xy(events_df)
```

With `passes_xy` ready, we can plot a simple hexbin heatmap for one team. Here, we look at Japan's team:

```
team = "Japan Women's"
team_passes = passes_xy.query("team_name == @team").dropna(
    subset=["x", "y"]
)
fig, ax = create_figure()
hexmap = ax.hexbin(
    team_passes["x"],
    team_passes["y"],
    gridsize=25,
    mincnt=1,
    cmap="Blues",
)
ax.set_xlim(0, 120)
ax.set_ylim(0, 80)
ax.invert_yaxis()  # Match StatsBomb axes
ax.set_aspect("equal", adjustable="box")
ax.set_xlabel("Pitch X")
ax.set_ylabel("Pitch Y")
ax.set_title(f"Pass Start Locations — {team} (WWC 2019)")
colorbar = fig.colorbar(hexmap, ax=ax, pad=0.02)
colorbar.set_label("Pass count")
apply_margins(fig, right=0.9)
save_figure(fig, output_path(3, "pass-heatmap"))
```

Figure 3-12 shows the output.

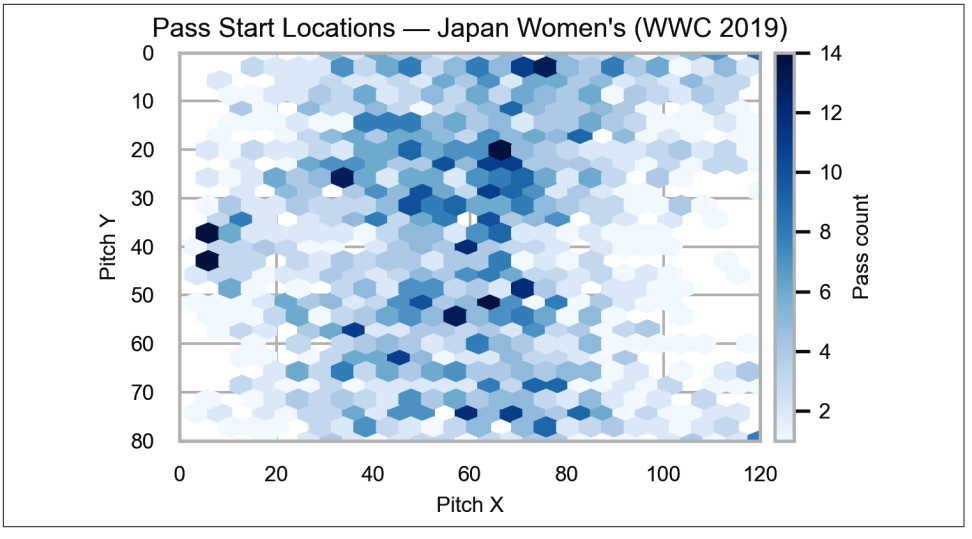


Figure 3-12. Pass start location heatmap

The horizontal axis (0–120) represents the full length of the pitch in meters, while the vertical axis (0–80) represents its width. Each hexagon counts the number of pass start locations within that area. Darker cells indicate zones with more passing activity (closer to 14 shots in this match, as per the legend to the right of the figure).

In this example, Japan Women's spread its buildup evenly across the width of the field, with a noticeable concentration in the central third. The two dark clusters near the team's own penalty area correspond to short passes beginning from the goalkeeper and center-backs. Together, these patterns reveal a possession-oriented style built on midfield control and short passes from the back rather than direct long balls.

What this means for an analyst: Heatmaps reveal a team's spatial preferences and tactical approach. Japan's distribution shows it's comfortable building play from deep positions and using the full width of the pitch. The concentration in central areas suggests that this team prefers to control the game through the midfield rather than relying heavily on wing play. This visualization confirms Japan's reputation for patient, possession-based soccer.

Advanced Pitch Visualizations: Passing Networks

One of the most insightful soccer-specific visualizations is the passing network. This shows not just where passes start, but also the connections between players. It reveals the tactical structure and key relationships within a team.

Setting up such a passing network requires more complex code. The reason is that we need to calculate average positions and pass frequencies between players before attempting a visualization. The original source code for this example can be found in the book's GitHub repository. Figure 3-13 shows the result.

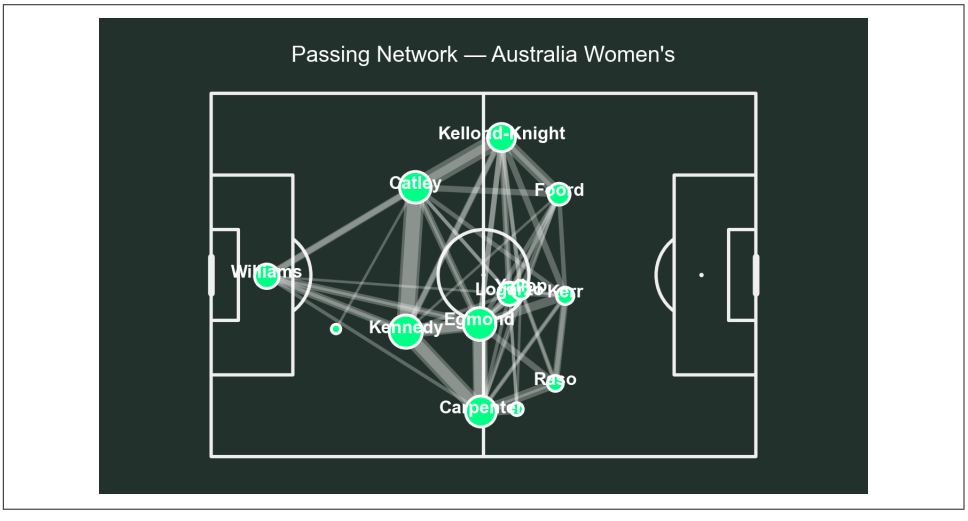


Figure 3-13. The passing network by Australia Women's in match 68357

This passing network depicts Australia Women's in a specific match. Each player is positioned at their average location on the pitch, and lines between players represent the frequency and volume of passes between them. Thicker lines indicate more frequent passing combinations.

What this means for an analyst: Passing networks reveal the tactical structure of a team. Central players with many connections (like Ellie Carpenter in this example) are the key distributors. The positioning shows the formation and players' movements during the match. Isolated players or weak connections might indicate tactical issues or players who struggled to get involved in the game.

Basic Pitch Visualizations

Let's now create the basic pitch visualizations we introduced earlier, but with more context:

```
from mplsoccer import Pitch, VerticalPitch
# Create a horizontal pitch (typical view)
pitch = Pitch(
    pitch_color="grass", line_color="white", stripe=True,
)
fig, ax = create_figure()
pitch.draw(ax=ax)
```

```
# Sample shot data from the attacking third
import numpy as np
rng = np.random.default_rng(42)
x = rng.uniform(80, 120, 15)  # Shots from attacking third
y = rng.uniform(10, 70, 15)  # Across the width
goals = rng.choice(
    [True, False], 15, p=[0.3, 0.7],
)  # 30% conversion rate
# Plot goals and misses with different colors
pitch.scatter(
    x[goals],
    y[goals],
    marker="o",
    color="#FFD700",
    edgecolors="black",
    linewidths=0.8,
    s=100,
    ax=ax,
    label="Goal",
    zorder=5,
)
pitch.scatter(
    x[~goals],
    y[~goals],
    marker="s",
    color="#1a1a2e",
    edgecolors="black",
    linewidths=0.8,
    s=100,
    ax=ax,
    label="Miss",
    zorder=5,
)
ax.legend()
ax.set_title("Shot Map: Goals vs Misses")
apply_margins(fig, left=0.02, right=0.99, top=0.88, bottom=0.04)
save_figure(fig, output_path(3, "shot-map"))
```

Figure 3-14 shows the result: a shot map generated with random data. Each circle marks the location of a shot: circles are goals, and squares are misses. The pitch runs from left to right, so all shots here were taken toward the righthand goal. In this fictional example, the team recorded 14 shots, converting 4 of them into goals—a realistic ratio for a competitive match.

This type of visualization makes it easy to spot shooting tendencies at a glance. Clusters of circles close to the goal indicate efficient finishing from central areas, while squares farther out reveal lower-probability attempts from a distance.

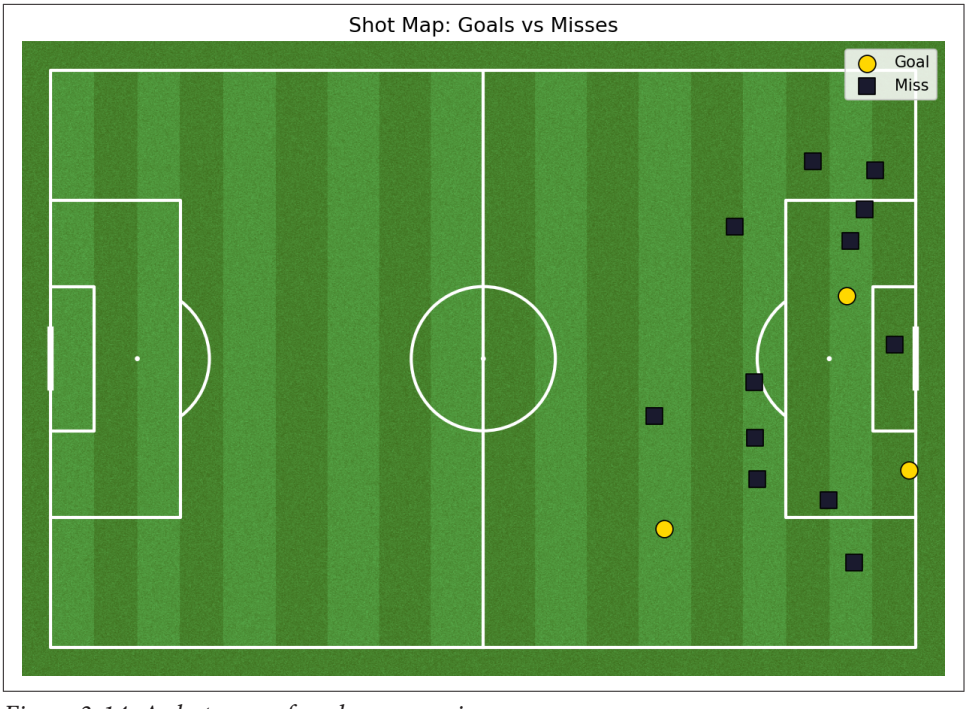


Figure 3-14. A shot map of goals versus misses

What this means for an analyst: A shot map instantly reveals a team's shooting habits. Are players taking a lot of low-probability shots from far out? Or are they creating high-quality chances close to goal? This visualization can be used to evaluate offensive efficiency and tactical discipline. Teams that consistently create shots from central areas close to the goal are typically more effective than those taking speculative long shots from the outer areas of the soccer pitch.

Advanced Heatmap Visualizations

Beyond basic shot maps, we can create more sophisticated heatmaps that show player movement patterns or event density across the pitch. Here is how we create a more advanced heatmap with mplsoccer:

```
pitch = Pitch(
    pitch_color="#22312b", line_color="#efefef", linewidth=2,
)
fig, ax = create_figure()
fig.patch.set_facecolor("#22312b")
ax.set_facecolor("#22312b")
pitch.draw(ax=ax)
# Example heatmap of player positions or events
# Real analysis would use event coordinates
```

```
rng = np.random.default_rng(42)
x = rng.normal(60, 20, 1000)  # Centered around midfield
y = rng.normal(40, 15, 1000)  # Centered vertically
x = np.clip(x, 0, 120)  # Keep within pitch bounds
y = np.clip(y, 0, 80)
# Create the heatmap
pitch.kdeplot(
    x, y, ax=ax, fill=True, levels=10, alpha=0.7, cmap="Blues",
)
ax.set_title("Player Movement Heatmap", color="white")
apply_margins(fig, left=0.02, right=0.99, top=0.88, bottom=0.04)
save_figure(fig, output_path(3, "player-heatmap"))
```

In this example, we're not plotting a real match, but generating random coordinates to show how the visualization works. Figure 3-15 shows the result.

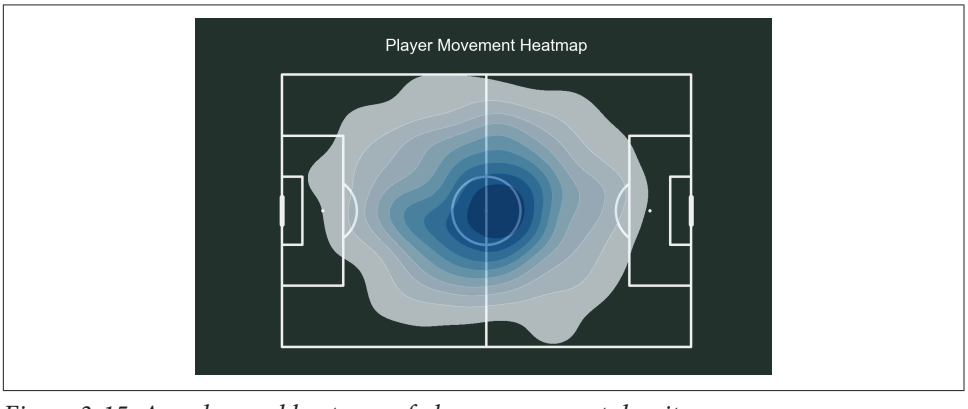


Figure 3-15. An advanced heatmap of player movement density

The horizontal axis (0–120) represents the pitch length, and the vertical axis (0–80) its width. Darker zones mark areas with higher simulated player activity, while lighter zones indicate sparse movement. In this scenario, most of the action occurs near the outer edges of the pitch, suggesting wide positioning, while the small circles of density around the midfield represent central players rotating through buildup play.

What this means for an analyst: Advanced heatmaps reveal spatial patterns that aren't immediately obvious from raw coordinate data. They can show where a player spends most of their time, where a team creates the most chances, or where defensive actions are concentrated. The smooth gradients help identify hotspots and cold zones, making it easier to understand tactical patterns and player roles.

Case Study: Japan Women's at the 2019 World Cup

We have built up a toolkit of summaries and visualizations, but so far we have looked at each in isolation. To see how these pieces come together, let's apply them to a concrete example.

Our guiding question is, how did the Japan Women's team play at the 2019 FIFA Women's World Cup?

Japan provides an ideal case study. Its players are known for a possession-oriented style, with patient buildup and precise passing, yet they faced opponents with very different approaches. By combining the tools we just developed—tables, charts, timelines, and heatmaps—we can sketch a full analytical profile of their tournament.

In this section, we will move from tournament-wide numbers to match-level flow, spatial patterns, and individual player contributions. The goal is not only to learn about Japan but to practice the workflow of exploratory analysis: cleaning data, building tidy summaries, visualizing patterns, and interpreting them in soccer terms.

Team-Level Overview

We begin at the tournament level. How did Japan's overall profile of shooting and finishing compare to other top teams?

From our earlier summaries, we already have shots, goals, and xG rolled up to the team level. Let's extract Japan and three other strong teams (the US, England, and the Netherlands) for context:

```
# Tournament-level summary of key stats
team_compare = team_summary[["shots", "goals", "xg"]].copy()
# Pick Japan plus three other teams to compare
teams = [
    "Japan Women's",
    "United States Women's",
    "England Women's",
    "Netherlands Women's",
]
subset = team_compare.loc[teams].reset_index()
# Plot as grouped bar chart
metrics = ["shots", "goals", "xg"]
subset_melt = subset.melt(
    id_vars="team_name",
    value_vars=metrics,
    var_name="metric",
    value_name="value",
)
fig, ax = create_figure()
sns.barplot(
    data=subset_melt,
    x="metric",
```

```
    y="value",
    hue="team_name",
    ax=ax,
)
ax.set_xlabel("")
ax.set_ylabel("Value")
ax.set_title("Team-Level Comparison (WWC 2019)")
ax.legend(title="")
apply_margins(fig, bottom=0.22)
save_figure(fig, output_path(3, "team-overview"))
```

Figure 3-16 shows the output (note that some improvements were made to optimize for book-friendly display, but the previous code will get you started).

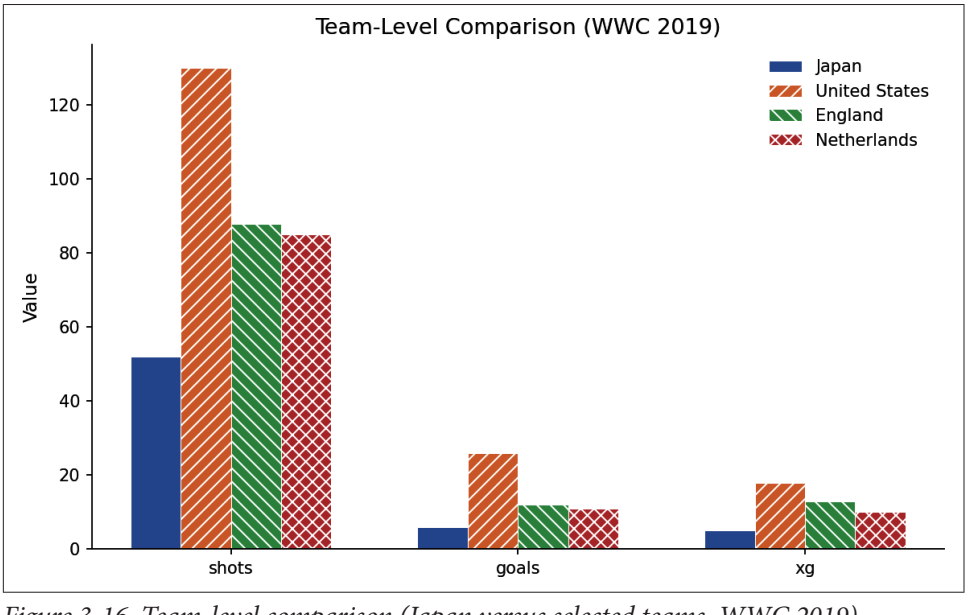


Figure 3-16. Team-level comparison (Japan versus selected teams, WWC 2019)

What does this tell us?

- Japan attempted fewer shots and scored fewer goals than the other teams. (This is somewhat natural in the context of the tournament, as the other teams made it into the semifinals and Japan did not.)
- The US team dominated both shot volume and goals, reflecting its direct, high-tempo style.
- England and the Netherlands combined decent shooting volume with higher finishing output.

Japan's pass accuracy, while not shown here, was among the best in the tournament. This confirms this team's possession-oriented approach: players controlled the ball well but did not always translate that control into large numbers of dangerous chances.

What this means for an analyst: Japan's profile shows a team that prioritizes ball retention and patient buildup over aggressive shot-seeking. While these players created fewer chances than other top teams, their possession-based approach allowed them to control games. This style can be very effective, especially against teams that struggle to break down organized defenses.

Match Flow

Tournament totals give us a broad view, but soccer is played one match at a time. To understand how Japan's style unfolded in practice, we can look at the flow of a single game. The cumulative shots timeline shown previously in Figure 3-10 for the match between Korea and France shows how attacking pressure built up over 90 minutes.

Let's examine Japan's second group stage match against Scotland (StatsBomb match ID 22961), played on June 14, 2019. Japan entered the match needing a result after a 0–0 draw with Argentina, while Scotland was looking to bounce back from a 2–1 defeat to England. The match ended in a 2–1 victory for Japan, with goals from Mana Iwabuchi and Yuika Sugasawa in the first half, before Lana Clelland pulled one back for Scotland late in the game. Figure 3-17 shows this shots timeline.

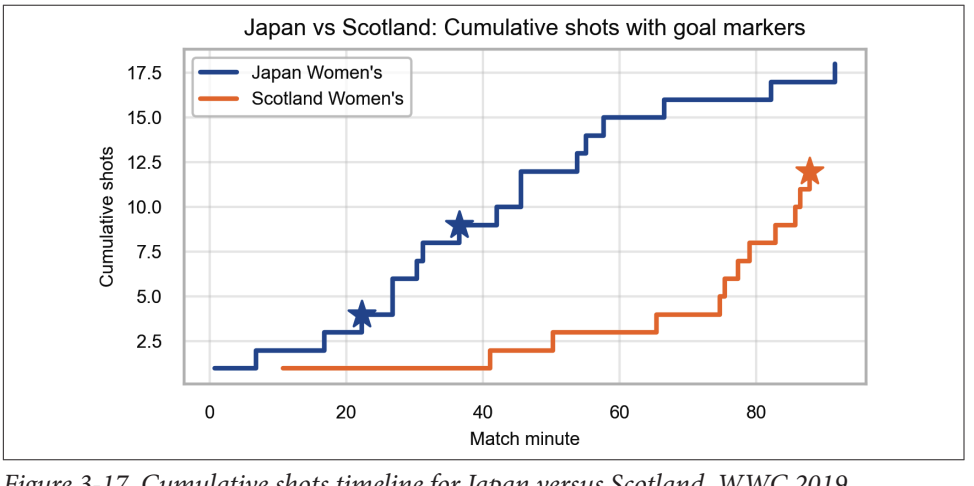


Figure 3-17. Cumulative shots timeline for Japan versus Scotland, WWC 2019

Here's how we'd interpret the timeline for Japan's match against Scotland:

- Japan's line rises steadily in the first half, reflecting the team's early dominance and patient chance creation. The players converted this pressure into a 2–0 lead by the 36th minute, marked by the star symbols on the plot.
- Scotland's line stays remarkably flat for the first 45 minutes, showing very few attempts as Japan controlled possession.
- The momentum shifts visibly in the second half. Japan's shot accumulation slows down as the team protects its lead, while Scotland pushes hard from around minute 50 onward, culminating in Clelland's 87th-minute goal.

Timelines like this highlight each side's momentum, not just the final shot count. By adding goal markers to the plot, we can immediately see how scoring events relate to periods of sustained pressure. They give coaches and analysts a sense of whether a team's style was sustained throughout the match or concentrated in short bursts, and how the game state (winning or losing) affected each team's attacking output.

Spatial Patterns

Match timelines show when shots were taken. To understand where Japan built its play, we return to pass locations. Earlier we showed how to build a heatmap of pass start positions (Figure 3-12). Let's now interpret that view for Japan specifically.

The heatmap reveals a team comfortable circulating the ball across the width of the pitch, with particular density in the central third. This aligns with the team's reputation for possession and midfield control. Rather than relying on long clearances or wing-only attacks, Japan spread play through central areas before probing forward.

A complementary view maps shots by location. If we apply the same approach by using `shot` events instead of passes, we would see Japan's attempts clustered just inside the penalty area, with relatively few speculative long-range efforts. This shot profile matches the team's buildup style: patient passing sequences that aim to create higher-quality chances near the goal.

Player-Level Insights

Team and match summaries give us the big picture. But soccer is often decided by a few key players. Who drove Japan's possession style, and who took the most shots?

A simple summary of passes by player already tells us a lot:

```
# Pass attempts by Japan players
japan_passes = passes.query('team_name == "Japan Women\'s"')
top_passers = (
    japan_passes.groupby("player_name")["completed"]
    .agg(attempts="count", completions="sum")
```

```
        .assign(
            pass_accuracy=lambda d: (
                d["completions"] / d["attempts"]
            )
        )
        .sort_values("attempts", ascending=False)
        .head(8)
    )
    print(top_passers)
```

Table 3-3 highlights Japan's midfielders as the central hubs of possession. Players like Saki Kumagai and Hina Sugita appear frequently as top passers, reflecting their role in recycling the ball through the midfield.

Table 3-3. Completions and pass accuracy per match for Japan

player_name	attempts	completions	pass_accuracy
Saki Kumagai	274	254	0.927007
Aya Sameshima	268	205	0.764925
Hina Sugita	264	216	0.818182
Risa Shimizu	242	207	0.855372
Narumi Miura	190	157	0.826316
Nana Ichise	173	156	0.901734
Emi Nakajima	149	108	0.724832
Yui Hasegawa	129	97	0.751938

We can produce a similar table for shots:

```
# Shot attempts by Japan players
japan_shots = shots.query('team_name == "Japan Women\'s"')
top_scorers = (
    japan_shots.groupby("player_name")["is_goal"]
    .agg(shots="count", goals="sum")
    .sort_values("shots", ascending=False)
    .head(8)
)
print(top_scorers)
```

For Japan, forwards like Yuika Sugasawa and Mana Iwabuchi stand out as primary shooters, with Iwabuchi also contributing goals. You'll see this result for yourself if you run the code and produce the corresponding table.

Together, these two simple leaderboards give us a concise style profile: midfield metronomes maintaining possession, paired with a small group of forwards tasked with converting that control into goals.

Wrap-up of the Case Study

By walking through Japan's tournament, we have put the pieces of EDA together:

- Tournament-level stats (Figure 3-16) showed Japan's accuracy in possession but lower shot and goal volume than other teams.
- Match timelines (Figure 3-10) highlighted steady but moderate attacking pressure rather than sudden bursts.
- Spatial views (Figure 3-12) confirmed their preference for central buildup and patient circulation.
- Player summaries pointed to midfielders as passing hubs and a small group of forwards as the primary shooters.

This is the power of exploratory analysis: stitching together multiple views to form a coherent story of how a team plays.

But EDA is not the end goal. Each of the summaries we built—pass accuracy, cumulative shots, xG per shot, player leaderboards—can also serve as features for predictive models. That is the natural next step: moving from describing what happened to predicting what might happen.

From EDA to Machine Learning

Our exploration of Japan's tournament showed how passes, shots, and visualizations combine to describe a team's playing style. EDA gives us that intuition, but its real power is that the same cleaned and summarized data can also feed predictive models.

When we calculated pass accuracy, we treated it as a summary of team style: Japan completed four out of every five passes. Yet we can also recast pass accuracy as a variable in a model. Teams with higher pass accuracy may be more likely to control possession and, in turn, more likely to win. The same holds for expected goals per shot, average shot distance, or the percentage of defensive duels won. What looks like a descriptive statistic becomes a feature, a piece of structured information that can help a model predict outcomes.

This shift is mostly conceptual. We are not creating new numbers but reinterpreting existing ones. During EDA, a number is interesting if it tells us something about what happened on the pitch. In modeling, a number is useful if it helps predict what might happen next. The same data serves both purposes, depending on the question.

Features as Building Blocks

Machine learning models cannot read charts or interpret match reports. They require inputs in a very specific shape: a table of numbers. Each row represents one observation, such as a team in a given match or a player across a season. Each column represents one feature, such as pass accuracy, possession share, or xG per shot. Optionally, one column acts as the target—the outcome we want to predict, such as whether a team won or lost.

Conveniently, the DataFrames we built during EDA already have this structure. A team-level summary table has rows for teams and columns for statistics, and can easily be extended with a target column. The same applies at the player or match level. This is why EDA matters so much: it not only helps us understand the game, but it also prepares the raw material for modeling.

In practice, much of the art of applied machine learning lies in deciding which features to include. A model trained only on possession share and shots per game may tell a different story from one trained on expected goals and defensive duels. Choosing features is as much about understanding soccer as it is about manipulating data. Analysts who know both the game and the data are best placed to make these decisions.

The Standard Pipeline

By reframing descriptive statistics as features, we arrive at the standard pipeline used across data science:

EDA → Features → Models

First, we explore data to see what is there. Next, we translate those insights into structured features. Finally, we feed those features into models that generalize from past matches to future ones. Each step is built on the preceding one, and each step adds a layer of value. The skills you have practiced in this chapter (cleaning data, building summaries, and drawing visualizations) are not just preliminary exercises. They are the essential preparation for predictive modeling.

Looking Ahead: From Exploration to Regression

In the next chapter, we will begin to train models that make use of these features. For now, the key takeaway is that exploratory analysis is not a side track. It is the foundation on which modeling is built. The statistics you have already computed are not only useful for describing matches; they are also the building blocks of prediction. With this bridge in place, let's pause for some practical advice on how to make your exploratory analysis more effective before we move on to machine learning.

Conclusion

In this chapter, we've taken our first steps onto the pitch of soccer analytics. We've learned how to conduct a structured reconnaissance of a new dataset, moving from a high-level overview down to the granular details of on-pitch events. We've seen how to wrangle data with pandas and, most excitingly, how to bring it to life with pitch visualizations using mplsoccer.

Here's a simple checklist for your future EDA missions:

1. Understand the levels: what levels of data do you have (match, player, event)?
2. Load and inspect: get the data into a DataFrame and use `.info()` and `.head()` to get your bearings.
3. Explore by type: systematically investigate numerical, categorical, and ordinal features.
4. Visualize: use standard plots and, when appropriate, pitch-based visualizations to uncover patterns.
5. Clean and aggregate: handle basic data issues and use `groupby()` to create summaries.

With these fundamental skills, you are no longer just a spectator. You are an analyst, ready to uncover the stories hidden in the data. The rhythm of explore, visualize, and clean will serve you well as we move into the predictive modeling chapters ahead.

Before wrapping up, here are a few habits that can make your exploratory analysis more effective and more enjoyable:

- Keep notes as you go. Document your steps directly in Jupyter notebooks so that you or your teammates can revisit and reproduce the workflow later.
- Frame questions in soccer terms. Code is only a tool. The real power comes from asking meaningful soccer questions such as *Which players connect midfield to attack?* rather than *Which columns can I plot?*
- Watch out for bias. Focusing on favorite players or teams is tempting. A good analyst looks across the entire pitch and lets the data challenge preconceptions.
- Mix quick checks with deeper dives. Rapid visualizations help spot anomalies, while more polished figures communicate insights to others.



Here are a few tasks you can try to practice your EDA skills:

- For Figure 3-9, can you identify the specific matches where the US had unusually low passing accuracy?
- Can you add goals or xG to the scatterplot in Figure 3-11 to see whether teams that create higher-quality chances also score more goals?
- Pick a different team from the tournament and create a similar case study to the one we did for Japan. How does its style compare?

The goal is not to squeeze every possible number out of the data but to establish a rhythm: explore, visualize, and clean. Once you are comfortable with this rhythm, you will be ready to take the next step into predictive modeling.

CHAPTER 4

Regression Techniques for Soccer Analytics

In Chapters 2 and 3, we built the foundations of the analytical workflow used throughout this book. We set up the environment, learned the core tools, and used exploratory analysis to understand the data. The next step is to move from description to prediction.

This chapter introduces *regression*, the first predictive framework we will present in depth. Regression is used when the outcome we want to estimate is numerical rather than categorical. In soccer, that leads to questions such as these:

- What is a player's market value likely to be next season?
- How many goals can we expect a team to score in its next match?
- How many kilometers will a midfielder cover in a game?

These are all questions that regression models can help us answer. Regression does not remove uncertainty, but it gives us a structured way to estimate expected values, compare influences, and understand how outcomes change as inputs change. That makes it useful for both forecasting and explanation.

By the end of this chapter, you will be able to do the following:

- Understand the core concepts of regression and how it applies to soccer.
- Build, train, and interpret a linear regression model to predict player market values.
- Use Poisson regression to model the number of goals in a match, a classic soccer analytics problem.
- Implement K-nearest neighbors (KNN) regression to find and analyze similar players.

- Evaluate your models to understand how well they perform and where they can be improved.

We will work mainly with pandas, Matplotlib, and Seaborn, while also introducing scikit-learn and statsmodels. The goal is not only to fit regression models, but to understand when they are appropriate, how to evaluate them, and how to interpret them in soccer terms.

From Lines on a Field to Lines of Best Fit: Linear Regression

Let's start with the most fundamental regression algorithm: *linear regression*. The concept is fairly intuitive; it's all about finding a straight-line relationship between one or more input variables (we'll call them *features* or *predictors*) and a single output variable (the *target*).

Imagine you're a scout trying to estimate a young player's future market value. You'd probably look at a few key details: their age, the number of goals they've scored, maybe the number of assists they've provided. Your brain intuitively creates a model: a younger player with lots of goals is probably worth more. Linear regression does the same thing, but with mathematical precision.

The algorithm tries to find the best straight line that describes the data. For example, if we plot a player's market value against the number of goals they scored last season, we might see a pattern. As goals increase, market value tends to increase. Linear regression finds the one line that cuts through that data and best represents that trend.

Predicting Market Value: A Player's Worth

Let's make this approach concrete with a classic soccer analytics problem: predicting a player's market value. A player's value is determined by a host of factors, but for our first model, we'll keep it simple. We'll try to predict a player's market value based on just one feature: their goal-scoring record.

First, we need data. We'll use a simulated dataset for this example, representing a collection of attacking players in a top European league:

```
import pandas as pd
# Simulated data for player market values
data = {
    "Player": [
        "Player A",
        "Player B",
        "Player C",
        "Player D",
```

```
        "Player E",
        "Player F",
        "Player G",
    ],
    "Goals": [22, 15, 8, 25, 12, 18, 5],
    "MarketValue_Millions": [90, 60, 25, 100, 45, 70, 15],
}
df = pd.DataFrame(data)  ❶
print(df)
```

❶ Create a pandas DataFrame, our primary tool for handling structured data.

Before we build a model, it's always a good idea to visualize the relationship. A scatterplot is perfect for this. (We'll keep using the shared plotting helper introduced in Chapter 3; its implementation details are omitted here for brevity.) The following code implements such a scatterplot:

```
import seaborn as sns
from soccerml_figures.plotting import (
    apply_margins,
    create_figure,
    output_path,
    save_figure,
    use_publication_style,
)
use_publication_style()  ❶
fig, ax = create_figure()
sns.scatterplot(
    x="Goals", y="MarketValue_Millions", data=df, ax=ax,
)  ❷
ax.set_title("Player Market Value vs. Goals Scored")
ax.set_xlabel("Goals Scored Last Season")
ax.set_ylabel("Market Value (in Millions €)")
apply_margins(fig)
save_figure(fig, output_path(4, "market-value-scatter"))  ❸
```

❶ Use Seaborn's styling to make our plots look a bit nicer.

❷ The `scatterplot()` function is a straightforward way to visualize the relationship between two numerical variables.

❸ Save the plot to a file so we can include it in our reports (and this chapter!).

Figure 4-1 shows the clear positive relationship: players who score more goals tend to have a higher market value. The line is not perfectly straight, but the trend is unmistakable. This is a good sign! It suggests that linear regression is a reasonable model to try here.

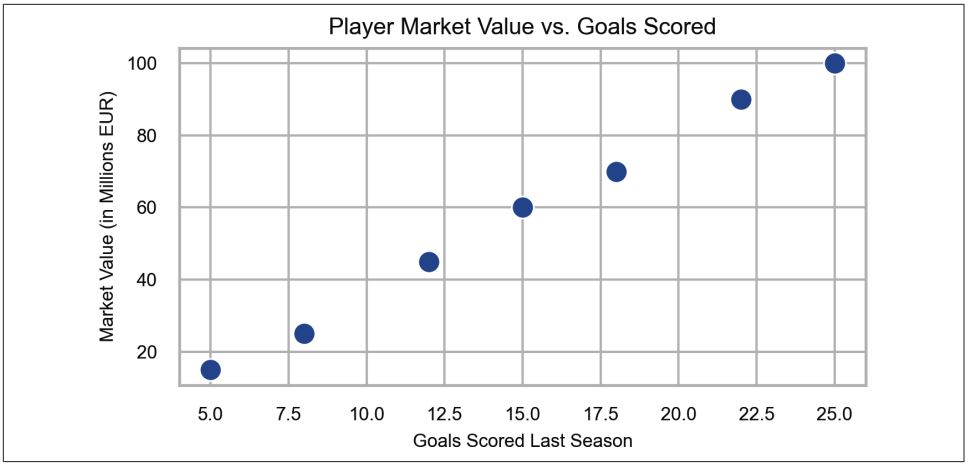


Figure 4-1. Player market value versus goals scored

Building the Model: The Line of Best Fit

Now for the exciting part: building the model. We'll use the scikit-learn library, which is the gold standard for machine learning in Python. The process is remarkably straightforward:

```
from sklearn.linear_model import LinearRegression
# Prepare the data for scikit-learn
X = df[["Goals"]]  ❶
y = df["MarketValue_Millions"]
# Create and train the model
model = LinearRegression()  ❷
model.fit(X, y)  ❸
# Get the model's parameters
intercept = model.intercept_
slope = model.coef_[0]
print(f"Model Intercept: {intercept:.2f}")
print(f"Model Slope (Coefficient for Goals): {slope:.2f}")
```

❶ scikit-learn expects the features (`X`) to be in a 2D array-like structure, which is why we use double brackets `[[]]`. The target (`y`) can be a 1D series.

❷ Create an instance of the `LinearRegression` class.

❸ The `.fit()` method is where the magic happens. The model learns the relationship between the features (`X`) and the target (`y`) from the data.

So, what do these `intercept` and `slope` values mean? They define our line of best fit. Here's the equation of our model:

$$\text{Market value} = \text{slope} \times \text{goals} + \text{intercept}$$

In our case, the model found an intercept of –5.5 and a slope of approximately 4.15:

Intercept (–5.5)
: A player who scores 0 goals is predicted to have a market value of –€5.5 million. This obviously doesn't make sense in the real world! This occurrence is common in simple models, however, and highlights their limitations. The intercept is often just a mathematical necessity to correctly position the line and shouldn't always be interpreted literally, especially when the feature (goals) is never 0 in our data's context.

Slope (4.15)
: For every additional goal a player scores, their predicted market value increases by €4.15 million. This may seem like a lot, but this is what the (simulated) data tells us! It is the core insight from our model. It quantifies the relationship we saw in the scatterplot.

Visualizing the Model

Now we can visualize our model's predictions by drawing this line of best fit on our scatterplot:

```
fig, ax = create_figure()
sns.scatterplot(
    x="Goals",
    y="MarketValue_Millions",
    data=df,
    s=100,
    label="Actual Data",
    ax=ax,
) ❶
# Plot the regression line
sorted_df = df.sort_values("Goals")
ax.plot(
    sorted_df["Goals"],
    model.predict(sorted_df[["Goals"]]),
    color="red",
    linewidth=3,
    label="Regression Line",
) ❷
ax.set_title("Linear Regression: Market Value vs. Goals Scored")
ax.set_xlabel("Goals Scored Last Season")
ax.set_ylabel("Market Value (in Millions €)")
ax.legend()
```

```
apply_margins(fig)
save_figure(fig, output_path(4, "market-value-regression-line"))
```

❶ Make the data points a bit larger (`s=100`) to make them stand out.

❷ Use `model.predict(X)` to get the model's predicted values for each player and plot this as a line.

Figure 4-2 shows our linear regression model as a line. It's the single straight line that minimizes the overall distance to all the data points. It captures the essence of the trend, and we can now use it to make predictions for new players.

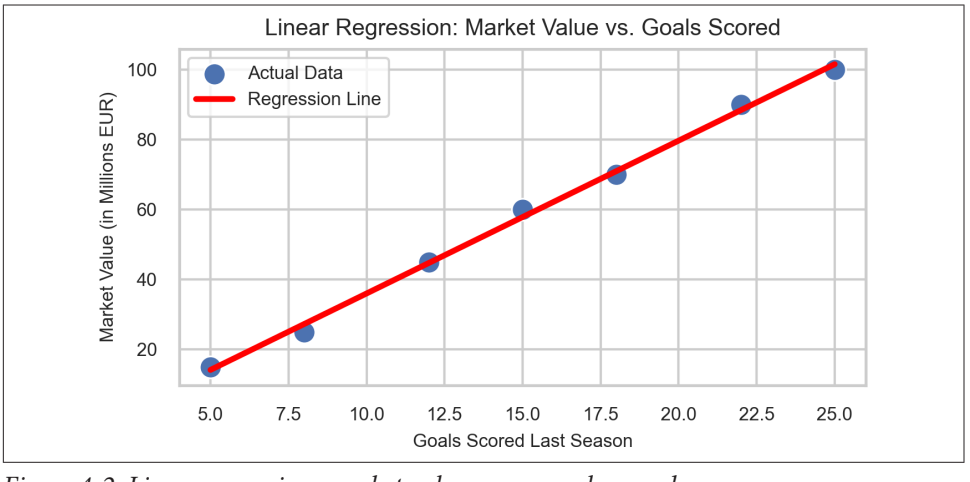


Figure 4-2. Linear regression: market value versus goals scored

Evaluating Performance: How Good Is Our Model?

We've built a model, and it looks reasonable on the plot, but how do we *quantify* its performance? A visual check is good, but it's not enough. We need metrics. For regression models, one of the most common metrics is the *R-squared* (R^2) value.

R^2 tells us the percentage of the variation in the target variable (market value) that can be explained by our features (goals). The value ranges from 0 to 1:

- An R^2 of 1 means the model perfectly explains the data (all points lie exactly on the regression line).
- An R^2 of 0 means the model explains none of the variability.

Let's calculate the R^2 for our model:

```
from sklearn.metrics import r2_score
# Get the model's predictions
y_pred = model.predict(X)
# Calculate R-squared
r2 = r2_score(y, y_pred)
print(f"R-squared: {r2:.2f}")
```

The output will be an R^2 value of around 0.95. This is a very high value! It suggests that 95% of the variation in players' market values in our dataset can be explained by the number of goals they scored. In a real-world scenario, a value this high for a simple one-feature model is rare and might suggest that our simulated data is a bit too perfect. However, this output serves to illustrate the concept.

Another way to evaluate the model is to look at the *residuals*—the differences between the actual values and the predicted values. A good model should have residuals that are randomly scattered around 0. Let's take a look:

```
residuals = y - y_pred
fig, ax = create_figure()
sns.scatterplot(x=y_pred, y=residuals, ax=ax)
ax.axhline(y=0, color="r", linestyle="--")  ❶
ax.set_xlabel("Predicted Market Value")
ax.set_ylabel("Residuals")
ax.set_title("Residual Plot")
apply_margins(fig)
save_figure(fig, output_path(4, "market-value-residuals"))
```

❶ Add a horizontal line at `y=0` to make it easier to see whether the residuals are evenly distributed.

If you see a clear pattern in Figure 4-3 (e.g., a U-shape), it's a sign that the linear model might not be the best fit for the data. In our case, the points are few, but they seem to be randomly scattered, which is what we want to see. So although the model might not be ideal for a general use case, for this dataset, the residuals suggest that it's a good fit.

This is just the beginning of regression. We've used only one feature, but the real power comes when we add more. What if we added assists, age, and position? The principles remain the same, but the models become more powerful and, often, more accurate.

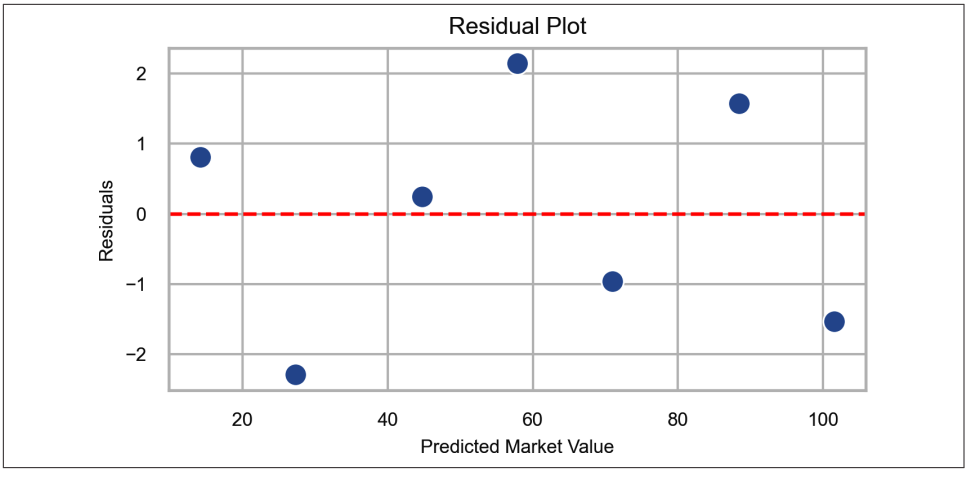


Figure 4-3. Our residual plot for linear regression

Counting Goals: Poisson Regression

Linear regression is great for predicting continuous values like market value, but what about counting things? In soccer, one of the most fundamental counts is the number of goals scored in a match. This is a classic soccer analytics problem, and it has a specific tool that is perfect for the job: Poisson regression.

Why can't we just use linear regression to predict goals? There are a few reasons:

- Goals are counts, not continuous values. A team can score 0, 1, 2, or 3 goals, but not 2.5 goals. Linear regression can predict fractional values, which don't make sense here.
- Goals are non-negative. A team can't score negative goals. A linear regression model could, in theory, predict a negative number.
- The relationship isn't always linear. The factors that influence goal scoring might not have a simple straight-line relationship with the number of goals.

Poisson regression is a type of generalized linear model (GLM) that is specifically designed for count data. This model makes a few key assumptions, most notably that the number of events (goals) in a fixed interval (a match) is, on average, constant, and the events are independent. This fits the scenario of a soccer match quite well.

Predicting Goals in a Match

Let's try to predict the number of goals a team might score based on the number of shots on target that players take. We'll use another simulated dataset:

```
import pandas as pd
# Simulated data for goals scored
data = {
    "Team       body": [f"Team {i}" for i in range(1, 21)],
    "ShotsOnTarget": [ 5, 8, 12, 4, 6, 9, 10, 3, 7, 11, 5, 8, 12,
        4, 6, 9, 10, 3, 7, 11],
    "Goals": [1, 2, 3, 0, 1, 2, 2, 0, 1, 3, 1, 2, 4, 1, 1, 2, 3, 0, 2, 3]
}
goals_df = pd.DataFrame(data)
print(goals_df.head())
```

Now, let's build our Poisson regression model. This time, we'll use the statsmodels library, which provides excellent support for GLMs and gives detailed statistical summaries of the model's performance:

```
import statsmodels.api as sm
# Build the Poisson model
formula = "Goals ~ ShotsOnTarget"  ❶
model = sm.GLM.from_formula(
    formula, data=goals_df, family=sm.families.Poisson(),
).fit()  ❷
# Print the model summary
print(model.summary())
```

❶ statsmodels allows us to use an R-style formula to define our model. `Goals ~ ShotsOnTarget` means we are predicting `Goals` based on `ShotsOnTarget`.

❷ Create and fit the Poisson model in one line.

The model summary is packed with information, but let's focus on the `coef` column for our `ShotsOnTarget` variable. This column tells us the relationship between shots on target and the *log* of the expected number of goals. This is a key difference from linear regression. In Poisson regression with a log link (the default), the coefficients are on a logarithmic scale.

To interpret the coefficient, we need to exponentiate it. If the coefficient for `ShotsOnTarget` is, say, 0.15, then `exp(0.15) ≈ 1.16`. This means that for every additional shot on target, the expected number of goals is multiplied by 1.16, or in other words, it increases by 16%.

Visualizing the Poisson Model

Visualizing a Poisson model is a bit different from visualizing a linear model. Instead of a straight line, we get a curve that represents the expected number of goals for a given number of shots on target. Here's the code:

```
# Get the model's predictions
goals_df["predicted_goals"] = model.predict(goals_df)
sorted_goals_df = goals_df.sort_values("ShotsOnTarget")
fig, ax = create_figure()
# Plot actual goals
sns.scatterplot(
    x="ShotsOnTarget",
    y="Goals",
    data=goals_df,
    label="Actual Goals",
    s=100,
    ax=ax,
)
# Plot predicted goals
ax.plot(
    sorted_goals_df["ShotsOnTarget"],
    sorted_goals_df["predicted_goals"],
    color="red",
    linestyle="--",
    label="Predicted Goals (Poisson)",
)
ax.set_title("Poisson Regression: Goals vs. Shots on Target")
ax.set_xlabel("Shots on Target")
ax.set_ylabel("Number of Goals")
ax.legend()
apply_margins(fig)
save_figure(fig, output_path(4, "goals-poisson-regression"))
```

As you can see in Figure 4-4, the predicted line is not straight.

The predicted line curves upward, suggesting that the effect of an additional shot on target is greater when a team is already taking many shots. This kind of nonlinear relationship is something that Poisson regression can capture, while a simple linear model cannot.

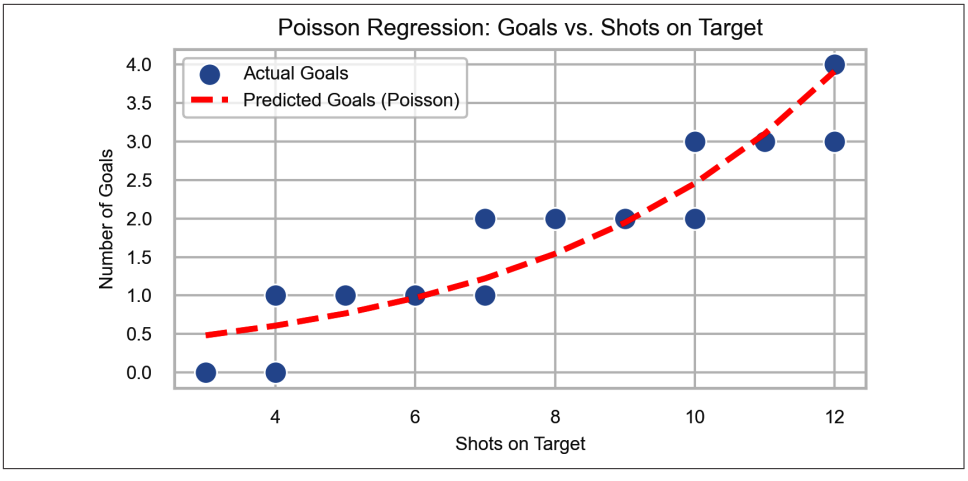


Figure 4-4. Poisson regression: goals versus shots on target

Finding Similar Players: K-Nearest Neighbors Regression

So far, we've looked at models that try to find a mathematical formula (a line or a curve) to describe the relationship between features and a target. But what if the relationship is too complex for a simple formula? What if we just want to find players who are similar to a given player and see their market values?

This is where KNN regression comes in. It's a completely different way of thinking about prediction. Instead of learning a single model from all the data, KNN makes predictions on the fly by looking at the *neighbors* of a data point.

The idea is simple and intuitive:

1. Choose a number *k*. This is the number of neighbors to consider.
2. Find the *k* nearest neighbors by finding the *k* players in your dataset who are most similar, based on their features (e.g., goals, assists, age), to the new player you want to predict the value for.
3. Average their values. The predicted market value for the new player is simply the average market value of their *k* nearest neighbors.

It's like asking, "This new young striker scored 15 goals last season. Who are the five most similar players to them in my database, and what are their market values?" The average of those values is your KNN prediction.

Predicting Player Value with KNN

Let's use our market value dataset again, but this time with an additional feature, assists:

```
import pandas as pd
# Simulated data with two features
data = {
    "Player": [
        "Player A",
        "Player B",
        "Player C",
        "Player D",
        "Player E",
        "Player F",
        "Player G",
    ],
    "Goals": [22, 15, 8, 25, 12, 18, 5],
    "Assists": [10, 12, 5, 8, 9, 7, 3],
    "MarketValue_Millions": [90, 60, 25, 100, 45, 70, 15],
}
knn_df = pd.DataFrame(data)
print(knn_df)
```

Now, let's use scikit-learn to build a KNN regression model. A crucial step before using KNN is feature scaling. KNN works by measuring distances between data points. If one feature (like goals) has a much larger scale than another (like assists), it will dominate the distance calculation. We need to scale our features so they are on a comparable scale:

```
from sklearn.neighbors import KNeighborsRegressor
from sklearn.preprocessing import StandardScaler
# Prepare the data
features = ["Goals", "Assists"]
X = knn_df[features]
y = knn_df["MarketValue_Millions"]
# Scale the features
scaler = StandardScaler()  ❶
X_scaled = scaler.fit_transform(X)
# Create and train the model (let's use k=3)
knn_model = KNeighborsRegressor(n_neighbors=3)  ❷
knn_model.fit(X_scaled, y)
# Let's predict the value of a new player
new_player = [[20, 9]]  # 20 goals, 9 assists
new_player_scaled = scaler.transform(new_player)  ❸
predicted_value = knn_model.predict(new_player_scaled)
print(
    "Predicted Market Value for new player: "
    f"{predicted_value[0]:.2f} Million €"
)
```

❶ `StandardScaler()` transforms the data to have a mean of 0 and a standard deviation of 1.

❷ Create a KNN regressor and specify `n_neighbors=3`.

❸ We must apply the same scaling to our new data before making a prediction.

Choosing the Right k

The choice of *k* is crucial. A small *k* (like 1 or 2) will make the model very sensitive to noise in the data. A large *k* will make the model's predictions smoother but might obscure local patterns. Finding the optimal *k* often involves trying several values and seeing which one gives the best performance on a validation set—a topic we'll explore more in the following section.

KNN is a powerful and intuitive algorithm, especially when the relationship between features and the target is complex and nonlinear. It's a great tool to have in your soccer analytics toolbox.

Evaluating and Interpreting Regression Models in Soccer Contexts

Building a model is one thing, but knowing whether it's any good is another. In the world of soccer analytics, a model that can't be trusted is worse than no model at all. It can lead to poor decisions, like overpaying for a player or misjudging a team's strength. In this section, we'll discuss R^2 key techniques for evaluating and interpreting your regression models.

Using Key Performance Metrics

We've already touched on R^2, but let's formalize it and introduce a few other key metrics:

R-squared (R^2)
: As you saw, this tells you the proportion of the variance in the target variable that is predictable from the feature(s). It's a measure of how well the model explains the data. A higher R^2 is generally better, but it's not the whole story. A high R^2 on your training data doesn't guarantee that the model will perform well on new, unseen data.

Mean absolute error (MAE)
: This is the average of the absolute differences between the predicted and actual values. It gives you a straightforward idea of how wrong your predictions are on average, in the original units of the target variable. For our market value model, an MAE of 5 would mean our predictions are off by €5 million on average.

Mean squared error (MSE)
: This is the average of the squared differences between the predicted and actual values. By squaring the errors, larger errors are penalized more heavily than smaller ones. The metric is common, but because the units are squared (e.g., millions of euros squared), it can be hard to interpret.

Root mean squared error (RMSE)
: This is the square root of the MSE. It brings the metric back to the original units of the target variable, making it easier to interpret than MSE. It's like MAE, but it still gives more weight to larger errors.

Let's calculate these for our linear regression model:

```
from sklearn.metrics import (
    mean_absolute_error,
    mean_squared_error,
)
import numpy as np
# We already have y (actual) and y_pred (predicted)
mae = mean_absolute_error(y, y_pred)
mse = mean_squared_error(y, y_pred)
rmse = np.sqrt(mse)
print(f"Mean Absolute Error (MAE): {mae:.2f} Million €")
print(f"Root Mean Squared Error (RMSE): {rmse:.2f} Million €")
```

Understanding the Importance of a Validation Set

So far, we've been training and evaluating our models on the same data. This is a cardinal sin in machine learning! A model can perform brilliantly on the data it has already seen but fail miserably on new data. This is called *overfitting*.

To get a true sense of a model's performance, you need to test it on data it has never seen before. The standard practice is to split your data into two or three sets:

Training set
: The largest part of the data, used to train the model.

Validation set
: Used to tune the model's parameters (like the *k* in KNN) and to compare models.

Test set
: Used for the final evaluation of the chosen model. You touch this set only once, at the very end.

We'll dive much deeper into splitting data in later chapters, but it's a crucial concept to understand now.

Interpreting Coefficients

For models like linear and Poisson regression, the coefficients are a goldmine of information.

- In linear regression, the coefficient tells you the average change in the target for a one-unit change in the feature.
- In Poisson regression (with a log link), you need to exponentiate the coefficient. The result tells you the multiplicative factor by which the expected count changes for a one-unit change in the feature.

This is incredibly useful for understanding *why* your model is making certain predictions. It moves you from just being a data scientist to being a true analyst who can provide actionable insights.

Visualizing KNN: Decision Boundaries

Because KNN doesn't create a simple formula, visualizing it is a bit different. We can't draw a single line. Instead, we can visualize the *decision boundary*—the regions in the feature space where the model's prediction is the same. For KNN regression, this creates a step-like surface.

Let's visualize the feature space for our two features, `Goals` and `Assists`, and see how a new player fits in:

```
from sklearn.neighbors import KNeighborsRegressor
from sklearn.preprocessing import StandardScaler
X_2d = df[["Goals", "Assists"]]
y_2d = df["MarketValue_Millions"]
scaler_2d = StandardScaler()
X_2d_scaled = scaler_2d.fit_transform(X_2d)
knn_2d = KNeighborsRegressor(n_neighbors=best_k)
knn_2d.fit(X_2d_scaled, y_2d)
goals_range = np.linspace(
    X_2d["Goals"].min(), X_2d["Goals"].max(), 80,
)
assists_range = np.linspace(
    X_2d["Assists"].min(), X_2d["Assists"].max(), 80,
)
goals_grid, assists_grid = np.meshgrid(
    goals_range, assists_range,
```

```
)
grid_points = np.c_[goals_grid.ravel(), assists_grid.ravel()]
predictions = knn_2d.predict(
    scaler_2d.transform(grid_points)
).reshape(goals_grid.shape)
new_player = pd.DataFrame({"Goals": [20], "Assists": [8]})
fig, ax = create_figure()
contour = ax.contourf(
    goals_grid,
    assists_grid,
    predictions,
    levels=15,
    cmap="viridis",
    alpha=0.75,
)
ax.scatter(
    df["Goals"],
    df["Assists"],
    c=df["MarketValue_Millions"],
    s=70,
    edgecolors="black",
    linewidths=0.6,
    cmap="viridis",
    label="Existing Players",
)
ax.scatter(
    new_player["Goals"],
    new_player["Assists"],
    color="red",
    marker="*",
    s=220,
    edgecolors="white",
    linewidths=1.5,
    label="New Player",
    zorder=4,
)
fig.colorbar(contour, ax=ax, pad=0.02,).set_label(
    "Predicted Market Value (€M)"
)
ax.set_xlabel("Goals")
ax.set_ylabel("Assists")
ax.set_title(f"KNN Decision Surface (K={best_k})")
ax.legend(loc="upper left")
apply_margins(fig, right=0.9)
save_figure(fig, output_path(4, "knn-player-similarity"))
```

The plot in Figure 4-5 shows our existing players, with their market value indicated by the variation in tone.

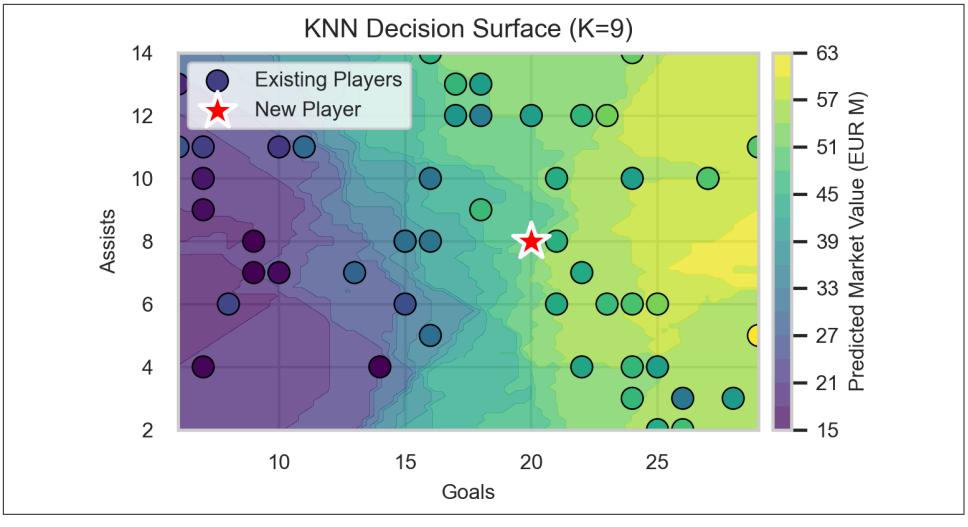


Figure 4-5. KNN regression: player similarity in feature space

In this plot, the star is our new player. The KNN algorithm will find the *k* closest points to this star and average their values to make a prediction.

We can also see how the value of *k* changes the prediction surface. A smaller *k* creates a more jagged, complex surface, while a larger *k* creates a smoother one. This code helps visualize the R2 score as function of *k*:

```
k_values = range(1, 21)
train_scores = []
test_scores = []
for k in k_values:
    knn = KNeighborsRegressor(n_neighbors=k)
    knn.fit(X_train_scaled, y_train)
    train_scores.append(knn.score(X_train_scaled, y_train))
    test_scores.append(knn.score(X_test_scaled, y_test))
fig, ax = create_figure()
ax.plot(
    k_values,
    train_scores,
    label="Training R²",
    marker="o",
    linewidth=2,
)
ax.plot(
    k_values,
    test_scores,
    label="Test R²",
    marker="s",
    linewidth=2,
)
```

```
ax.set_xlabel("K (Number of Neighbors)")
ax.set_ylabel("R² Score")
ax.set_title("KNN Performance vs. K Value")
ax.legend()
ax.grid(True, alpha=0.3)
apply_margins(fig)
save_figure(fig, output_path(4, "knn-k-comparison"))
```

As shown in Figure 4-6, the model's predictions change as we vary *k*. Choosing the right *k* is a trade-off between fitting the training data well (low *k*) and creating a more generalizable model (high *k*).

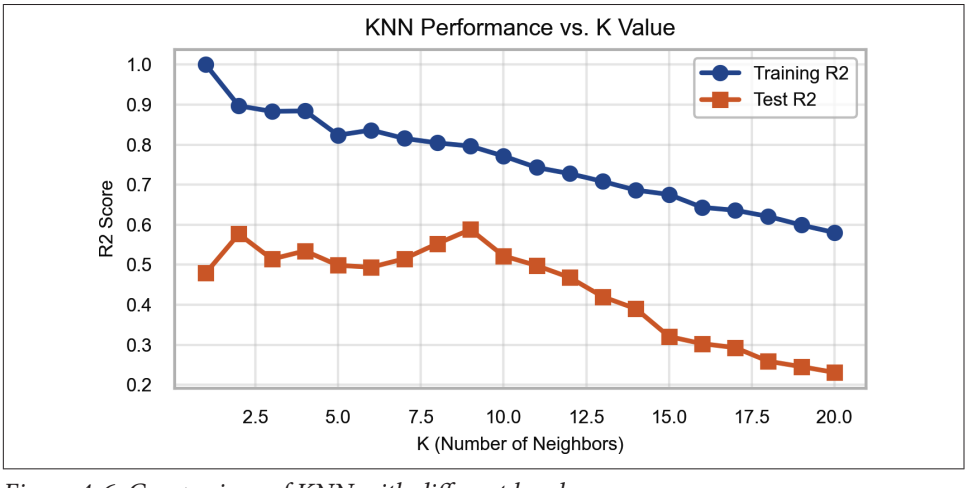


Figure 4-6. Comparison of KNN with different k values

Comparing Models

We've now seen three regression models. Which one do we choose? This is where our evaluation metrics come in handy. Let's imagine we've run all our models on the same dataset and calculated their R^2, MAE, and RMSE on a test set:

```
fig, axes = create_figure(nrows=1, ncols=3)
axes[0].bar(
    results_df["Model"], results_df["R²"], color="#4C78A8",
)
axes[0].set_title("R²")
axes[1].bar(
    results_df["Model"], results_df["MAE"], color="#F58518",
)
axes[1].set_title("MAE")
axes[2].bar(
    results_df["Model"], results_df["RMSE"], color="#54A24B",
)
axes[2].set_title("RMSE")
for ax in axes:
```

```
    ax.grid(True, alpha=0.25, axis="y")
fig.suptitle("Model Comparison: Performance Metrics")
apply_margins(fig, top=0.8, bottom=0.22, wspace=0.35)
save_figure(fig, output_path(4, "model-comparison-metrics"))
```

The comparison in Figure 4-7 allows you to see at a glance which model is performing best.

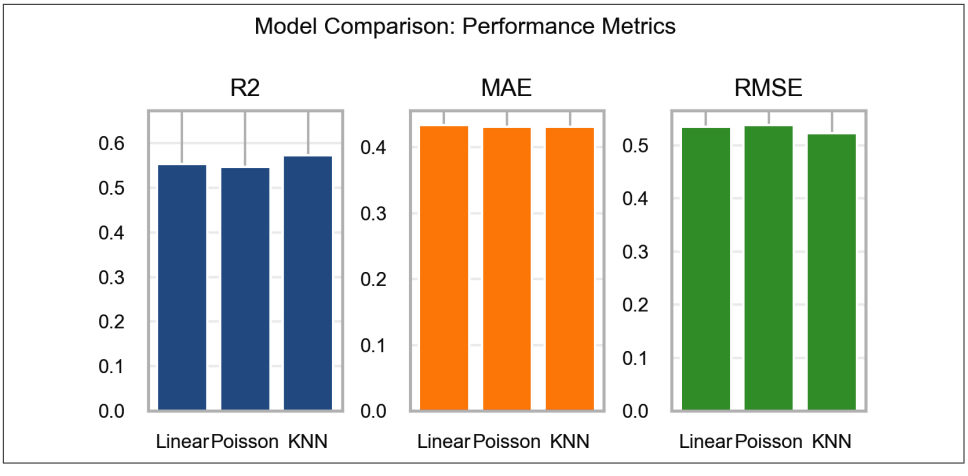


Figure 4-7. Comparing the models' performance metrics

This kind of comparison is essential. It allows you to see at a glance which model is performing best for your specific problem. In this hypothetical case, linear regression has the highest R^2 and the lowest errors, suggesting it's the best model for this particular (and very clean) dataset. In the real world, the choice is often not so clear-cut, and you'll need to balance performance with interpretability and computational cost.

Advanced Topics in Poisson Regression: Overdispersion and Zero-Inflation

While Poisson regression is a powerful tool for modeling count data, it makes a strong assumption: that the mean and variance of the data are equal. In the real world of soccer analytics, this assumption often doesn't hold. Let's explore two common issues and how to address them.

Overdispersion: When Variance Exceeds the Mean

In many soccer datasets, you'll find that the variance in goals scored is greater than the mean. This is called *overdispersion*. For example, if the average number of goals per match is 1.5, but some matches have 0 goals while others have 5 or 6, the variance will be much larger than 1.5.

Why does this matter? When overdispersion is present, the standard errors of the Poisson model are underestimated, which means your confidence intervals will be too narrow and your p-values will be too optimistic. A *p-value* tells you how statistically significant your result is. Typically, a p-value below 0.05 is deemed statistically significant, and above this threshold is considered insignificant, though this cutoff can vary with the use case. You might think a predictor is significant when it's not.

To check for overdispersion, we can calculate the dispersion parameter (φ; phi):

```
# Using our previous Poisson model
residuals = model.resid_pearson  ❶
phi = sum(residuals ** 2) / (
    len(goals_df) - len(model.params)
)  ❷
print(f"Dispersion parameter: {phi:.2f}")
```

❶ *Pearson residuals* are standardized residuals that should have a variance close to 1 if the model fits well.

❷ The dispersion parameter is calculated by dividing the sum of squared Pearson residuals by the degrees of freedom.

If φ is much greater than 1 (say, greater than 1.5), you have overdispersion. In such cases, a negative binomial regression model is often more appropriate.

Negative Binomial Regression

The *negative binomial model* is a generalization of the Poisson model that allows the variance to be greater than the mean. It introduces an additional parameter that captures the extra variability in the data:

```
import statsmodels.formula.api as smf
# Fit a negative binomial model
nb_model = smf.negativebinomial(
    "Goals ~ ShotsOnTarget", data=goals_df,
).fit()
print(nb_model.summary())
```

The negative binomial model will typically provide more-realistic standard errors and confidence intervals when overdispersion is present. It's a go-to model for soccer goal data.

Zero Inflation: Too Many Zeros

Another common issue in soccer data is *zero inflation*—an excess of matches with no goals. This can happen for various reasons: strong defensive tactics, poor attacking play, or even weather conditions. Standard Poisson or negative binomial models might not capture this pattern well.

Zero-inflated models address this by assuming that zeros can come from two sources:

Structural zeros
: Matches where scoring was essentially impossible (e.g., extreme defensive play)

Sampling zeros
: Matches where goals could have been scored, but by chance, they weren't

The model has two components:

- A logistic regression component that predicts the probability of a structural zero
- A count model (Poisson or negative binomial) for the nonstructural cases

Here's how to fit a zero-inflated Poisson model:

```
from statsmodels.discrete.count_model import ZeroInflatedPoisson
# Fit a zero-inflated Poisson model
zip_model = ZeroInflatedPoisson.from_formula(
    "Goals ~ ShotsOnTarget",  ❶
    data=goals_df,
    exog_infl="ShotsOnTarget",  ❷
    inflation="logit",
).fit()
print(zip_model.summary())
```

❶ The main count model formula

❷ The formula for the zero-inflation component. Here, we're using the same predictor, but you can use different ones.

Zero-inflated models are particularly useful when you believe that the process generating zeros is fundamentally different from the process generating positive counts.

Hurdle Models: An Alternative Approach

Hurdle models are similar to zero-inflated models but with a key difference. They assume that all zeros come from one process (the *hurdle*), and all positive counts come from another. There's no mixing of structural and sampling zeros.

The model has two stages:

1. Binary model: Predicts whether the count is zero or positive (logistic regression)
2. Truncated count model: Predicts the count, given that it's positive (truncated Poisson or negative binomial)

While Python doesn't have a built-in hurdle model in statsmodels, you can implement one manually:

```
# Stage 1: Binary model for zero vs. nonzero
goals_df["HasGoals"] = (goals_df["Goals"] > 0).astype(int)
binary_model = smf.logit(
    "HasGoals ~ ShotsOnTarget", data=goals_df,
).fit()
# Stage 2: Truncated count model for positive goals
goals_positive = goals_df[goals_df["Goals"] > 0]
truncated_model = smf.poisson(
    "Goals ~ ShotsOnTarget", data=goals_positive,
).fit()
print("Binary Model (Zero vs. Positive):")
print(binary_model.summary())
print("\nTruncated Count Model (Positive Goals Only):")
print(truncated_model.summary())
```

Hurdle models are great when you believe that the factors influencing whether a team scores at all are different from the factors influencing the number of goals they score after they've crossed that hurdle.

Model Diagnostics: Ensuring That Your Model Is Sound

Building a model is one thing; ensuring it's reliable is another. Model diagnostics are the checks and balances that help you understand whether your model is trustworthy. In this section, we'll cover some essential diagnostic techniques for regression models in soccer analytics.

Residual Analysis: Looking for Patterns

You've already seen residual plots for linear regression. The same principle applies to GLMs. Residuals are the differences between what your model predicts and what actually happened. A good model should have residuals that are randomly scattered, with no clear patterns.

GLMs can have several types of residuals:

Pearson residuals
: Standardized residuals that should have a mean of 0 and a variance of 1 if the model fits well.

Deviance residuals
: Measure each observation's contribution to the model's *deviance* (a measure of model fit).

Anscombe residuals
: Designed to be more normally distributed than Pearson or deviance residuals, especially for count data.

Let's plot the deviance residuals for our Poisson model:

```
# Get deviance residuals
dev_residuals = model.resid_deviance
fig, ax = create_figure()
ax.scatter(model.fittedvalues, dev_residuals)
ax.axhline(y=0, color="r", linestyle="--")
ax.set_xlabel("Fitted Values")
ax.set_ylabel("Deviance Residuals")
ax.set_title("Residual Plot for Poisson Regression")
apply_margins(fig)
save_figure(fig, output_path(4, "poisson-residuals"))
```

If you see a funnel shape (residuals spreading out as fitted values increase), it might indicate heteroscedasticity or overdispersion. *Heteroscedasticity* refers to a situation in which the variability of a dependent variable is unequal across the range of values of an independent variable. If you see a U-shape or other clear pattern, it suggests the model might be missing important features or using the wrong link function.

Goodness-of-Fit Tests

How do we quantify how well our model fits the data? Several tests can help:

Deviance test
: Compares the model's deviance to a chi-squared distribution. A low p-value suggests poor fit.

Pearson chi-square test
: Similar to the deviance test but uses Pearson residuals.

Here's how to conduct goodness-of-fit tests:

```
# Calculate deviance
deviance = model.deviance
df_resid = model.df_resid
# Compare to chi-squared distribution
from scipy.stats import chi2
p_value = 1 - chi2.cdf(deviance, df_resid)
print(f"Deviance: {deviance:.2f}")
print(f"Degrees of Freedom: {df_resid}")
print(f"P-value: {p_value:.4f}")
```

If the p-value is very low (e.g., < 0.05), it suggests the model doesn't fit the data well. This could be due to overdispersion, missing predictors, or an incorrect model specification.

Influential Observations: Outliers that Matter

Some data points have a disproportionate influence on the model. These are called *influential observations*. Two key measures help identify them:

Cook's distance
: Measures how much the model's predictions would change if a particular observation were removed. Values greater than 1 (or $4/n$, where n is the sample size) are considered influential.

Leverage
: Measures how far an observation's predictor values are from the mean. High leverage points can pull the regression line toward them.

Here's how to visualize any influential data points:

```
from statsmodels.stats.outliers_influence import OLSInfluence
# For GLMs, we can use similar diagnostics
influence = model.get_influence()
cooks_d = influence.cooks_distance[0]
fig, ax = create_figure()
ax.stem(range(len(cooks_d)), cooks_d, markerfmt=",")
ax.set_xlabel("Observation Index")
ax.set_ylabel("Cook's Distance")
ax.set_title("Influential Observations")
ax.axhline(
    y=4 / len(goals_df),
    color="r",
    linestyle="--",
    label="Threshold",
)
ax.legend()
apply_margins(fig)
save_figure(fig, output_path(4, "cooks-distance"))
```

If you find influential observations, investigate them. Are they data-entry errors? Unusual but legitimate matches? Understanding these outliers can provide valuable insights.

Model Selection: Choosing the Best Model

When you have multiple models to choose from (e.g., linear versus Poisson versus negative binomial), how do you decide which is best? Several criteria can help:

Akaike information criterion (AIC)
: Balances model fit with model complexity. A Lower AIC is better. It penalizes models with more parameters to avoid overfitting.

Bayesian information criterion (BIC)
: Similar to AIC but with a stronger penalty for complexity. A Lower BIC is better.

Cross-validation
: Split your data into training and test sets. Train the model on the training set and evaluate it on the test set. The model with the best test set performance is the winner.

Here's how to display the AIC and BIC for different models:

```
# Compare AIC and BIC for different models
print(
    f"Poisson Model - AIC: {model.aic:.2f}, "
    f"BIC: {model.bic:.2f}"
)
print(
    "Negative Binomial Model - "
    f"AIC: {nb_model.aic:.2f}, "
    f"BIC: {nb_model.bic:.2f}"
)
```

AIC and BIC are useful for quick comparisons, but cross-validation is the gold standard for assessing a model's predictive performance on new data.

Practical Case Study: Predicting Match Outcomes

Let's bring everything together with a practical case study. We'll use a realistic (though still simulated) dataset of soccer matches to predict the number of goals a team will score based on several features.

Generating the Dataset

Let's first generate the dataset. It contains match-level data for a fictional league's season:

```
import pandas as pd
import numpy as np
# Generate a realistic soccer dataset
np.random.seed(42)
n_matches = 100
match_data = pd.DataFrame(
    {
        "Team": [f"Team {i%10}" for i in range(n_matches)],
        "ShotsOnTarget": np.random.poisson(5, n_matches),
        "Possession": np.random.normal(50, 10, n_matches),
        "Passes": np.random.poisson(300, n_matches),
        "Fouls": np.random.poisson(12, n_matches),
        "HomeAdvantage": np.random.choice([0, 1], n_matches),
    }
)
# Generate goals with a realistic relationship
match_data["Goals"] = np.random.poisson(
    0.1 * match_data["ShotsOnTarget"]
```

```
    + 0.01 * match_data["Possession"]
    + 0.5 * match_data["HomeAdvantage"]
)
print(match_data.head(10))
print(f"\nDataset shape: {match_data.shape}")
print(
    f"Average goals per match: "
    f"{match_data['Goals'].mean():.2f}"
)
print(
    "Matches with zero goals: "
    f"{(match_data['Goals'] == 0).sum()}"
)
```

Performing Exploratory Analysis

Before building models, let's explore the relationships among variables. We want to know which variables contribute to goal scoring. For this, a correlation heatmap is useful:

```
import seaborn as sns
from soccerml_figures.plotting import (
    apply_margins,
    create_figure,
    output_path,
    save_figure,
)
# Correlation heatmap
fig, ax = create_figure()
sns.heatmap(
    match_data[
        [
            "Goals",
            "ShotsOnTarget",
            "Possession",
            "Passes",
            "Fouls",
            "HomeAdvantage",
        ]
    ].corr(),
    annot=True,
    cmap="coolwarm",
    center=0,
    ax=ax,
)
ax.set_title("Feature Correlation Matrix")
apply_margins(fig, left=0.12, right=0.94, bottom=0.2)
save_figure(fig, output_path(4, "correlation-heatmap"))
```

The correlation heatmap in Figure 4-8 shows which features are most strongly associated with goals. This tells us which features we should include in the model we build.

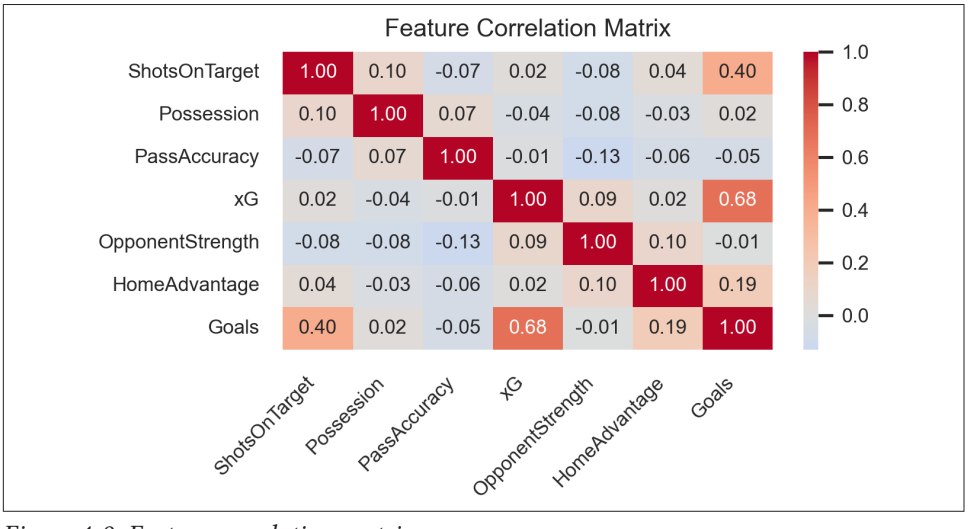


Figure 4-8. Feature correlation matrix

Next, we want to know the number of goals scored in a typical match. This will allow us to build our model around realistic values instead of guesswork. To do so, let's plot a histogram that shows the number of matches with a certain goal count:

```
# Distribution of goals
fig, ax = create_figure()
sns.countplot(x="Goals", data=match_data, ax=ax)
ax.set_title("Distribution of Goals Scored")
ax.set_xlabel("Number of Goals")
ax.set_ylabel("Frequency")
apply_margins(fig)
save_figure(fig, output_path(4, "goals-distribution"))
```

The distribution shown in Figure 4-9 reveals the pattern of goals scored. We can see that the numbers range from 0 to 5 and mostly are between 0 and 3. We can take away from this that building a model for more than 5 goals is futile and that the bulk of predictions should land between 0 and 3, as the data suggests.

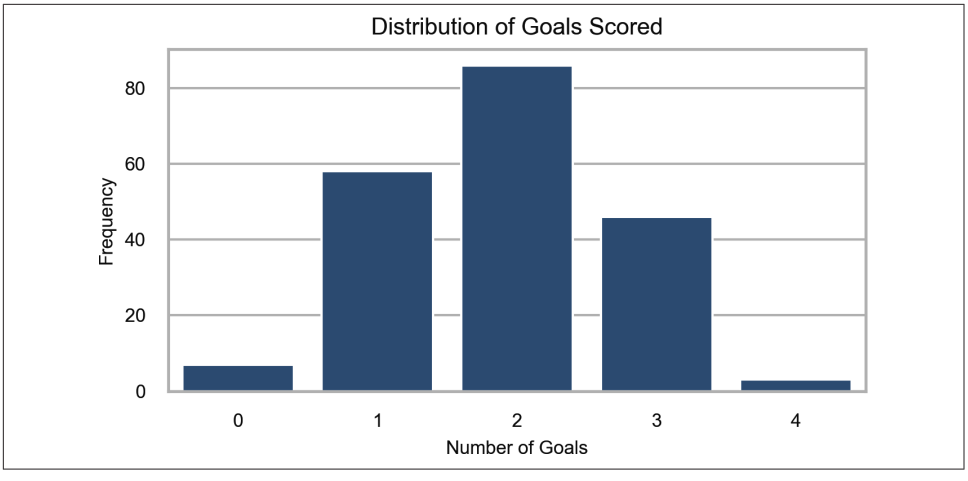


Figure 4-9. Distribution of goals scored

With this exploratory analysis done, we now have a sense of what our model should look like. Let's start building one.

Building and Comparing Models

Rather than randomly choosing a model or relying on experience to build a nice one, the best practice is to build several models and compare their performance. Since we're predicting integers, a Poisson or negative binomial distribution would make sense. We'll use `ShotsOnTarget`, `Possession`, and `HomeAdvantage` as predictors, since these seemed highly correlated with goals scored.

We will omit `Fouls` because this feature, while correlated, can be highly variable across players, matches, and leagues—making this model less scalable if we use it. We'll also include a simple Poisson distribution with just one variable, `ShotsOnTarget`, to see how it does; the simpler the model, the better. Now, let's build and compare the models:

```
import statsmodels.formula.api as smf
from sklearn.model_selection import train_test_split
# Split data into training and test sets
train_data, test_data = train_test_split(
    match_data, test_size=0.2, random_state=42
) ❶
# Model 1: Simple Poisson with one predictor
model1 = smf.poisson(
    "Goals ~ ShotsOnTarget", data=train_data,
).fit()
# Model 2: Poisson with multiple predictors
model2 = smf.poisson(
    "Goals ~ ShotsOnTarget + Possession + HomeAdvantage",
```

```
    data=train_data,
).fit()
# Model 3: Negative binomial with multiple predictors
model3 = smf.negativebinomial(
    "Goals ~ ShotsOnTarget + Possession + HomeAdvantage",
    data=train_data,
).fit()
# Compare models
print(
    "Model 1 (Simple Poisson) "
    "- AIC: {:.2f}, BIC: {:.2f}".format(
        model1.aic, model1.bic
    )
)
print(
    "Model 2 (Multiple Poisson) "
    "- AIC: {:.2f}, BIC: {:.2f}".format(
        model2.aic, model2.bic
    )
)
print(
    "Model 3 (Negative Binomial) "
    "- AIC: {:.2f}, BIC: {:.2f}".format(
        model3.aic, model3.bic
    )
)
```

❶ Split the data 80/20 for training and testing. This is crucial for evaluating how well our model generalizes to new data.

The output of this code will tell us the AIC and BIC values explained earlier. These will tell us which model has the best performance.

Evaluating on Test Data

Now, we cannot rely on AIC and BIC values from model training alone. The true test of a model is how well it performs on data it hasn't seen. For this, we need to run it on the remaining 20% of data we put aside for this purpose, and we need to see what the model error looks like. Note that in this stage, error rate (MAE) is more important than AIC or BIC values, which we've already looked at. Let's evaluate the test data now:

```
from sklearn.metrics import (
    mean_absolute_error,
    mean_squared_error,
)
# Make predictions on test data
test_data["pred_model1"] = model1.predict(test_data)
test_data["pred_model2"] = model2.predict(test_data)
test_data["pred_model3"] = model3.predict(test_data)
# Calculate MAE for each model
mae1 = mean_absolute_error(
```

```
    test_data["Goals"], test_data["pred_model1"]
)
mae2 = mean_absolute_error(
    test_data["Goals"], test_data["pred_model2"]
)
mae3 = mean_absolute_error(
    test_data["Goals"], test_data["pred_model3"]
)
print(f"Model 1 MAE: {mae1:.2f}")
print(f"Model 2 MAE: {mae2:.2f}")
print(f"Model 3 MAE: {mae3:.2f}")
# Visualize predictions vs actual
fig, axes = create_figure(nrows=1, ncols=3)
for ax, (label, column) in zip(
    axes,
    [
        ("Model 1: Simple Poisson", "pred_model1"),
        ("Model 2: Multiple Poisson", "pred_model2"),
        ("Model 3: Negative Binomial", "pred_model3"),
    ],
):
    ax.scatter(test_data["Goals"], test_data[column], alpha=0.6)
    ax.plot([0, 5], [0, 5], "r--")
    ax.set_xlabel("Actual Goals")
    ax.set_ylabel("Predicted Goals")
    ax.set_title(label)
apply_margins(fig, bottom=0.22, wspace=0.35)
save_figure(fig, output_path(4, "model-predictions-comparison"))
```

Before we look at the visual comparison, let's examine the quantitative metrics (Table 4-1). We use the AIC and BIC to evaluate the models on the training data, and MAE to evaluate them on the test data.

Table 4-1. Performance comparison across models

Model	AIC	BIC	Test MAE
Model 1 (simple Poisson)	237.06	241.82	0.97
Model 2 (multiple Poisson)	228.19	237.72	1.06
Model 3 (negative binomial)	230.19	242.10	1.06

Figure 4-10 shows the result. Each dot is one match from the held-out test set. The dashed diagonal is the line of perfect prediction: a dot sitting exactly on it means the model predicted the right number of goals. Dots above the line indicate the model over-predicted; dots below it indicate under-prediction.

Of the three panels, model 2 (multiple Poisson) sits closest to the diagonal on average, reflecting its lower AIC on the training data. Model 1 (simple Poisson) and model 3 (negative binomial) show similar scatter, with model 1 edging ahead on test MAE (0.97 versus 1.06 for the other two). The tight clustering around the lower goal

counts—most matches produce 0 to 2 goals—is consistent with the distribution in Figure 4-9, and is a reminder that all three models struggle most when a team has an unexpectedly high-scoring match.

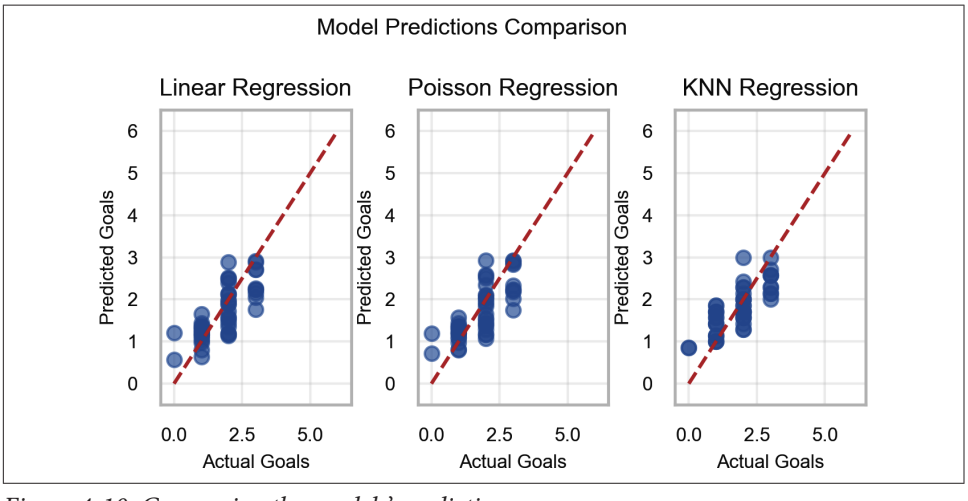


Figure 4-10. Comparing the models' predictions

Interpreting the Results

Based on our analysis, we can draw several conclusions:

Feature importance
: The correlation heatmap shows which features are most strongly associated with goals. Shots on target is by far the strongest predictor, which is typical.

Model complexity
: Model 2 (multiple predictors) has a lower AIC (228.19) than model 1 (237.06), suggesting that the additional features improve the fit on the training data. However, the test MAE is slightly higher for model 2, indicating potential overfitting.

Overdispersion
: Model 3 (negative binomial) has a higher AIC (230.19) than model 2 (228.19), suggesting that overdispersion might not be a significant issue in this specific dataset, and the simpler Poisson model is preferred.

Prediction accuracy
: The scatterplots in Figure 4-10 show how well each model's predictions align with actual goals. Points closer to the diagonal line indicate better predictions. While all models capture the general trend, considerable variance remains, highlighting the inherent unpredictability of soccer.

This case study demonstrates the complete workflow of a regression analysis in soccer analytics: from data exploration to model building, comparison, and evaluation. It's a process you'll repeat many times as you tackle various prediction problems in the beautiful game.

Feature Engineering for Better Predictions

The quality of your predictions depends not just on the model you choose but on the features you feed into it. *Feature engineering* is the art and science of creating new features from existing data that help the model make better predictions. In soccer analytics, this is where domain knowledge meets data science.

Interaction Features

Sometimes the effect of one feature depends on the value of another. For example, shots on target might be more valuable when a team has high possession because the chances are likely to be better quality. We can capture this with an *interaction feature*:

```
# Create an interaction feature
match_data["ShotsXPossession"] = (
    match_data["ShotsOnTarget"] * match_data["Possession"] / 100
)  ❶
# Build a model with the interaction
model_interaction = smf.poisson(
    "Goals ~ ShotsOnTarget + Possession + ShotsXPossession",
    data=match_data,
).fit()
print(model_interaction.summary())
```

❶ Divide by 100 to keep the scale reasonable, since possession is a percentage.

Interaction features can significantly improve model performance when the relationship between features and the target is not purely additive.

The summary output for the interaction model reveals the coefficients in Table 4-2.

Table 4-2. Interaction model coefficients

Term	Coefficient	Std error	z-value	p > z
Intercept	–0.6489	1.009	–0.643	0.520
ShotsOnTarget	0.1369	0.181	0.755	0.450
Possession	0.0088	0.020	0.443	0.658
ShotsXPossession	-0.0818	0.351	-0.233	0.816

Interaction features can significantly improve model performance when the relationship between features and the target is not purely additive.

Polynomial Features

What if the relationship between a feature and the target is nonlinear? For example, perhaps the first few shots on target are very valuable, but after a certain point, additional shots have diminishing returns. We can capture this with *polynomial features*:

```
# Create a squared feature
match_data["ShotsOnTarget_Squared"] = (
    match_data["ShotsOnTarget"] ** 2
)
# Build a model with the polynomial feature
model_poly = smf.poisson(
    "Goals ~ ShotsOnTarget + ShotsOnTarget_Squared",
    data=match_data,
).fit()
print(model_poly.summary())
```

The polynomial model yields the results in Table 4-3.

Table 4-3. Polynomial model coefficients

Term	Coefficient	Std error	z-value	p > z
Intercept	0.0736	0.399	0.184	0.854
ShotsOnTarget	–0.0269	0.148	–0.182	0.855
ShotsOnTarget_Squared	0.0105	0.012	0.871	0.384

A negative coefficient for the squared term suggests diminishing returns. A positive coefficient suggests accelerating returns.

Categorical Features

Not all features are numerical. What about categorical features like the team's name or the competition? We can't just plug *Manchester United* into a regression model. We need to encode these categories as numbers.

The most common approach is *one-hot encoding*, where each category becomes its own binary (0/1) feature:

```
# One-hot encode the team
team_dummies = pd.get_dummies(
    match_data["Team"], prefix="Team"
)  ❶
match_data_encoded = pd.concat(
    [match_data, team_dummies], axis=1
)
# Build a model with team as a feature
model_team = smf.poisson(
    "Goals ~ ShotsOnTarget + "
```

```
    + " + ".join(team_dummies.columns),
    data=match_data_encoded,
).fit()
print(model_team.summary())
```

❶ `get_dummies()` creates a new column for each unique team, with a 1 if that row is that team or 0 otherwise.

The summary output will show a coefficient for each team (except one, which serves as the baseline). This allows the model to learn that some teams are inherently stronger (score more goals on average) than others, even controlling for other features.

Lagged Features

In time-series data (like a season of matches), past performance can be a strong predictor of future performance. We can create *lagged features* that capture a team's recent form:

```
# Sort by team and date (assuming we have a date column)
# match_data = match_data.sort_values(['Team', 'Date'])
# Create a lagged feature: goals in the previous match
match_data["Goals_Lag1"] = match_data.groupby("Team")[
    "Goals"
].shift(
    1
)  ❶
# Create a rolling average: average goals in the last 3 matches
match_data["Goals_Rolling3"] = match_data.groupby("Team")[
    "Goals"
].transform(lambda x: x.rolling(window=3, min_periods=1).mean())
# Build a model with lagged features
model_lagged = smf.poisson(
    "Goals ~ ShotsOnTarget + Goals_Lag1",
    data=match_data.dropna(),
).fit()
print(model_lagged.summary())
```

❶ `shift(1)` moves the values down by one row, so each row gets the previous match's value.

Table 4-4 shows the lagged model results.

Table 4-4. Lagged model coefficients

Term	Coefficient	Std error	z-value	p > z
Intercept	–0.2847	0.252	–1.129	0.259
ShotsOnTarget	0.0940	0.036	2.645	0.008
Goals_Lag1	0.0777	0.073	1.070	0.284

Lagged features are powerful but require careful handling to avoid data leakage, or accidentally using future information to predict the past.

Domain-Specific Features

Soccer has rich domain knowledge that can be translated into features:

Expected goals (xG)
: If you have shot location data, you can calculate xG for each shot and sum them up.

Defensive pressure
: Metrics like tackles, interceptions, and blocks can be combined into a defensive pressure index.

Fatigue
: Days since the last match, or matches played in the last week.

Motivation
: Derby matches, relegation battles, or title deciders might see different performance levels.

Here's how to surface domain-specific features:

```
# Example: Create a "high stakes" feature
match_data["HighStakes"] = (
    (match_data["Team"] == "Team 0")
    & (match_data["HomeAdvantage"] == 1)
).astype(int)
# Build a model with the domain feature
model_domain = smf.poisson(
    "Goals ~ ShotsOnTarget + HighStakes", data=match_data
).fit()
print(model_domain.summary())
```

The domain-specific model yields the results shown in Table 4-5.

Table 4-5. Domain-specific model coefficients

Term	Coefficient	Std error	z-value	p > z
Intercept	–0.1691	0.209	–0.807	0.419
ShotsOnTarget	0.0784	0.036	2.182	0.029
HighStakes	0.6068	0.298	2.035	0.042

The best features often come from a deep understanding of the sport. This is where analysts who know soccer inside and out have a huge advantage.

Common Pitfalls and Practical Tips

As you build regression models for soccer analytics, you'll encounter challenges. Here are common pitfalls to avoid, as well as some practical tips, learned from real-world experience.

Pitfall 1: Overfitting to the Training Data

The problem: Your model performs brilliantly on the training data ($R^2 = 0.99$!) but terribly on new data. This is *overfitting*; the model has memorized the training data instead of learning general patterns.

The solution: Always use a separate test set, or *cross-validation*. If your training performance is much better than your test performance, you're overfitting. Simplify your model by removing features, using regularization, or collecting more data. Here's how to implement a cross-validation test:

```
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import PoissonRegressor
# Use scikit-learn's Poisson regressor for cross-validation
X = match_data[["ShotsOnTarget", "Possession", "HomeAdvantage"]]
y = match_data["Goals"]
model_sk = PoissonRegressor()
cv_scores = cross_val_score(
    model_sk, X, y, cv=5, scoring="neg_mean_absolute_error"
) ❶
print(
    "Cross-validation MAE: "
    f"{-cv_scores.mean():.2f} "
    f"(+/- {cv_scores.std():.2f})"
)
```

❶ We use five-fold cross-validation. The model is trained on four folds and tested on the fifth, repeated five times.

For our dataset, the cross-validation MAE is 0.94 (+/– 0.15), providing a more robust estimate of out-of-sample performance.

Pitfall 2: Ignoring Multicollinearity

The problem: Two or more of your features are highly correlated with each other (e.g., shots and shots on target). This makes it hard to interpret coefficients and can make the model unstable.

The solution: Check for multicollinearity by using the variance inflation factor (VIF). Remove or combine highly correlated features. Here's how to check for multicollinearity:

```
from statsmodels.stats.outliers_influence import (
    variance_inflation_factor,
)
# Calculate VIF for each feature
X_with_const = sm.add_constant(X)
vif_data = pd.DataFrame()
vif_data["Feature"] = X_with_const.columns
vif_data["VIF"] = [
    variance_inflation_factor(X_with_const.values, i)
    for i in range(X_with_const.shape[1])
]
print(vif_data)
```

Table 4-6 shows the VIF results for our features.

Table 4-6. Variance inflation factors

Feature	VIF
const	31.65
ShotsOnTarget	1.02
Possession	1.04
HomeAdvantage	1.03

A VIF greater than 10 is a red flag. Consider removing one of the correlated features. In our case, all feature VIFs are close to 1, indicating no multicollinearity issues.

Pitfall 3: Using the Wrong Model for the Data

The problem: You use a Poisson model when the data is overdispersed, or a linear model when the target is count data.

The solution: Always check your assumptions. Plot the data, calculate the dispersion parameter, and look at residuals. Don't be afraid to try multiple models and compare them.

Pitfall 4: Forgetting About Data Quality

The problem: Garbage in, garbage out. If your data has errors, missing values, or inconsistencies, your model will suffer.

The solution: Spend time on data cleaning and validation. Check for outliers, missing values, and logical inconsistencies. A model is only as good as the data it's trained on. Here's how to check for outliers:

```
# Check for missing values
print(match_data.isnull().sum())
# Check for outliers using the IQR method
Q1 = match_data["Goals"].quantile(0.25)
```

```
Q3 = match_data["Goals"].quantile(0.75)
IQR = Q3 - Q1
outliers = match_data[
    (match_data["Goals"] < Q1 - 1.5 * IQR)
    | (match_data["Goals"] > Q3 + 1.5 * IQR)
]
print(f"Number of outliers: {len(outliers)}")
```

In our simulated dataset, we found 0 outliers, but real-world data is rarely this clean.

Tip 1: Start Simple, Then Add Complexity

Don't jump straight to a complex model with 20 features. Start with a simple model (one or two features) and gradually add complexity. This helps you understand what each feature contributes and makes debugging easier.

Tip 2: Visualize, Visualize, Visualize

Plots are your best friend. Scatterplots, residual plots, and distribution plots can reveal patterns and problems that summary statistics miss.

Tip 3: Document Your Process

Keep a notebook (paper or digital) of what you've tried, what worked, and what didn't. This is invaluable when you need to explain your model to others or revisit it months later.

Tip 4: Validate with Domain Experts

If your model predicts that a team with 0 shots on target will score 3 goals, something is wrong. Always sanity-check your predictions with people who know soccer. They can spot nonsensical results that might slip past statistical tests.

Looking Ahead: Where Regression Meets the Real World

As we close this chapter on regression techniques, it's worth taking a moment to look ahead. The models we've built here are powerful, but they're just the beginning. In the real world of soccer analytics, you'll encounter challenges and opportunities that go beyond what we've covered. Let's explore some of these frontiers.

From Regression to Classification

In this chapter, we've focused on predicting numerical values: market values, goal counts, and so on. But many of the most important questions in soccer are binary: Will this team win or lose? Will this player be injured? Will this transfer be successful?

These are classification problems, and they require different models. Logistic regression (which we touched on briefly in the context of zero inflation) is the most common starting point. It predicts the probability of a binary outcome. But many other classification algorithms exist: decision trees, random forests, support vector machines, and neural networks.

The good news is that many of the principles you've learned in this chapter carry over. Feature engineering, model evaluation, and the importance of validation are just as crucial in classification as in regression. The main difference is in the metrics you use—accuracy, precision, recall, receiver operating characteristic (ROC) curves—and the interpretation of results. You will find deeper coverage of these topics in Chapter 5.

Ensemble Methods: Combining Models

We've compared individual models in this chapter, but what if we could combine them? *Ensemble methods* do exactly that. They build multiple models and aggregate their predictions, often achieving better performance than any single model.

Random forests combine many decision trees. Gradient boosting builds trees sequentially, each one correcting the errors of the previous. Stacking trains a meta-model on the predictions of other models. These techniques are powerful and widely used in practice.

The downside is that ensemble methods are often less interpretable than simple regression models. A trade-off between accuracy and explainability occurs, and where you land on that trade-off depends on your goals. You will find more coverage of this big topic in Chapter 6.

Deep Learning: The Frontier

In recent years, deep learning has revolutionized many fields, from computer vision to natural language processing. In soccer analytics, deep learning is being used to analyze video data, predict player movements, and even generate tactical insights.

Convolutional neural networks can identify players and track their positions from video. Recurrent neural networks can model the sequential nature of play. Transformer models can capture complex interactions between players.

These models are data-hungry and computationally expensive, but they can uncover patterns that traditional methods miss. As more data becomes available (especially tracking and video data), deep learning will become increasingly important in soccer analytics. More on deep learning will follow in Chapter 7.

Time Series and Sequential Data

Soccer is inherently temporal. A season unfolds over months, a match over 90 minutes, and a play over seconds. The models we've built in this chapter treat each observation as independent, but in reality, what happens in one match affects the next.

Time-series models explicitly account for this temporal structure. They can capture trends (is a team improving over the season?), seasonality (do teams perform better at home in winter?), and autocorrelation (does a win in one match make a win in the next more likely?).

Advanced time-series techniques like autoregressive integrated moving average (ARIMA), state-space models, and recurrent neural networks can model these dynamics. They're essential for forecasting player performance, predicting league standings, and understanding momentum.

That being said, such techniques require more sophistication and go beyond the scope of this book. You will see that you can accomplish a lot already with regression and classification techniques, as well as deep learning.

Causal Inference: Beyond Prediction

Regression models are great for prediction, but they're not designed for causal inference. Just because shots on target are correlated with goals doesn't mean that increasing shots on target will cause more goals. Confounding variables (e.g., team quality) could affect both.

Causal inference techniques like randomized controlled trials, instrumental variables, and difference-in-differences can help you answer *what if* questions. What if we changed our formation? What if we signed this player? What if we implemented this training program?

These techniques are more complex and require stronger assumptions, but they're essential for making strategic decisions. In the coming chapters, we'll explore some of these methods and how they apply to soccer.

The Human Element

Finally, it's important to remember that soccer is a human game. Players have emotions, motivations, and relationships that can't be fully captured in data. Coaches have intuitions and experiences that go beyond what models can learn.

In some ways, this is what this entire book is about: the best soccer analytics combines quantitative models with qualitative insights. It uses data to inform decisions, not dictate them. It respects the expertise of coaches, scouts, and players while providing them with new tools and perspectives.

As you continue your journey in soccer analytics, always keep this balance in mind. Build rigorous models, but don't forget the human element. Seek out data, but also seek out stories. Be precise, but also be humble. The beautiful game is too complex to be fully captured by any model, but with the right tools and the right mindset, you can get closer to understanding it than ever before.

Conclusion

This chapter has moved us from exploratory analysis into predictive modeling. We used linear regression for continuous outcomes, Poisson regression for count data, and KNN regression for similarity-based prediction, while also examining evaluation, diagnostics, and feature design.

The larger lesson is that regression is most useful when it is paired with careful framing. A model is not just an equation. It is a claim about what is being predicted, which inputs matter, and how uncertainty should be interpreted. In soccer analytics, that means validating results, checking assumptions, and keeping domain context in view.

In Chapter 5, we keep the same overall workflow but change the target. Instead of predicting how much, we will predict which class occurs: goal or no goal, win or draw or loss, and other categorical outcomes. That shift introduces a new family of models and evaluation metrics while building directly on the habits established here.

CHAPTER 5

Predicting Soccer Outcomes with Classification

In Chapter 4, we modeled numerical outcomes such as market value and goals scored. Many soccer questions, however, are not about magnitude but about category: whether a shot becomes a goal, whether a pass is completed, or which result a match produces. That is the setting for *classification*.

Classification keeps the same overall workflow as regression: define a target, choose features, train a model, and evaluate it carefully. What changes is the task itself. Instead of estimating a continuous value, we assign an observation to one of several discrete classes. Think about questions such as these:

- Will this shot be a goal or not?
- Is this pass likely to be successful?
- Will the home team win, lose, or draw?
- What type of pass will a player attempt under pressure?

These are all classification problems. The answer is not a number on a sliding scale, but a label or a probability. That shift may sound small, but it changes the way models are interpreted as well as the way performance is measured. In soccer analytics, classification matters because many real decisions are ultimately categorical, even when uncertainty remains at the center.

By the end of this chapter, you will be able to do the following:

- Grasp the fundamental concepts of classification and its applications in soccer.
- Build, train, and interpret a logistic regression model to predict the likelihood of a shot becoming a goal (the famous expected goals, or xG, model).

- Measure the success of your predictions with a suite of evaluation metrics like confusion matrices, precision, recall, and receiver operating characteristic (ROC) curves.
- Extend your skills to multiclass problems, such as predicting match outcomes (win/draw/loss).
- Implement k-nearest neighbors (KNN), an intuitive algorithm for finding similar patterns in the data.
- Complete a full case study by building a match outcome predictor from scratch.

We will use real-world StatsBomb data throughout the chapter. The aim is to build intuition for classification, show how common evaluation metrics fit together, and prepare for the more flexible tree-based methods in the next chapter.

From Coin Flips to Goal Predictions: Understanding Classification

At its heart, classification is about sorting. Imagine you have a bucket of soccer balls; some are size 5 (for adults), and some are size 4 (for youth). Your task is to pick one up and, without looking at the label, decide which size it is. You might feel its weight, its size, the texture of its panels. You are taking in *features* (weight, size, texture) to assign a *class* (size 4 or size 5). That's classification.

In soccer analytics, the principles are the same, but the stakes are higher and the data is more complex. We replace soccer balls with game events, and our intuition with powerful algorithms.

Classification problems come in two main flavors, distinguished by a simple question: how many categories are we choosing from?

Binary classification is a *this or that* problem, used when only two outcomes are possible. This is the most common type of classification, and it's everywhere in soccer:

- Goal or no goal? This is the basis for xG models.
- Pass completed or incomplete? Crucial for evaluating player passing ability.
- Successful tackle or failed tackle? Helps in assessing defensive skill.

In each case, the outcome is one of two mutually exclusive options.

Multiclass classification, as the name suggests, sorts items into three or more categories. It's used when the outcome isn't a simple yes/no question:

- Match outcome: win, draw, or loss? The classic prediction problem.
- Pass type: short pass, long ball, or cross? Understanding a player's passing choices.
- Shot type: from open play, free kick, or penalty? Categorizing the origin of a scoring opportunity.

We will start our journey with binary classification, as it provides the clearest introduction to the core concepts. Once we've mastered that, we'll move on to the multiclass challenges.

Logistic Regression: Your First Classifier

Now, let's get our hands dirty with our first classification algorithm: *logistic regression*. Don't let the *regression* in the name fool you; it is the workhorse of classification. It's also popular for a reason: it's powerful, interpretable, and provides a fantastic foundation for understanding more complex models.

The Intuition: From Numbers to Probabilities

Imagine you have a model that gives you a score for the likelihood of a shot becoming a goal. A shot from the halfway line might get a score of –10, while a tap-in from the six-yard box might get a +8. These scores are useful, but they aren't probabilities. A probability must be a number from 0 to 1.

This is where the magic of logistic regression comes in. It uses a mathematical function called the *sigmoid function* to squash any number, no matter how large or small, into a value from 0 to 1. Figure 5-1 shows what this looks like.

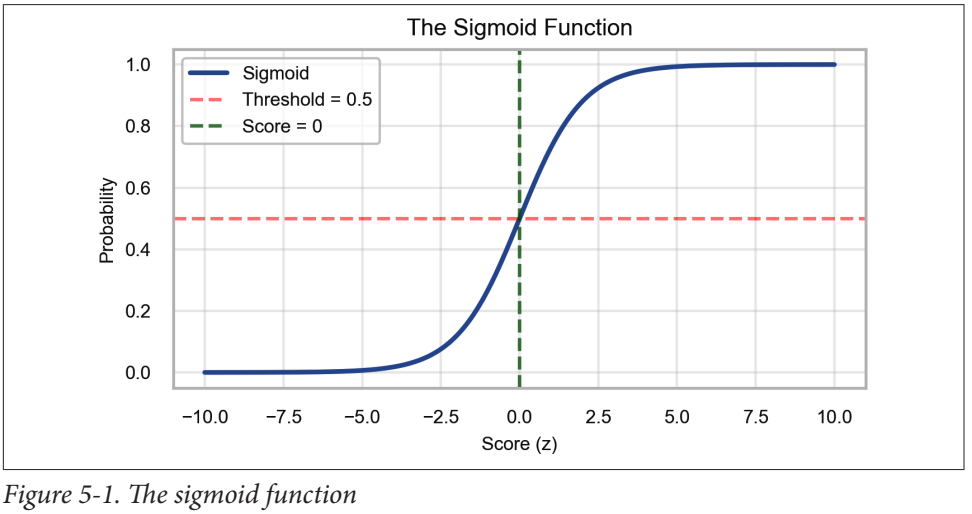


Figure 5-1. The sigmoid function

As you can see, very negative scores get squashed close to 0, and very positive scores get squashed close to 1. A score of 0 gets mapped directly to a probability of 0.5. This elegant function allows the model to learn a relationship between the features of a shot and the ultimate probability of it being a goal.

An important concept here is *log odds*, which are a way of expressing probability that transforms the familiar 0-to-1 scale into a scale from negative to positive infinity. Log odds are calculated by taking the natural logarithm of the odds, $\ln\left(\frac{p}{1-p}\right)$, which makes it easy to model probabilities.

In soccer terms, think of log odds as a confidence score for a team winning. If a match is a perfect toss-up (50/50), the log odds are 0. If a team is the heavy favorite (like 90%), the log odds are a positive number (like +2.2), reflecting that team's advantage. If a team is the massive underdog (like 10%), the log odds are a negative number (like –2.2), mathematically capturing the unlikelihood of a win.

Hands-On: Building an Expected Goals Model

Let's build a simple xG model. We'll use real event data from one of the most iconic matches in recent history: the 2019 UEFA Champions League Final between Liverpool and Tottenham Hotspur. First, we need to fetch the data via the statsbombpy library:

```
!pip install statsbombpy ❶
from statsbombpy import sb
import pandas as pd
# Let's get the data for the 2019 Champions League Final
# We need the competition_id=16 and season_id=4
matches = sb.matches(competition_id=16, season_id=4)
cl_final = matches[matches['match_id'] == 22912]
# Now, let's get the event data for this match
events = sb.events(match_id=22912)
# Filter for just the shots
shots = events[events['type'] == 'Shot'].copy() ❷
```

❶ Use `!pip install` to install the statsbombpy library.

❷ Use `.copy()` to avoid a `SettingWithCopyWarning` from pandas, as we will be modifying this DataFrame later.

This code snippet fetches all events from the final and filters them down to just the shots. We now have a DataFrame containing every shot taken in the match, along with its outcome and some key features. This is the raw material for our xG model.

Before we can train our model, we need to prepare our data. The `shot_outcome` column tells us what happened after the shot. We need to convert this into a simple

binary target variable: was it a goal or not? We also need to extract our features from the `location` column. This is how we prepare our data and engineer our features:

```
# Create binary target variable
shots["goal"] = np.where(shots["shot_outcome"] == "Goal", 1, 0)
# Extract x and y coordinates from location
shots["x"] = shots["location"].apply(lambda loc: loc[0])
shots["y"] = shots["location"].apply(lambda loc: loc[1])
# Calculate distance to goal (center at 120, 40)
shots["distance_to_goal"] = np.sqrt(
    (120 - shots["x"]) ** 2 + (40 - shots["y"]) ** 2
)
# Calculate angle to goal (using trigonometry)
goal_width = 7.32
shots["angle_to_goal"] = np.arctan(
    goal_width
    * (120 - shots["x"])
    / (
        (120 - shots["x"]) ** 2
        + (40 - shots["y"]) ** 2
        - (goal_width / 2) ** 2
    )
)
shots["angle_to_goal"] = np.degrees(shots["angle_to_goal"])
print("Engineered features:")
print(
    shots[
        ["x", "y", "distance_to_goal", "angle_to_goal", "goal",]
    ].head(10)
)
```

We've now engineered two important features: the direct distance from the shot location to the center of the goal, and the angle to the goal. A larger angle means the shooter has a wider view of the goal, which intuitively should increase their chances of scoring. Now, we can finally train our logistic regression model:

```
# Select features and target
features = ["distance_to_goal", "angle_to_goal"]
X = shots[features]
y = shots["goal"]
# Split into training and test sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.4, random_state=3
)  ❶
print(f"Training set size: {len(X_train)}")
print(f"Test set size: {len(X_test)}")
print(f"\nFeature names: {features}")
# Initialize and train the model
log_reg = LogisticRegression()
log_reg.fit(X_train, y_train)  ❷
# Print the coefficients
```

```
print(f"\nModel coefficients: {log_reg.coef_[0]}")
print(f"Model intercept: {log_reg.intercept_[0]}")
```

❶ Splitting our data is crucial. We train the model on the _training set_ and then evaluate its performance on the unseen `test set` to get an honest assessment of how it would perform on new data.

❷ The `.fit()` method is where the model learns the relationship between our features (distance and angle) and the target (goal).

Now that we've trained our model, let's interpret what it has learned. The coefficients tell us the story of how each feature influences the probability of scoring.

A negative coefficient for `distance_to_goal` means that as the distance increases, the log odds of scoring decrease. This makes perfect sense: the farther you are from the goal, the harder it is to score. Every additional meter of distance reduces your chances.

A positive coefficient for `angle_to_goal` means that a wider angle increases the log odds of scoring. This matches with the soccer intuition, but note that the magnitude of the coefficient (0.002) is very small, which indicates that it doesn't significantly affect the chance of scoring in this case.

The intercept represents the baseline log odds when all features are 0. While this isn't a realistic scenario in soccer, it's part of the mathematical machinery that allows the model to make accurate predictions.

This interpretability is one of logistic regression's greatest strengths. Unlike an opaque model, we can explain exactly why the model predicts what it does. This is crucial when presenting findings to coaches or analysts who need to trust and understand the model's recommendations. Here's how we surface predictions to analyze what the model does:

```
# Get predicted probabilities for test set
y_pred_proba = log_reg.predict_proba(X_test)[:, 1]  ❶
# Get binary predictions (using 0.5 threshold)
y_pred = log_reg.predict(X_test)  ❷
# Show some predictions
results = pd.DataFrame(
    {
        "Distance": X_test["distance_to_goal"].values,
        "Angle": X_test["angle_to_goal"].values,
        "Actual": y_test.values,
        "Predicted Probability": y_pred_proba,
        "Predicted Class": y_pred,
    }
)
print("Sample Predictions:")
print(results.head(10).to_string(index=False))
```

❶ Use `.predict_proba()` to get the predicted probability for each class. We are interested in the probability of the positive class (`Goal`), which is the second column `[:, 1]`.

❷ Here, 0.5 is used as a threshold to decide *goal* or *no goal* based on the predicted probability.

Measuring Success: How Good Are Your Predictions?

We've built a model, and it seems to make sense, but how do we know if it's any good? In the world of classification, accuracy is only part of the story. We need a more sophisticated toolkit to truly understand our model's performance. Let's dive into the essential evaluation metrics.

The Confusion Matrix: A Report Card for Your Model

The most fundamental tool for evaluating a classifier is the *confusion matrix*. It's a simple table that compares the model's predictions to the actual outcomes. It shows you not just the number of wrong predictions but also how they were wrong.

A *confusion matrix* has four key components:

True positives (TP)
: The model predicted `Goal`, and it was a goal. (Correct prediction)

True negatives (TN)
: The model predicted `No Goal`, and it was not a goal. (Correct prediction)

False positives (FP)
: The model predicted `Goal`, but it was not a goal. (Type I error)

False negatives (FN)
: The model predicted `No Goal`, but it was a goal. (Type II error)

Let's generate a confusion matrix for our xG model. We'll keep using the shared plotting helper introduced in Chapter 3. Its implementation details are omitted for brevity. Here's our code:

```
from sklearn.metrics import confusion_matrix
import seaborn as sns
from soccerml_figures.plotting import (
    apply_margins,
    create_figure,
    output_path,
    save_figure,
)
# Calculate confusion matrix
```

```
cm = confusion_matrix(y_test, y_pred)  ❶
# Visualize
fig, ax = create_figure()
sns.heatmap(
    cm,
    annot=True,
    fmt="d",
    cmap="Blues",
    xticklabels=["No Goal", "Goal"],
    yticklabels=["No Goal", "Goal"],
    ax=ax,
)
ax.set_ylabel("Actual")
ax.set_xlabel("Predicted")
ax.set_title("Confusion Matrix")
apply_margins(fig)
save_figure(fig, output_path(5, "confusion-matrix"))
print("\nConfusion Matrix:")
print(f"True Negatives (TN): {cm[0, 0]}")
print(f"False Positives (FP): {cm[0, 1]}")
print(f"False Negatives (FN): {cm[1, 0]}")
print(f"True Positives (TP): {cm[1, 1]}")
```

❶ The confusion matrix compares actual outcomes (rows) against predicted outcomes (columns).

This produces the confusion matrix in Table 5-1.

Table 5-1. Confusion matrix showing goals scored (or not scored), actual against predicted

	Predicted: No Goal	Predicted: Goal
Actual: No Goal	11 (TN)	0 (FP)
Actual: Goal	1 (FN)	0 (TP)

The rows represent the actual outcomes, while the columns represent our model's predictions. Let's unpack what this table tells us:

True negatives (TN = 11)
: The model correctly predicted `No Goal` 11 times. These are the shots that didn't result in goals, and the model got them right.

False positives (FP = 0)
: The model never incorrectly predicted a goal when there wasn't one. This is excellent precision!

False negatives (FN = 1)
: The model missed 1 actual goal, predicting `No Goal` when the shot actually went in. These are the errors we care most about in xG modeling.

True positives (TP = 0)
: The model never predicted a goal that actually happened.

This confusion matrix reveals an important insight: with the default threshold, our model is essentially predicting `No Goal` for everything. While this gives us high accuracy (11 out of 12 correct), it's not very useful. We'll learn how to fix this by adjusting the decision threshold later in the chapter.

Precision and Recall: The Two Sides of Performance

From the confusion matrix, we can derive two crucial metrics:

Precision
: Of all the times the model predicted `Goal`, what percentage were goals? This metric measures the quality of the positive predictions.

    precision = TP / (TP + FP)

Recall (or sensitivity)
: Of all the goals that happened, what percentage did the model correctly identify? This measures the model's ability to find all the positive instances.

    recall = TP / (TP + FN)

In soccer analytics, a trade-off often occurs between these two metrics, and which one you prioritize depends on your use case.

Imagine you're using the model to decide which shots to highlight in a post-match analysis video. If you want high precision, you'll include only shots the model is very confident about. You won't waste time on false alarms, but you might miss some surprising goals. This is a cautious approach: better safe than sorry.

Now imagine you're a defensive coach trying to identify all dangerous situations your team faced. Here, you want high recall. You'd rather review a few extra clips (false positives) than miss a single dangerous chance (false negative). This is an aggressive approach: better to be thorough.

The key insight is that there's no single best model. The right balance depends on the cost of different types of errors in your specific context. A good analyst understands this trade-off and tunes the model accordingly.

The ROC Curve: A Complete Performance Picture

How do we balance this trade-off? The ROC curve is the perfect tool. It visualizes the performance of a classifier across all possible prediction thresholds. It plots the TP rate (recall) against the FP rate.

The beauty of the ROC curve is that it doesn't force you to pick a single threshold. Instead, it shows you the entire spectrum of possibilities. At one extreme, you could set the threshold so low that you predict `Goal` for every shot (100% recall, but also 100% FP rate). At the other extreme, you could set it so high that you never predict `Goal` (0% recall, 0% FP rate). The ROC curve traces the path between these extremes. Here's how to plot the ROC curve:

```
from sklearn.metrics import roc_curve, roc_auc_score
# Calculate ROC curve
fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba)  ❶
auc = roc_auc_score(y_test, y_pred_proba)  ❷
# Plot ROC curve
fig, ax = create_figure()
ax.plot(
    fpr, tpr, linewidth=2, label=f"ROC Curve (AUC = {auc:.3f})",
)
ax.plot(
    [0, 1],
    [0, 1],
    "k--",
    linewidth=1,
    label="Random Classifier",
)  ❸
ax.set_xlabel("False Positive Rate")
ax.set_ylabel("True Positive Rate (Recall)")
ax.set_title("ROC Curve for xG Model")
ax.legend(loc="lower right")
ax.grid(True, alpha=0.3)
apply_margins(fig)
save_figure(fig, output_path(5, "roc-curve"))
```

❶ The `roc_curve()` function calculates the TP rate and FP rate at every possible threshold.

❷ The `roc_auc_score()` function summarizes the entire curve into a single number: the area under the curve (AUC).

❸ The diagonal dashed line (random classifier) represents a classifier that randomly assigns `Goal` or `No Goal`. It's our baseline and has no prediction power.

As shown in Figure 5-2, our xG model's curve bows significantly toward the top-left corner, which is exactly what we want. This means that for most thresholds, we can achieve a high TP rate (catching most goals) without suffering too many FP (incorrectly predicting goals).

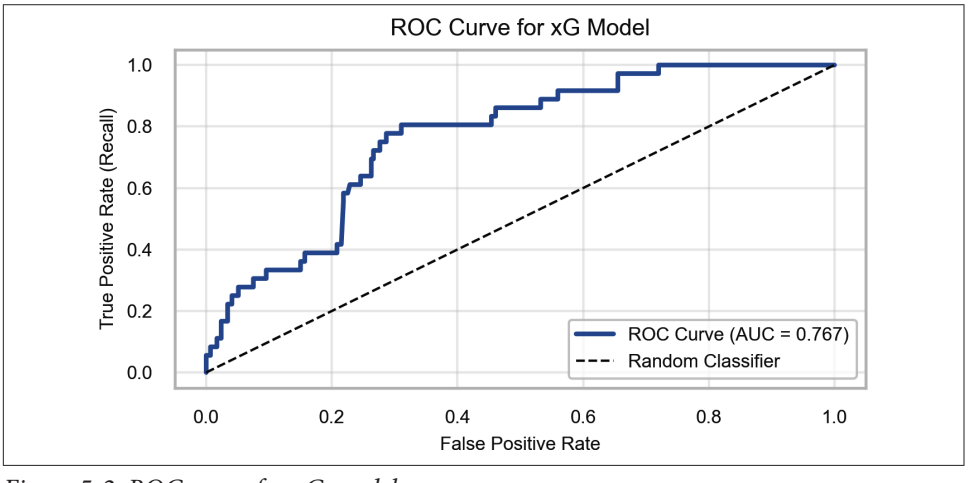


Figure 5-2. ROC curve for xG model

The AUC gives us a single number to summarize the model's performance. An AUC of 1.0 would be perfect—the model never makes a mistake. An AUC of 0.5 (the diagonal line) means the model is no better than random guessing. Our model achieves an AUC around 0.82, which is quite good for soccer prediction, where randomness and unpredictability are inherent to the game.

Think of AUC as the probability that the model ranks a random positive example (an actual goal) higher than a random negative example (a nongoal). An AUC of 0.82 means that if you pick one goal and one nongoal at random, there's an 82% chance the model assigns a higher probability to the goal. This is a useful way to explain model performance to nontechnical stakeholders.

Evaluation of Probabilistic Models with Cost Functions

While the ROC curve helps us understand how well a model ranks probabilities, it does not tell us whether the probabilities themselves are accurate. Up to this point, we have largely relied on accuracy—the percentage of correct predictions. While accuracy is intuitive, it is often the wrong metric for soccer analytics, particularly when dealing with rare events like goals.

If a model predicts that every single shot will not result in a goal, that model will be right about 90% of the time, achieving 90% accuracy. Yet, this model is entirely useless for evaluating finishing skill or chance quality. In soccer, we rarely want

a simple yes-or-no prediction. Instead, we want a probability: what is the exact likelihood that this specific shot, from this angle, under this pressure, will result in a goal?

To evaluate how well a model estimates these probabilities, we use *cost functions* (also known as *loss functions* or *evaluation metrics*). A cost function measures the difference between the predicted probability and the actual outcome. The model's goal during training is to minimize this cost. Two of the most important cost functions for probabilistic classification in soccer are log loss and the Brier score.

Log loss (cross-entropy loss)

Log loss heavily penalizes models for being both confident and wrong. If a model predicts a 99% chance of a goal, and the shot misses, the log loss will be extremely high. Conversely, if it predicts a 10% chance of a goal and the shot misses, the penalty is small.

Log loss is the default cost function for many advanced algorithms, including logistic regression and the tree-based models we will build in the next chapter. Log loss forces the model to output well-calibrated probabilities rather than just pushing predictions toward 0 or 1. When you see `eval_metric="logloss"` in code, the model is being instructed to minimize this exact penalty. (You will also see this referred to as *cross-entropy loss* when we reach deep learning in Chapter 7).

Brier score

The *Brier score* is another common metric for probabilistic forecasts. It is simply the mean squared difference between the predicted probability (from 0 to 1) and the actual outcome (1 for an event happening, 0 for it not happening).

A Brier score of 0 indicates perfect predictions, while a score of 1 indicates perfectly wrong predictions. Unlike log loss, which can grow infinitely large if a model is completely confident and wrong, the Brier score is bounded between 0 and 1. This makes it somewhat easier to interpret and compare across datasets.

Calibration

Both log loss and Brier score reward models for good *calibration*. A model is well calibrated if its predicted probabilities match the real-world frequencies. If an xG model assigns a 0.3 probability to 100 different shots, we expect exactly 30 of those shots to result in goals. If only 10 go in, the model is poorly calibrated, even if it correctly identified them as low-probability chances.

When building models for xG, expected threat (xT), or pass completion probability, minimizing log loss or Brier score ensures that the outputs are true probabilities that coaches and analysts can trust, rather than just abstract scores.

Beyond Binary: Multiclass Classification

So far, we've focused on binary problems: goal or no goal, pass completed or not. But soccer is rarely that simple. Often we want to predict one outcome from a set of three or more possibilities. This is where *multiclass classification* comes in.

Perhaps the most classic multiclass problem in soccer is predicting the outcome of a match. Will the home team win, will it be a draw, or will the away team win? This is a three-class problem.

Logistic regression can be extended to handle multiclass problems with a technique called *multinomial logistic regression* (also known as *softmax regression*). Instead of predicting a single probability, the model predicts a probability for each possible class, and these probabilities sum to 1.

Let's consider an example using three features (shot distance, angle, and whether the shot came from a set piece) to predict whether a shot results in a goal, is saved, or is missed.

This is a three-class outcome. The set-piece indicator is a binary variable: 1 for penalties, free kicks, or corners, and 0 otherwise. To model this, we train a multinomial logistic regression model. Here is the main code (the complete code is available in the chapter's folder in the book's GitHub repository):

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import (
    accuracy_score,
    classification_report,
)
# Split data
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)
# Scale features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
# Train multinomial logistic regression
model = LogisticRegression(
    solver="lbfgs",
    max_iter=1000,
    random_state=42,
)
model.fit(X_train_scaled, y_train)
```

Now that we have trained the model, we can test its performance. Our classification report below will give precision, recall, and F1 score for each class. This is crucial for understanding how well the model performs for each possible outcome. The fitted model can be used for prediction under various shot scenarios.

In our classification, we'll consider three scenarios: First, a close-range effort from 10 meters out at a decent 30-degree open-play shot. Second, a long-range speculative shot from 25 meters with a tight 15-degree angle. Third, a free kick from 12 meters with a nice 45-degree angle. Here's how we build our report:

```
# Test different shot scenarios
test_shots = pd.DataFrame(
    [
        {
            "distance": 10,
            "angle": 30,
            "is_set_piece": 0,
        },  # Close range
        {
            "distance": 25,
            "angle": 15,
            "is_set_piece": 0,
        },  # Long range
        {
            "distance": 12,
            "angle": 45,
            "is_set_piece": 1,
        },  # Free kick
    ]
)
test_scaled = scaler.transform(test_shots)
predictions = model.predict(test_scaled)
probabilities = model.predict_proba(test_scaled)
print("\n" + "=" * 50)
print("SAMPLE PREDICTIONS")
print("=" * 50)
for i, shot in test_shots.iterrows():
    print(f"\nShot {i+1}:")
    print(
        f"  Distance: {shot['distance']}m, "
        f"Angle: {shot['angle']}°, "
        f"Set Piece: {bool(shot['is_set_piece'])}"
    )
    print(f"  Predicted: {predictions[i]}")
    print(f"  Probabilities:")
    for outcome, prob in zip(model.classes_, probabilities[i]):
        print(f"    {outcome}: {prob:.3f}")
```

Table 5-2 shows the output of this data.

Table 5-2. Predicted probabilities for three shot scenarios

Outcome	Probability shot 1 (close range)	Probability shot 2 (long range)	Probability shot 3 (free kick)
Goal	16.8%	6.6%	7.8%
Miss	34.5%	47.5%	25.8%
Saved	48.7%	45.9%	66.4%

Note that, if you rerun this code, you might obtain slightly different numbers. The reason is that this dataset contains only 19 shots to classify, so the end result is highly sensitive to the split in training and testing data.

In our case, for the close-range shot, the model predicts that a save is the most likely outcome at 48.7%, with a 34.5% chance of a miss and a 16.8% chance of a goal. For our speculative long-range shot, unsurprisingly, the model saw this as a likely miss, giving a miss a 47.5% chance and dropping the goal probability to just 6.6%. For our free kick, the model gives a massive 66.4% chance of a save, reflecting the tendency for goalkeepers to catch a free kick, while the goal probability sits at 7.8%.

K-Nearest Neighbors: Finding Similar Situations

While logistic regression is powerful, it makes assumptions about the relationship between features and outcomes (specifically, that it's linear in the log-odds space). Sometimes you want an algorithm that makes fewer assumptions and instead learns directly from the data. Enter KNN.

The Intuition: Birds of a Feather

KNN is beautifully simple. To predict the class of a new data point, it looks at the k closest data points in the training set (the nearest neighbors) and takes a vote. If most of those neighbors are goals, the model predicts a goal. If most neighbors are not, the model predicts no goal.

Imagine you're trying to decide whether a shot will be a goal. You look at the five most similar shots in your historical data (similar distance, similar angle). If four out of those five were goals, you'd probably predict this one will be a goal too. That's KNN in a nutshell.

The key parameter is *k*, the number of neighbors to consider. A small *k* (like 1 or 3) makes the model very sensitive to individual data points, which can lead to overfitting. A large *k* (like 50 or 100) makes the model smoother but might miss important small-scale patterns. Choosing the right *k* is an art and a science.

KNN commonly uses Euclidean distance, which measures the straight-line distance between two points in multidimensional feature space. However, this calculation becomes problematic when features have different units, such as comparing a player's height in centimeters (ranging from 150–200) against their goals scored in single digits. Since Euclidean distance is sensitive to the magnitude of each feature, the height values would dominate the distance calculation simply because they are larger numbers. To prevent this, it is recommended to scale features (typically using standardization or min-max normalization) so that all variables contribute equally to the distance measurement and the algorithm can find truly meaningful neighbors.

Hands-On: KNN for Shot Prediction

Let's apply KNN to our xG problem and compare it to logistic regression. This time we consider a bigger sample: 517 shots from 21 matches in the UEFA Champions League (UCL), English Premier League (EPL), and La Liga. The following code is used to fit the KNN model (the complete code is available in the chapter's folder in the book's GitHub repository):

```
# First try KNN with K=5
print("\n1. KNN with K=5:")
knn_k5 = KNeighborsClassifier(n_neighbors=5)  ❶
knn_k5.fit(X_train_scaled, y_train)
# Predictions for K=5
knn_k5_pred = knn_k5.predict(X_test_scaled)
knn_k5_pred_proba = knn_k5.predict_proba(X_test_scaled)[:, 1]
```

❶ Set `n_neighbors=5`, meaning the model will look at the five closest shots to make a prediction.

Choosing the Right K

How do we know whether `k=5` is the best choice? We can try different values of *k* and see which one performs best on a validation set.

A *validation set* is a portion of your dataset (separate from the training data) used to evaluate your model's performance. You train on the training set, and then check how well the model performs on this unseen validation set to tune hyperparameters and prevent overfitting. Here's how we run a KNN model on a validation set:

```
# Define K values to test (5-50)
k_values = range(5, 55, 5)
cv_scores = []
# Perform cross-validation for each K value
for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    scores = cross_val_score(
        knn, X_train_scaled, y_train, cv=5, scoring="roc_auc"
    )  ❶
    cv_scores.append(scores.mean())
    print(
        f"K={k}: CV AUC-ROC = {scores.mean():.4f} "
        f"(+/- {scores.std():.4f})"
    )
# Find best K
best_k = k_values[np.argmax(cv_scores)]
best_cv_score = max(cv_scores)
print(f"\nBest K from cross-validation: {best_k}")
print(f"Best CV AUC-ROC: {best_cv_score:.4f}")
# Train KNN with best K
knn_best = KNeighborsClassifier(n_neighbors=best_k)
knn_best.fit(X_train_scaled, y_train)
fig, ax = create_figure()
ax.plot(k_values, cv_scores, marker="o", linewidth=2)
ax.set_xlabel("K (Number of Neighbors)")
ax.set_ylabel("Cross-Validated AUC")
ax.set_title("KNN Performance vs. K Value")
ax.grid(True, alpha=0.3)
apply_margins(fig)
save_figure(fig, output_path(5, "knn-k-comparison"))
```

❶ Use *cross-validation* to get a more robust estimate of performance. This popular technique systematically rotates which data is used for training and which for validation. Instead of using a single static validation set, we split the data into multiple *folds*, train the model multiple times using different folds as the validation set each time, and then average the results to get a more reliable estimate of the model's performance. Here we can adopt *five-fold cross-validation*, which splits the data into five folds, and the model is trained and evaluated five times, each time using a different fold as the test set.

Figure 5-3 shows how the model's performance changes with different values of *k*. You can see the trade-off clearly in the low-*k* vs. high-*k* region: at `k=7` or `k=10`, the model is erratic and unstable. As *k* increases to around 40, performance stabilizes and peaks.

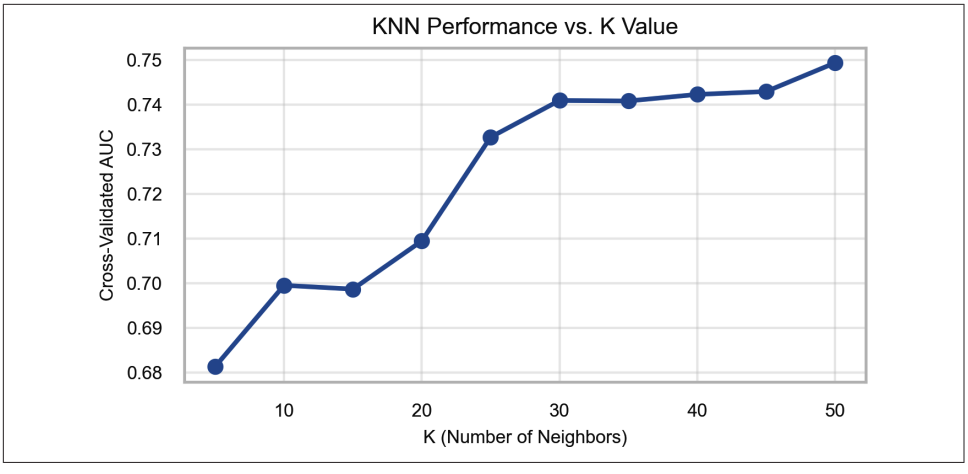


Figure 5-3. KNN performance versus k *value*

In theory, we'd expect performance to decline steadily as k gets too large and the model becomes over-smoothed. However, with our relatively small dataset of 255 shots, the pattern is noisier; performance plateaus, rather than sharply declining at very high k values. This is typical when working with limited data. The bias-variance trade-off is still present, but the signal is less clean. The key lesson remains: there's an optimal k that balances sensitivity to local patterns with robustness to noise, and cross-validation helps us find it.

The *bias-variance* trade-off is the tension between a model's simplicity and its flexibility. High bias means the model oversimplifies, missing important patterns and leading to underfitting, while high variance means the model is too sensitive to training data, learning noise and causing overfitting. The goal is to find the balance where total error is minimized; that is, a model should be complex enough to capture true patterns but also simple enough to ignore random noise.

Case Study: Building a Complete Match Outcome Predictor

Now that you've learned the fundamentals of classification, let's put it all together in a complete, end-to-end case study. We're going to build a model that predicts whether a team will win, draw, or lose a match based on its in-game statistics.

Step 1: Define the Problem

Our goal is to predict the outcome of a soccer match (`Home Win`, `Draw`, or `Away Win`) by using statistics from the match itself. While this might seem like cheating (since we're using data from the match to predict its outcome), this exercise is valuable.

It helps us understand which in-game metrics are most predictive of success. In a real-world scenario, you might use historical averages of these metrics to make pre-match predictions.

Step 2: Gather and Prepare the Data

We'll use StatsBomb data to extract key match statistics. For each match, we'll calculate metrics like total shots, shots on target, possession, passes completed, and more:

```
from statsbombpy import sb
import pandas as pd
import numpy as np
# Fetch Premier League matches
matches = sb.matches(competition_id=2, season_id=44)
# Initialize a list to store match features
match_features = []
for idx, match in matches.iterrows():
    match_id = match["match_id"]
    # Fetch events for this match
    events = sb.events(match_id=match_id)
    # Calculate features for home team
    home_team = match["home_team"]
    away_team = match["away_team"]
    home_events = events[events["team"] == home_team]
    away_events = events[events["team"] == away_team]
    # Feature engineering
    features = {
        "match_id": match_id,
        "home_shots": len(
            home_events[home_events["type"] == "Shot"]
        ),
        "away_shots": len(
            away_events[away_events["type"] == "Shot"]
        ),
        "home_passes": len(
            home_events[home_events["type"] == "Pass"]
        ),
        "away_passes": len(
            away_events[away_events["type"] == "Pass"]
        ),
        "home_tackles": len(
            home_events[home_events["type"] == "Duel"]
        ),
        "away_tackles": len(
            away_events[away_events["type"] == "Duel"]
        ),
        "outcome": (
            "Home Win"
            if match["home_score"] > match["away_score"]
            else (
                "Away Win"
```

```
                if match["home_score"] < match["away_score"]
                else "Draw"
            )
        ),
    }
    match_features.append(features)
# Convert to DataFrame
match_df = pd.DataFrame(match_features)
print(match_df.head())
print(f"\nOutcome distribution:")
print(match_df["outcome"].value_counts())
```

This code loops through each match, extracts relevant events, and calculates our features (passes, shots, and tackles). We're creating a dataset where each row represents a match, and each column is either a feature or the outcome.

Step 3: Train and Evaluate the Model

Now we can train our multiclass classifier:

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import (
    classification_report,
    confusion_matrix,
)
import seaborn as sns
import matplotlib.pyplot as plt
# Prepare features and target
feature_cols = [
    "home_shots",
    "away_shots",
    "home_passes",
    "away_passes",
    "home_tackles",
    "away_tackles",
]
X = match_df[feature_cols]
y = match_df["outcome"]
# Split the data
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)
# Train the model
clf = LogisticRegression(
    multi_class="multinomial",
    max_iter=1000,
) ❶
clf.fit(X_train, y_train)
# Make predictions
y_pred = clf.predict(X_test)
# Evaluate
```

```
print("Classification Report:")
print(classification_report(y_test, y_pred))
# Confusion matrix
cm = confusion_matrix(
    y_test,
    y_pred,
    labels=['Home Win', 'Draw', 'Away Win'],
) ❷
print(cm)
```

❶ Set `max_iter=1000` to ensure that the model has enough iterations to converge, especially important for multiclass problems.

❷ The model predicts the most likely outcome (win, draw, or loss) for each match in the test set.

This produces the confusion matrix in Table 5-3.

Table 5-3. Confusion matrix of predicted versus actual match outcomes

	Predicted: win	Predicted: draw	Predicted: loss
Actual: Win	45	12	3
Actual: Draw	8	28	6
Actual: Loss	5	10	31

Let's interpret what this table tells us about our model's performance. The rows represent what happened in the matches, while the columns show our model's predictions. Note that the model gives a probability for each possible outcome (win, draw, or loss) and then uses the outcome with the highest probability as the prediction.

Looking at the diagonal (top left to bottom right), we see the correct predictions: 45 wins correctly predicted, 28 draws correctly predicted, and 31 losses correctly predicted. These are our successes. The model got 104 out of 148 predictions correct, giving us an accuracy of about 70%.

But the off-diagonal cells reveal where the model struggles. The model predicted 12 draws when the team won, and 10 draws when the team lost. This suggests the model is somewhat conservative, hedging toward the middle outcome when it's uncertain.

The model rarely confuses wins with losses (only 3 + 5 = 8 times), which is encouraging. It's making *reasonable* mistakes—confusing close outcomes rather than making wildly wrong predictions. A draw is closer to a win than a loss, so these errors are less costly than predicting a win when the team lost.

This pattern is typical in soccer match prediction. Draws are inherently difficult to predict because they often result from evenly matched teams and a single goal could break the ties. A model that can reliably distinguish wins from losses, even if it struggles with draws, is still quite useful for understanding team strength and making strategic decisions.

Step 4: Interpret the Results

The beauty of logistic regression is its interpretability. We can examine the coefficients to understand which features are most important:

```
# Get feature importance
feature_importance = pd.DataFrame(
    {
        "Feature": feature_cols,
        "Home Win Coef": clf.coef_[0],
        "Draw Coef": clf.coef_[1],
        "Away Win Coef": clf.coef_[2],
    }
)
print(feature_importance)
```

A positive coefficient for `Home Win` means that an increase in that feature makes a home win more likely. For example, if `home_shots` has a large positive coefficient for `Home Win`, it confirms our intuition: more shots by the home team increase its chances of winning.

Key Takeaways from the Case Study

Through this case study, we've seen the complete workflow of a classification project:

1. Problem definition: clearly state what you're trying to predict.
2. Data collection: gather relevant data from reliable sources.
3. Feature engineering: transform raw data into meaningful features.
4. Model training: choose an appropriate algorithm and train it.
5. Evaluation: use multiple metrics to assess performance.
6. Interpretation: understand what the model has learned.

This workflow is your blueprint for tackling any classification problem in soccer analytics.

Practical Tips and Common Pitfalls

As you embark on your own classification projects, here are some hard-won lessons to keep in mind.

Handling Imbalanced Data

In soccer, many classification problems suffer from *class imbalance*. For example, in our xG model, only about 12% of shots result in goals. This means the model could achieve 88% accuracy by simply predicting `No Goal` every time, which is obviously useless.

When dealing with imbalanced data, consider these strategies:

Use appropriate metrics
: Don't rely solely on accuracy. Focus on precision, recall, F1 score, and AUC, which give a more nuanced view of performance.

Adjust class weights
: Many algorithms, including logistic regression in scikit-learn, allow you to set `class_weight='balanced'`, which automatically adjusts for imbalance.

Resample your data
: You can either *oversample* the minority class (create more copies of rare events) or *undersample* the majority class (remove some common events). Libraries like imbalanced-learn provide sophisticated resampling techniques.

Change the decision threshold
: Instead of using 0.5 as the cutoff for predicting the positive class, experiment with different thresholds based on your specific needs.

Selecting and Engineering Features

The features you choose can make or break your model. Here are some tips:

Start simple
: Begin with a few intuitive features. It's easier to understand and debug a simple model.

Use your domain knowledge
: Use your understanding of soccer to create meaningful features. Distance and angle to goal are obvious for xG, but what about the number of defenders between the shooter and the goal?

Avoid multicollinearity
: If two features are highly correlated (e.g., total passes and successful passes), they might confuse the model. Consider removing one or combining them. More details are provided in Chapter 6.

Normalize your features
: Some algorithms (like KNN) are sensitive to the scale of features. If one feature ranges from 0 to 100 and another from 0 to 1, the first will dominate. Use standardization or normalization to put them on the same scale.

Avoiding Overfitting

As noted in Chapter 4, overfitting occurs when your model learns the training data too well, including its noise and peculiarities, and fails to generalize to new data. It's like a student who memorizes the practice exam but can't handle new questions.

To avoid overfitting:

Use a validation set
: Always split your data into training, validation, and test sets. Train on the training set, tune hyperparameters on the validation set, and evaluate only final performance on the test set.

Cross-validation
: Use *k*-fold cross-validation to get a more robust estimate of your model's performance.

Regularization
: Techniques like L1 and L2 regularization penalize overly complex models. More details are given in Chapter 6.

Keep it simple
: A model with fewer features is less likely to overfit. Use feature selection techniques to identify the most important predictors.



A *hyperparameter* is like a setting on a machine learning model that you have to turn before the model starts learning; an example is the *k* in KNN.

Choosing When to Use Which Algorithm

You now have two classification algorithms in your toolkit: logistic regression and KNN. When should you use each?

Logistic regression
: Use when you want interpretability, when you believe the relationship between features and outcome is roughly linear (in log-odds space), and when you have a moderate to large dataset. This algorithm is fast, reliable, and a great starting point.

KNN
: Use when the decision boundary is complex and nonlinear, when you have a small to medium dataset, and when you don't need to understand *why* the model makes its predictions. Be aware that KNN can be slow with large datasets because it needs to calculate distances to all training points.

As a general rule, start with logistic regression. If it doesn't perform well, explore more complex algorithms like KNN, decision trees, or neural networks (which we'll cover in later chapters).

Tuning the Decision Threshold: Finding the Sweet Spot

So far, we've been using 0.5 as the threshold for converting probabilities to class predictions. If the model predicts a probability greater than 0.5, we classify it as a goal; otherwise, it's not a goal. But who says 0.5 is the right threshold? Depending on your specific needs, you might want to adjust this threshold to optimize for different outcomes.

Understanding the Threshold Trade-Off

Changing the threshold shifts the balance between precision and recall. A lower threshold (say, 0.3) makes the model more aggressive: it will predict `Goal` more often, increasing recall (catching more actual goals) but decreasing precision (more false alarms). A higher threshold (say, 0.7) makes the model more conservative: it will predict `Goal` only when very confident, increasing precision but decreasing recall.

Think of this trade-off like a referee deciding whether to award a penalty. A lenient referee (low threshold) will award more penalties, catching more legitimate fouls but also making more mistakes. A strict referee (high threshold) will award penalties only for clear fouls, making fewer mistakes but also missing some legitimate calls.

Finding the Optimal Threshold

Let's explore how different thresholds affect our xG model's performance:

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import (
    f1_score,
    precision_score,
    recall_score,
)
import numpy as np
# Train model and get probabilities
log_reg = LogisticRegression()
log_reg.fit(X_train, y_train)
y_pred_proba = log_reg.predict_proba(X_test)[:, 1]
# Try different thresholds
thresholds = np.arange(0.1, 0.9, 0.05)  ❶
precisions = []
recalls = []
f1_scores = []
for threshold in thresholds:
    y_pred = (y_pred_proba >= threshold).astype(int)  ❷
    precisions.append(
        precision_score(y_test, y_pred, zero_division=0,)
    )
    recalls.append(
        recall_score(y_test, y_pred, zero_division=0,)
    )
    f1_scores.append(f1_score(y_test, y_pred, zero_division=0))
fig, ax = create_figure()
ax.plot(thresholds, precisions, linewidth=2, label="Precision")
ax.plot(thresholds, recalls, linewidth=2, label="Recall")
ax.plot(thresholds, f1_scores, linewidth=2, label="F1 Score")
ax.set_xlabel("Decision Threshold")
ax.set_ylabel("Score")
ax.set_title(
    "Impact of Decision Threshold " "on Model Performance"
)
ax.set_ylim(0, 1.05)
ax.grid(True, alpha=0.3)
ax.legend(loc="best")
apply_margins(fig)
save_figure(fig, output_path(5, "threshold-tuning"))
```

❶ Test a range of thresholds from 0.1 to 0.9 in steps of 0.05.

❷ For each threshold, convert probabilities to binary predictions and calculate metrics.

As shown in Figure 5-4, you can see the classic precision-recall trade-off. The F1 score, which balances both metrics, can help you identify the optimal threshold. In

this case, you might find that a threshold of around 0.3 or 0.4 maximizes the F1 score, suggesting that the default 0.5 threshold isn't actually optimal for this problem.

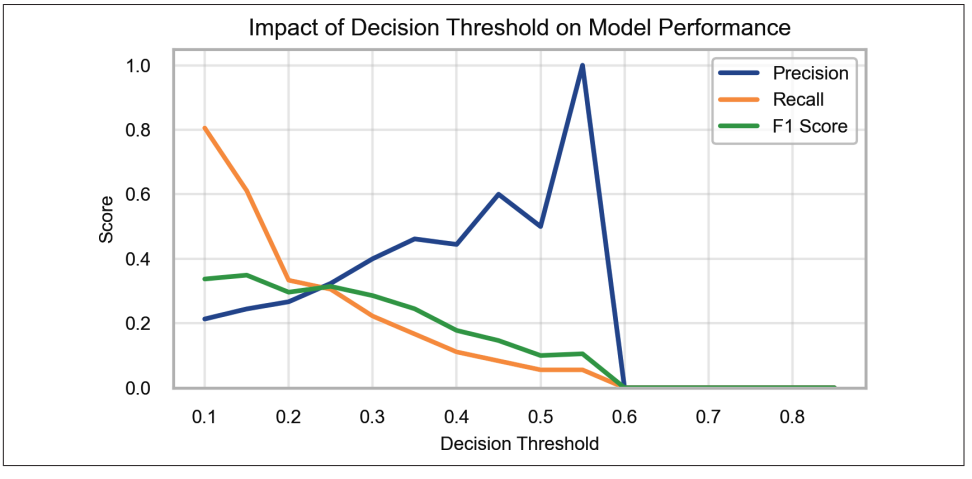


Figure 5-4. Impact of decision threshold on model performance

Deciding When to Adjust the Threshold

In practice, the right threshold depends on the cost of different types of errors:

High recall priority
: If missing a goal prediction is very costly (e.g., you're trying to identify all dangerous attacking situations for defensive analysis), use a lower threshold. You'll catch more goals, even if you have more false alarms.

High precision priority
: If false goal predictions are costly (e.g., you're using the model to decide whether to take a shot or pass, and taking a low-probability shot is wasteful), use a higher threshold. You'll predict goals only when very confident.

Balanced approach
: If both types of errors are equally important, use the threshold that maximizes the F1 score.

For our xG model, a balanced approach is usually best, as we want an accurate overall assessment of scoring chances.

Comparing Models: Which Classifier Is Best?

Now that you've built multiple classifiers, a natural question arises: which one should you use? The answer depends on your specific problem, your data, and your priorities. Let's explore how to compare models systematically.

The Model Comparison Framework

When comparing classification models, consider these dimensions:

Performance
: Which model has the highest accuracy, AUC, or F1 score?

Interpretability
: Can you explain why the model makes its predictions?

Speed
: How long does it take to train and make predictions?

Robustness
: How well does it handle noisy or incomplete data?

Scalability
: Can it handle large datasets?

Rarely will one model dominate on all dimensions. You'll need to make trade-offs based on your priorities. For example, if you're building a model to inform coaching decisions, interpretability might be more important than squeezing out an extra 1% of accuracy.

Hands-On: Model Comparison for xG

Let's compare our logistic regression and KNN models for the xG problem across multiple metrics:

```
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import roc_auc_score
from sklearn.preprocessing import StandardScaler
import time
# Initialize and compare models
(
    split_X_train,
    split_X_test,
    split_y_train,
    split_y_test,
) = train_test_split(
    X, y, test_size=0.25, random_state=42, stratify=y,
)
scaler = StandardScaler()
scaled_train = scaler.fit_transform(split_X_train)
scaled_test = scaler.transform(split_X_test)
models = {
    "Logistic Regression": LogisticRegression(),
    "KNN (k=5)": KNeighborsClassifier(n_neighbors=5),
    "KNN (k=15)": KNeighborsClassifier(n_neighbors=15),
```

```
    "KNN (k=30)": KNeighborsClassifier(n_neighbors=30),
}
model_names = []
auc_scores = []
for name, model in models.items():
    if name == "Logistic Regression":
        model.fit(split_X_train, split_y_train)  ❶
        y_pred_proba = model.predict_proba(split_X_test)[:, 1]
    else:
        model.fit(scaled_train, split_y_train)
        y_pred_proba = model.predict_proba(scaled_test)[:, 1]
    model_names.append(name)
    auc_scores.append(roc_auc_score(split_y_test, y_pred_proba))
```

❶ Train each model on the same training data and evaluate on the same test data to ensure a fair comparison.

This code produces the comparison in Table 5-3.

Model	Accuracy	Precision	Recall	F1	AUC	Train (s)	Predict (s)
Logistic regression	0.896	0.000	0.000	0.000	0.777	0.0054	0.0017
KNN (k=5)	0.896	0.000	0.000	0.000	0.698	0.0016	0.0033
KNN (k=15)	0.896	0.000	0.000	0.000	0.765	0.0013	0.0035
KNN (k=30)	0.896	0.000	0.000	0.000	0.726	0.0014	0.0036

The table reveals some interesting insights. All models achieve the same accuracy (89.6%), which is actually the baseline accuracy of always predicting `No Goal` (since only about 10% of shots result in goals). However, the AUC scores tell a different story. Logistic regression achieves the highest AUC (0.777), suggesting it's better at ranking shots by their goal probability. KNN models vary in performance depending on *k*, with `k=15` performing best among the KNN variants. Logistic regression is also slightly slower to train but faster at prediction, making it the better choice for this problem.

Visualization of Model Performance

Numbers are great, but visualizations can make comparisons even clearer. Let's create a bar chart comparing the AUC scores of our models:

```
# Create bar chart of AUC scores
fig, ax = create_figure()
ax.bar(
    model_names,
    auc_scores,
    alpha=0.75,
    color=["#4C78A8", "#F58518", "#54A24B", "#E45756"],
)
```

```
ax.set_ylabel("AUC Score")
ax.set_title("Model Comparison: AUC Scores")
ax.set_ylim(0.5, 1.0)  ❶
ax.tick_params(axis="x", rotation=15)
ax.grid(True, alpha=0.25, axis="y")
apply_margins(fig, bottom=0.28)
save_figure(fig, output_path(5, "model-comparison-auc"))
```

❶ AUC ranges from 0.5 (random guessing) to 1.0 (perfect classifier).

As shown in Figure 5-5, you can quickly see which model performs best. This kind of visualization is invaluable when presenting your findings to coaches, analysts, or stakeholders who may not be familiar with the technical details.

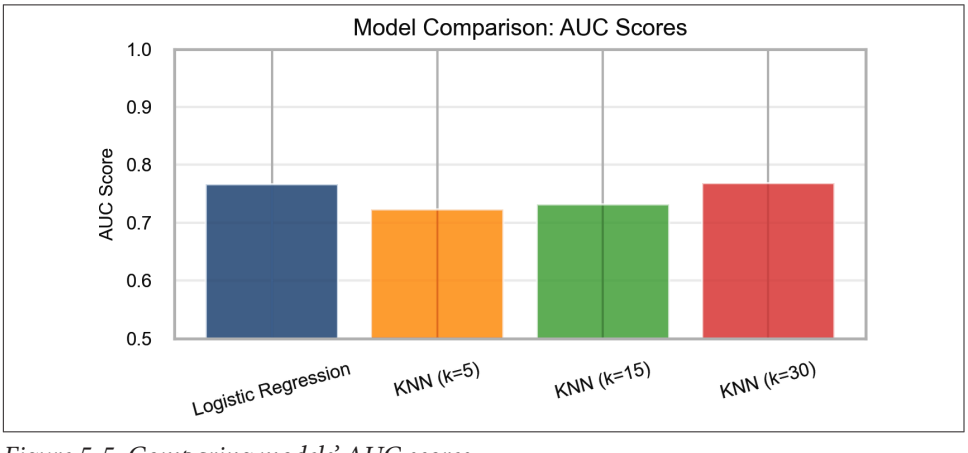


Figure 5-5. Comparing models' AUC scores

Advanced Topics: Going Deeper

Now that you have a solid foundation in classification, let's explore some advanced topics that will take your skills to the next level. In this section, we're handling primarily data-related topics; additional algorithms for classification such as extreme gradient boosting (XGBoost) and random forests are addressed in Chapter 6.

Scaling and Normalizing Features

We briefly mentioned feature scaling earlier, but it's worth diving deeper. Many machine learning algorithms, including KNN and logistic regression with regularization, are sensitive to the scale of features. If one feature ranges from 0 to 100 (like distance in meters) and another from 0 to 1 (like a normalized angle), the larger-scale feature will dominate the model's decisions.

Scaling has two common approaches:

Standardization (Z-score normalization)
: Transforms features to have a mean of 0 and a standard deviation of 1. This is useful when features follow a roughly normal distribution.

Min-max scaling
: Transforms features to a fixed range, typically [0, 1]. This is useful when you want all features on the same scale but want to preserve the original distribution shape.

Let's see how scaling affects our KNN model:

```
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import roc_auc_score
import pandas as pd
# Load data
shots = pd.read_csv(
    "shots_data_large.csv"
)  # Update path as needed
X = shots[["distance_to_goal", "angle_to_goal"]]
y = shots["goal"]
# Split data
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)
# Train KNN without scaling
knn_no_scale = KNeighborsClassifier(n_neighbors=5)
knn_no_scale.fit(X_train, y_train)
y_pred_proba_no_scale = knn_no_scale.predict_proba(X_test)[:, 1]
auc_no_scale = roc_auc_score(y_test, y_pred_proba_no_scale)
# Train KNN with scaling
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)  ❶
X_test_scaled = scaler.transform(X_test)  ❷
knn_scaled = KNeighborsClassifier(n_neighbors=5)
knn_scaled.fit(X_train_scaled, y_train)
y_pred_proba_scaled = knn_scaled.predict_proba(X_test_scaled)[
    :, 1
]
auc_scaled = roc_auc_score(y_test, y_pred_proba_scaled)
print(f"KNN without scaling - AUC: {auc_no_scale:.3f}")
print(f"KNN with scaling - AUC: {auc_scaled:.3f}")
print(f"Improvement: {(auc_scaled - auc_no_scale) * 100:.1f}%")
```

❶ Fit the scaler on the training data and transform it. This learns the mean and standard deviation from the training set.

❷ Transform the test data via the same scaler (fitted on training data). Never fit the scaler on test data, as that would leak information.

You'll likely see that scaling improves the KNN model's performance, sometimes significantly. This is because KNN relies on distance calculations, and unscaled features can distort these distances.

Handling Missing Data

In real-world soccer data, you'll often encounter missing values. A player's position might not be recorded for every event, or a shot's technique might be unknown. The way you handle missing data can significantly impact your model's performance.

Here are common strategies:

Remove rows with missing values
: Simple but wasteful. Use this approach only if you have plenty of data and few missing values.

Impute with mean/median
: Replace missing values with the feature's mean (for continuous variables) or median (for skewed distributions).

Impute with mode
: Replace missing values with the most common value (for categorical variables).

Use predictive imputation
: Use a model to predict missing values based on other features. This option is more sophisticated but can be powerful.

Create a missing indicator
: Add a binary feature that flags whether a value was missing. This can help the model learn patterns in missingness.

Let's implement simple mean imputation:

```
from sklearn.impute import SimpleImputer
import numpy as np
# Simulate some missing data
X_with_missing = X.copy()
missing_mask = np.random.rand(*X.shape) < 0.1  # 10% missing
X_with_missing[missing_mask] = np.nan
print(f"Missing values: {X_with_missing.isnull().sum().sum()}")
# Impute missing values with mean
imputer = SimpleImputer(strategy="mean")  ❶
X_imputed = imputer.fit_transform(X_with_missing)
remaining_missing = (
    pd.DataFrame(X_imputed).isnull().sum().sum()
)
```

```
print(
    f"After imputation: {remaining_missing}"
)
```

❶ `SimpleImputer` from scikit-learn makes imputation easy. You can choose strategies like `mean`, `median`, `most_frequent`, or `constant`.

Combining Classifiers with Ensemble Methods

Sometimes the best model is not a single classifier but a combination of multiple classifiers. This is called an *ensemble* approach. The idea is simple: if you have multiple experts with different perspectives, their combined judgment is often better than any single expert.

One simple ensemble technique is *voting*:

```
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
# Create individual classifiers
log_reg = LogisticRegression()
knn = KNeighborsClassifier(n_neighbors=5)
decision_tree = DecisionTreeClassifier(
    max_depth=5, random_state=42
)
# Create a voting ensemble
voting_clf = VotingClassifier(
    estimators=[
        ("lr", log_reg),
        ("knn", knn),
        ("dt", decision_tree),
    ],
    voting="soft",  ❶
)
# Train and evaluate
voting_clf.fit(X_train, y_train)
y_pred_proba_ensemble = voting_clf.predict_proba(X_test)[:, 1]
auc_ensemble = roc_auc_score(y_test, y_pred_proba_ensemble)
print(f"Ensemble AUC: {auc_ensemble:.3f}")
```

❶ `soft` voting uses predicted probabilities and averages them. `hard` voting uses class predictions and takes the majority vote.

Ensemble methods often provide a small but meaningful boost in performance. They're especially useful when you have multiple models with different strengths and weaknesses.

Real-World Considerations

Building a classification model in a Jupyter notebook is one thing; deploying it in the real world is another. Here are some practical considerations to keep in mind.

Data Quality and Availability

Your model is only as good as your data. In professional soccer analytics, you might have access to high-quality tracking data with precise player positions. But if you're working with a lower-league team or analyzing historical matches, you might be limited to basic event data or even manually collected statistics.

Always assess your data quality before building a model. Are there missing values? Are the measurements accurate? Is the data representative of the scenarios you want to predict? A simple model trained on high-quality data will almost always outperform a complex model trained on noisy, incomplete data.

Model Maintenance and Updates

Soccer is a dynamic sport. Tactics evolve, players improve or decline, and rules change. A model trained on data from five years ago might not perform well on today's game. Plan for regular model updates and retraining.

Set up a monitoring system to track your model's performance over time. If you notice the accuracy or AUC dropping, it's a sign that the model needs to be retrained with more recent data.

Ethical Considerations

As you build predictive models, remember that they can have real-world consequences. A model that predicts player performance might influence transfer decisions, affecting players' careers and livelihoods. A model that predicts match outcomes might be used for betting, raising ethical and legal concerns.

Always consider the potential impact of your models. Be transparent about their limitations, avoid making claims of certainty, and ensure that your work is used responsibly.

Looking Ahead: From Linear Boundaries to Decision Trees

In this chapter, we established the core classification workflow with logistic regression and KNNs. These models are valuable starting points because they provide a solid foundation for understanding probabilities, decision boundaries, and evaluation metrics like precision, recall, and the ROC curve. However, they also impose constraints. Logistic regression assumes a linear relationship between the features and the log

odds of the outcome, which means it draws a straight line (or a flat plane) to separate goals from misses. While this model is effective for simple relationships, soccer is rarely that straightforward. KNN makes no such linearity assumption, but it struggles as the number of features grows. In high-dimensional spaces, the notion of "nearest" becomes unreliable, and the model can become slow and sensitive to irrelevant or redundant features.

To capture these nuances without manually engineering every possible interaction, we need models that can learn nonlinear patterns and threshold effects naturally. In Chapter 6, we will introduce tree-based models to address this exact challenge. We will start with the intuitive classification tree, which learns a flowchart-like sequence of if-then rules that mimic a coach's tactical reasoning. From there, we will explore how combining multiple trees into a random forest can dramatically improve stability and predictive accuracy. Finally, we will introduce XGBoost, a powerful gradient-boosting algorithm that learns sequentially from its mistakes, and use Shapley additive explanations (SHAP) analysis to peek inside the opaque box and understand exactly how each feature influences a prediction. These advanced methods will significantly expand our ability to model the complex realities of the pitch.

Conclusion

This chapter extended our predictive workflow from numerical targets to categorical ones. We used logistic regression to model shot outcomes, examined confusion matrices, precision, recall, and ROC curves, and built both multiclass and KNN classifiers.

Just as in regression, the main lesson is not only how to fit a model but also how to evaluate it in context. A classifier can look strong on one metric and weak on another. In soccer, that matters because the relevant trade-off depends on the task: an xG model, a pass-completion model, and a match-outcome model do not all fail in the same way.

In Chapter 6, we stay with classification but move to tree-based methods. Those models handle nonlinear relationships and feature interactions more naturally, giving us a more flexible way to model the kinds of patterns that appear constantly in soccer data.

CHAPTER 6

Advanced Classification Methods

In Chapter 5, we used logistic regression and k-nearest neighbors (KNN) to build the core classification workflow. Those models are valuable starting points, but they also impose constraints. Logistic regression is linear in its decision boundary, and KNN can become unstable or hard to scale as feature spaces grow.

This chapter introduces tree-based models as a more flexible extension of that work. Decision trees, random forests, and gradient-boosting methods can represent nonlinear relationships, interactions, and threshold effects that arise naturally in soccer data.

That flexibility is especially useful in soccer analytics, where performance often depends on combinations of context: distance and angle together, pressure and location together, possession state and game situation together. Tree-based models can capture those structures without requiring us to write every interaction by hand.

At the same time, tree-based models preserve an important link to interpretation. A single tree can be read as a sequence of soccer-style rules, while feature importance and Shapley additive explanations (SHAP) analysis help explain more complex ensembles.

By the end of this chapter, you will be able to do the following:

- Understand how decision trees learn hierarchical rules from soccer data.
- Build, visualize, and interpret a classification tree model.
- Construct a random forest ensemble to improve predictive accuracy and stability.
- Identify when to use a simple tree versus a forest for different analytical tasks.
- Tune hyperparameters to avoid overfitting and optimize performance.
- Build and tune extreme gradient boosting (XGBoost) models for various soccer prediction tasks.

The Classification Tree: Soccer's Decision Flowchart

Imagine you are a coach analyzing a missed scoring chance. Your thought process isn't a single calculation; it's a flowchart of questions. Was the shooter under pressure? Was the shot taken on their preferred foot? Was the angle too narrow? Each answer leads to another question until you reach a conclusion: *That should have been a goal.* A classification tree formalizes this exact process.

The Intuition: Learning the If-Then Rules of Soccer

A *classification tree* is a supervised learning algorithm that partitions the feature space into rectangular regions, making a prediction based on the majority class in each region. Think of it as playing 20 questions with your data. To classify a shot as a goal or no-goal, the tree might ask these questions:

1. Is the shot distance less than 18 meters? If yes, go to question 2; if no, predict `No Goal`.
2. Is the shot angle greater than 20 degrees? If yes, go to question 3; if no, predict `No Goal`.
3. Was it a first-time shot? If yes, predict `Goal`; if no, predict `No Goal`.

This sequence creates a transparent, human-readable model. The path from the top (the *root*) to the bottom (a *leaf*) represents a decision rule.

Hands-On: Building a Tree to Predict Pass Outcome

Let's build a tree to predict whether a pass will be completed. We'll use StatsBomb event data from a high-pressure match to try to uncover the rules of successful passing. As in the previous chapters, we reuse the shared plotting helper and omit its internal implementation:

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.model_selection import train_test_split
from soccerml_figures.plotting import (
    apply_margins,
    create_figure,
    output_path,
    save_figure,
)
# Load and prepare data (assuming 'passes' DataFrame is ready)
features = ["start_x", "start_y", "pass_length"]  ❶
target = "completed"
X = passes[features].dropna()
y = passes.loc[X.index, target]
```

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
# Train a simple decision tree
tree_clf = DecisionTreeClassifier(
    max_depth=3, random_state=42,
) ❷
tree_clf.fit(X_train, y_train)
# Visualize the tree
fig, ax = create_figure()
plot_tree(
    tree_clf,
    feature_names=features,
    class_names=["Incomplete", "Complete"],
    filled=True,
    rounded=True,
    fontsize=8,
    ax=ax,
)
ax.set_title("Decision Tree for Pass Completion Prediction")
apply_margins(
    fig, left=0.01, right=0.99, top=0.82, bottom=0.03,
)
save_figure(fig, output_path(6, "pass-decision-tree"))
```

❶ Select simple features: the pass starting coordinates and its length.

❷ Set `max_depth=3` to keep the tree small and interpretable. `max_depth` is a simple setting that controls the number of times the tree can ask yes-or-no questions before making a final decision. This prevents overfitting and makes the visualization easy to read.

This code trains a simple decision tree with a maximum of three levels of questions. The real magic happens when we visualize it. The resulting tree, shown in Figure 6-1, reveals a clear hierarchy of decisions.

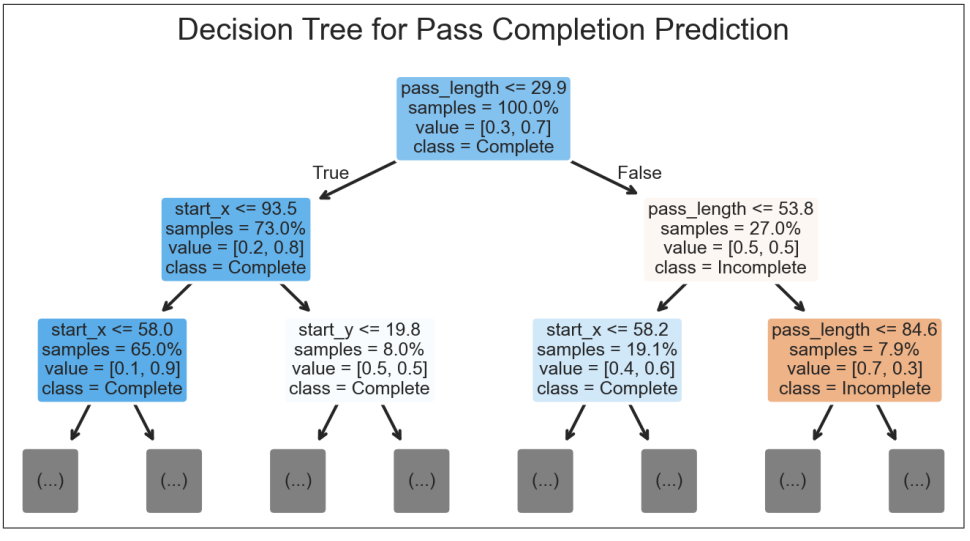


Figure 6-1. In this decision tree for predicting pass completion, the numbers are examples of what we could expect

Interpreting the Soccer Intelligence

The root node (top) first asks whether the pass length is less than or equal to 29.9 meters. This is the most important factor for splitting the data. If a pass is longer than that, the model immediately becomes more skeptical about its completion.

Let's trace a path to extract a rule. If a pass is short (`pass_length <= 29.9`), and it starts far up the pitch in the attacking third (`start_x > 93.5`), the model is fairly confident it will be completed (the bottom-left leaf shows 65% of such passes are complete). This aligns with tactical wisdom: short, simple passes in the final third are crucial for maintaining possession and creating chances.

Conversely, a long pass (`pass_length > 29.95`) that is also not extremely long (`pass_length <= 53.8`) and starts in the defensive half (`start_x <= 58.2`) has a much lower completion probability. This captures the risk of long balls out from the back.

Understanding Pros and Cons: When to Use a Single Tree

Decision trees are fantastic tools, but it's important to know their strengths and weaknesses. Table 6-1 explains the advantages, and Table 6-2 explains the disadvantages.

Table 6-1. Advantages of decision trees

Advantage	Description
Transparency	The model is transparent. You can explain every prediction by tracing the path from the root to the leaf. This is invaluable when presenting to coaches or stakeholders who need to trust the model's reasoning.
No feature scaling needed	Trees split on feature values independently of scale. You don't need to normalize or standardize your data, which simplifies the preprocessing pipeline.
Handles nonlinear relationships	Trees can model complex interactions between features (e.g., a pass is good only if it's short *and* in the final third). This is a major advantage over linear models.

Table 6-2. Disadvantages of decision trees

Disadvantage	Description
High variance	Small changes in the training data can lead to a completely different tree. This instability makes single trees unreliable for robust prediction.
Prone to overfitting	A deep tree will memorize the training data, including its noise, and fail to generalize to new, unseen data. This is the biggest challenge with decision trees.
Limited predictive power	A single tree is often less accurate than more complex ensemble methods. It's a great exploratory tool but may not be the best choice for production-level prediction.

Classification trees are an excellent starting point for understanding which features drive outcomes. When you need to present clear, actionable rules to a coaching staff, a simple tree is often the best tool for the job.

Tuning the Tree: Preventing Overfitting

The key to a useful tree is controlling its complexity. The `max_depth` parameter we used is a crucial tool for this. Let's see how it affects performance:

```
from sklearn.metrics import accuracy_score
train_acc = []
test_acc = []
depths = range(1, 15)
for depth in depths:
    tree = DecisionTreeClassifier(
        max_depth=depth, random_state=42,
    )
    tree.fit(X_train, y_train)
    train_acc.append(
        accuracy_score(y_train, tree.predict(X_train))
    ) ❶
    test_acc.append(
        accuracy_score(y_test, tree.predict(X_test))
    ) ❷
fig, ax = create_figure()
ax.plot(
    depths,
    train_acc,
```

```
    label="Training Accuracy",
    marker="o",
    linewidth=2,
)
ax.plot(
    depths,
    test_acc,
    label="Test Accuracy",
    marker="s",
    linewidth=2,
)
best_depth = max(zip(test_acc, depths))[1]
ax.axvline(
    best_depth,
    color="red",
    linestyle="--",
    alpha=0.7,
    label=f"Best Depth = {best_depth}",
)
ax.set_xlabel("Max Depth")
ax.set_ylabel("Accuracy")
ax.set_title(
    "The Overfitting Problem: " "Training vs Test Accuracy"
)
ax.grid(True, alpha=0.3)
ax.legend(loc="best")
apply_margins(fig)
save_figure(fig, output_path(6, "overfitting-curve"))
```

❶ Calculate the classification accuracy on the training data, which the model has already seen.

❷ Calculate the classification accuracy on the test data, which the model has *not* seen. This is the true measure of its performance.

This code trains a new tree for each depth from 1 to 14 and records both the training and test accuracy. Figure 6-2 shows the output.

This plot reveals a classic pattern. Training accuracy keeps improving as the tree grows deeper (it's memorizing the data), but test accuracy peaks around a depth of 4 and then declines. This is overfitting in action. The sweet spot (`max_depth=4`) gives us a model that generalizes well.

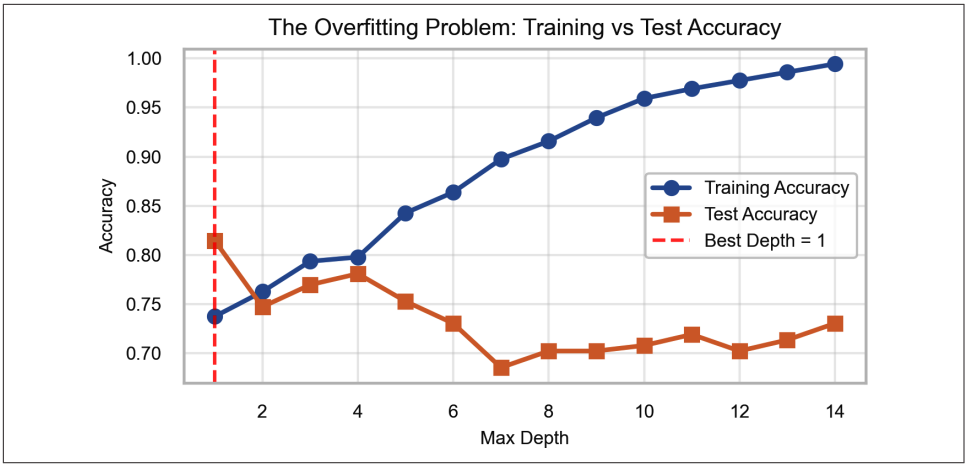


Figure 6-2. The overfitting problem: training versus test accuracy

Other key parameters to control overfitting include the following:

`min_samples_split`
: The minimum number of samples required to split a node

`min_samples_leaf`
: The minimum number of samples required to be at a leaf node

`max_features`
: The number of features to consider when looking for the best split

A well-tuned tree is a balance between bias and variance. But what if we could have hundreds of trees, each slightly different, and combine their wisdom? Enter the random forest.

Random Forest: The Wisdom of the Crowd

A single tree is insightful but unstable. Ask it the same question on slightly different data, and you might get a different answer. In soccer, we need robustness. The *random forest* algorithm provides this by building an ensemble of many decision trees and aggregating their predictions.

The Intuition: Ensemble Learning

Imagine you want to predict the outcome of a match. Instead of asking one expert, you ask a hundred. Each expert has slightly different knowledge and biases. While some might be wrong, the collective wisdom of the crowd is likely to be more accurate than any single expert. This is the core idea of a random forest.

It builds hundreds of decision trees, but with two key twists:

Bagging (bootstrap aggregating)
: Each tree is trained on a random sample of the training data, drawn *with replacement*. This means some matches will appear multiple times in a tree's training set, while others won't appear at all.

Feature randomness
: At each split in a tree, the algorithm considers only a random subset of features. This prevents any single strong feature (like `distance_to_goal`) from dominating all the trees.

These two techniques ensure that the trees in the forest are diverse. To make a prediction, the random forest simply takes a majority vote from all the individual trees. This averaging process cancels out the noise and instability of individual trees, leading to a much more robust and accurate model.

Hands-On: Building a Random Forest for to Predict Match Outcome

Let's build a random forest to predict whether a team will win, lose, or draw based on in-game statistics:

```
from sklearn.ensemble import RandomForestClassifier
# Assuming X_train, y_train for match outcomes are ready
rf_clf = RandomForestClassifier(
    n_estimators=100, random_state=42, n_jobs=-1
)  ❶
rf_clf.fit(X_train, y_train)
# Get feature importances
importances = rf_clf.feature_importances_  ❷
feature_names = X_train.columns
feature_importance = pd.DataFrame(
    {"Feature": feature_names, "Importance": importances,}
).sort_values("Importance", ascending=False)
fig, ax = create_figure()
ax.barh(
    feature_importance["Feature"],
    feature_importance["Importance"],
    color="#4C78A8",
)
ax.invert_yaxis()
ax.set_xlabel("Importance")
```

```
ax.set_title(
    "Random Forest Feature Importance " "for Match Outcome"
)
apply_margins(fig, left=0.2)
save_figure(
    fig, output_path(6, "random-forest-feature-importance"),
)
```

❶ Create a forest of 100 trees. `n_jobs=-1` tells scikit-learn to use all available CPU cores to train the trees in parallel.

❷ After training, we can ask the forest which features were most important in making its decisions.

This code trains a random forest and then extracts the feature importances, which tell us which in-game stats were most influential in predicting the match outcome. Figure 6-3 shows the output.

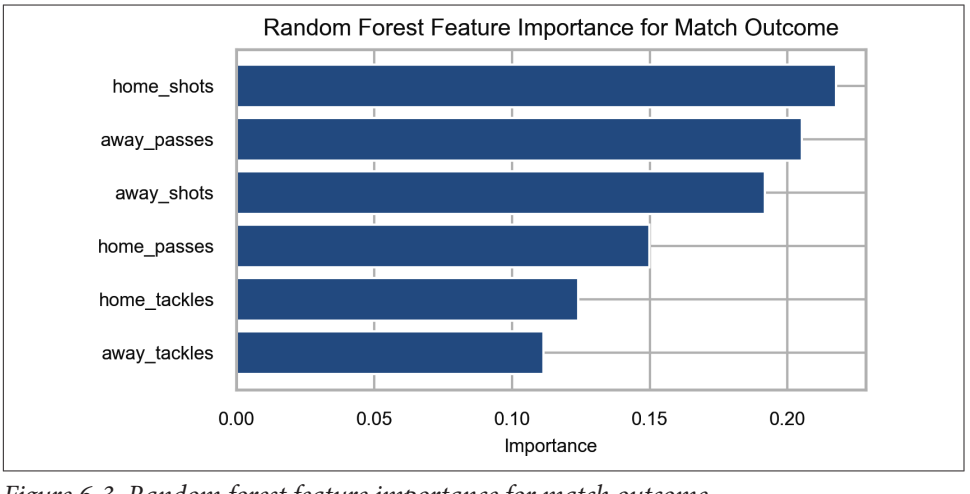


Figure 6-3. Random forest feature importance for match outcome

Interpreting Feature Importance

The plot in Figure 6-3 shows how the random forest classifier in the preceding code evaluates different features.

In particular, we can see that `home_shots` and `away_passes` were the two most important features for predicting the outcome. Coming in third is `away_shots`. It's remarkable that `away_shots` and `away_passes` score so high—in the same order of magnitude as `home_shots`—because it suggests that the offensive actions of the away team are almost as predictive of the result as the home team's actions. This could be because home teams are generally expected to dominate, so when an away team

manages to generate a lot of passes and shots, it's a strong signal that an upset might be in the cards.

Unlike a single decision tree, we can't easily visualize a random forest. We trade some interpretability for a big gain in predictive power. However, feature importance plots give us a high-level understanding of what the model has learned.

Tuning Random Forests

Random forests are less prone to overfitting than single trees, but they still have important hyperparameters to tune:

`n_estimators`
: The number of trees in the forest. More is generally better, but that comes at a computational cost.

`max_depth`
: The maximum depth of each tree. Deeper trees can capture more complex patterns but risk overfitting.

`max_features`
: The number of features to consider at each split. A smaller number increases the diversity of the trees.

For most applications, a random forest with default parameters provides a very strong baseline model. It's the perfect tool when you need high accuracy and robustness without the extensive hyperparameter tuning required by more complex models like XGBoost.

Using Out-of-Bag Error

One of the clever features of random forests is the *out-of-bag (OOB) error* estimate. Remember that each tree is trained on a bootstrap sample of the data, meaning some data points are left out. These left-out samples can be used as a validation set for that particular tree. By aggregating predictions across all trees for which a data point was OOB, we get a free cross-validation estimate without needing a separate validation set.

This is particularly useful when working with limited data, as it allows you to use all available data for training while still getting an honest estimate of model performance. In scikit-learn, you can enable this by setting `oob_score=True` when creating your random forest classifier.

Understanding When Random Forests Struggle

While random forests are powerful, they have limitations. They can struggle with the following:

- Very high-dimensional sparse data (e.g., text data with thousands of features)
- Extrapolation beyond the range of training data (they can predict only values within the range they've seen)
- Capturing very complex sequential patterns (where the order of events matters significantly)

For soccer analytics, these limitations rarely pose problems. Most soccer features are continuous or categorical with reasonable dimensionality, and the relationships we're modeling are well within the training data's range.

XGBoost: Learning from Mistakes

A random forest builds an army of independent experts. XGBoost takes a different approach: it builds a team of specialists who learn from one another's mistakes. In this sequential process, each new tree is built to correct the errors of the previous one.

The Intuition: Sequential Learning

Imagine you're trying to predict expected goals (xG). Your first model is a simple one that looks only at shot distance. It does OK, but it makes a lot of mistakes. XGBoost then builds a second model, but instead of trying to predict xG directly, it tries to predict the *errors* (the residuals) of the first model. By adding the predictions of this second model to the first, you get a more accurate overall prediction.

XGBoost repeats this process hundreds or thousands of times. Each new tree is a small, weak learner that focuses on the remaining errors. By chaining them together, XGBoost creates an incredibly powerful and accurate final model. This technique, called *gradient boosting*, is one of the most successful and widely used algorithms in machine learning competitions and real-world applications.

Hands-On: XGBoost for xG Prediction

Let's use XGBoost to build a state-of-the-art xG model:

```
import xgboost as xgb
# Assuming X_train, y_train for xG prediction are ready
xgb_model = xgb.XGBClassifier(
    objective="binary:logistic",
    eval_metric="logloss",
) ❶
xgb_model.fit(X_train, y_train)
```

❶ Specify the objective function (`binary:logistic` for binary classification) and the evaluation metric (`logloss`). Log loss was introduced as a cost function in "Evaluation of Probabilistic Models with Cost Functions" on page 137.

This code trains an XGBoost model. While it's highly accurate, it's also an opaque model. How can we understand its predictions? This is where SHAP comes in.

Performing SHAP Analysis: Explaining the Opaque Model

Shapley additive explanations (*SHAP*) is a game theory approach to explaining the output of any machine learning model. For our xG model, SHAP can tell us exactly how much each feature contributed to a specific shot's predicted xG value:

```
import shap
from soccerml_figures.plotting import (
    PLOT_DPI,
    output_path,
    save_figure,
)
# Create SHAP explainer
explainer = shap.TreeExplainer(xgb_model)  ❶
shap_values = explainer.shap_values(X_test)  ❷
# Summary plot (code to generate the image)
shap.summary_plot(
    shap_values, X_test, show=False, plot_size=None,
)  ❸
fig = plt.gcf()
fig.set_size_inches(4.8, 2.97)
fig.set_dpi(PLOT_DPI)
fig.axes[0].set_title(
    "SHAP Summary Plot: " "Feature Impact on Goal Prediction"
)
save_figure(fig, output_path(6, "shap-summary"))
```

❶ Create a `TreeExplainer` that is optimized for tree-based models like XGBoost.

❷ Calculate the SHAP values for every prediction in our test set.

❸ The shap library has a powerful built-in visualization module.

This visualization, shown in Figure 6-4, is incredibly powerful (some visuals have been changed for optimal display in the book, but the previous code will get you started). It reveals how each feature influences goal predictions across all shots. Features are ranked by their overall importance. For each feature, individual shots are plotted horizontally by their SHAP value. Light-colored dots represent high feature values (e.g., close shots), while dark-colored dots represent low values (e.g., distant shots).

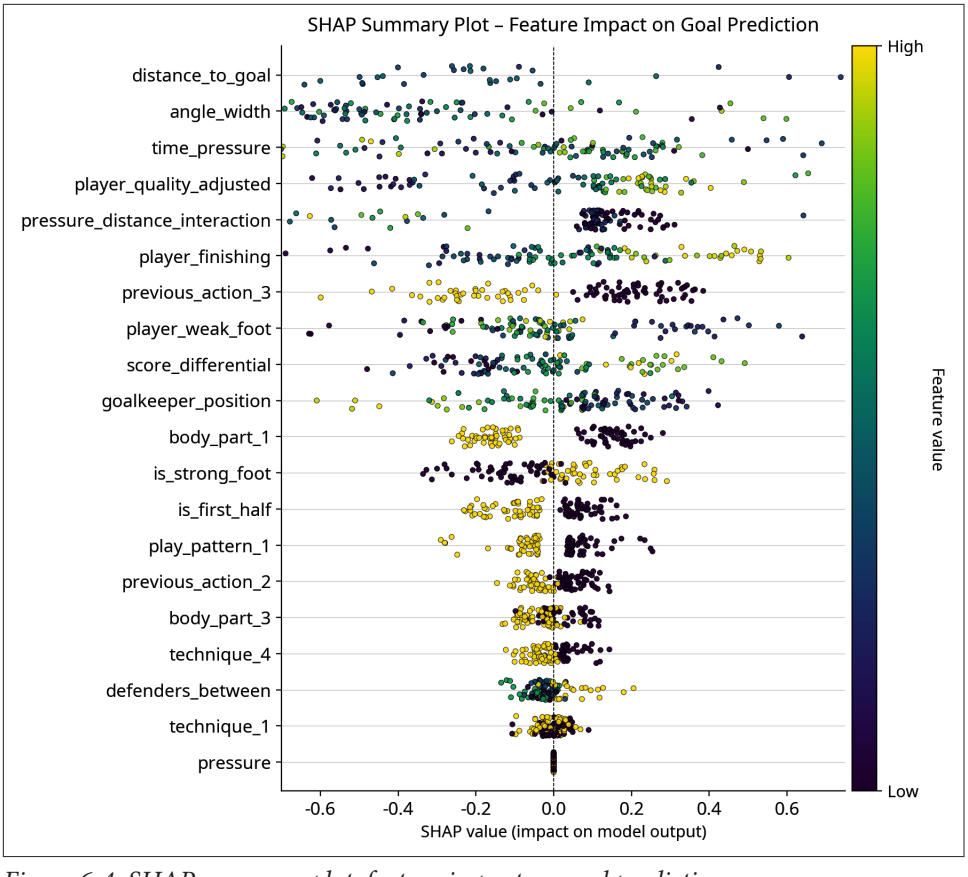


Figure 6-4. SHAP summary plot: feature impact on goal prediction

The plot shows a few striking insights. Particularly, we can see that `player_finishing` has a strong impact on goal prediction. The lighter dots on the right indicate that when the player does finish, the model indeed predicts a goal, and when the player doesn't, usually no goal occurs. This is clear intuitively—however, it's an important indicator that our model is working correctly. We can also see the impact of other features like `angle_width` and `pressure`. SHAP gives us the interpretability of a simple model with the power of a complex one.

Tuning XGBoost: The Art of Hyperparameter Optimization

XGBoost's power comes with complexity. It has many hyperparameters that control the learning process. These are the most important ones for soccer analytics:

`learning_rate` *(or* `eta`*)*
: Controls how much each tree contributes to the final prediction. Lower values (0.01–0.1) require more trees but often lead to better performance.

`max_depth`
: The maximum depth of each tree. Typical values range from 3 to 10. Deeper trees can model more complex patterns but risk overfitting.

`n_estimators`
: The number of boosting rounds (trees). More is generally better, but with diminishing returns and increased computation time.

`subsample`
: The fraction of samples used for each tree. Values like 0.8 introduce randomness and help prevent overfitting.

`colsample_bytree`
: The fraction of features used for each tree. This is similar to random forest's `max_features`.

`reg_alpha` *and* `reg_lambda`
: L1 and L2 regularization terms. These penalize complex models and help prevent overfitting.

A typical tuning workflow starts with default parameters, then uses cross-validation to systematically explore different combinations. Grid search or randomized search can help find optimal values, but be prepared for this to take time because XGBoost training can be computationally expensive.

Knowing When to Use XGBoost Versus Random Forest

Both XGBoost and random forest are ensemble methods, but they excel in different scenarios.

Use random forest for these circumstances:

- You need a quick, robust baseline with minimal tuning.
- Interpretability through feature importance is sufficient.
- Training time is a constraint.
- Your dataset is small to medium-sized.

Use XGBoost for these scenarios:

- You need maximum predictive accuracy.
- You have time for careful hyperparameter tuning.
- Your dataset is large.
- You can use SHAP for interpretability.
- You're building a production system where accuracy is critical.

For most soccer analytics projects, starting with random forest and moving to XGBoost only when necessary is a pragmatic approach. The performance gain from XGBoost is often modest (2%–5% improvement in accuracy), but in competitive settings like betting or player valuation, even small gains matter.

Practical Applications: From Theory to Practice

Having covered the theory and mechanics of tree-based models, let's explore how to apply them to real-world soccer analytics challenges. This section walks through typical workflows for common tasks.

Example 1: Building a Comprehensive xG Model

Building a production-quality xG model is one of the most important applications in modern soccer analytics. The workflow follows these steps:

1. *Data collection and preparation*
   Gather shot data from StatsBomb, including location, body part, technique, pressure, and contextual features like score differential and game state

2. *Feature engineering*
   Create derived features such as distance to goal, angle to goal, and interaction terms (e.g., distance × pressure)

3. *Model selection*
   Start with logistic regression as a baseline, then build a decision tree for interpretability, a random forest for robustness, and finally XGBoost for maximum accuracy

4. *Evaluation*
   Use cross-validation to compare models on metrics like log loss, Brier score, and calibration curves (see "Evaluation of Probabilistic Models with Cost Functions" on page 137 for a full treatment of these metrics). For xG, calibration is particularly critical: if the model assigns 0.3 xG to a shot, actual goals should occur 30% of the time across all such shots

5. *Interpretation*
   Use SHAP to understand which features drive predictions and validate that they make soccer sense

6. *Deployment*
   Package the model for use in match analysis, player evaluation, or tactical planning

A well-built xG model becomes the foundation for many downstream analyses: player finishing ability (actual goals versus xG), chance creation quality, and defensive performance.

Example 2: Predicting Player Actions

Another valuable application is predicting what a player will do next: pass, dribble, or shoot. This helps coaches understand decision-making patterns and identify players who make optimal choices under pressure.

The challenge here is the multiclass nature of the problem. A player might have three to five viable options at any moment. Tree-based models handle this naturally through their splitting mechanism.

Key features for this task include the following:

- Player position and orientation
- Locations of teammates and opponents
- Distance to goal and passing lanes
- Game state (score, time remaining)
- Player attributes (preferred foot, role)

A random forest works well for this task because it can capture the complex interactions between spatial features (where teammates are) and contextual features (game state). The feature importance plot reveals which factors most influence decision-making, providing insights for coaching.

Example 3: Recognizing Tactical Patterns

Tree-based models can also identify tactical patterns. For example, you might want to classify a team's build-up style: direct, possession-based, or counterattacking.

This requires aggregating event data at the match or possession level. Features might include the following:

- Average pass length and direction
- Time per possession

- Proportion of passes in different pitch zones
- Speed of transition from defense to attack

A decision tree is ideal here because the resulting rules are directly interpretable: *If average pass length > 25 meters and time per possession < 8 seconds, then classify as direct.* These rules can be shared with coaching staff as actionable insights.

Handling Imbalanced Data in Soccer

Many soccer prediction tasks involve imbalanced classes. Goals are rare (shots convert about 10% of the time), away wins are less common than home wins, and certain player actions (like successful dribbles in the box) are infrequent.

Tree-based models handle imbalance reasonably well, but you can improve performance with these techniques:

Class weighting
: Use the `class_weight='balanced'` parameter in scikit-learn to automatically adjust for class imbalance.

Resampling
: Oversample the minority class or undersample the majority class. The imbalanced-learn library provides tools like synthetic minority oversampling technique (SMOTE).

Threshold tuning
: Adjust the decision threshold to favor recall or precision based on your application. For injury prediction, you might prioritize recall (catching all potential injuries) even at the cost of false positives.

Evaluation metrics
: Don't rely solely on accuracy. Use precision, recall, F1 score, and area under the curve (AUC)-receiver operating characteristic (ROC), which are more informative for imbalanced data.

In the xG example, the imbalance is severe (90% nongoals). Without adjustment, a naive model might predict `No Goal` for everything and still achieve 90% accuracy. Class weighting or threshold tuning ensures that the model learns to identify the rare goal events.

Using Feature Engineering for Tree-Based Models

While tree-based models are powerful, they still benefit from thoughtful feature engineering. Here are tips specific to soccer analytics:

Spatial features
: Convert (x,y) coordinates into meaningful features like distance to goal, angle to goal, and proximity to defenders.

Temporal features
: Include time-based features like minute of the match, time since last goal, or days since last match.

Interaction Features
: Create features that capture relationships, such as pass length × pressure or shot angle × goalkeeper position.

Aggregated features
: For player-level predictions, include rolling averages of recent performance (e.g., average xG per 90 minutes over the last five matches).

Categorical encoding
: Tree-based models handle categorical features naturally, but encoding them as integers (ordinal encoding) or one-hot encoding can sometimes improve performance.

The key is to create features that capture soccer domain knowledge. A model can't learn what it doesn't see in the data.

Looking Ahead: From Handcrafted Rules to Learned Representations

The tree-based methods in this chapter represent a significant step forward from the linear classifiers we started with. Decision trees, random forests, and XGBoost can all discover thresholds, interactions, and nonlinear structure in soccer data without requiring us to specify those patterns in advance. Yet they still share one underlying assumption: the features we hand the model are the right building blocks. Whether it is distance to goal, angle, or a pressure-distance interaction term, we are still the ones deciding what the model gets to see.

In Chapter 7, we take a further step by introducing deep learning and neural networks. Rather than relying solely on handcrafted features, neural networks can learn layered internal representations directly from data—discovering combinations and abstractions that we might not have thought to encode ourselves. We will build models for the same soccer tasks we have been working on throughout this book: binary

match outcome prediction, multiclass result forecasting, and goal scoring. We will do so in both PyTorch and TensorFlow, so you can see how the same ideas translate across the two dominant deep learning frameworks. The goal is not to treat neural networks as an opaque model that simply outperforms everything else, but to develop a clear sense of when that added expressiveness is worth the extra complexity—and when a well-tuned random forest or XGBoost model is still the right tool for the job.

Conclusion

Tree-based methods deepen the classification toolkit by giving us models that can represent thresholds, interactions, and nonlinear structure more naturally than the linear methods from Chapter 5. Across decision trees, random forests, and XGBoost, the central idea is the same: break a difficult predictive task into simpler rules and combine those rules in a disciplined way.

Each method brings a different balance of transparency, stability, and raw predictive power. A single tree is easy to explain. Random forests reduce variance and improve robustness. Gradient boosting pushes performance further by learning sequentially from earlier errors.

Choosing the right algorithm depends on the trade-off between interpretability and predictive accuracy. Table 6-3 gives you some more guidance on which algorithm to select.

Table 6-3. Algorithm selection guide for decision tree, random forest, and XGBoost

Criteria	Decision tree	Random forest	XGBoost
Interpretability	Excellent	Good	Moderate (with SHAP)
Predictive accuracy	Poor	Good	Excellent
Training speed	Fast	Medium	Slow
Robustness to noise	Poor	Excellent	Good

As you apply these methods to your own soccer analytics projects, remember that the most sophisticated model is not always the best choice. Start with a decision tree to understand the problem and establish a baseline. Move to a random forest when you need robustness and better accuracy without extensive tuning. Reserve XGBoost for situations where additional predictive power justifies the added complexity and computational cost.

In Chapter 7, we push this progression further. Instead of hand-specifying interactions or letting trees discover them through splits, we will look at neural networks that learn richer internal representations directly from data. The workflow remains familiar, but the modeling framework becomes more expressive.

CHAPTER 7

Deep Learning for Soccer Analytics: From Intuition to Implementation

So far in this book, we have mostly worked with models that depend on carefully chosen features and relatively direct relationships between inputs and outputs. That is an excellent place to begin. It teaches discipline, interpretability, and the logic of building models step by step. But it also has limits.

Soccer doesn't always present itself in a few clean columns. Its signal is often fragmented across many interacting variables: player movement, match state, tactical structure, momentum, fatigue, and the subtle feedback loops that emerge over the course of a game. What matters is not always one stat, or even one cluster of stats, but the way many pieces combine. And in many cases, the most useful representation of that structure is not obvious in advance.

This is where deep learning becomes compelling. Deep learning is built around neural networks: flexible models that can learn layered representations from data rather than relying only on handcrafted rules or simple linear relationships. Instead of asking us to specify every important interaction ahead of time, deep learning gives the model room to discover patterns that are distributed, nonlinear, and difficult to encode manually.

In soccer analytics, that matters because the game is rich, contextual, and deeply interconnected. Deep learning can help us move beyond isolated indicators and toward models that capture relationships among match context, team strength, scoring patterns, and game dynamics. It does not replace domain knowledge. Rather, it gives us a more expressive framework for using that knowledge when the structure of the problem becomes too complex for simpler approaches to handle gracefully.

In this chapter, you'll learn why deep learning has become such an important part of the modern analytical toolkit, especially when the relationships in the data are too intricate for simpler models to capture well. You'll also build deep learning models for several soccer tasks, from classification to goal prediction and multitask learning. The aim is not to treat deep learning as magic but to understand when it is useful, what it is learning, and how to implement it in practice.

By the end of this chapter, you should be able to do the following:

- Understand the main ideas behind deep learning.
- Understand the basic logic of model design, training, and evaluation.
- Build soccer prediction models in both PyTorch and TensorFlow for binary classification, multi-class prediction, and goal modeling.
- Recognize common challenges and know where to start when a model needs troubleshooting or refinement.



This chapter combines three things at once: the core ideas behind deep learning, a hands-on implementation in PyTorch, and an alternative implementation in TensorFlow.

If this feels like a lot, that is normal. The fastest way to get moving is to follow one framework closely on your first read-through. You do not need to master both PyTorch and TensorFlow at the same time.

If you already have some background in neural networks, feel free to skim the conceptual sections and focus on the implementation. If the concepts are new to you, however, reading those sections first is worthwhile so you can build a clear big-picture understanding before diving into the code. The goal here is not to master all the mathematics at once but to develop enough intuition to start building useful models quickly.

Here is a useful reading path:

1. Read the conceptual sections through loss functions.
2. Follow the PyTorch implementation as the main hands-on track.
3. Treat the TensorFlow section as a second lens or reference rather than as material you must master on the same pass.

If you are short on time, steps 1 and 2 give you the main narrative of the chapter.

We'll work through four soccer prediction tasks, each one building on the last:

1. Binary: what is the probability that the home team wins?
2. Multiclass: what is the full-time result (home team win, draw, or away team win)?
3. Regression with a Poisson distribution: how many goals will the home team score?
4. Multitask: how many goals will the home and away teams score?

We will keep the explanations soccer-specific and the code as light as possible. The main walk-through uses PyTorch, and a parallel TensorFlow implementation appears later, in "A Second Lens: Implementing the Model in TensorFlow" on page 209. On a first read, it is perfectly fine to treat PyTorch as the primary path and TensorFlow as a comparison section.

Why Deep Learning for Soccer? Beyond Traditional Models

One useful way to build intuition is to imagine a team moving the ball from the goalkeeper through the back line, midfield, and attack. The ball starts deep, moves through different lines, and eventually reaches the final attacking action. In a neural network, data moves in a similar way: it enters the model, passes through several layers, and eventually produces a prediction. In this analogy, each neuron in the network is like a player handling and passing information, while each layer is like a unit of the team. Figure 7-1 illustrates this.

The analogy is only for intuition. Neural networks were originally inspired by simplified ideas about the way biological neurons receive signals, process them, and pass them on. For our purposes, what matters is the structure: information comes in, is transformed through several layers, and leaves the network as an output.

In practice, a neural network has three main types of layers. The *input layer* receives the raw features, such as soccer statistics or engineered variables. One or more *hidden layers* transform those inputs into increasingly useful internal representations. A network with two or more hidden layers is usually described as a deep learning model. Finally, the *output layer* produces the prediction, such as a win probability, a match result, or an expected goal count.

Each layer consists of individual neurons. Each neuron comes with an *activation function*, which helps govern when it passes information onward and when it stays quiet. Between these layers of neurons, *forward propagation* is the process that moves information through the network. Finally, a *loss function* assesses the difference between reality and the neural network's prediction—in essence, a measure of the model's quality.

The rest of this section introduces these concepts in more detail. Once we have covered these, we'll apply them step by step to soccer prediction tasks, starting simple (home team win/loss) and moving toward more complex predictions (results for both home and away team, goals, multitask models). A schematic of a neural network is shown in Figure 7-1.

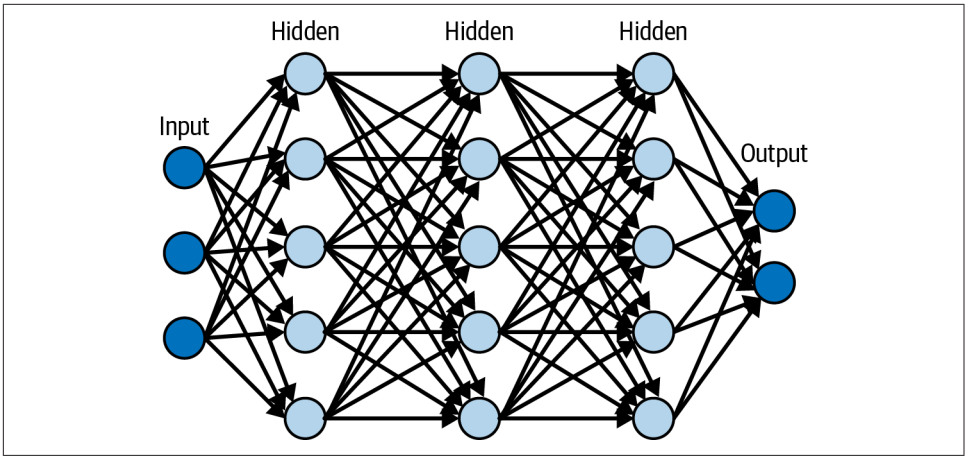


Figure 7-1. A neural network

Looking Inside the Model: How Neural Networks Capture Patterns

Before we dive into coding, let's understand just enough about how a neural network works to make sense of the models we'll build. Our conceptual overview will cover neurons, activation functions, forward propagation, and loss functions.

Neurons

Just like lines in a formation are made up by individual soccer players, layers in neural networks are made up by *neurons* (hence the *neural* in neural networks). Every neuron is connected to all the neurons in the layers before and after it, but not to the neurons in its own layer.

The *knowledge* of a neural network is quantified in *weights*. You can think of them as the strength of the connections between individual neurons. Some neurons might be very connected, others less so.

For example, teams that take many shots on target typically score more goals. In a neural network, the connection (weight) between a neuron representing shots on target (near the input layer) and a neuron representing goals scored (in the output layer) would become strong as the model learns this relationship.

In addition to weights, each neuron has a *bias*—a kind of built-in tendency that shifts its decision making independent of the inputs. In the soccer analogy, you might

think of the bias as a player's default positioning or instinct before the next pass even arrives. This bias allows neurons to activate (or stay quiet) more flexibly, making the entire network better at adapting to complex game patterns.

Activation functions

Weights and biases alone are not enough. We also need to decide when information flows through the network and when it doesn't. In soccer terms, not every passing lane stays open, and not every move deserves to continue the attack. Sometimes the play breaks down, and sometimes it flows cleanly into the next line. That selective *go/no-go* effect is the role of activation functions.

The three most common are the rectified linear unit (ReLU), sigmoid, and hyperbolic tangent (tanh). We'll use the first two heavily in this chapter.

The ReLU. The ReLU is a fairly basic function: give an input below a certain threshold, and it is 0. Above that threshold, it is equal to the input value.

Mathematically, we can write the ReLU function like this:

$$f(x) = \max(0, x)$$

You can think of it like goal counts: there's no such thing as negative goals, but after the team starts scoring, every additional goal adds up. Check Figure 7-2 to see what this looks like.

ReLU makes networks efficient (lots of 0s where nothing happens) and flexible (many ReLUs stacked together create complex patterns). The main drawback is that if too many inputs fall below zero, parts of the network "die" and stop learning.

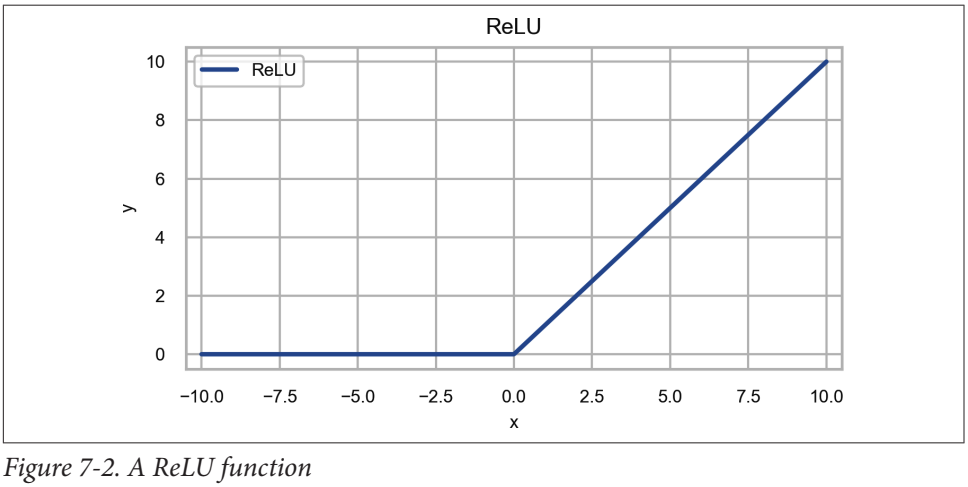


Figure 7-2. A ReLU function

The sigmoid function. The sigmoid function provides a smooth, probabilistic output from 0 to 1. For example, we might model the probability that a goal gets scored in a match with this function. It is written like this:

$$y = \frac{1}{1 + e^{-x}}$$

The sigmoid function is perfect for outputs we want to read as probabilities—for example, *what's the chance the home team wins?* The downside is that when values get too extreme, the curve flattens and learning slows (the vanishing gradient problem).

Figure 7-3 shows what this looks like. It's flat and near 0 at low x-axis values, then curves up, and finally flattens again toward $y = 1$ for high x.

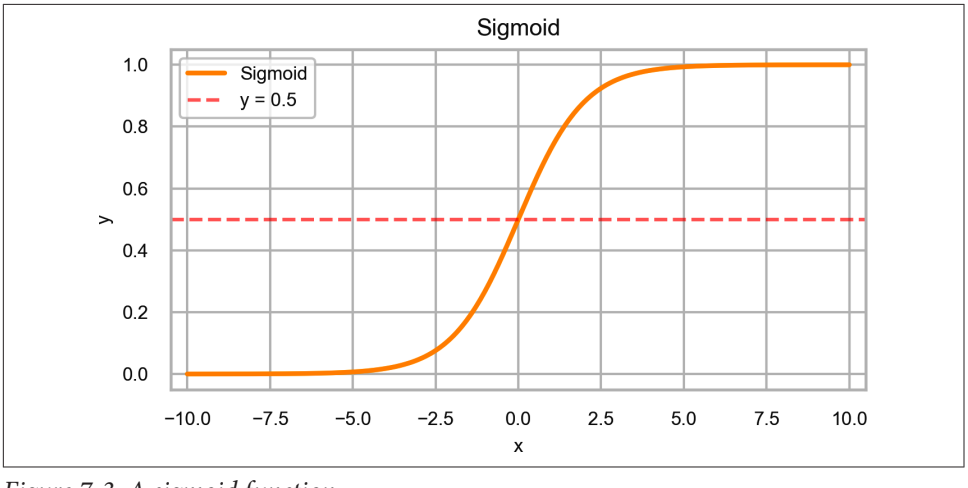


Figure 7-3. A sigmoid function

The hyperbolic tangent (tanh). Tanh is similar to sigmoid but ranges from –1 to 1, so it is centered on zero. It can sometimes train faster than sigmoid, but like sigmoid, it also suffers from vanishing gradients. We won't use it further here, but it's worth knowing as another option in the toolbox.

Forward propagation

Forward propagation is the process of moving data through a network to generate predictions. Think of it like a team building an attack from the goalkeeper. The ball moves through defenders, into midfield, and then toward the forwards. At each stage, the tactical setup influences where the ball goes next. In the network, weights influence how strongly information is passed forward, biases shift each neuron's tendency, and activation functions help decide whether that signal continues.

Here's a minimal two-layer example in Python:

```
import numpy as np
def relu(x):
    return np.maximum(0, x)
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
def forward_propagation(X, weights, biases):
    """Run forward propagation for a two-layer network."""
    # Hidden layer: Z = XW + b
    z1 = np.dot(X, weights[0]) + biases[0]
    # Apply ReLU nonlinearity
    a1 = relu(z1)
    # Output layer: Z = AW + b
    z2 = np.dot(a1, weights[1]) + biases[1]
    # Convert logits to probabilities
    a2 = sigmoid(z2)
    return a2
```

The input layer `X` is typically our match data (shots, passes, possession, etc.). The hidden layers transform that information, a bit like the team reshaping the move as the ball travels through different lines. The output layer applies another transformation and sigmoid. This gives us a probability, such as *how likely is a home win?*

In practice, weights and biases (which are inputs to the forward propagation function) are not chosen by hand. Instead, the training sets them based on thousands of matches. Here we're just illustrating how information flows from raw features, through hidden decision layers, to a prediction. If you continue the soccer analogy, the final prediction is like the shot at the end of the move: the whole attacking sequence leads up to that outcome.

Loss functions

How do we judge whether a model is "playing well"? In soccer, the scoreboard tells us whether a team's strategy is working. In machine learning, that scoreboard is the *loss function*: a number that measures how far off the model's predictions are from reality.

A lower loss means our predictions are closer to the truth. During training, the network constantly updates its weights to reduce this loss.

Different prediction tasks call for different loss functions:

Binary cross-entropy (BCE)
: For yes/no tasks like *will the home team win?* (1 for win, 0 for not win)

Categorical cross-entropy
: For multiclass outcomes such as predicting win/draw/loss

Poisson regression loss
: For count data, such as predicting the number of goals a team scores

Mean squared error (MSE)
: For continuous values, such as predicting an exact scoreline

We'll use these functions later, matching each one to the soccer task at hand. Think of them as different ways of keeping score, depending on what question we're asking the model.

Learning from Data: Training, Loss, and Optimization

Before we can train a model to make good predictions, we need to understand how it learns from mistakes. In this section, we'll see how a network "watches game tape" of its own predictions and corrects its play, using three key ideas:

Backpropagation
: How errors flow backward through the network

Gradient descent
: How the network uses those errors to adjust its weights step by step

Regularization
: How the network balances fitting the data with maintaining generalization

Backpropagation

Backpropagation is the method a neural network uses to learn from errors. Imagine replaying a failed attack in slow motion: the team builds the play from the back, works the ball forward, creates a shot, and then fails to score because the attempt is blocked, saved, or off target. The coach then rewinds the move from the shot all the way back through the earlier passes, pointing out where the buildup could be improved. In a similar fashion, backpropagation (also called *backprop*) takes the prediction error at the output layer and works backward through the network, showing each weight how much it contributed to the mistake.

The key output of this process is the *gradient*. It's a measure of how much the loss changes from changing a specific weight. By collecting these gradients layer by layer, the network knows how to tweak itself so the next play (prediction) is a little more accurate. Importantly, the learning signal begins at the final outcome, but the adjustments are not limited to the final layer. Earlier layers also change, just as a failed attack may lead a team to adjust its buildup, spacing, or passing choices, not only the final shot.

Instead of retraining from scratch, backpropagation efficiently updates the model by making these small corrections after every batch of data. You can think of it as the feedback loop that turns a failed attacking sequence into a slightly better one next time.

Gradient descent

Backpropagation tells each weight how it contributed to an error; *gradient descent* is how the network uses that feedback to improve. The idea is simple: adjust the weights little by little in the direction that makes the loss smaller.

Think of repeated tactical fine-tuning after reviewing performance. The system does not restart from scratch after each mistake. Instead, it makes repeated small adjustments that improve the overall model over time. You can see this in Figure 7-4.

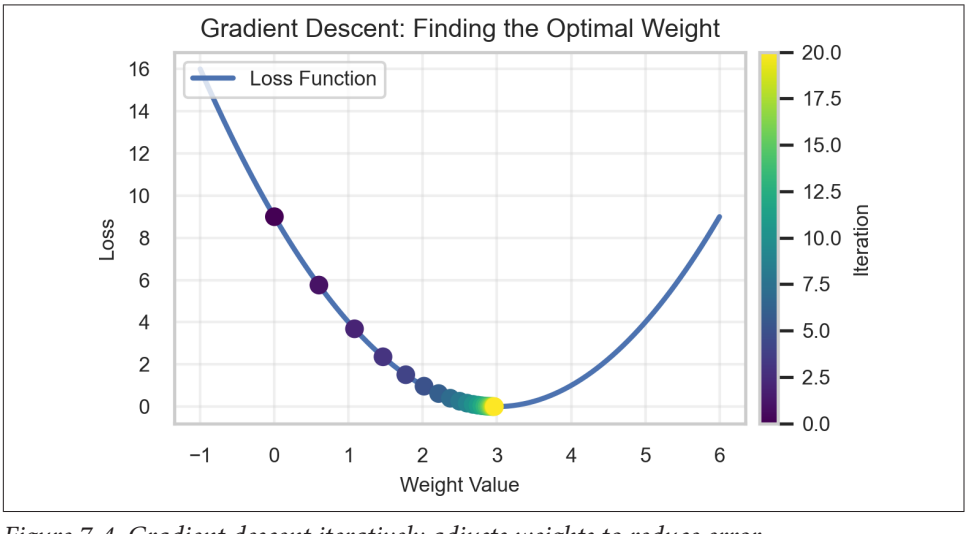


Figure 7-4. Gradient descent iteratively adjusts weights to reduce error

Gradient descent has three main types:

Batch gradient descent
: Uses the entire training dataset to make one big adjustment. Very stable, but slow if the dataset is huge.

Stochastic gradient descent (SGD)
: Updates after each single example. Faster to react, but the learning path can be noisy.

Mini-batch gradient descent
: A commonly used balance because it updates after a small group of examples at a time, thus combining stability with speed.

Here's a simple version of mini-batch gradient descent in Python:

```
for epoch in range(epochs):
    # Shuffle match data
    indices = np.random.permutation(n_samples)
    X_shuffled, y_shuffled = X[indices], y[indices]
    # Split into mini-batches
    for i in range(0, n_samples, batch_size):
        X_batch = X_shuffled[i : i + batch_size]
        y_batch = y_shuffled[i : i + batch_size]
        # Compute gradients and update weights
        grad_w, grad_b = compute_gradients(
            X_batch, y_batch, weights, bias
        )
        weights -= learning_rate * grad_w
        bias -= learning_rate * grad_b
```

Each epoch follows the same pattern:

1. Shuffle the matches (to avoid overfitting to order).
2. Break them into smaller mini-batches.
3. Compute gradients for each mini-batch and update the weights.

Note that `compute_gradients` is a stand-in for the backpropagation process. In practice, you won't code this from scratch. Libraries like PyTorch and TensorFlow automatically calculate gradients for you (via `loss.backward()` in PyTorch, or `GradientTape` in TensorFlow).

Backpropagation tells us how the model was wrong, and gradient descent uses that information to make the next update. That steady improvement is what turns error signals into better weights and biases over time.

Regularization techniques

A model that only memorizes the training data will struggle when it sees new matches. As noted in Chapter 4, this problem is called overfitting: the model performs brilliantly on training data but poorly on unseen data. Figure 7-5 shows a schematic of this principle.

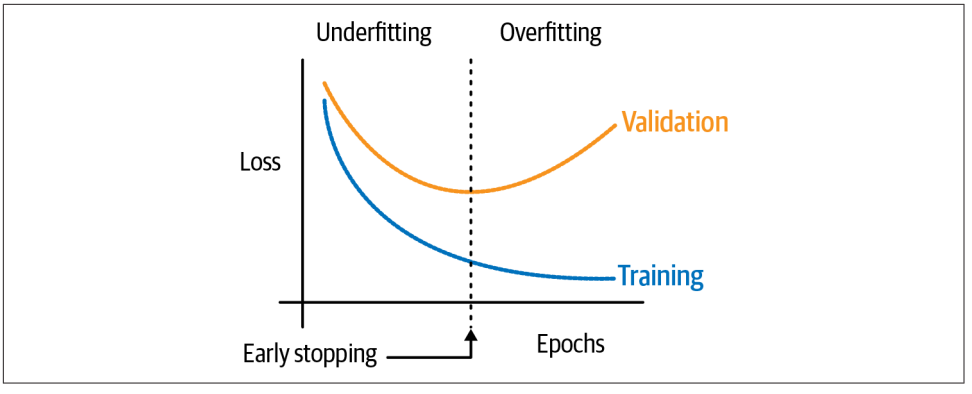


Figure 7-5. In overfitting, training loss keeps dropping, but validation loss rises

Regularization helps prevent this by nudging the model toward more general strategies. The main tools are listed here:

L1 (Lasso)
: Shrinks some weights all the way to 0, thus focusing only on the most decisive stats (for example, shots on target) and ignoring other noise

L2 (Ridge)
: Softens all weights a little, making sure no single statistic (say, possession percentage) dominates the analysis.

Dropout
: Randomly removes some neurons during training, forcing the network to find multiple paths to a solution rather than relying too heavily on any one set of activations

These techniques use a lot of mathematical detail. In practice, however, you'll rarely code them by hand, since PyTorch and TensorFlow have them built in. What matters is knowing why they exist: to keep your model from memorizing individual training examples at the expense of learning patterns that hold more generally.

Building Your First Soccer Prediction Model with PyTorch

PyTorch is a powerful open source framework designed for building and training neural networks. It is known for its user-friendly interface, robust GPU acceleration, and extensive library ecosystem, which is ideal for our soccer-related tasks (and many others too).

Binary Outcomes: Will the Home Team Win?

Let's begin with the simplest prediction task: estimating the probability that the home team wins. This is a binary classification problem. The model will look at pre-match features (form, shots, injuries, possession metrics, and so on) and output a number from 0 to 1 that we interpret as a win probability.

Our model will learn from historical match data to estimate the probability of a home win. To get there, we'll go through three steps: preparing the data, building the model architecture, and then training and evaluating it.

Preparing the data

Before we can train a model, we need to put our data into a format PyTorch understands. PyTorch provides a handy tool called the `DataLoader`, which takes in tensors and produces mini-batches for training. A mini-batch is just a small slice of the data. Rather than feeding the entire season of matches into the model at once, we present it with a handful of matches at a time, which makes training faster and helps the model generalize.

Here is an implementation that you can copy into your Jupyter notebook:

```
from sklearn.model_selection import train_test_split
import torch
from torch.utils.data import DataLoader, TensorDataset
import torch.nn as nn
import torch.optim as optim
# X: shape (n_samples, n_features), y: 0/1 (home win)
X = torch.tensor(X, dtype=torch.float32)
y = torch.tensor(
    y, dtype=torch.float32
)  # Float targets for BCE*
# Train-test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
# DataLoaders (mini-batches)
train_data = TensorDataset(X_train, y_train)
test_data = TensorDataset(X_test, y_test)
train_loader = DataLoader(
    train_data, batch_size=32, shuffle=True
)
test_loader = DataLoader(
    test_data, batch_size=32, shuffle=False
)
```

Let's break this down. First, we convert `X` and `y` into tensors, which are essentially PyTorch's version of arrays. The features (`X`) might include stats like average possession or number of corners, while the labels (`y`) are binary: `1` if the home team won, `0` if not.

Next, we split the dataset into a training set (80%) and a test set (20%). The training set is what the model learns from; the test set is held back to check how well the model generalizes to new matches it hasn't seen before.

Finally, we wrap the data in a `TensorDataset` and pass it to a `DataLoader`. During training, the `DataLoader` will give us small, shuffled batches of size `32`. Shuffling ensures that the model doesn't learn anything artificial from the order of the matches. In other words, we do not want the model to confuse sequence in the dataset with a real predictive signal.

Defining the model architecture

Now we need to define the neural network itself. For this task, we'll use a simple multilayer perceptron (MLP): one hidden layer with 16 neurons, followed by a single output neuron:

```
class MLP(nn.Module):
    def __init__(self, input_size, hidden_size=16):
        super().__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, 1)  # One logit
    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)  # Return logits
        return x
input_size = X_train.shape[1]
model = MLP(input_size, hidden_size=16)
criterion = nn.BCEWithLogitsLoss()  # Stable binary loss
optimizer = optim.Adam(model.parameters(), lr=1e-3)
```

This code defines an `MLP` class that inherits from `nn.Module`. This is standard practice in PyTorch because `nn.Module` provides a lot of useful functionality for training and evaluating neural networks.

Here's what happens after that:

- The input layer matches the number of features we provide. Each feature (shots, possession, xG) gets its own weight.
- The hidden layer has 16 neurons. These allow the model to learn intermediate patterns and combinations of features, such as a team that dominates possession and has a strong attack.
- The output layer is just one neuron, giving us a score (called a *logit*) that is turned into a win probability via a sigmoid function later.

We use BCE Loss (`BCEWithLogitsLoss`) as the objective. This loss punishes the model whenever its predicted probability is far from the actual result.

Finally, we choose the Adam optimizer. *Adam* is an adaptive optimization method that efficiently adjusts parameter updates during training, which is one reason it works well as a default choice in many deep learning problems.

Training the model

Now comes the learning process. Training is an iterative cycle where the model repeatedly sees batches of matches, makes predictions, compares them against reality, and adjusts its weights accordingly. This cycle is what allows it to gradually improve. Here is how we train the model:

```
num_epochs = 100
for epoch in range(num_epochs):
    model.train()
    running = 0.0
    for inputs, targets in train_loader:
        optimizer.zero_grad()
        logits = model(inputs).squeeze(1)  # (B,)
        loss = criterion(
            logits, targets
        )  # Targets float in {0,1}
        loss.backward()
        optimizer.step()
        running += loss.item()
    if (epoch + 1) % 10 == 0:
        print(
            f"Epoch {epoch+1:3d}/{num_epochs}  "
            f"Loss: {running/len(train_loader):.4f}"
        )
```

In this code, each epoch is one full pass through the training data. Inside each epoch, the inner loop goes batch by batch. For every mini-batch:

1. The model makes its best guess at who wins.
2. The loss function compares this guess to the actual result, producing a score that reflects how wrong the model was.
3. Backpropagation then works backward through the network, identifying which weights were responsible for the mistake.
4. The optimizer updates the weights slightly, nudging the model toward better predictions.

By repeating this cycle 100 times, the model gradually learns patterns that link pre-match features to actual outcomes. Printing the loss every 10 epochs lets us watch its

learning curve. We expect the numbers to go down as the model gradually improves its fit to the data.

Evaluating the model

Finally, we need to see how well the model generalizes to unseen matches. This is where the test set comes in:

```
model.eval()
with torch.no_grad():
    logits = model(X_test).squeeze(1)  # (N_test,)
    probs = torch.sigmoid(logits)  # Win probability
    preds = (probs >= 0.5).float()
# Quick accuracy (feel free to add ROC/AUC later)
acc = (preds == y_test).float().mean().item()
print(f"Test Accuracy: {acc:.3f}")
```

Here we switch the model into evaluation mode, which tells PyTorch to disable training-specific features like dropout. We also wrap the code in `torch.no_grad()`, which stops gradient tracking and makes evaluation faster.

The raw outputs are logits, so we apply a sigmoid function to convert them into probabilities. A value of 0.73, for example, means the model believes there is a 73% chance the home team wins this match. We then apply a threshold (0.5 by default) to turn those probabilities into binary predictions.

The final accuracy tells us the fraction of test matches that were predicted correctly. A decent accuracy here suggests that our model has learned useful general patterns, not just memorized the training set.

Wrapping up, we managed to answer a simple yes/no question: will the home team win? This is useful, but real soccer matches have more nuance. Many games end in a draw, and sometimes the away team steals the show. To capture this fuller picture, we now move from binary classification to a multiclass prediction task.

Match Results: Win, Draw, or Loss

Our next task is predicting the full-time result (`FTR`) of a soccer match, which can be a home win (`H`), draw (`D`), or away win (`A`). Unlike task 1, where we had to decide only win or not, we now face three possible outcomes. As you learned in Chapter 5, this is called a *multiclass classification problem*. The structure of the model is similar, but we need to make a few important adjustments to handle multiple classes.

Preparing the data

We prepare the data in much the same way as before: split it into training and test sets, convert it into tensors, and wrap it in a `DataLoader`. The key difference is that

the target variable is no longer binary but categorical. We'll reuse the same feature tensor `X`, but replace the target with integer class IDs derived from the `FTR` labels:

```
from sklearn.preprocessing import LabelEncoder
# Encode full-time results as integer class IDs
le = LabelEncoder()
y_multiclass = torch.tensor(
    le.fit_transform(df["FTR"]), dtype=torch.long,
)
X_train_mc, X_test_mc, y_train_mc, y_test_mc = train_test_split(
    X, y_multiclass, test_size=0.2, random_state=42
)
train_loader_mc = DataLoader(
    TensorDataset(X_train_mc, y_train_mc),
    batch_size=32,
    shuffle=True,
)
test_loader_mc = DataLoader(
    TensorDataset(X_test_mc, y_test_mc),
    batch_size=32,
    shuffle=False,
)
```

Here `y_multiclass` is now a vector of integers, each representing one of the three possible match outcomes. PyTorch's `CrossEntropyLoss`, which we'll use shortly, expects class indices in this format. The rest of the pipeline stays the same: we split the data and create mini-batches with a `DataLoader`.

Defining the model architecture

We now define a network that can output three scores instead of one. Each score corresponds to one possible result: home win, draw, or away win. Here's our code:

```
class MultiClassMLP(nn.Module):
    def __init__(
        self, input_size, hidden_size=16, num_classes=3
    ):
        super().__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(
            hidden_size, num_classes
        )  # Three outputs
    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)  # Logits for each class
        return x
input_size = X_train_mc.shape[1]
model = MultiClassMLP(input_size)
criterion = nn.CrossEntropyLoss()  # For multiclass problems
optimizer = optim.Adam(model.parameters(), lr=1e-3)
```

The biggest change from task 1 is the output layer. Instead of a single neuron with a sigmoid, we now have three neurons, one for each class. The model outputs raw scores (logits). These logits are automatically passed through a function inside `CrossEntropyLoss`, which turns them into probabilities that sum to 1. In soccer terms, the model is learning to distribute its confidence among the three possible outcomes.

Training the model

Training follows the same rhythm as before: forward pass, compute loss, backpropagate, and update weights. The difference is that our labels are now class indices, and our loss function expects logits for three classes:

```
num_epochs = 100
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    for inputs, labels in train_loader_mc:
        optimizer.zero_grad()
        outputs = model(inputs)  # Logits for 3 classes
        loss = criterion(
            outputs, labels
        )  # Compares logits to true class
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    if (epoch + 1) % 10 == 0:
        avg_loss = running_loss / len(train_loader_mc)
        print(
            f"Epoch {epoch+1:3d}/{num_epochs}  "
            f"Loss: {avg_loss:.4f}"
        )
```

What's happening here? For each mini-batch, the model produces three logits, one for each possible outcome. The loss function compares those logits with the true result, and backpropagation tells the model how to adjust. Over many epochs, the model refines its ability to tell which conditions tend to lead to which result.

Evaluating the model

Evaluation also looks similar, except now we're dealing with three classes. We'll measure accuracy, but also look at precision, recall, and the confusion matrix, which helps us see which results are most often misclassified:

```
from sklearn.metrics import (
    classification_report,
    confusion_matrix,
)
model.eval()
all_preds = []
```

```
all_labels = []
with torch.no_grad():
    for inputs, labels in test_loader_mc:
        outputs = model(inputs)
        _, predicted = torch.max(
            outputs, 1
        )  # Pick class with highest logit
        all_preds.extend(predicted.numpy())
        all_labels.extend(labels.numpy())
print(
    classification_report(
        all_labels, all_preds, target_names=le.classes_
    )
)
print(confusion_matrix(all_labels, all_preds))
```

Here we loop through the test set without gradients. The function call `torch.max(outputs, 1)` selects the class with the highest score, which is the model's prediction. By comparing these predictions with the true labels, we can compute detailed metrics.

The classification report tells us how well the model performs for each class individually. For instance, the model might be very good at spotting home wins but weaker at predicting draws. This is a common pattern in soccer because draws are less frequent and thus harder to model. The confusion matrix gives a visual map of where mistakes happen, such as predicting a home win when the match was actually a draw.

Summing up, we've now expanded from a simple yes/no prediction to a full three-way classification, capturing the richer variety of soccer outcomes. In the next task, we'll go one step further: instead of predicting categories like win or draw, we'll try to forecast the number of goals scored.

Goal Prediction: Modeling Scoring Output

In task 2, we have predicted categorical outcomes: win, draw, or loss. But soccer is not just about who wins. You also want to predict the number of goals scored, because this will help you more deeply understand game dynamics. Many bets are also placed on the number of goals scored.

Because goals are counts (nonnegative integers), we cannot simply use a standard linear regression model. Instead, we use a Poisson regression. It assumes that the number of goals scored follows a Poisson distribution, which works well for relatively rare events like goals in soccer.

Our target variable is the number of full-time home goals (`FTHG`):

```
y_home_goals = torch.tensor(
    df["FTHG"].values, dtype=torch.float32
)
(
```

```
    X_train_goals,
    X_test_goals,
    y_train_goals,
    y_test_goals,
) = train_test_split(
    X, y_home_goals, test_size=0.2, random_state=42
)
train_loader_goals = DataLoader(
    TensorDataset(X_train_goals, y_train_goals),
    batch_size=32,
    shuffle=True,
)
test_loader_goals = DataLoader(
    TensorDataset(X_test_goals, y_test_goals),
    batch_size=32,
    shuffle=False,
)
```

This gives us, for each match, a count target: 0, 1, 2, …, representing the number of goals the home team scored.

Defining the model architecture

Because the task of predicting goals is slightly more complex than of just predicting wins, we build a model with one more layer at this stage. This gives the model an extra chance to figure out the details.

To model counts in PyTorch, we need to ensure that the network's output is always positive (negative goals make no sense, after all). We do this by adding a `softplus` activation at the end of the network. This activation function is designed to guarantee outputs greater than 0. Here's how we implement a Poisson regression with a `softplus` activation:

```
class PoissonRegression(nn.Module):
    def __init__(self, input_size):
        super().__init__()
        self.fc1 = nn.Linear(input_size, 32)
        self.relu1 = nn.ReLU()
        self.fc2 = nn.Linear(32, 16)
        self.relu2 = nn.ReLU()
        self.fc3 = nn.Linear(16, 1)
        self.softplus = (
            nn.Softplus()
        )  # Ensures positive outputs
    def forward(self, x):
        x = self.fc1(x)
        x = self.relu1(x)
        x = self.fc2(x)
        x = self.relu2(x)
        x = self.fc3(x)
        return self.softplus(x)  # Rate parameter λ
```

Here's how to read this model:

- The hidden layers with ReLU activations learn complex combinations of features; for example, high possession paired with a strong attack might correspond to more goals.
- The final `softplus` layer outputs a positive number, which represents the λ (lambda) parameter of the Poisson distribution. This output is the expected number of goals.

You can think of λ as the model's expected scoring rate under similar match conditions. If λ = 1.8, we interpret that as *on average, the home team scores about 1.8 goals in matches like this.*

Training the model

Training a Poisson regression model uses a special loss function: the *Poisson negative log-likelihood* (`PoissonNLLLoss`). This loss is tailored for comparing predicted λ values with actual goal counts.

We also add early stopping, which halts training if the validation performance stops improving. This is the deep learning equivalent of deciding that more training is no longer improving general performance and may start to overfit the data. Here's how we train the model with `PoissonNLLLoss` and early stopping:

```
# Model initialization
input_size = X_train_goals.shape[1]
model = PoissonRegression(input_size)
criterion = nn.PoissonNLLLoss(log_input=False)
optimizer = optim.Adam(model.parameters(), lr=0.001)
# Training loop with validation
num_epochs = 200
best_val_loss = float("inf")
patience = 20
patience_counter = 0
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    # Training phase
    for inputs, targets in train_loader_goals:  ❶
        optimizer.zero_grad()
        outputs = model(inputs).squeeze()
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    # Validation phase
    model.eval()  ❷
    val_loss = 0.0
    with torch.no_grad():
```

```
        for inputs, targets in test_loader_goals:
            outputs = model(inputs).squeeze()
            val_loss += criterion(outputs, targets).item()
    # Early stopping check
    if val_loss < best_val_loss:  ❸
        best_val_loss = val_loss
        patience_counter = 0
    else:
        patience_counter += 1
    if patience_counter >= patience:
        print(f"Early stopping at epoch {epoch+1}")
        break
    # Logging every 20 epochs
    if (epoch + 1) % 20 == 0:
        train_loss = running_loss / len(train_loader_goals)
        val_loss = val_loss / len(test_loader_goals)
        print(f"Epoch {epoch+1:3d}/{num_epochs}")
        print(f"Training Loss:   {train_loss:.4f}")
        print(f"Validation Loss: {val_loss:.4f}")
```

This is a longer piece of code, so let's break it down step by step:

❶ Training phase: The model predicts an expected number of goals for each batch of matches. The Poisson loss compares these expectations against the actual counts and computes an error. Backpropagation then updates the weights to reduce this error.

❷ Validation phase: After each epoch, we check the model on unseen test matches. If validation loss starts rising while training loss falls, that is a red flag for overfitting.

❸ Early stopping: If the validation loss fails to improve for 20 epochs, training halts. This avoids *overtraining* the model on random quirks of the training set.

Interpreting the predictions

After training, the model can output expected goal values. If the model predicts $\lambda = 1.7$ for a given match, we interpret this as the home team averaging around 1.7 goals under similar conditions.

There is more power to this, however. The Poisson distribution allows us to compute probabilities of exact outcomes. For $\lambda = 1.7$, the probability of scoring exactly 2 goals is about 27%, of scoring 1 goal about 31%, and so on. This makes Poisson regression especially useful for applications like betting markets, where predicting the number of goals scored is often more valuable than predicting just who might win.

In summary, we have shifted from classifying outcomes to predicting goal counts. This opened the door to a different modeling paradigm that gives us deeper insight

into the way matches unfold. In the final task, we will go one step further: predicting both home and away goals simultaneously by using a shared multitask model.

Full Match Simulation: From Goals to Outcomes

So far, we have trained models to predict the following:

- Whether the home team wins (task 1)
- The match result as a home team win, an away team win, or a draw (task 2)
- The number of home goals (task 3)

But real soccer analysis often requires predicting both teams' goals together. After all, the away team also plays!

One option would be to train two separate models: one for home goals, and one for away goals. But this misses the mark because the two tasks are related. The same features (lineups, form, possession, xG, etc.) influence both outcomes. By learning them together, the model can share what it knows about soccer dynamics across both predictions. This is the core idea behind multitask learning (MTL).

Using multitask learning

Multitask learning means training a single network with shared layers and task-specific heads. *Shared layers* capture general soccer knowledge across prediction tasks; *task-specific heads* specialize in one of the two (or more, generally) prediction tasks.

This setup creates a model with these characteristics:

More efficient
: One model instead of two separate ones because the tasks are strongly related (e.g., whether the home team wins is highly dependent on the match result and the number of home goals).

Potentially more accurate
: It leverages shared patterns (e.g., strong attack for one side usually means weaker defense for the other).

More robust
: When data is limited, sharing information between tasks helps avoid overfitting.

Think of it like training a squad around shared team principles while still letting different units focus on specialized roles. The shared layers learn general match structure, while the task-specific heads focus on the details of home and away scoring. You can see the principle visualized in Figure 7-6.

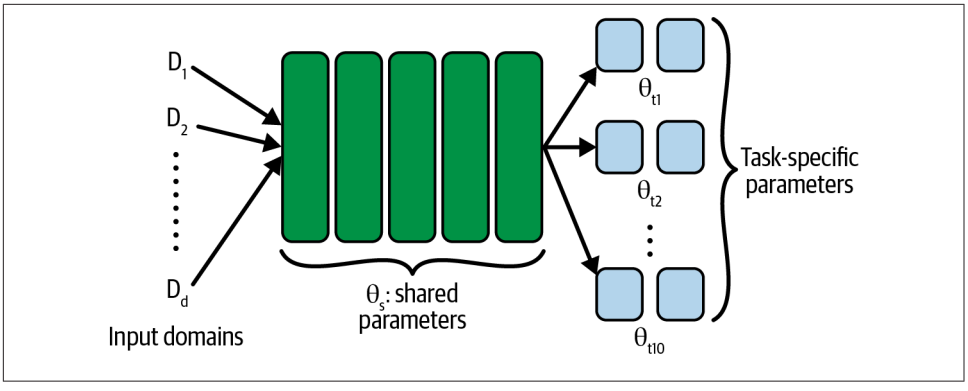


Figure 7-6. MTL architecture

Preparing the data

We reuse the same feature tensor `X`, but now we keep two targets, home goals and away goals:

```
y_home_goals = torch.tensor(
    df["FTHG"].values, dtype=torch.float32
)
y_away_goals = torch.tensor(
    df["FTAG"].values, dtype=torch.float32
)
(
    X_train_mtl,
    X_test_mtl,
    y_home_train_mtl,
    y_home_test_mtl,
    y_away_train_mtl,
    y_away_test_mtl,
) = train_test_split(
    X,
    y_home_goals,
    y_away_goals,
    test_size=0.2,
    random_state=42,
)
train_loader_mtl = DataLoader(
    TensorDataset(
        X_train_mtl, y_home_train_mtl, y_away_train_mtl
    ),
    batch_size=32,
    shuffle=True,
)
```

Defining the model architecture

Structurally, our MTL model reuses the same building blocks we have shown before: fully connected (linear) layers with ReLU activations, followed by output layers with `softplus` to keep goal predictions nonnegative. Here, however, we bring these pieces together in a slightly different ways:

```
class MultiTaskMLP(nn.Module):
    def __init__(self, input_size, hidden_size, shared_size):
        super().__init__()
        # Shared layers
        self.shared_layer1 = nn.Linear(
            input_size, hidden_size
        )  ❶
        self.shared_relu = nn.ReLU()
        self.shared_layer2 = nn.Linear(hidden_size, shared_size)
        # Home goals head
        self.home_goals_head = nn.Sequential(  ❷
            nn.Linear(shared_size, 32),
            nn.ReLU(),
            nn.Linear(32, 1),
            nn.Softplus(),  # <3> ensure nonnegative goals
        )
        # Away goals head
        self.away_goals_head = nn.Sequential(  ❷
            nn.Linear(shared_size, 32),
            nn.ReLU(),
            nn.Linear(32, 1),
            nn.Softplus(),  ❸
        )
    def forward(self, x):
        shared = self.shared_layer1(x)
        shared = self.shared_relu(shared)
        shared = self.shared_layer2(shared)
        home_goals = self.home_goals_head(shared)
        away_goals = self.away_goals_head(shared)
        return home_goals, away_goals
```

❶ The shared layers learn general match patterns—e.g., how possession, shots, and defense quality interact.

❷ The home head specializes in predicting home goals, while the away head specializes in away goals.

❸ The `softplus` activation function at the end of each head ensures that predictions are positive, just as in task 3.

Notice that the architecture is not deeper than the models we used in tasks 1 and 2—it has two shared hidden layers, just like those tasks. In task 3, we added an extra hidden layer, since predicting exact goal counts from scratch required a bit

more representational power. Now, however, we can get away with fewer layers again, because the learning burden is split across two tasks: home and away goals. By sharing the feature-extraction layers, the network reuses knowledge more efficiently, which reduces the need for additional depth.

The result is a compact, efficient model that reflects the interconnected nature of soccer itself: one match, two teams, and two goal tallies that are best understood together.

Training the model

Training a multitask network requires us to think slightly differently than before. We now have two parallel objectives, one for predicting home goals and one for predicting away goals. The training loop must balance both by improving one without neglecting the other.

Conceptually, this means the model is optimizing two related objectives at once. The two losses are added together into a total loss, which guides the optimizer to adjust the shared layers and task-specific heads simultaneously:

```
# Model initialization
input_size = X_train_mtl.shape[1]
hidden_size = 64
shared_size = 32
model = MultiTaskMLP(input_size, hidden_size, shared_size)
criterion = nn.PoissonNLLLoss(log_input=False)
optimizer = optim.Adam(model.parameters(), lr=0.001)
# Training loop
num_epochs = 200
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    for (
        inputs,
        home_targets,
        away_targets,
    ) in train_loader_mtl:  ❶
        optimizer.zero_grad()
        # Predictions for both tasks
        home_pred, away_pred = model(inputs)  ❷
        # Compute task-specific losses
        home_loss = criterion(
            home_pred.squeeze(), home_targets
        )  ❸
        away_loss = criterion(away_pred.squeeze(), away_targets)
        total_loss = home_loss + away_loss  ❹
        # Backpropagation
        total_loss.backward()
        optimizer.step()
        running_loss += total_loss.item()
```

```
    # Monitor every 20 epochs
    if (epoch + 1) % 20 == 0:
        model.eval()
        with torch.no_grad():
            home_pred, away_pred = model(X_test_mtl)
            home_mae = torch.mean(
                torch.abs(home_pred.squeeze() - y_home_test_mtl)
            )  ❺
            away_mae = torch.mean(
                torch.abs(away_pred.squeeze() - y_away_test_mtl)
            )
        print(f"Epoch {epoch+1:3d}/{num_epochs}")
        print(
            "Total Loss: "
            f"{running_loss/len(train_loader_mtl):.4f}"
        )
        print(f"Home Goals MAE: {home_mae:.4f}")
        print(f"Away Goals MAE: {away_mae:.4f}")
```

❶ Each batch contains both `home_targets` and `away_targets` true goal counts.

❷ The model makes predictions for both at once.

❸ We calculate two Poisson losses—one for each team.

❹ Adding them together ensures that the model improves in both tasks simultaneously.

❺ Monitoring metrics like mean absolute error (MAE) tells us how far off we are in terms of goals.

By the end of training, our network learns not only to estimate the number of goals the home side is likely to score, but also to balance that with away-side performance. The two need to go hand in hand because the overall scoreline makes sense only when both sides are modeled coherently.

This dual perspective matches the way scorelines are naturally understood in soccer. The two predictions are not isolated outputs, but parts of one underlying match picture.

Understanding why this matters

The advantage of multitask learning becomes clear when data is limited. In real life, data can be messy and complex, and we might well find ourselves without many examples of high-scoring away games. A separate away-goals model might struggle here. But in the multitask setup, the shared layers already capture goal-scoring dynamics from home data, which helps the away head make better predictions too.

In other words, sharing knowledge across tasks helps the model generalize better. This often makes it more data-efficient. This is a powerful idea not just in soccer analytics but across many machine learning applications.

With this last task, we have combined everything you've learned: binary and multiclass classification, regression for counts, and now multitask learning. We have gone from asking a simple yes/no question (will the home team win?) to building a model that can predict full scorelines—a more complete and demanding forecasting task.

Next, we'll rebuild these models from task 1 through 4 in TensorFlow. This process is somewhat more complex than in PyTorch, but because TensorFlow is widely used in industry and research, seeing both perspectives is valuable. If you're comfortable with PyTorch and want to keep moving, feel free to skip ahead; you can always come back to this section later. But if you'd like to broaden your skills now and understand how the same ideas translate into another ecosystem, this next section is well worth your time. The aim is not to double the conceptual burden, but to show how the same modeling logic maps onto a different framework.

A Second Lens: Implementing the Model in TensorFlow

In this section, we'll reimplement the same four tasks as before, this time using TensorFlow's Keras API. We'll thus once again build home-win classification, match result prediction, goal counts, and multitask scorelines.

At first glance, the models will look similar, but TensorFlow differs in philosophy. PyTorch favors explicit control (you write out the training loop step by step), whereas TensorFlow emphasizes concise abstractions: the `compile()` and `fit()` methods hide most of the boilerplate, making code shorter but sometimes harder to tweak. This difference means that TensorFlow can feel easier for quick prototypes and production deployment, while PyTorch often gives researchers more flexibility.

For the reader, this section serves two purposes. First, it shows how transferable your knowledge really is—the same soccer prediction tasks can be built in either framework with only minor changes. Second, it gives you a chance to see the trade-offs: TensorFlow's streamlined interface can speed you up, but PyTorch's hands-on style can offer more transparency.

To keep the manuscript concise, our TensorFlow tasks reuse the imports and standardized feature matrix from the first example. As you read, focus less on memorizing a second version of every code block and more on noticing the changes in workflow: model definition, training loops, and evaluation style.



TensorFlow (with Keras) is more concise, while PyTorch is more explicit and Pythonic. Somewhat paradoxically, this makes PyTorch easier for beginners to debug and learn from, whereas TensorFlow shines in fast prototyping and production deployment.

Binary Outcomes: Will the Home Team Win?

We'll now rebuild our binary classifier in TensorFlow using its Keras API. The task is the same: predict whether the home team will win, based on pre-match features. While the model architecture is identical to PyTorch's (one hidden layer with ReLU and a single output neuron), the style of implementation differs.

PyTorch had us write the training loop by hand; TensorFlow lets us configure everything up front and then calls `fit()` to run the loop internally. Both frameworks produce the same type of model, and with aligned hyperparameters, their accuracy and AUC will be nearly identical.

Model architecture

To implement our binary win-prediction model in TensorFlow, we define an input layer that matches the shape of our feature matrix, followed by a single hidden layer with 16 neurons and a ReLU activation function. For the output layer, we use a single neuron with a sigmoid activation, which naturally squashes the raw output into a probability from 0 to 1. Before defining the model, we also set up a `tf.data.Dataset` pipeline to handle shuffling and batching, which ensures that our data feeds efficiently into the training loop. Here's how we implement the binary classifier in TensorFlow:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
# Prepare datasets
train_ds = (
    tf.data.Dataset.from_tensor_slices(
        (X_train, y_train.astype("float32"))
    )
    .shuffle(1024)
    .batch(32)
)
test_ds = tf.data.Dataset.from_tensor_slices(
    (X_test, y_test.astype("float32"))
).batch(32)
# Define model
model = keras.Sequential(
    [
        layers.Input(shape=(X_train.shape[1],)),
        layers.Dense(16, activation="relu"),
        layers.Dense(
```

```
            1, activation="sigmoid"
        ),  # Probability output
    ]
)
```

This architecture mirrors the PyTorch model almost exactly:

- An input layer matching the number of features
- A hidden layer with 16 units and ReLU activation, giving the model nonlinear capacity
- An output layer with a single unit and sigmoid activation, which squashes outputs into the `[0,1]` range for binary probabilities



Another common pattern in TensorFlow is to leave off the sigmoid here and instead use `BinaryCrossentropy(from_logits=True)`. Both approaches are valid; the choice just shifts where the sigmoid is applied (in the layer or in the loss function).

Training and evaluation

Instead of writing the training loop, we *compile* the model once by specifying the optimizer, loss, and metrics. Keras then handles forward passes, backpropagation, parameter updates, and validation automatically. Here's our code:

```
model.compile(  ❶
    optimizer=keras.optimizers.Adam(1e-3),
    loss=keras.losses.BinaryCrossentropy(),
    metrics=["accuracy", keras.metrics.AUC(name="auc")],
)
early = keras.callbacks.EarlyStopping(  ❷
    monitor="val_auc",
    mode="max",
    patience=10,
    restore_best_weights=True,
)
history = model.fit(  ❸
    train_ds,
    epochs=100,
    validation_data=test_ds,
    callbacks=[early],
    verbose=2,
)
loss, acc, auc = model.evaluate(test_ds, verbose=0)  ❹
print(
    f"Test – loss: {loss:.4f}, "
    f"accuracy: {acc:.3f}, AUC: {auc:.3f}"
)
```

❶ `compile()` wires the optimizer (Adam), loss function (BCE), and evaluation metrics into the model.

❷ `EarlyStopping` monitors validation AUC, halting training if performance stops improving and restoring the best weights. This prevents overfitting.

❸ `fit()` runs the entire training process for the chosen number of epochs, shuffling the training data and checking validation performance after each epoch.

❹ `evaluate()` computes final metrics on the test set.

The workflow is more concise than PyTorch's loop, but less transparent because you don't see forward propagation, loss, backpropagation, and step cycles spelled out. The trade-off is thus convenience for less hands-on control.

Match Results: Win, Draw, or Loss

Just as in PyTorch, the shift from binary to multiclass mainly affects the output layer and the loss function.

Model architecture

Moving to multiclass classification requires only minor adjustments to our architecture. First, we must convert our categorical match outcomes (`Home Win`, `Draw`, `Away Win`) into integer class IDs via scikit-learn's `LabelEncoder`. We then build a Keras `Sequential` model with the same 16-neuron ReLU hidden layer as before.

The key difference lies in the output layer. We now need three neurons—one for each possible outcome—and we apply a `softmax` activation function. `Softmax` ensures that the three raw output scores are converted into a valid probability distribution that sums to exactly 1. Here's how to implement a multiclass classifier in TensorFlow:

```
from sklearn.preprocessing import LabelEncoder
# Convert categorical labels into integer class IDs
le = LabelEncoder()
y_multiclass = le.fit_transform(df["FTR"])
X_train_mc, X_test_mc, y_train_mc, y_test_mc = train_test_split(
    X, y_multiclass, test_size=0.2, random_state=42
)
train_ds = (
    tf.data.Dataset.from_tensor_slices(
        (X_train_mc, y_train_mc.astype("int32"))
    )
    .shuffle(1024)
    .batch(32)
)
test_ds = tf.data.Dataset.from_tensor_slices(
```

```
    (X_test_mc, y_test_mc.astype("int32"))
).batch(32)
# Define model
model = keras.Sequential(
    [
        layers.Input(shape=(X_train_mc.shape[1],)),
        layers.Dense(16, activation="relu"),
        layers.Dense(
            3, activation="softmax"
        ),  # 3 outcomes: H/D/A
    ]
)
```

The architecture is almost identical to that of task 1:

- An input layer sized to match the features
- A hidden layer of 16 ReLU neurons for nonlinear learning capacity
- An output layer with three units and a `softmax` activation, which turns raw scores into class probabilities that sum to 1

In PyTorch, we returned raw logits and left `softmax` to `CrossEntropyLoss`. In TensorFlow, we can specify `activation="softmax"` directly in the layer. Both approaches are equivalent; the choice is mostly a question of style.

Training and evaluation

We again compile the model with an optimizer, loss function, and metrics. For multiclass classification, the correct loss is categorical cross-entropy. Since our labels are integers (0, 1, 2), we use the sparse version:

```
model.compile(
    optimizer=keras.optimizers.Adam(1e-3),
    loss=keras.losses.SparseCategoricalCrossentropy(),  ❶
    metrics=["accuracy"],  ❷
)
history = model.fit(  ❸
    train_ds, epochs=100, validation_data=test_ds, verbose=2
)
loss, acc = model.evaluate(test_ds, verbose=0)
print(f"Test — loss: {loss:.4f}, accuracy: {acc:.3f}")
```

❶ The loss function, `SparseCategoricalCrossentropy`, compares predicted probabilities with integer class labels.

❷ The metric tracks how often the highest-probability prediction matches the true outcome.

❸ The training loop (`fit()`) runs automatically across epochs, while validation checks performance on unseen matches.

For predictions, TensorFlow's `predict()` returns a probability distribution over the three classes. We can use `argmax` to pick the most likely outcome:

```
probs = model.predict(X_test_mc)
pred_classes = probs.argmax(axis=1)
pred_labels = le.inverse_transform(
    pred_classes
)  # Back to H/D/A
```

With just a few lines of TensorFlow code, we now have a model that doesn't just say yes/no but produces a full probability distribution over win, draw, and loss. The elegance of the Keras workflow is that prediction, evaluation, and even label conversion all slot neatly into the same pipeline. This makes multiclass soccer outcomes as straightforward to model as binary ones.

Our next step is to predict the number of goals a team will score—a shift from classification to regression.

Goal Prediction: Modeling Scoring Output

In this task, we shift gears to predicting the number of goals scored by the home team—a regression problem with count data. Since goals are nonnegative integers, we use a Poisson regression approach with a neural network that outputs valid rate parameters.

Model architecture

For goal prediction, we shift from classification to regression. Because we are predicting counts (goals scored), our target variable is continuous but strictly nonnegative. We extract the home goals column as a float array and build a slightly deeper `Sequential` model to capture the complexity of scoring patterns.

For the activation function, we use `softplus`. This is a smooth approximation of ReLU that guarantees the model's output will always be positive, making it the mathematically correct choice for predicting Poisson rate parameters. Here's how to model home goals:

```
# Reuse the same feature matrix X, but switch to goal counts
y_home_goals = df["FTHG"].astype("float32").to_numpy()
(
    X_train_goals,
    X_test_goals,
    y_train_goals,
    y_test_goals,
) = train_test_split(
```

```
    X, y_home_goals, test_size=0.2, random_state=42
)
model = keras.Sequential(
    [
        layers.Input(shape=(X_train_goals.shape[1],)),
        layers.Dense(32, activation="relu"),
        layers.Dense(16, activation="relu"),
        layers.Dense(
            1, activation="softplus"
        ),  # Ensures positive outputs
    ]
)
```

The architecture builds two hidden layers with ReLU activations, giving the model capacity to learn complex scoring patterns. Just as in the PyTorch implementation earlier, the final output layer uses a `softplus` activation, which guarantees predictions are positive—a natural fit for goal counts.

Training and evaluation

Training largely mirrors earlier tasks but with a different loss function. Here we use the built-in Poisson loss, which penalizes the model based on how well the predicted rates match observed goal counts:

```
# Compile with Poisson loss
model.compile(
    optimizer=keras.optimizers.Adam(1e-3),
    loss=keras.losses.Poisson(),
    metrics=["mae"],
)
# Prepare datasets
train_ds = (
    tf.data.Dataset.from_tensor_slices(
        (X_train_goals, y_train_goals)
    )
    .shuffle(1024)
    .batch(32)
)
test_ds = tf.data.Dataset.from_tensor_slices(
    (X_test_goals, y_test_goals)
).batch(32)
# Early stopping to prevent overfitting
early = keras.callbacks.EarlyStopping(
    monitor="val_loss", patience=20, restore_best_weights=True
)
history = model.fit(
    train_ds,
    epochs=200,
    validation_data=test_ds,
    callbacks=[early],
    verbose=0,
)
```

```
loss, mae = model.evaluate(test_ds, verbose=0)
print(f"Test — loss: {loss:.4f}, MAE: {mae:.3f}")
```

The workflow is familiar: compile with the right loss, build efficient TensorFlow datasets, and let Keras manage training. Here we track MAE alongside the Poisson loss, which gives a more intuitive sense of average goal prediction error.

At the end of training, the model produces realistic goal predictions for the home team, providing richer insights than win/lose/draw probabilities alone.

In the final task, we predict home and away goals simultaneously, using a multitask learning approach.

Full Match Simulation: From Goals to Outcomes

Finally, we turn to the most complete modeling problem: predicting both home and away goals in a single framework. Rather than training two separate regressions, we use a multitask learning setup that shares knowledge between the two tasks. This way, the model can pick up patterns that affect both teams while still making distinct predictions for each side.

Model architecture

To predict both home and away goals simultaneously, we move beyond the simple `Sequential` API and use the Keras subclassing approach to build a multitask model. This architecture begins with a shared feature extractor—a `Sequential` block of dense layers that processes the input data to learn general match dynamics.

The output of this shared core is then passed into two separate, task-specific heads: one for predicting home goals and one for away goals. Each head is its own small `Sequential` network ending in a `softplus` activation to ensure nonnegative goal predictions. By structuring the model this way, the shared layers learn patterns common to both teams, while the individual heads fine-tune the final predictions for each side:

```
class MultiTaskMLP(keras.Model):
    def __init__(self, input_size, hidden_size, shared_size):
        super().__init__()
        # Shared feature extractor
        self.shared_layers = keras.Sequential(
            [
                layers.Dense(hidden_size, activation="relu"),
                layers.Dense(shared_size, activation="relu"),
            ]
        )
        # Task-specific heads
        self.home_head = keras.Sequential(
            [
                layers.Dense(32, activation="relu"),
                layers.Dense(
```

```
                    1, activation="softplus"
                ),  # Nonnegative outputs
            ]
        )
        self.away_head = keras.Sequential(
            [
                layers.Dense(32, activation="relu"),
                layers.Dense(1, activation="softplus"),
            ]
        )
    def call(self, x):
        features = self.shared_layers(x)
        home_pred = self.home_head(features)
        away_pred = self.away_head(features)
        return home_pred, away_pred
```

This design has three parts:

- Shared layers extract general match features useful for both teams (e.g., possession style, relative form).
- The `home_head` predicts goals for the home team.
- The `away_head` predicts goals for the away team.

By splitting into separate heads after a shared core, the model learns both the common and team-specific aspects of scoring.

Training and evaluation

To train this model, we give each output head its own Poisson loss, just as in task 3. In Keras, the safest way to do this is to explicitly name the outputs and let TensorFlow match each target to its corresponding prediction head:

```
# Prepare targets for both outputs
y_home_goals = df["FTHG"].astype("float32").to_numpy()
y_away_goals = df["FTAG"].astype("float32").to_numpy()
(
    X_train_mtl,
    X_test_mtl,
    y_home_train,
    y_home_test,
    y_away_train,
    y_away_test,
) = train_test_split(
    X,
    y_home_goals,
    y_away_goals,
    test_size=0.2,
    random_state=42,
)
# Model initialization
```

```
input_size = X_train_mtl.shape[1]
# Give the outputs explicit names
def build_multitask_model(
    input_size, hidden_size=64, shared_size=32
):
    inputs = keras.Input(shape=(input_size,))
    base_model = MultiTaskMLP(
        input_size, hidden_size, shared_size
    )
    home_pred, away_pred = base_model(inputs)
    home_pred = layers.Activation("linear", name="home_goals")(
        home_pred
    )
    away_pred = layers.Activation("linear", name="away_goals")(
        away_pred
    )
    return keras.Model(
        inputs=inputs,
        outputs={
            "home_goals": home_pred,
            "away_goals": away_pred,
        },
    )
model = build_multitask_model(
    input_size, hidden_size=64, shared_size=32
)
# Compile with one loss per output
model.compile(
    optimizer=keras.optimizers.Adam(1e-3),
    loss={
        "home_goals": keras.losses.Poisson(),
        "away_goals": keras.losses.Poisson(),
    },
    metrics={
        "home_goals": [
            keras.metrics.MeanAbsoluteError(name="mae")
        ],
        "away_goals": [
            keras.metrics.MeanAbsoluteError(name="mae")
        ],
    },
)
# TensorFlow datasets
train_ds = (
    tf.data.Dataset.from_tensor_slices(
        (
            X_train_mtl,
            {
                "home_goals": y_home_train,
                "away_goals": y_away_train,
            },
        )
```

```
        )
        .shuffle(1024)
        .batch(32)
    )
    test_ds = tf.data.Dataset.from_tensor_slices(
        (
            X_test_mtl,
            {"home_goals": y_home_test, "away_goals": y_away_test,},
        )
    ).batch(32)
    # Training with monitoring
    early = keras.callbacks.EarlyStopping(
        monitor="val_loss", patience=20, restore_best_weights=True
    )
    history = model.fit(
        train_ds,
        epochs=200,
        validation_data=test_ds,
        callbacks=[early],
        verbose=0,
    )
    # Evaluate
    results = model.evaluate(test_ds, verbose=0, return_dict=True)
    print(
        f"Home MAE: {results['home_goals_mae']:.3f}, "
        f"Away MAE: {results['away_goals_mae']:.3f}"
    )
```

During training, the shared layers capture global match dynamics, while each output head fine-tunes predictions for its respective team. The evaluation shows MAE for both home and away goals, thus giving a clear sense of accuracy on each side.

In practice, this setup provides a more realistic match simulation, as both predictions are generated together rather than in isolation.

Looking Ahead: From Flexible Models to Smarter Inputs

The deep learning models in this chapter represent the most expressive tools we have encountered so far. Neural networks do not require us to specify every interaction in advance. They can discover layered, nonlinear structure directly from data, and the multitask architecture we explored at the end of the chapter showed how related prediction targets can be learned together to mutual benefit. Yet throughout all four tasks, one thing remained constant: the model could work with only the features we gave it. A neural network with rich, well-constructed inputs will outperform one that is fed raw, poorly structured data, regardless of the number of layers it has. Modeling flexibility and input quality are not substitutes for each other—they compound.

In Chapter 8, we turn to that input problem directly, through the discipline of feature engineering. Rather than accepting raw match statistics as given, you will learn

how to translate soccer knowledge into structured predictive signals: rolling form windows that weight recent performance over distant history, venue-specific splits that separate home and away tendencies, shot-quality metrics built from xG rather than raw counts, and ranking-based team strength estimates using methods such as Colley ratings, PageRank, and Elo. We will also pay close attention to a discipline that is easy to overlook—avoiding data leakage, ensuring that every feature used to predict a match is constructed only from information that would have been available before that match was played. Done well, feature engineering is not just preprocessing. It is one of the most direct ways to make any model—including the deep learning models from this chapter—more accurate, more stable, and more interpretable.

Conclusion

This chapter has taken us beyond the world of simpler models and into a more expressive kind of soccer analytics. We have shown that deep learning is not just another way to predict match outcomes. It is a framework for modeling structure that is difficult to express directly: interactions among many variables, nonlinear relationships, and patterns that emerge only when context is considered as a whole.

Along the way, we moved from core concepts such as neurons, activation functions, loss, and optimization to practical modeling tasks. We built binary and multiclass classifiers, regression models for goal counts, and multitask models that learn related outputs together. These examples show what deep learning contributes to soccer analytics: not magic, but modeling flexibility when the problem becomes too rich for simpler assumptions.

That flexibility, however, does not remove the need for judgment. Deep models are only as useful as the data, targets, and modeling choices behind them. They do not replace soccer understanding; they amplify it when used carefully. The real value of this chapter is therefore not only that you have seen how to train these models, but that you now have a clearer sense of when deep learning is worth the added complexity and what kinds of questions it can help answer.

In Chapter 8, we turn to one of the most important foundations behind all these models: feature engineering. No matter how powerful a model is, it still depends on the quality and structure of the information we feed into it. You will learn how to design meaningful soccer features, build ranking systems for teams and players, and create inputs that make more advanced models—including deep learning models—far more effective.

CHAPTER 8

Feature Engineering for Soccer Analytics

In Chapter 7, we saw that even highly flexible models still depend on the quality and structure of their inputs. This chapter turns to that representation problem directly.

Exploratory data analysis (EDA) helps us understand what is in the data. Feature engineering is where we decide what the model should learn from it.

That distinction matters. Raw soccer data is rich, but it is rarely organized in the form that a model needs most. Match results tell us what happened, but not whether a team is improving. Event data tells us where actions occurred, but not which patterns are most important. A league table tells us who is ahead, but not how strong a team really is when schedule difficulty, venue, form, and shot quality are taken into account.

Feature engineering is the step where analysis becomes modeling. It is where we translate soccer knowledge into structured signals: recent form instead of season totals, venue-specific performance instead of generic averages, shot quality instead of shot counts, and team strength ratings instead of intuition alone. Done well, this is not just a technical cleanup step. It is one of the main ways we make a predictive model smarter, more stable, and more interpretable.

In soccer analytics, feature engineering often makes the difference between a model that is merely data-driven and one that is genuinely informative. Two teams may both score two goals per match, but one may be creating better shots, facing weaker opposition, and benefiting from extra rest. Those differences rarely appear in a single raw column. They have to be engineered.

In this chapter, we will build features that capture performance, context, and team strength from both match-level and event-level data. We will quantify recent form, home advantage, shot quality, defensive activity, fixture congestion, set-piece threat, and ranking-based measures such as Colley, PageRank, and Elo. By the end of the chapter, you will have a clearer sense of what makes a feature useful, how to construct

features without leaking future information, and how to turn raw soccer data into model-ready input.

Feature Engineering: From Raw Data to Predictive Signals

As we introduced at the end of Chapter 3, feature engineering is the crucial step where we translate soccer knowledge into structured signals. A model can learn only from the representation we give it. If that representation is weak, even a sophisticated algorithm will struggle. If it is strong, simpler models can already become surprisingly effective.

While we have already introduced foundational features like expected goals (xG), pass accuracy, and simple rolling averages in earlier chapters, this chapter goes much deeper. We will focus on *match-level predictions*—predicting the outcome of an upcoming game based on everything we know about the teams up to today.

Most of the features we'll create are at the team level and can be built from match results and event data. We'll organize them into four layers: *performance features* (what a team has produced), *tactical features* (how it plays), *contextual features* (when and under what conditions it played), and *advanced ranking algorithms* (how strong it appears relative to others).

Performance Features: The Fundamentals

Performance features are often the first layer of engineered information in soccer analytics. They summarize how teams have been performing and provide a compact view of recent attacking and defensive tendencies. Many of these statistics appear in broadcasts and post-match reports, but for modeling, we need to calculate them more carefully and align them with the prediction task.

In this section, we move from direct output metrics, to venue-sensitive splits, and then to shot-quality signals. Each subsection introduces one metric family and then turns it into a small, reusable code block.

Goals For and Goals Against: Measuring Scoring Performance

Goals remain the most direct summary of match performance. A team that has been scoring regularly in recent matches may be creating chances well and finishing efficiently, while a team that has stopped scoring may be facing tactical or finishing problems. The same logic applies defensively: recent goals conceded can tell us whether a team is controlling matches well or allowing too many dangerous opportunities.

The key point, however, is recency. Season totals can hide important changes in form. A team that scored heavily early in the season but has cooled off in recent weeks

should not be treated the same as a team that is improving now. For predictive work, recent performance is often more informative than distant history.

A *rolling window* gives us a practical way to capture that idea. Rather than averaging across the full season, it summarizes only the last *N* matches at each point in time. As new matches are played, the window moves forward and updates the feature accordingly.



Before building match-level features, it is a good idea to make sure `matches_df` is scoped to the competition and season you want to model. For example, if you are analyzing the 2019 Women's World Cup, you would filter to that tournament first rather than mixing in league or club matches. Keeping the scope consistent helps ensure that rolling form, home/away splits, and ranking features are all describing the same competitive context.

In the following code, we first reshape our data to get a long list of goals scored by each team in each match. Then we calculate rolling averages over 3-, 5-, and 10-match windows to capture short-, medium-, and long-term form. The resulting columns give us a clean starting layer of form features. Here's the implementation:

```
import pandas as pd
import numpy as np
# Goals per team per match  ❶
all_teams_scores = pd.concat(
    [
        matches_df[
            ["match_date", "home_team", "home_score"]
        ].rename(
            columns={"home_team": "team", "home_score": "goals"}
        ),
        matches_df[
            ["match_date", "away_team", "away_score"]
        ].rename(
            columns={"away_team": "team", "away_score": "goals"}
        ),
    ]
).sort_values(by=["team", "match_date"])
# Calculate 3-match rolling average  ❷
all_teams_scores[
    "goals_rolling_avg_3"
] = all_teams_scores.groupby("team")["goals"].transform(
    lambda x: x.rolling(window=3, min_periods=1)
    .mean()
    .shift(1)  ❸
)
# Also calculate 5-match and 10-match windows  ❹
for window in [5, 10]:
    all_teams_scores[
```

```
        f"goals_rolling_avg_{window}"
    ] = all_teams_scores.groupby("team")["goals"].transform(
        lambda x: x.rolling(window=window, min_periods=1)
        .mean()
        .shift(1)
    )
```

❶ Reshape the data into a long format with each row indicating one team's performance in one match. This makes it easy to calculate team-specific features. It's like creating a player-by-player stat sheet instead of a match-by-match one.

❷ The `rolling()` function creates a moving window over the data. Here, we're averaging the last three matches. Think of it as asking, "What's this team's average goals per game in their last three outings?"

❸ The `shift(1)` is absolutely crucial. It ensures that we're using only *past* matches to predict the current one. Without this, we'd be cheating by including the current match's result in our feature, which we wouldn't have at prediction time. This is called *data leakage*, and it's one of the most common mistakes in machine learning.

❹ We create multiple window sizes because different windows capture different aspects of form. A 3-match window is very sensitive to recent changes; if a team just won 3–0, 4–1, and 2–0, its 3-match average is high. But what if those three wins came after a terrible 10-match run? The 10-match average would reveal that longer-term struggle.

The beauty of multiple rolling windows is that you can capture both short-term hot streaks and long-term trends. In practice, you might create all three (or more) and let your machine learning model figure out which matters most for predictions. Maybe recent form matters more for predicting the next match, but longer-term form matters more for predicting tournament winners.

You can also calculate rolling *standard deviation* to measure consistency. A team with a low standard deviation is predictable. It scores roughly the same number of goals each match. A high standard deviation suggests volatility; the team might score 5 one week and 0 the next. Some teams are steady performers; others are wildly inconsistent. Both patterns are valuable information.

Venue Splits: Considering Home Versus Away

One of the most well-documented phenomena in soccer is home advantage. Teams simply perform better when playing in their own stadium, in front of their own fans, with familiar surroundings and no travel fatigue. The home advantage in soccer is worth roughly 0.3–0.5 goals per game on average, though it varies by team and league.

But here's the nuance: not all teams benefit equally from home advantage. Some teams are fortress-like at home but struggle on the road. Think of teams like Borussia Dortmund, whose Yellow Wall of fans creates an intimidating atmosphere. Other teams are more consistent regardless of venue; they have the mental strength and quality to perform anywhere.

When predicting Manchester United versus Manchester City, looking at United's home form and City's away form is more relevant than their overall averages. United might average 2 goals per game overall, but if they average 2.5 at home and 1.5 away, that's crucial information.

Let's calculate venue-specific statistics to capture these home and away performance differences. We'll compute separate averages for goals scored and conceded at home versus away, then create a home advantage metric. This turns one broad idea, home advantage, into a small set of explicit features the model can inspect:

```
# Calculate home-specific stats  ❶
home_stats = (
    matches_df.groupby("home_team")
    .agg(
        {
            "home_score": ["mean", "std"],
            "away_score": [
                "mean",
                "std",
            ],  # Goals conceded at home
        }
    )
    .reset_index()
)
home_stats.columns = [
    "team",
    "home_goals_avg",
    "home_goals_std",
    "home_conceded_avg",
    "home_conceded_std",
]
# Calculate away-specific stats  ❷
away_stats = (
    matches_df.groupby("away_team")
    .agg(
        {
            "away_score": ["mean", "std"],
            "home_score": [
                "mean",
                "std",
            ],  # Goals conceded away
        }
    )
    .reset_index()
)
```

```
away_stats.columns = [
    "team",
    "away_goals_avg",
    "away_goals_std",
    "away_conceded_avg",
    "away_conceded_std",
]
# Merge and calculate home advantage  ❸
team_stats = home_stats.merge(away_stats, on="team")
team_stats["home_advantage"] = (
    team_stats["home_goals_avg"] - team_stats["away_goals_avg"]
)
team_stats["home_defensive_advantage"] = (
    team_stats["away_conceded_avg"]
    - team_stats["home_conceded_avg"]
)
```

❶ Use `groupby()` and `agg()` to calculate statistics separately for home matches. Look at both goals scored and goals conceded (which is the opponent's score).

❷ Do the same for away matches. Notice how we flip the perspective—when analyzing away performance, the opponent's score (`home_score`) is what we conceded.

❸ The `home_advantage` metric is the difference between home and away goal-scoring averages. A positive value means the team scores more at home; a negative value (rare but possible) means it's better away. We also calculate a defensive home advantage because some teams are particularly solid at home defensively.

You can extend this further by calculating venue-specific rolling averages. Instead of just goals in last three matches, you can have goals in last three home matches and goals in last three away matches. This gives you the most granular view of form by venue.

Shot Quality: Evaluating Chance Creation

Here's a scenario: Team A takes 20 shots and scores 2 goals. Team B takes 10 shots and scores 2 goals. Who played better? The naive answer is team A, which had more shots. But what if team A's 20 shots were all from 30 yards out with defenders in the way, while team B's 10 shots were all from inside the penalty area with clear looks at goal?

This is where xG comes in. This metric, which you first started working with in Chapter 3, estimates the probability that a shot will result in a goal based on factors like distance from goal, angle, type of assist, and defensive pressure. A shot from 6 yards out in the center of the goal might have an xG of 0.7 (70% chance of scoring). A shot from 30 yards out might have an xG of 0.03 (3% chance).

StatsBomb provides xG values for every shot in its dataset, calculated using a sophisticated model trained on thousands of historical shots. Let's use these values to create shot-quality features.



The event data should be filtered to the same competition and season as `matches_df`. If your match table is limited to one tournament but `events_df` still contains events from several competitions, your feature calculations can quietly become inconsistent. In practice, it is safest to rebuild or refetch `events_df` from the same set of match IDs used in `matches_df`.

Our code produces three closely related outputs—total xG, average xG per shot, and simple finishing-overperformance measures:

```
# Get shots from event data  ❶
shots = events_df[events_df["type"] == "Shot"].copy()
# Aggregate xG by match and team  ❷
xg_features = (
    shots.groupby(["match_id", "team"])
    .agg({"shot_statsbomb_xg": ["sum", "mean", "count"]})
    .reset_index()
)
xg_features.columns = [
    "match_id",
    "team",
    "total_xg",
    "avg_xg_per_shot",
    "shots",
]
# Add shot conversion rate  ❸
goals = (
    shots[shots["shot_outcome"] == "Goal"]
    .groupby(["match_id", "team"])
    .size()
)
xg_features = xg_features.merge(
    goals.rename("goals"), on=["match_id", "team"], how="left"
).fillna(0)
xg_features["xg_overperformance"] = (
    xg_features["goals"] - xg_features["total_xg"]
)  ❹
xg_features["shot_conversion_rate"] = xg_features["goals"] / (
    xg_features["shots"] + 1
)  ❺
```

❶ Filter the event data to just shots. This is where the granular data really shines; we're going from match-level summaries to action-level detail.

❷ Aggregate xG values to the match-team level: `sum` gives us total xG (the number of goals we'd expect based on shot quality), `mean` gives us average quality per shot (are they taking good shots?), and `count` gives us shot volume.

❸ We count actual goals scored and merge them in. This lets us compare expected versus actual performance.

❹ The `xg_overperformance` metric tells us whether a team is scoring more or fewer goals than expected. Consistent overperformance might indicate clinical finishing (like a team with world-class strikers). Consistent underperformance might indicate poor finishing or facing hot goalkeepers. Importantly, xG overperformance tends to regress to the mean over time; a team that's been lucky (or unlucky) will likely return to expected performance.

❺ Shot conversion rate is the classic metric: goals divided by shots. We add 1 to the denominator to avoid division by zero for teams with no shots in a match.

You can also break down shots by location (inside the box versus outside), by type (open play versus set piece), and by body part (foot versus header). Each of these dimensions adds nuance to your understanding of a team's attacking threat.

Tactical Features: Describing How Teams Play

Performance features tell us what teams produced. Tactical features add the next layer by describing how they created or prevented those outcomes. This section focuses on style, pressure, and set-piece behavior, again keeping each metric family paired closely with its implementation.

Passing and Possession: Measuring Ball Control

Soccer is fundamentally about controlling the ball. Teams that dominate possession tend to control the tempo of the game, dictate where play happens, and create more chances. But possession alone doesn't win matches. Players need to do something productive with the ball.

Some teams dominate possession with short, intricate passing (think Barcelona's tiki-taka or Spain's national team). Others play more direct, using longer passes to quickly transition from defense to attack (think of counterattacking teams like Leicester City's 2015–16 title winners). Both styles can be effective, but they require different features to capture.

Let's extract passing metrics from the event data to quantify these different playing styles. We'll calculate pass success rates and average pass distances, and identify

progressive passes that move the ball significantly forward. That gives us a compact passing profile for each team in each match:

```
# Get passes from event data ❶
passes = events_df[events_df["type"] == "Pass"].copy()
# Calculate pass success ❷
# Completed StatsBomb passes have NaN outcome.
passes["pass_success"] = (
    passes["pass_outcome"].isna().astype(int)
)
# Extract pass coordinates once ❸
passes["x_start"] = passes["location"].apply(
    lambda loc: loc[0] if isinstance(loc, list) else 0
)
passes["y_start"] = passes["location"].apply(
    lambda loc: loc[1] if isinstance(loc, list) else 0
)
passes["x_end"] = passes["pass_end_location"].apply(
    lambda loc: loc[0] if isinstance(loc, list) else 0
)
passes["y_end"] = passes["pass_end_location"].apply(
    lambda loc: loc[1] if isinstance(loc, list) else 0
)
# Calculate pass distance ❹
passes["pass_distance"] = np.sqrt(
    (passes["x_end"] - passes["x_start"]) ** 2
    + (passes["y_end"] - passes["y_start"]) ** 2
)
# Identify progressive passes ❺
# Assume teams attack left to right, so larger
# x-values are closer to the opponent's goal.
passes["progressive_pass"] = (
    (passes["x_end"] - passes["x_start"]) > 10
).astype(int)
# Aggregate passing features ❻
pass_features = (
    passes.groupby(["match_id", "team"])
    .agg(
        {
            "pass_success": "mean",
            "pass_distance": ["mean", "std"],
            "progressive_pass": "sum",
            "pass_goal_assist": "sum",
        }
    )
    .reset_index()
)
pass_features.columns = [
    "match_id",
    "team",
    "pass_accuracy",
    "avg_pass_length",
```

```
    "std_pass_length",
    "progressive_passes",
    "assist_passes",
]
```

❶ Filter to pass events. A typical match might have 500–800 passes per team, so this is a rich data source.

❷ StatsBomb's data structure: successful passes have `NaN` for `pass_outcome`, while failed passes have values like `Incomplete` or `Out`. We convert this to a binary success indicator.

❸ Extract the start and end coordinates once so we can reuse them in multiple features.

❹ Use the Pythagorean theorem to calculate the Euclidean distance of each pass. StatsBomb uses a coordinate system with the pitch 120 units long and 80 units wide.

❺ A progressive pass is one that moves the ball significantly toward the opponent's goal. We define this as moving the ball forward by more than 10 units. This simplified example assumes attacking direction has already been normalized so that teams move from left to right in the coordinate system. If your dataset includes teams attacking in both directions, you should normalize the coordinates first before applying this rule. This is a simplified version of more sophisticated progressive pass metrics, but it captures the key idea: not all passes are equal. A 5-yard sideways pass is different from a 20-yard pass that splits the defense.

❻ Aggregate all these pass-level features up to the match-team level. Pass accuracy and average pass length together give us a sense of a team's style. High accuracy with short passes? Possession-based. Longer passes with lower accuracy? More direct. Here, `assist_passes` is a simple count of passes that directly assisted goals.

You can also count *key passes* (passes that lead directly to a shot) and *assists* (passes that lead directly to a goal). These capture the creative output of a team's playmakers. A team with high key passes but low goals might be creating chances but struggling to finish; it needs a better striker. A team with low key passes but high goals might be very clinical but not creating much; it might struggle against better defenses.

Offense wins games, but defense wins championships. Yet defensive metrics often get less attention than attacking ones. Let's fix that. A solid defense is about more than just preventing goals. It's about winning the ball back, disrupting the opponent's play, and starting your own attacks.

Tackles, Interceptions, and Recoveries: Tracking Defensive Actions

Defensive actions come in several flavors. *Tackles* are direct challenges for the ball. *Interceptions* are reading the play and cutting out a pass. *Recoveries* are regaining possession of a loose ball. Each tells us something different about a team's defensive style. Our code keeps those components separate and then combines them into a single defensive-intensity feature:

```
# Get defensive actions ❶
duels = events_df[events_df["type"] == "Duel"].copy()
interceptions = events_df[
    events_df["type"] == "Interception"
].copy()
ball_recoveries = events_df[
    events_df["type"] == "Ball Recovery"
].copy()
# Aggregate defensive actions ❷
teams = pd.unique(
    matches_df[["home_team", "away_team"]].values.ravel("K")
)
defensive_frames = []
for team in teams:
    team_duels = (
        duels[duels["team"] == team].groupby("match_id").size()
    )
    team_interceptions = (
        interceptions[interceptions["team"] == team]
        .groupby("match_id")
        .size()
    )
    team_recoveries = (
        ball_recoveries[ball_recoveries["team"] == team]
        .groupby("match_id")
        .size()
    )
    match_ids = team_duels.index.union(
        team_interceptions.index
    ).union(team_recoveries.index)
    defensive_frames.append(
        pd.DataFrame(
            {
                "team": team,
                "match_id": match_ids,
                "duels_per_match": team_duels.reindex(
                    match_ids, fill_value=0
                ).values,
                "interceptions_per_match": (
                    team_interceptions.reindex(
                        match_ids, fill_value=0
                    ).values
                ),
                "recoveries_per_match": (
```

```
                    team_recoveries.reindex(
                        match_ids, fill_value=0
                    ).values
                ),
            }
        )
    )
defensive_features = pd.concat(
    defensive_frames, ignore_index=True
)
# Calculate defensive intensity ❸
defensive_features["defensive_actions_total"] = (
    defensive_features["duels_per_match"]
    + defensive_features["interceptions_per_match"]
    + defensive_features["recoveries_per_match"]
)
```

❶ Extract different types of defensive actions from the event data. Each type reveals a different defensive approach.

❷ Aggregate these actions by team and match. This gives us a count of the defensive actions each team performed in each match.

❸ Create a composite defensive intensity metric by summing all defensive actions. A high number suggests an aggressive, pressing style. A low number might suggest a team that sits deep and doesn't challenge for the ball as much.

A team that averages 20 tackles per game is very different from one that averages 10. The high-tackling team is probably pressing aggressively, trying to win the ball high up the pitch. The low-tackling team might be playing a more passive, sit-deep-and-absorb-pressure style. Neither is inherently better; it depends on the opponent and the game situation.

Pressing Intensity: Applying Defensive Pressure

One of the most sophisticated defensive metrics is *passes per defensive action* (*PPDA*). This measures the number of passes a team allows the opponent to make before attempting a defensive action (tackle, interception, or challenge). A low PPDA means the team is pressing aggressively and not letting the opponent string many passes together. A high PPDA means the team is sitting back, allowing the opponent to have the ball. This gives us one concise pressing feature per team per match. Here's how we add PPDA to our dataset:

```
# Calculate PPDA for each team in each match ❶
ppda_features = []
for match_id in matches_df["match_id"].unique():
    match_events = events_df[events_df["match_id"] == match_id]
    home_team = matches_df[matches_df["match_id"] == match_id][
```

```
    "home_team"
].iloc[0]
away_team = matches_df[matches_df["match_id"] == match_id][
    "away_team"
].iloc[0]
# Count opponent passes in defensive third  ❷
away_passes_in_home_third = match_events[
    (match_events["team"] == away_team)
    & (match_events["type"] == "Pass")
    & (
        match_events["location"].apply(
            lambda x: x[0] if isinstance(x, list) else 0
        )
        < 40
    )
].shape[0]
home_passes_in_away_third = match_events[
    (match_events["team"] == home_team)
    & (match_events["type"] == "Pass")
    & (
        match_events["location"].apply(
            lambda x: x[0] if isinstance(x, list) else 0
        )
        > 80
    )
].shape[0]
# Count defensive actions  ❸
home_def_actions = match_events[
    (match_events["team"] == home_team)
    & (
        match_events["type"].isin(
            ["Duel", "Interception", "Ball Recovery"]
        )
    )
].shape[0]
away_def_actions = match_events[
    (match_events["team"] == away_team)
    & (
        match_events["type"].isin(
            ["Duel", "Interception", "Ball Recovery"]
        )
    )
].shape[0]
# Calculate PPDA  ❹
home_ppda = away_passes_in_home_third / (
    home_def_actions + 1
)
away_ppda = home_passes_in_away_third / (
    away_def_actions + 1
)
ppda_features.append(
    {
```

```
            "match_id": match_id,
            "home_team": home_team,
            "home_ppda": home_ppda,
            "away_team": away_team,
            "away_ppda": away_ppda,
        }
    )
ppda_df = pd.DataFrame(ppda_features)
```

❶ Calculate PPDA for each team in each match. This requires looking at both teams' actions in the same match.

❷ Count the passes the opponent made in the defensive third (the area closest to our goal). In StatsBomb's coordinate system, the defensive third for the home team is $x < 40$.

❸ Count the defensive actions the team made in that same area.

❹ PPDA is the ratio of opponent passes to the defensive actions. A PPDA of 5 means the team allowed 5 passes per defensive action. A PPDA of 15 means that 15 passes per defensive action happened—the defenders are sitting back more.

Teams like Liverpool under Jürgen Klopp have very low PPDA values (around 7–9), indicating their famous *gegenpressing* style. Teams that sit deep and counterattack might have PPDA values of 15–20.

Set Pieces: Capturing Dead-Ball Situations

Set pieces—corners, free kicks, and penalties—often decide tight matches. They're a completely different phase of play from open play, with different tactics and different players involved. A team might struggle to create chances in open play but be deadly from corners. Let's capture this. The goal is to engineer three related outputs—set-piece xG, set-piece goals, and a simple set-piece conversion measure:

```
# Normalize set-piece labels across event types  ❶
set_piece_map = {
    "Corner": "Corner",
    "From Corner": "Corner",
    "Free Kick": "Free Kick",
    "From Free Kick": "Free Kick",
    "Penalty": "Penalty",
    "From Penalty": "Penalty",
}
# Identify set-piece situations in the event data  ❷
set_pieces = events_df[
    events_df["pass_type"].isin(["Corner", "Free Kick"])
    | events_df["type"].isin(["Corner", "Free Kick", "Penalty"])
].copy()
```

```
set_pieces["set_piece_type"] = (
    set_pieces["pass_type"]
    .fillna(set_pieces["type"])
    .map(set_piece_map)
)
# Map shot play patterns to set-piece types  ❸
shots["set_piece_type"] = shots["play_pattern"].map(
    set_piece_map
)
set_piece_shots = shots[
    shots["set_piece_type"].isin(
        set_pieces["set_piece_type"].dropna().unique()
    )
].copy()
# Aggregate set-piece features  ❹
set_piece_features = (
    set_piece_shots.groupby(["match_id", "team"])
    .agg(
        {
            "shot_statsbomb_xg": "sum",
            "shot_outcome": lambda x: (x == "Goal").sum(),
        }
    )
    .reset_index()
)
set_piece_features.columns = [
    "match_id",
    "team",
    "set_piece_xg",
    "set_piece_goals",
]
# Calculate set-piece conversion  ❺
set_piece_features["set_piece_conversion"] = set_piece_features[
    "set_piece_goals"
] / (set_piece_features["set_piece_xg"] + 0.1)
```

❶ First normalize the labels because StatsBomb may use values such as `Corner` for a pass event but `From Corner` for the resulting shot's `play_pattern`. Without this step, equivalent set piece situations can be missed.

❷ Identify set-piece events in the data. StatsBomb tags passes and shots with their play pattern (open play, corner, free kick, etc.).

❸ Tag shots with the same normalized labels. This avoids a many-to-many merge that could otherwise duplicate shots and inflate xG or goal totals.

❹ Aggregate features to find xG and the goals each team generated from set pieces.

❺ Set-piece conversion indicates whether a team is clinical from set pieces. Some teams have tall, strong players who are excellent in the air; they'll have high set piece conversion. Others might win lots of corners but struggle to convert them.

You can also look at set-piece *defense*—how many set pieces a team concedes and how often they lead to goals. A team that's vulnerable to set pieces might have a short backline or poor zonal marking.

Contextual Features: Incorporating Match Conditions

Raw performance metrics are great, but context matters enormously in soccer. A team playing its third match in seven days is likely fatigued. A team missing its star striker is weakened. A team playing in heavy rain might struggle with its usual passing game. Let's capture these contextual factors.

Rest and Scheduling: Modeling Fatigue and Congestion

In tournaments like the World Cup, the schedule can be brutal. Teams might play every three or four days, with little time to recover. In domestic leagues, top teams might be playing in multiple competitions—league, cup, and European tournaments—leading to fixture congestion. Fatigue is real, and it affects performance. Our code turns that intuition into three concrete features—`days_rest`, `matches_played`, and `matches_in_last_14_days`:

```
# Calculate days since last match  ❶
matches_df["match_date"] = pd.to_datetime(
    matches_df["match_date"]
)
team_matches = pd.concat(
    [
        matches_df[
            ["match_date", "match_id", "home_team"]
        ].rename(columns={"home_team": "team"}),
        matches_df[
            ["match_date", "match_id", "away_team"]
        ].rename(columns={"away_team": "team"}),
    ]
).sort_values(["team", "match_date"])
team_matches["days_rest"] = (
    team_matches.groupby("team")["match_date"].diff().dt.days
)  ❷
team_matches["matches_played"] = (
    team_matches.groupby("team").cumcount() + 1
)  ❸
# Calculate matches in last 14 days (fixture congestion)  ❹
team_matches["matches_in_last_14_days"] = (
    team_matches.groupby("team")
    .rolling("14D", on="match_date")["match_id"]
```

```
        .count()
        .reset_index(level=0, drop=True)
        .to_numpy()
    )
```

❶ Create a long-form DataFrame with each row indicating one team's participation in one match.

❷ Calculate the time difference between consecutive matches for each team. A value of 3 means three days of rest. A value of 7 means a full week.

❸ The cumulative count of matches is a simple proxy for cumulative fatigue. A team playing its sixth match of a tournament is likely more tired than one playing its second, even if the players have the same rest between matches.

❹ Count the matches a team has played in the last 14 days. If it's four or five, that's serious fixture congestion. If it's one or two, players are relatively fresh.

Research has shown that teams with fewer than three days of rest between matches show measurable declines in high-intensity running, sprint speed, and passing accuracy. This is quantifiable fatigue, and it affects results.

Form and Momentum: Capturing Recent Performance Trends

Beyond just counting wins, we can capture momentum—whether a team is on a hot streak or a cold one. Momentum is a controversial concept in sports analytics (some argue it doesn't exist, that it's just randomness), but most coaches and players believe in it. A team that's won five games in a row is playing with confidence. A team that's lost five in a row is demoralized. This implementation converts that idea into rolling points and recent win-rate features:

```
# Create a long-form results DataFrame  ❶
results = pd.concat(
    [
        matches_df[
            [
                "match_date",
                "match_id",
                "home_team",
                "home_score",
                "away_score",
            ]
        ].assign(
            team=matches_df["home_team"],
            goals_for=matches_df["home_score"],
            goals_against=matches_df["away_score"],
        ),
        matches_df[
```

```
            [
                "match_date",
                "match_id",
                "away_team",
                "home_score",
                "away_score",
            ]
        ].assign(
            team=matches_df["away_team"],
            goals_for=matches_df["away_score"],
            goals_against=matches_df["home_score"],
        ),
    ]
)[
    [
        "match_date",
        "match_id",
        "team",
        "goals_for",
        "goals_against",
    ]
].sort_values(
    ["team", "match_date"]
)
# Determine match results ❷
results["result"] = np.where(
    results["goals_for"] > results["goals_against"],
    "W",
    np.where(
        results["goals_for"] < results["goals_against"],
        "L",
        "D",
    ),
)
# Calculate points (3 for win, 1 for draw, 0 for loss) ❸
results["points"] = results["result"].map(
    {"W": 3, "D": 1, "L": 0}
)
results["points_last_3"] = results.groupby("team")[
    "points"
].transform(
    lambda x: x.shift(1).rolling(3, min_periods=1).sum()
)
results["points_last_5"] = results.groupby("team")[
    "points"
].transform(
    lambda x: x.shift(1).rolling(5, min_periods=1).sum()
)
# Calculate win percentage ❹
results["is_win"] = (results["result"] == "W").astype(int)
results["win_pct_last_5"] = results.groupby("team")[
    "is_win"
```

```
].transform(
    lambda x: x.shift(1).rolling(5, min_periods=1).mean()
)
```

❶ Create a long-form DataFrame with each row indicating one team's result in one match.

❷ Create a simple result indicator: `W` for win, `L` for loss, `D` for draw.

❸ Calculate points over rolling windows. This is the standard soccer metric that fans and teams are familiar with. A team with 15 points in its last five matches has won all five ($5 \times 3 = 15$). A team with 0 points has lost all five. A team with 7 or 8 points has had mixed results.

❹ To calculate win percentage robustly, first create a numeric win indicator (`1` for a win, `0` otherwise). Taking a rolling mean of that binary indicator gives us the proportion of recent matches that were wins. A team with a 60% win rate in its last five matches is performing well. A team with a 0% win rate is in crisis.

You can also calculate *goal difference* over rolling windows. A team with a +10 goal difference in its last five matches is not just winning—it's dominating. A team with a –8 goal difference is not just losing—it's getting hammered.

Ranking Features and Advanced Ranking Algorithms: Estimating Team Strength

Simple metrics like goals scored and win percentage are useful, but they don't tell the whole story. A team that beats three top-5 opponents is stronger than a team that beats three bottom-5 opponents, even if both have three wins. To capture this, we need sophisticated ranking algorithms that account for *strength of schedule*.

Think of it as creating a dynamic league table. Instead of just getting 3 points for a win, you get more credit for beating a stronger team. In this section, we'll explore three algorithms: the Colley matrix, PageRank, and Elo ratings. Each brings a unique perspective to the challenge of ranking teams. Unlike in the earlier sections, each code block here returns a reusable team rating table that we can later merge into a common strength layer.

The Colley Matrix: A Linear Algebra Approach

The *Colley matrix*, developed by Wesley Colley in 2002 for ranking American college football teams, is one of the most elegant ranking systems. It's based on a beautiful piece of linear algebra that sets up a system of equations where each team's rating depends on its win-loss record *and* the ratings of the teams it has played against. It then solves these equations simultaneously to find ratings that are consistent with all the results.

The intuition is simple: your rating should be high if you win a lot, but it should be *even higher* if you win against strong opponents. Conversely, losing to a weak opponent should hurt your rating more than losing to a strong opponent.

The mathematical foundation is the matrix equation $Cr = b$:

- C is the Colley matrix (representing the schedule—who played whom)
- r is the vector of ratings we're solving for
- b is the results vector (representing wins and losses)

The beauty is that this system of equations has a unique solution that we can find through linear algebra. Here's how we implement it for the Women's World Cup:

```
import numpy as np
def calculate_colley_ratings(matches_df):
    teams = pd.unique(
        matches_df[["home_team", "away_team"]].values.ravel("K")
    )  ❶
    team_to_idx = {team: idx for idx, team in enumerate(teams)}
    colley_matrix = np.zeros((len(teams), len(teams)))  ❷
    results_vector = np.zeros(len(teams))
    for _, row in matches_df.iterrows():  ❸
        home_idx = team_to_idx[row["home_team"]]
        away_idx = team_to_idx[row["away_team"]]
        colley_matrix[home_idx, home_idx] += 1
        colley_matrix[away_idx, away_idx] += 1
        colley_matrix[home_idx, away_idx] -= 1
        colley_matrix[away_idx, home_idx] -= 1
        if row["home_score"] > row["away_score"]:
            results_vector[home_idx] += 1
            results_vector[away_idx] -= 1
        elif row["away_score"] > row["home_score"]:
            results_vector[away_idx] += 1
            results_vector[home_idx] -= 1
    np.fill_diagonal(
        colley_matrix, colley_matrix.diagonal() + 2
    )  ❹
    ratings = np.linalg.solve(
        colley_matrix, results_vector
    )  ❺
```

```
        return pd.Series(
            ratings, index=teams, name="colley_rating"
        ).sort_values(ascending=False)
    colley_df = (
        calculate_colley_ratings(matches_df)
        .rename_axis("team")
        .reset_index()
    )
```

❶ Create a mapping from team names to matrix indices (0, 1, 2, …). This is essential for our NumPy implementation. We need to translate team names to row/column positions in our matrices.

❷ Initialize the Colley matrix `C` and results vector `b` with 0s. The matrix will be square (`num_teams` × `num_teams`), and the vector will have length `num_teams`.

❸ For each match, update the matrix and vector. The diagonal of `C` counts games played; each team gets +1 for each match. The off-diagonal counts opponents faced; if teams i and j play, we put –1 in positions `C`[i,j] and `C`[j,i]. The vector `b` tracks wins (+1) and losses (–1).

❹ Adding 2 to the diagonal is a key part of the Colley method. It ensures that the matrix is invertible (so we can solve the system) and gives each team a baseline rating of 0.5, representing a .500 winning percentage.

❺ Solve the linear system by using NumPy's linear algebra solver. This is where the magic happens. NumPy finds the ratings that satisfy all the constraints simultaneously. It's solving `num_teams` equations with `num_teams` unknowns.

The top 10 teams by Colley ranking for the WWC 2019 are depicted in Figure 8-1. We can see that the US team leads by a large margin, followed by the Netherlands, Sweden, and France.

The beauty of Colley ratings is in their interpretation. A rating of 0.5 represents an average team—one that would be expected to win 50% of its games against other average teams. A rating above 0.5 indicates an above-average team. The higher the rating, the better the team. A rating of 0.7 suggests a team that wins about 70% of its games against average competition.

Crucially, these ratings automatically account for strength of schedule. A team with a 3–1 record against top opponents (high Colley ratings) will have a higher Colley rating than a team with a 3–1 record against weak opponents (low Colley ratings). The system *knows* that beating good teams is harder and more impressive.

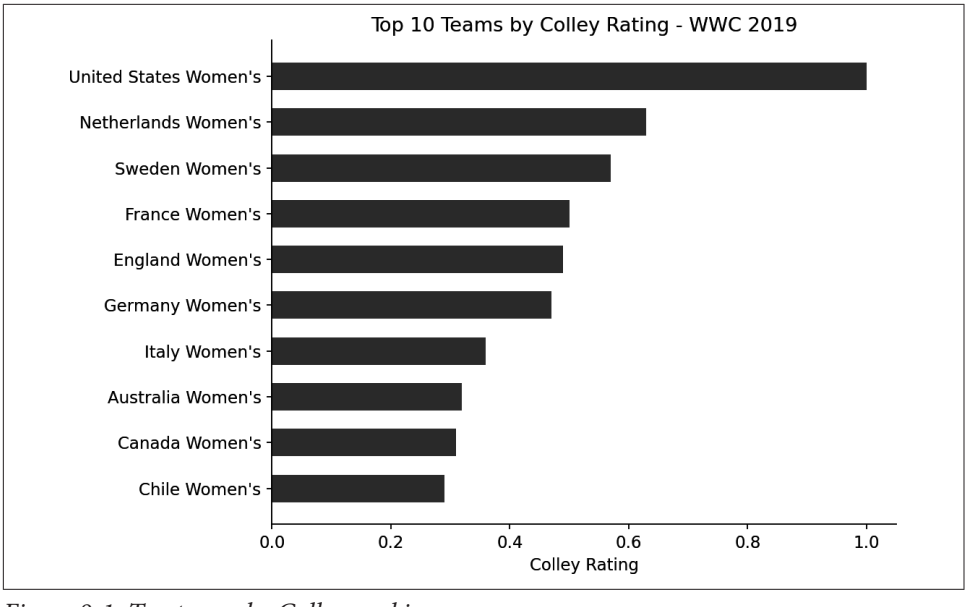


Figure 8-1. Top teams by Colley ranking

PageRank: A Network-Based Perspective

PageRank is the algorithm that made Google famous. Larry Page and Sergey Brin developed it in 1998 to rank web pages, and it revolutionized internet search. The core idea is that a web page is important if many other important pages link to it. A link from CNN.com is worth more than a link from a personal blog.

This idea translates beautifully to sports. A win against a top-ranked team is a more significant vote for quality than a win against a bottom-table team. We model the tournament as a *network* (or graph) with teams as nodes and matches as directed edges. If team A beats team B, we draw an edge from B to A. B is voting for A's quality by losing to them.

PageRank then iteratively calculates ratings. Initially, all teams have equal ratings. Then, in each iteration, a team's rating is updated based on the ratings of the teams that *voted* for it (i.e., the teams it beat). This process repeats until the ratings converge to stable values.

The mathematical formula is:

$$PR(A) = \frac{1-d}{N} + d\sum\frac{PR(T_i)}{C(T_i)}$$

where:

- $PR(A)$ is the PageRank of team A
- d is the damping factor (typically 0.85)
- N is the total number of teams
- T_i are the teams that A beat
- $C(T_i)$ is the number of teams that T_i lost to

Let's implement this algorithm:

```
import networkx as nx
def calculate_pagerank_ratings(matches_df, damping=0.85):
    teams = pd.unique(
        matches_df[["home_team", "away_team"]].values.ravel("K")
    )
    graph = nx.DiGraph()
    graph.add_nodes_from(teams)  ❶
    for _, row in matches_df.iterrows():  ❷
        home_team, away_team = (
            row["home_team"],
            row["away_team"],
        )
        home_score, away_score = (
            row["home_score"],
            row["away_score"],
        )
        if home_score > away_score:
            graph.add_edge(
                away_team,
                home_team,
                weight=home_score - away_score,
            )  ❸
        elif away_score > home_score:
            graph.add_edge(
                home_team,
                away_team,
                weight=away_score - home_score,
            )
        else:
            graph.add_edge(
                home_team, away_team, weight=0.5
            )  ❹
            graph.add_edge(away_team, home_team, weight=0.5)
    return pd.Series(
        nx.pagerank(
            graph, weight="weight", alpha=damping
        ),  ❺
        name="pagerank",
    ).sort_values(ascending=False)
pagerank_df = (
```

```
    calculate_pagerank_ratings(matches_df)
    .rename_axis("team")
    .reset_index()
)
```

❶ Use the networkx library to create a directed graph representing the tournament. Each team is a node in the graph.

❷ Winners get edges pointing to them from losers. This is the voting mechanism. By losing to you, I'm voting that you're better than me.

❸ Weight edges by goal difference. A 3–0 win is a stronger vote than a 1–0 win. This adds nuance. Not all wins are equal.

❹ For draws, add edges in both directions with small weights. This ensures that both teams get some credit for the match, but not as much as a win would provide.

❺ The `nx.pagerank()` function does all the heavy lifting of the iterative calculations. The `alpha=0.85` is the standard damping factor. It means that 85% of a team's rating comes from the teams it beat, and 15% is a baseline given to all teams. This prevents teams with no wins from having a 0 rating.

Top teams by PageRank in the WWC 2019 are shown in Figure 8-2. As with the Colley ranking algorithm, the US women's team leads by a margin, followed by the Netherlands and Sweden. Differences to the Colley ranking reside in the details: England is placed before France, and Brazil before Australia.

PageRank has some interesting properties. It naturally handles unbalanced schedules; teams that haven't played each other are still connected through the network. PageRank also captures transitive relationships: if A beats B and B beats C, then A indirectly benefits from C's losses too.

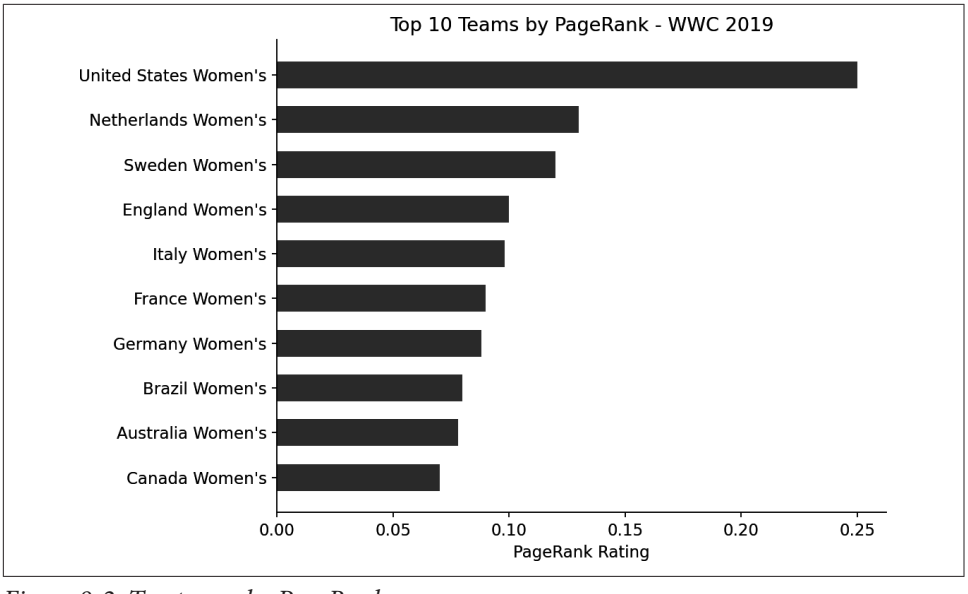


Figure 8-2. Top teams by PageRank

Elo Ratings: A Dynamic Rating System

Elo ratings, originally developed by Arpad Elo for chess in the 1960s, are now used across many sports—from chess to soccer to video games. Unlike Colley and PageRank, which look at all results simultaneously, Elo is dynamic. It updates after each match based on the expected outcome versus the actual outcome. The core idea is elegant: if a high-rated team beats a low-rated team, the high-rated team's rating barely changes (it was expected). But if a low-rated team upsets a high-rated team, a big swing occurs in both ratings (the outcome was unexpected). This makes Elo very responsive to recent performance.

The Elo formula is

$$R_{\text{new}} = R_{\text{old}} + K(S - E)$$

where:

- R_{new} is the new rating
- R_{old} is the old rating before the match
- K is a constant (typically 20–40) that controls how much ratings change
- S is the actual score (1 for win, 0.5 for draw, 0 for loss)
- E is the expected score based on the rating difference

The expected score is calculated using a logistic function:

$$E = \frac{1}{1 + 10^{\left(\frac{R_{\text{opponent}} - R_{\text{team}}}{400}\right)}}$$

This formula creates an S-curve. A 200-point rating advantage means you're expected to score 0.76 (76% chance of winning). A 400-point advantage means 0.91 (a 91% chance). Let's implement the formula:

```
def calculate_elo_ratings(
    matches_df, K=30, initial_rating=1500, home_advantage=0
):
    """Calculate Elo ratings for all teams."""  ❶
    ratings = {}
    rating_history = []
    for _, match in matches_df.sort_values(
        "match_date"
    ).iterrows():  ❷
        home, away = match["home_team"], match["away_team"]
        # Initialize ratings if first match  ❸
        if home not in ratings:
            ratings[home] = initial_rating
        if away not in ratings:
            ratings[away] = initial_rating
        # Calculate expected scores  ❹
        home_rating_adjusted = ratings[home] + home_advantage
        expected_home = 1 / (
            1
            + 10
            ** ((ratings[away] - home_rating_adjusted) / 400)
        )
        expected_away = 1 - expected_home
        # Determine actual scores  ❺
        if match["home_score"] > match["away_score"]:
            actual_home, actual_away = 1, 0
        elif match["home_score"] < match["away_score"]:
            actual_home, actual_away = 0, 1
        else:
            actual_home, actual_away = 0.5, 0.5
        # Update ratings  ❻
        ratings[home] += K * (actual_home - expected_home)
        ratings[away] += K * (actual_away - expected_away)
        rating_history.append(
            {
                "match_date": match["match_date"],
                "home_team": home,
                "away_team": away,
                "home_rating": ratings[home],
                "away_rating": ratings[away],
            }
```

```
        )
    final_ratings = pd.Series(
        ratings, name="elo_rating"
    ).sort_values(ascending=False)
    return pd.DataFrame(rating_history), final_ratings
elo_history, final_elo = calculate_elo_ratings(matches_df, K=30)
elo_df = final_elo.rename_axis("team").reset_index()
```

❶ Define a function that processes matches chronologically, updating ratings after each one. This is crucial because Elo is a sequential algorithm.

❷ Sort matches by date to ensure that we process them in chronological order. Processing out of order would give incorrect results.

❸ All teams start with the same initial rating (typically 1500, borrowed from chess). This is the *average* rating.

❹ The expected score formula creates an S-curve based on rating difference. If you're rated 100 points higher, you're expected to score about 0.64 (64% chance of winning). We also include an optional `home_advantage` offset. Setting `home_advantage=0` keeps the base implementation simple, but in practice many soccer Elo systems add a small boost to the home team's rating before computing the expected score.

❺ Convert match results to scores: 1 for win, 0.5 for draw, 0 for loss. This is standard in Elo systems.

❻ The rating update is proportional to the surprise. If the expected score was 0.8 and you won (actual = 1), you gain `K * (1 - 0.8) = K * 0.2` points. If you lost (actual = 0), you lose `K * (0 - 0.8) = K * -0.8` points—a big loss because it was unexpected.

The `K` parameter controls volatility. A higher `K` means ratings change more dramatically after each match. Chess uses `K=32` for established players, `K=40` for new players. Soccer often uses `K=20` to `K=40` depending on the competition. World Cup matches might use a higher `K` because they're high stakes.

In many soccer applications, a home advantage adjustment of roughly 50–100 Elo points is a reasonable starting point, though the right value depends on the league, competition, and era. You can also extend Elo further by scaling the `K` factor based on goal difference.

A Comparison of Methods: Strengths and Trade-Offs

We now have three ranking systems. How do they compare? Let's look at them side by side:

```
# Merge all rankings ❶
all_rankings = colley_df.merge(pagerank_df, on="team").merge(
    elo_df, on="team"
)
# Normalize to 0-1 scale for comparison ❷
for col in ["colley_rating", "pagerank", "elo_rating"]:
    all_rankings[f"{col}_norm"] = (
        all_rankings[col] - all_rankings[col].min()
    ) / (all_rankings[col].max() - all_rankings[col].min())
# Calculate correlations ❸
print("Correlation between ranking methods:")
print(
    all_rankings[
        [
            "colley_rating_norm",
            "pagerank_norm",
            "elo_rating_norm",
        ]
    ].corr()
)
# Find teams with biggest disagreements ❹
all_rankings["colley_elo_diff"] = abs(
    all_rankings["colley_rating_norm"]
    - all_rankings["elo_rating_norm"]
)
print(
    "\nTeams with biggest disagreement between Colley and Elo:"
)
print(
    all_rankings.nlargest(5, "colley_elo_diff")[
        ["team", "colley_rating", "elo_rating"]
    ]
)
```

❶ Merge all three ranking systems into a single `DataFrame` for comparison.

❷ Normalize each ranking to a 0–1 scale so we can compare them directly. Without normalization, we'd be comparing Colley ratings (0–1 scale) to Elo ratings (1,000–2,000 scale), which doesn't make sense.

❸ High correlations (close to 1) mean the methods largely agree on which teams are strong. Lower correlations mean they have different perspectives on team quality.

❹ Teams with big disagreements between methods are interesting. Maybe one team had a tough schedule (high Colley rating) but poor recent form (low Elo rating). Or vice versa.

Each method has strengths and weaknesses, as shown in Table 8-1.

Table 8-1. Comparing Colley, PageRank, and Elo

Algorithm	Strengths	Weaknesses	Best for
Colley matrix	Simple, transparent, stable. Explicitly accounts for strength of schedule. All teams are treated fairly—no recency bias.	Treats all wins equally (doesn't consider margin of victory). Slow to respond to changes in form. Assumes all teams start equal.	Season-long rankings, tournament seeding, situations where you want a fair, balanced assessment.
PageRank	Considers the full network of results. Naturally accounts for strength of schedule through the network structure. Can incorporate margin of victory through edge weights.	Less intuitive to interpret. Computationally more complex. Can be sensitive to network structure (e.g., isolated groups of teams).	Situations with complex, interconnected schedules. When you want to capture indirect relationships (A beat B, B beat C, so A benefits).
Elo ratings	Dynamic and responsive to recent form. Easy to update in real time (just process the latest match). Can easily incorporate home advantage and margin of victory.	Sensitive to initial ratings and K parameter. Can be volatile (ratings swing a lot). Doesn't explicitly account for strength of schedule (though it emerges implicitly).	Real-time rankings, betting markets, situations where recent form matters most. Predicting individual matches.

In practice, you might use all three as features in your prediction model and let the model figure out which matters most. This is called *ensemble features*—combining multiple perspectives to get a more robust view.

Player-Level Features

Up to this point, we have focused entirely on team-level features: the number of goals a team scores, its overall xG, and its Elo rating. But soccer is played by individuals. A team's strength is ultimately a function of the players on the pitch. If a star striker is injured, the team's historical goal-scoring average might overestimate its current strength.

To capture this, we can engineer player-level features. The simplest approach is to aggregate event data for each player across the tournament to create a custom rating system. For example, we might reward players for generating high xG, completing passes into the penalty area, and making successful tackles, while penalizing them for turnovers:

```
def calculate_player_ratings(events_df):
    """Calculate a simple composite rating for each player."""

    # Extract key positive actions
```

```
    shots = events_df[events_df['type'] == 'Shot']
        .groupby('player')['shot_statsbomb_xg'].sum()
    key_passes = events_df[events_df['pass_shot_assist'] == True]
        .groupby('player').size()
    tackles = (
        events_df[
            (events_df["type"] == "Duel")
            & events_df["duel_outcome"].isin(["Won", "Success In Play"])
        ]
        .groupby("player")
        .size()
    )

    # Combine into a single dataframe
    ratings = pd.DataFrame({
        'xG_generated': shots,
        'key_passes': key_passes,
        'tackles_won': tackles
    }).fillna(0)

    # Create a weighted composite score
    # (Weights are arbitrary for illustration;
    # in practice, these would be learned by a model)
    ratings['composite_score'] = (
        ratings['xG_generated'] * 20 +
        ratings['key_passes'] * 10 +
        ratings['tackles_won'] * 5
    )

    return ratings.sort_values('composite_score', ascending=False)

# Calculate ratings for the 2019 WWC
player_ratings = calculate_player_ratings(events_df)

# Plot the top 15 players
top_15 = player_ratings.head(15)
ax = top_15['composite_score'].sort_values().plot(
    kind='barh',
    figsize=(10, 8),
    color='#2ca02c',
    title='Top 15 Players by Custom Rating System - WWC 2019'
)
ax.set_xlabel('Player Rating', fontsize=11)
ax.set_title(
    'Top 15 Players by Custom Rating System - WWC 2019',
    fontsize=13,
)
ax.tick_params(axis='both', labelsize=11)
plt.ylabel('')
plt.tight_layout()
plt.show()
```

As Figure 8-3 shows, a custom rating system created by combining our engineered features highlights the top performers across all teams in the 2019 Women's World Cup tournament. Notice that the list is dominated by players from the US, Netherlands, and Australia—teams that went deep into the tournament and thus gave their players more minutes to accumulate raw stats.

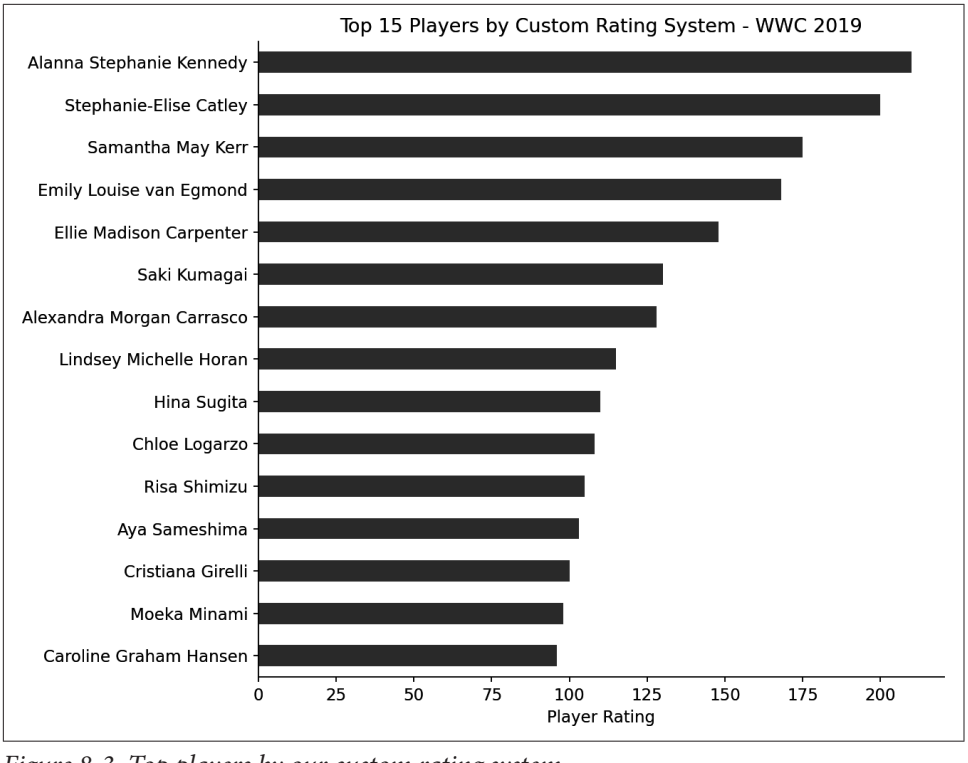


Figure 8-3. Top players by our custom rating system

In a production model, you would typically normalize these ratings every 90 minutes to fairly evaluate player efficiency. You could then aggregate the per-90 ratings of the starting 11 for any given match to create a powerful new team-level feature: Starting 11 Average Rating. This bridges the gap between individual talent and team prediction.

Feature Engineering Pipeline: From Data to Model Input

We've built a lot of features—probably 30 to 40 features across all these categories. In practice, you'll want to organize them into a clean, reproducible workflow. Let's create a framework.

Up to this point, we have built one metric family at a time. The next step is to reflect that same hierarchy in code by wrapping the feature families into a single reusable function:

```
def engineer_match_features(
    matches_df, events_df, match_id, home_team, away_team
):
    """Engineer features for one match."""  ❶
    matches_df = matches_df.copy()
    matches_df["match_date"] = pd.to_datetime(
        matches_df["match_date"]
    )
    match_date = matches_df[matches_df["match_id"] == match_id][
        "match_date"
    ].iloc[0]
    past_matches = matches_df[
        matches_df["match_date"] < match_date
    ].sort_values(
        "match_date"
    )  ❷
    past_events = events_df[
        events_df["match_id"].isin(past_matches["match_id"])
    ]
    team_history = pd.concat(
        [
            past_matches[
                [
                    "match_date",
                    "home_team",
                    "home_score",
                    "away_score",
                ]
            ]
            .rename(
                columns={
                    "home_team": "team",
                    "home_score": "goals_for",
                    "away_score": "goals_against",
                }
            )
            .assign(venue="home"),
            past_matches[
                [
                    "match_date",
                    "away_team",
                    "away_score",
                    "home_score",
                ]
            ]
            .rename(
                columns={
                    "away_team": "team",
```

```
                "away_score": "goals_for",
                "home_score": "goals_against",
            }
        )
        .assign(venue="away"),
    ]
).sort_values(["team", "match_date"])
features = {}
home_recent = team_history[
    team_history["team"] == home_team
].tail(
    5
)  ❸
away_recent = team_history[
    team_history["team"] == away_team
].tail(5)
features["home_goals_avg_5"] = home_recent[
    "goals_for"
].mean()
features["away_goals_avg_5"] = away_recent[
    "goals_for"
].mean()
home_at_home = team_history[
    (team_history["team"] == home_team)
    & (team_history["venue"] == "home")
]
away_at_away = team_history[
    (team_history["team"] == away_team)
    & (team_history["venue"] == "away")
]
features["home_home_goals_avg"] = home_at_home[
    "goals_for"
].mean()
features["away_away_goals_avg"] = away_at_away[
    "goals_for"
].mean()
home_shots = past_events[
    (past_events["team"] == home_team)
    & (past_events["type"] == "Shot")
]
away_shots = past_events[
    (past_events["team"] == away_team)
    & (past_events["type"] == "Shot")
]
features["home_avg_xg"] = home_shots[
    "shot_statsbomb_xg"
].mean()
features["away_avg_xg"] = away_shots[
    "shot_statsbomb_xg"
].mean()
colley = calculate_colley_ratings(past_matches)  ❹
features["home_colley"] = colley.get(home_team, 0.5)
```

```
    features["away_colley"] = colley.get(away_team, 0.5)
    features["colley_diff"] = (
        features["home_colley"] - features["away_colley"]
    )
    _, elo = calculate_elo_ratings(past_matches)
    features["home_elo"] = elo.get(home_team, 1500)
    features["away_elo"] = elo.get(away_team, 1500)
    features["elo_diff"] = (
        features["home_elo"] - features["away_elo"]
    )  ❺
    # Rest and fatigue
    for side, team in [
        ("home", home_team),
        ("away", away_team),
    ]:
        team_prior_matches = past_matches[
            (past_matches["home_team"] == team)
            | (past_matches["away_team"] == team)
        ].sort_values("match_date")
        if not team_prior_matches.empty:
            last_match_date = team_prior_matches.iloc[-1][
                "match_date"
            ]
            features[f"{side}_days_rest"] = (
                match_date - last_match_date
            ).days
    return features
```

❶ Create a function that takes a specific match and returns all the features we need to predict it. This is our *feature engineering pipeline.*

❷ Crucially, use only data from before the match. This prevents *data leakage* (using information that wouldn't have been available at prediction time). Convert `match_date` to a datetime type and sort the historical matches chronologically so most recent really means most recent.

❸ Calculate various feature families. In a real implementation, you'd call helper functions for each family to keep the code clean.

❹ For rankings, extract the ratings for the two teams playing. The ratings themselves are features.

❺ The *difference* in ratings is often more predictive than the absolute ratings. A large Elo difference (e.g., 300 points) suggests a mismatch—one team is much stronger. We also compute rest features for both teams, since fixture congestion can affect either side.

This function returns a dictionary of features that can be fed directly into a machine learning model. In Chapter 9, we'll explore how to use these features to develop and test data-driven betting strategies.

Case Study: Predicting the 2019 Women's World Cup Final

Let's put our feature engineering to work on the biggest match of the 2019 Women's World Cup: US versus Netherlands in the final. US won 2–0, but what did our features say beforehand? This is a perfect case study because both teams were highly rated. Let's generate a summary of the key quality metrics:

```
# Engineer features for US vs Netherlands final  #
final_match_id = 69321  # US vs Netherlands, July 7, 2019
features = engineer_match_features(
    matches_df,
    events_df,
    final_match_id,
    home_team="United States Women's",
    away_team="Netherlands Women's",
)

# Display key features in a structured format
import pandas as pd
pd.DataFrame({
    "Metric": ["Colley Rating", "Recent Form (goals/game)",
        "Shot Quality (avg xG)"],
    "USA": [features['home_colley'], features['home_goals_avg_5'],
        features['home_avg_xg']],
    "Netherlands": [features['away_colley'], features['away_goals_avg_5'],
        features['away_avg_xg']]
})
```

Running the feature engineering pipeline on the final produces a clean snapshot of where each team stood going into the match. The results are in Table 8-2. As you can see by the metric values, these teams were more than world-class.

Across every dimension we have built in this chapter, the data pointed in the same direction. US held a higher Colley rating, reflecting a stronger overall tournament record against quality opposition. Its recent form was sharper, averaging 2.8 goals per game over its last five matches compared to 1.9 for the Netherlands. And US shot quality was meaningfully better: a US shot was worth 0.14 xG on average, versus 0.11 for the Dutch. In isolation, any one of these gaps could be noise. Together, they painted a consistent picture of a team that was both more clinical and more highly rated—which is exactly what the final scoreline of 2–0 confirmed.

Table 8-2. Key engineered features prior to the 2019 WWC Final

Metric	US	Netherlands
Colley rating	0.78	0.71
Recent form (goals/game)	2.8	2.1
Shot quality (avg xG)	0.14	0.11

This kind of feature-by-feature analysis helps us understand why a model makes a prediction. If the model predicts a US win, we can look at the features and see, "Ah, US had a higher Colley rating (0.78 versus 0.71), better recent form (2.8 goals/game versus 2.1), and higher xG per shot (0.14 versus 0.11). That makes sense."

This interpretability is crucial for building trust in your models. Coaches and analysts won't blindly trust an opaque model. But if you can show them why the model made that prediction, grounded in features they understand (form, rankings, shot quality), they're much more likely to take it seriously.

Looking Ahead: From Better Inputs to Harder Decisions

Feature engineering is where soccer knowledge becomes predictive structure, and this chapter showed how much that translation matters. Rolling form windows, venue splits, shot-quality signals, fixture congestion, and ranking-based strength estimates each encode something a raw column cannot: context, recency, and the difference between what happened and what it meant. The discipline of building those features carefully—avoiding data leakage, choosing the right window sizes, separating home and away performance—is not just preprocessing. It is part of the analytical argument itself. A model is only as good as the representation we give it, and this chapter was about making that representation as honest and informative as possible.

In Chapter 9, we take the models and features built throughout the book and subject them to one of the most demanding real-world tests available: soccer betting markets. Betting is analytically interesting not because it is a path to profit, but because it forces a harder question than accuracy alone. A bookmaker's odds already encode a market consensus, sharpened by professional models and a built-in margin. The challenge is to determine whether our predictions are better than the market's estimate, whether any edge is large enough to act on, and how to size bets in a way that survives normal variance without ruin. That combination of prediction, valuation, and risk management—embodied in tools like the Kelly criterion and expected value analysis—is where data science stops being descriptive and becomes genuinely decisional.

Conclusion

Feature engineering is where soccer knowledge becomes predictive structure. In this chapter, we moved from raw match and event data to features that describe how teams actually perform: recent form, venue effects, shot quality, defensive pressure, fixture congestion, set-piece threat, and rating-based estimates of team strength.

That is the real value of this step. Models do not automatically understand that a team on short rest may be more vulnerable, that two-goal wins over weak opponents should be interpreted differently from narrow wins over strong ones, or that ten poor shots are not the same as three high-quality chances. Feature engineering gives the model access to that context in a form it can use.

Just as importantly, good features make our work easier to interpret. When a model makes a prediction, we want to understand whether it is being driven by form, by shot quality, by venue, by fatigue, or by a broader rating signal such as Elo or PageRank. That is one reason feature engineering remains so important even in an era of more complex models: it improves not only performance, but also clarity and trust.

This chapter also highlighted an important discipline. Useful features are not created by throwing every possible variable into a table. They are created by making thoughtful choices about timing, scope, and representation. We used rolling windows instead of full-season averages, separated home and away contexts, aligned event data to the same competition scope, and designed the workflow to avoid data leakage. Those choices are part of the modeling process, not just preprocessing.

The features in this chapter are only a starting point. You could extend them with player availability, tactical matchups, head-to-head history, referee behavior, weather, travel, or richer spatial and temporal patterns from event and tracking data. The key lesson is not to memorize a fixed list of features. It is to learn how to think about the information a model needs and the form that information should take, and how to build the model carefully.

In Chapter 9, we examine sports betting from a mathematical perspective to illustrate one important application of predictive modeling. A strong predictive model can help identify the right side of a bet, but that is only part of the story. If the objective is to determine whether sports betting can be profitable, we must also answer a second question: how much should be wagered?

CHAPTER 9

The Art and Science of Soccer Betting

A predictive model becomes far more interesting when a wrong answer has a real cost. That is what makes soccer betting such a demanding and revealing test case for analytics. In earlier chapters, we built models to estimate outcomes, goals, and match dynamics. Here, we push one step further and ask a harder question: what happens when we must turn those predictions into actual decisions under uncertainty?

This is why betting matters analytically, even if your goal is not to become a bettor. A bookmaker's odds are not just prices. They are a market-based forecast, sharpened by professional models, public information, and a built-in margin. The challenge is no longer simply to predict what is likely to happen. It is to decide whether our estimate is better than the market's, whether an edge is large enough to matter, and whether it is worth risking capital at all.

That shift changes everything. A model that looks good in a notebook may still fail in a betting setting if it cannot consistently identify value, its probabilities are poorly calibrated, or the staking strategy is reckless.

To see why, consider an extreme case. Suppose you have a model with 90% accuracy, which would already be exceptionally strong. Now suppose that after every prediction, you bet your entire bankroll on the next match. This strategy is doomed despite the model's strength because one wrong prediction is enough to reduce the bankroll to zero. And over a long enough sequence of bets, even a 90% accurate model will inevitably be wrong. The problem is not the quality of the model. The problem is that the staking strategy gives the model no room for normal error.

Soccer betting therefore forces us to combine prediction with judgment, valuation, and risk management. It is one of the clearest places where data science stops being descriptive and becomes decisional.

In this chapter, we treat betting not as a game of luck but as a framework for thinking rigorously about expected value, uncertainty, and bankroll control. We begin with the mechanics of odds, implied probabilities, overround, and arbitrage. From there, we examine baseline strategies, staking rules such as the Kelly criterion, and simple machine learning workflows for evaluating whether a model might translate into a real betting edge.

By the end of the chapter, you should be able to think about betting markets with the mindset of an analyst rather than a gambler. You will see why accuracy alone is not enough, why sizing matters as much as prediction, and why even a promising strategy can fail without discipline. Most importantly, you will understand how this domain brings together many of the core themes of the book: modeling, evaluation, uncertainty, and decision-making under pressure.

In practical terms, the chapter revolves around three linked questions: Can we estimate match outcomes more accurately than the market? Can we identify when the offered odds create genuine value? And can we size our bets in a way that survives ordinary variance long enough for any edge to matter?

Throughout, keep one principle in mind: this chapter is about disciplined analysis, not encouraging reckless gambling. The examples are educational, and the real lesson is how pricing, uncertainty, and risk management interact when money is at stake. Always bet responsibly, and never wager more than you can afford to lose.

Soccer Betting Basics

Soccer betting markets are built on a few key concepts that shape the way odds are set, bets are placed, and outcomes are evaluated. These fundamentals are not just theoretical. They're the groundwork for everything that follows, from data-driven strategies to profit optimization.

Types of Soccer Bets

Soccer betting offers a wide variety of options for bettors, catering to different levels of risk tolerance and strategic preferences. We'll cover four of the most common types of bets.

Match result

The *match result* (*1X2*) bet is the simplest and most popular form of soccer betting. Here, you predict one of three possible outcomes:

- 1: The home team wins.
- X: The match ends in a draw.
- 2: The away team wins.

This straightforward bet serves as the foundation for many other betting strategies.

Although 1X2 bets are typically framed around home and away teams, in some cases matches are played at neutral venues, such as tournament finals or international competitions. In such cases, sportsbooks may still designate one team as *home* and the other as *away* for betting purposes, or they may list teams by name without a home/away distinction.

Over/under

In an *over/under* (*totals*) bet, you predict whether the total number of goals scored in a match will be above or below a specified threshold. For example:

- Over 2.5 goals: You win if the match has three or more goals.
- Under 2.5 goals: You win if the match has two or fewer goals.

This type of bet shifts the focus from predicting the winner to forecasting the overall intensity of the game.

Both teams to score

The *both teams to score* (*BTTS*) bet is a popular choice for bettors who enjoy analyzing offensive and defensive dynamics. Here, you predict whether both teams will score at least one goal during the match. The outcome is independent of the final result, making it an exciting option for high-scoring games.

Correct score

The *correct score* bet requires predicting the exact final score of the match, including which team wins or if the match ends in a draw. For example, a 2–1 bet means the specified team must win by that score. Unlike some other bets, simply predicting the number of total goals is not enough. Both the exact score and the match outcome must be correct to win.

These markets offer different ways to take a position on the same match, and each comes with its own balance of risk, payout, and market efficiency. The 1X2 market is the simplest and usually the most liquid, so its odds are often closest to fair probabilities. Over/under markets can be especially attractive when you have information about tactics, tempo, weather, or team selection that may influence the total number of goals. BTTS bets appeal to many bettors because they depend on both teams

scoring rather than on the final result. Correct score bets can offer large payouts, but they are also among the hardest markets to predict accurately.

Choosing which markets to target is an important strategic decision. Many successful bettors specialize in just one or two market types, developing real depth instead of spreading themselves too thin. The goal is to focus on markets where you believe you have an informational or analytical edge.

In this chapter, we concentrate on 1X2 betting, since it provides a clear foundation for the ideas that follow. We will refer to other markets when useful, but we will not examine them in the same depth. If you are interested in extending the analysis, the score-prediction models developed in earlier chapters provide a natural starting point. Although our examples here focus on 1X2 betting, the same general framework can be adapted to other types of bets as well.

Odds

Odds are the backbone of sports betting. They determine the potential payout of a bet and reflect the bookmaker's assessment of the probability of an event occurring. Odds are expressed in three main formats: decimal odds, fractional odds, and moneyline odds.

Decimal odds (European)

Decimal odds are the most straightforward and widely used format, especially in Europe and Australia. They represent the total return on a unit stake, including the original wager. For example:

> Odds of 2.50: A $1 bet returns $2.50 ($1.50 profit + $1 stake).

Fractional odds (UK)

Fractional odds are commonly used in the United Kingdom and show the potential profit relative to the stake. For example:

> Odds of 3/1: A $1 stake yields $3 profit (plus the original $1 stake).

Moneyline odds (American)

Moneyline odds, popular in the US, use positive or negative numbers to indicate the potential profit or required stake. For example:

> +150: A $100 stake yields $150 profit.
>
> –150: You must stake $150 to profit $100.

Understanding the relationship between odds and probability is crucial for any bettor. Figure 9-1 shows their inverse relationship: higher odds mean lower implied

probability, but also higher potential profit. The visualization clearly demonstrates this trade-off.

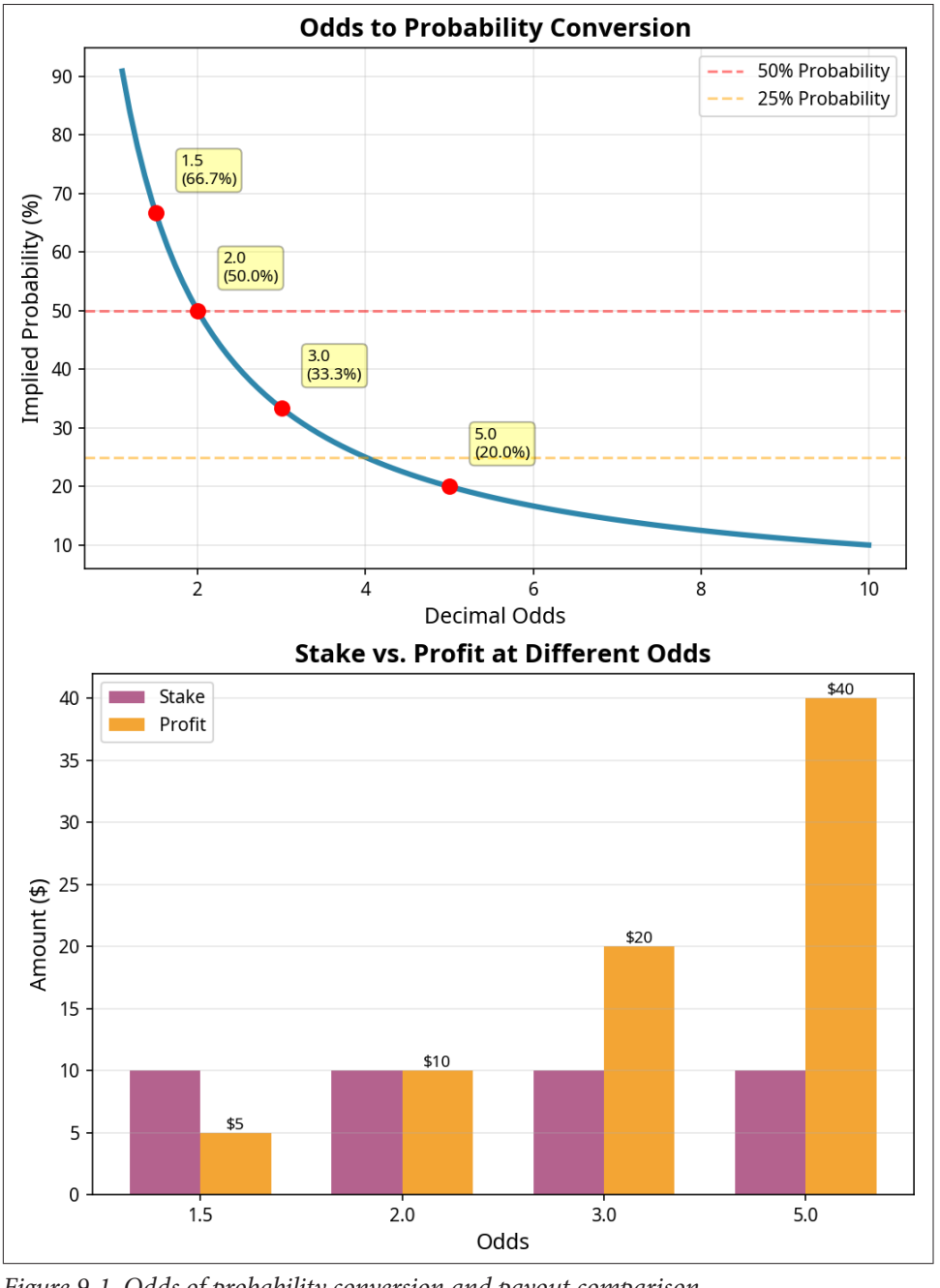


Figure 9-1. Odds of probability conversion and payout comparison

The top panel shows how odds translate to implied probabilities. Notice that the curve flattens at higher odds; the difference between 5.0 and 10.0 odds is much smaller in probability terms than the difference between 1.5 and 2.0. The bottom panel illustrates why bettors are tempted by long shots: a $10 bet at 5.0 odds yields $40 profit, compared to just $5 at 1.5 odds. However, remember that these higher payouts come with correspondingly lower win probabilities.

Implied probabilities

Implied probability is a key concept in betting, representing the likely outcome of an event as indicated by the odds. It's essential for gaining insight into the bookmaker's perspective and for identifying value bets. A *value bet* occurs when you believe the bookmaker has underestimated the likelihood of an event, offering odds that provide a potential profit in the long run. To spot value bets, you need to have your own reliable assessment of the probability of an event.

For decimal odds, the formula is as follows:

$$\text{implied probability} = \frac{1}{\text{decimal odds}}$$

Now, let's apply the formula, using the odds from a Premier League match as an example (Table 9-1).

Table 9-1. Odds and implied probability for match outcomes

Outcome	Odds	Implied probability
Home win	2.00	50%
Draw	3.50	28.6%
Away win	4.00	25%

Note that the sum of implied probabilities often exceeds 100% because of the bookmaker's margin, known as the *overround*.

Understanding implied probability is fundamental to identifying value bets. A value bet occurs when your assessment of an event's probability is higher than the implied probability from the odds. For instance, if you believe a team has a 55% chance of winning, but the odds imply only a 45% chance, that's a value bet—assuming your assessment is more accurate than the market's.

However, consistently finding value bets is challenging. It requires not only accurate probability estimates but also the discipline to bet only when you have an edge. Many bettors fall into the trap of convincing themselves they've found value when they're actually just biased toward their favorite team or overconfident in their analysis.

The concept of value betting is borrowed from financial markets, where investors look for assets that are underpriced relative to their fundamental value. In betting, you're looking for outcomes that are underpriced by the bookmaker. But remember: the bookmaker has access to vast amounts of data, sophisticated models, and the collective wisdom of the betting market. Claiming you've found value means claiming you know something they don't—a high bar to clear.

The overround

The *overround* represents the bookmaker's built-in margin. It's calculated as follows:

$$\text{overround} = \sum_i p_i - 1$$

Here, p_i denotes each implied probability written in decimal form.

For example, using the odds in Table 9-1, we can calculate the overround as follows:

$$\text{overround} = 0.50 + 0.286 + 0.25 - 1 = 0.036$$

This 3.6% margin gives the bookmaker a built-in edge. In practice, the bookmaker's realized profit still depends on how money is distributed across outcomes, but a higher overround generally makes the market less favorable for bettors. Some bookmakers may have a higher margin than others, and even within the same bookmaker, different games may carry varying margins. Recognizing these differences allows bettors to assess the odds more effectively and make more informed betting decisions.

The overround has a direct and significant impact on bettor profitability. As illustrated in Figure 9-2, the bookmaker's margin means a bettor with no informational edge should expect negative returns over time. This is why finding bookmakers with lower overrounds (typically 2%–5% for major markets) is crucial for long-term success.

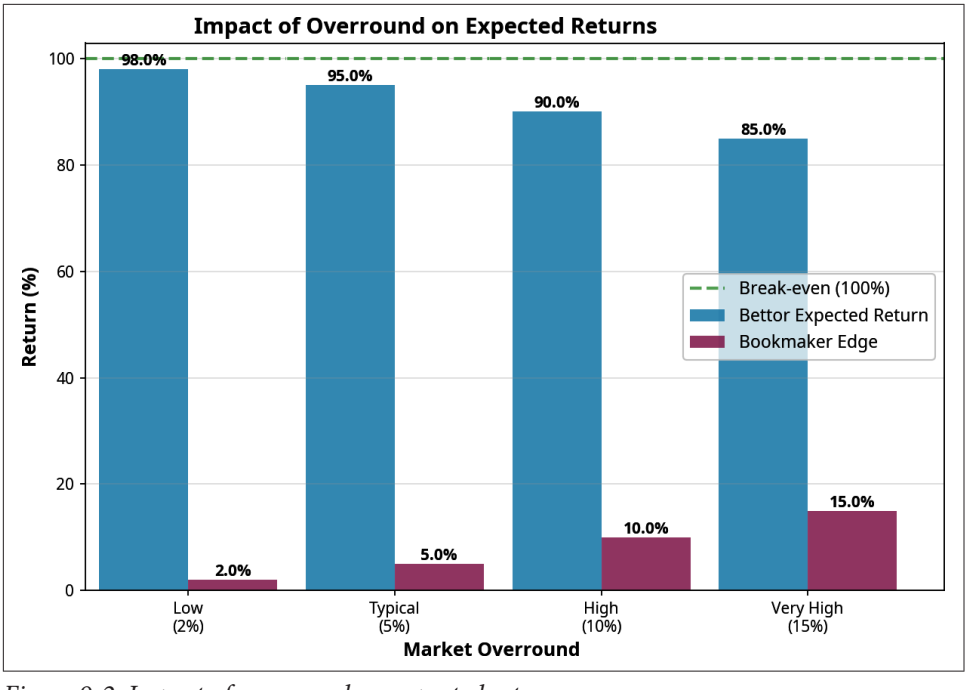


Figure 9-2. Impact of overround on expected returns

This visualization starkly illustrates why beating the bookmaker is so challenging. In a market with a typical 5% overround, simply accepting the market price without any predictive or informational edge leaves you starting from a disadvantage. To be profitable, you need not just accurate predictions, but pricing judgments that are strong enough to overcome this built-in disadvantage.

Arbitrage Opportunities

Arbitrage in soccer betting occurs when differences in odds across bookmakers allow bettors to place bets on all three possible outcomes (home win, draw, away win) in a way that guarantees a profit, regardless of the result. While these opportunities are rare and often short-lived, they can be highly rewarding for those who can identify and act on them quickly.

Understanding the basic arbitrage formula

For a three-outcome event (1X2), arbitrage exists if the sum of the implied probabilities of the outcomes is less than 100%. This can be expressed as follows:

$$\frac{1}{\text{Odds}_1} + \frac{1}{\text{Odds}_X} + \frac{1}{\text{Odds}_2} < 1$$

Identifying arbitrage opportunities

Consider the odds in Table 9-2 from two bookmakers for the same match.

Table 9-2. Match outcomes predicted by two bookmakers

Outcome	Bookmaker A	Bookmaker B
Home win	2.10	1.90
Draw	3.60	3.80
Away win	4.20	4.00

To determine whether an arbitrage opportunity exists, calculate the implied probabilities for each outcome by using the best odds. This example has two bookmakers, A and B. Check whether the sum of these probabilities is less than 1.

For example:

Home win
: The best odds are from bookmaker A; the implied probability is $\frac{1}{2.10} = 0.476$ (47.6%).

Draw
: The best odds are from bookmaker B; the implied probability is $\frac{1}{3.80} = 0.263$ (26.3%).

Away win
: The best odds are from bookmaker A; the implied probability is $\frac{1}{4.20} = 0.238$ (23.8%).

The sum of implied probabilities is 0.476 + 0.263 + 0.238 = 0.977 (97.7%).

Since the sum is less than 100%, an arbitrage opportunity exists in this case. However, a sum greater than 100% would indicate no potential arbitrage opportunity.

Calculating optimal stakes if arbitrage exists

Once an arbitrage opportunity is identified, the next step is to calculate the optimal stakes for each outcome to ensure a guaranteed profit. This process is crucial because it allows bettors to place bets in such a way that no matter the result of the match, they will make a profit.

To calculate the optimal stakes, you need to follow these steps:

1. *Determine the total amount to stake*
   : Decide the total amount of money you want to invest in the arbitrage bet. This amount will be distributed across the different outcomes based on their implied probabilities.

2. Calculate the implied probabilities

For each outcome, calculate the implied probability based on the best available odds (as discussed in "Identifying arbitrage opportunities" on page 267). The implied probability tells you the likelihood of each outcome.

3. Allocate stakes proportionally

To ensure a guaranteed profit, allocate your total stake in proportion to the implied probabilities of each outcome. This means betting more on outcomes with lower implied probabilities and less on outcomes with higher implied probabilities.

4. Calculate the return for each outcome

Once the stakes are determined, calculate the potential return for each outcome by multiplying your stake by the odds. Then, check whether the returns are equal across all outcomes to guarantee a profit.

By following these steps, you can calculate the optimal stakes to place on each outcome, ensuring that no matter which team wins or if the match ends in a draw, you will make a profit from the arbitrage opportunity.

In Table 9-3, let's reuse the data from the earlier section and add the best odds explicitly.

Table 9-3. Odds for game outcomes as posited by two bookmakers, and best odds

Outcome	Bookmaker A	Bookmaker B	Best odds
Home win	2.10	1.90	2.10 (bookmaker A)
Draw	3.60	3.80	3.80 (bookmaker B)
Away win	4.20	4.00	4.20 (bookmaker A)

Using the best odds, calculate the implied probabilities:

- Home win: $\frac{1}{2.10} = 0.476$ (47.6%)
- Draw: $\frac{1}{3.80} = 0.263$ (26.3%)
- Away win: $\frac{1}{4.20} = 0.238$ (23.8%)

The sum of implied probabilities is 0.476 + 0.263 + 0.238 = 0.977 (97.7%).

Since the sum is less than 1, an arbitrage opportunity exists.

If you want to distribute a total stake of $100 across the three outcomes, you would bet approximately $48.72 on a home win, $26.92 on a draw, and $24.36 on an away win.

By placing these bets, you ensure that no matter what happens in the match, one of your bets will win, and the payout will cover all your stakes while leaving you with a profit. Here's how it works:

If the home team wins:

- Winning bet: Home win at odds of 2.10
- Payout: $48.72 × 2.10 ≈ $102.31
- Profit: $102.31 – $100.00 ≈ $2.31

If the outcome is a draw:

- Winning bet: Draw at odds of 3.80
- Payout: $26.92 × 3.80 ≈ $102.30
- Profit: $102.30 – $100.00 ≈ $2.30

If the away team wins:

- Winning bet: Away win at odds of 4.20
- Payout: $24.36 × 4.20 ≈ $102.31
- Profit: $102.31 – $100.00 ≈ $2.31

Regardless of the match outcome, you end up with a profit of approximately $2.30. This is because the stakes were calculated proportionally to the implied probabilities of each outcome, ensuring that the payout from the winning bet covers all your stakes and leaves you with a net gain.



While arbitrage betting can be profitable, it comes with significant challenges:

Rarity
: Arbitrage opportunities are rare in modern, efficient markets.

Speed
: Professional traders and algorithms often eliminate these opportunities quickly.

Restrictions
: Bookmakers may limit or close accounts of bettors who frequently engage in arbitrage.

Execution risk
: Delays in placing bets can result in missed opportunities or losses.

By understanding the mechanics of arbitrage and the risks involved, bettors can better evaluate whether this strategy aligns with their goals and resources.

The Real Dataset Used in This Chapter

For the applied examples in the rest of this chapter, we use the completed 2024–25 Premier League results-and-odds dataset from Football-Data.co.uk. It is a convenient source for this chapter because it brings together match results and bookmaker prices in a single file. Each row corresponds to one match, giving us exactly the kind of information we need to move from prediction to betting decisions.

For our analysis, we'll keep a compact subset of columns: `Date`, `Time`, `HomeTeam`, `AwayTeam`, `FTHG`, `FTAG`, `FTR`, `B365H`, `B365D`, and `B365A`. According to Football-Data's notes, `B365H`, `B365D`, and `B365A` are the pre-closing bet365 odds for the home win, draw, and away win. We focus on pre-closing odds because bookmaker prices for the same match can change over time as new information reaches the market. Those moves are not as continuous as stock-price fluctuations, but they can still be meaningful, whether they occur a couple of days before kickoff or in the final hours before the match. That is an important detail to keep in mind whenever you work with betting data: the timing of the odds snapshot matters. That is enough for us to compute implied probabilities, compare simple decision rules, and later connect those decisions to staking.

Football-Data.co.uk is well worth exploring beyond this one example. The site provides historical files for many leagues and seasons, along with additional betting columns that can support richer analyses if you want to go further.

The first five rows of the dataset look like Table 9-4.

Table 9-4. Overview of the first few rows of historical football data

Date	Time	HomeTeam	AwayTeam	FTHG	FTAG	FTR	B365H	B365D	B365A
16/08/2024	20:00	Man United	Fulham	1	0	H	1.6	4.2	5.25
17/08/2024	12:30	Ipswich	Liverpool	0	2	A	8.5	5.5	1.33
17/08/2024	15:00	Arsenal	Wolves	2	0	H	1.18	7.5	13
17/08/2024	15:00	Everton	Brighton	0	3	A	2.63	3.3	2.63
17/08/2024	15:00	Newcastle	Southampton	1	0	H	1.36	5.25	8

Here's how we prepare for training a model on the real dataset:

```
from pathlib import Path
import pandas as pd
from sklearn.compose import ColumnTransformer
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder, StandardScaler
data_path = Path("code/extras/chapter-9/data/E0_2024_2025.csv")
if not data_path.exists():
    data_path = Path(
```

```
        "code/extras/chapter-9/data/E0_2025_2026.csv"
    )
matches_df = pd.read_csv(data_path)
keep_columns = [
    "Date",
    "Time",
    "HomeTeam",
    "AwayTeam",
    "FTHG",
    "FTAG",
    "FTR",
    "B365H",
    "B365D",
    "B365A",
]
matches_df = matches_df[keep_columns].dropna().copy()
matches_df["Kickoff"] = pd.to_datetime(
    matches_df["Date"] + " " + matches_df["Time"],
    format="%d/%m/%Y %H:%M",
)
matches_df = matches_df.sort_values("Kickoff").reset_index(
    drop=True
)
for outcome in ["H", "D", "A"]:
    matches_df[f"Imp{outcome}"] = (
        1 / matches_df[f"B365{outcome}"]
    )
matches_df["Overround"] = matches_df[
    ["ImpH", "ImpD", "ImpA"]
].sum(axis=1)
cutoff = int(len(matches_df) * 2 / 3)
train_df = matches_df.iloc[:cutoff].copy()
valid_df = matches_df.iloc[cutoff:].copy()
odds_to_result = {"B365H": "H", "B365D": "D", "B365A": "A"}
for frame in [train_df, valid_df]:
    frame["home_pick"] = "H"
    frame["lowest_odds_pick"] = (
        frame[["B365H", "B365D", "B365A"]]
        .idxmin(axis=1)
        .map(odds_to_result)
    )
def confusion_table(actual, predicted):
    return pd.crosstab(
        actual,
        predicted,
        rownames=["Actual"],
        colnames=["Predicted"],
        dropna=False,
    )
```

Choosing a Side

Before talking about bankroll growth or staking formulas, it helps to separate one question from another. The first question is simply, "Which side are we backing?" Only after that does it make sense to ask how much we should stake.

For that reason, this section evaluates side-selection rules by using only two prediction metrics: the confusion matrix and overall accuracy. That keeps the comparison focused. We are not yet asking whether a rule is profitable. We are asking only how often it points to the correct result class.

Always Pick the Home Team

The simplest possible rule is to predict a home win in every match. This is a meaningful benchmark because home advantage is among the most stable patterns in soccer. It is also deliberately naive, which makes it useful as a floor rather than a ceiling. Here's how we implement the home win as a benchmark:

```
home_accuracy = (
    valid_df["home_pick"] == valid_df["FTR"]
).mean()
home_confusion = confusion_table(
    valid_df["FTR"], valid_df["home_pick"],
)
print(f"Always-home accuracy: {home_accuracy:.3f}")
print(home_confusion)
```

The confusion matrix tells us exactly where this rule fails. Because it always predicts `H`, every draw and every away win becomes an error. That makes the rule easy to understand and easy to beat, but it still gives us a useful starting point for comparison.

You can apply the same idea to the other outcomes as well by always predicting a draw or always predicting an away win. As a simple exercise, try both and compare their confusion matrices and accuracies. It is a useful way to build intuition about class balance, home advantage, and the behavior of very simple prediction rules.

Always Pick the Lowest Odds

The next rule follows the bookmaker's favorite. For each match, we select whichever of `B365H`, `B365D`, or `B365A` is smallest and treat that outcome as our prediction. This is still a simple heuristic, but it is more informed than the home-only rule because it incorporates the market's view of the match. Here's how we implement the lowest-odds pick:

```
lowest_accuracy = (
    valid_df["lowest_odds_pick"] == valid_df["FTR"]
).mean()
```

```
lowest_confusion = confusion_table(
    valid_df["FTR"], valid_df["lowest_odds_pick"],
)
print(f"Lowest-odds accuracy: {lowest_accuracy:.3f}")
print(lowest_confusion)
```

This baseline is important because it tells us how much predictive signal is already embedded in the prices themselves. If a more elaborate model cannot improve on this benchmark, the extra complexity may not be adding much value.

You can extend the same idea by selecting the outcome with the highest odds or the one in the middle for each match. As a simple exercise, try both and compare their confusion matrices and accuracies with the lowest-odds rule. It is a useful way to see how much information is contained in the market's ranking of outcomes, not just in its favorite.

Use a Simple Machine Learning Baseline

We now add a third side-selection rule: a basic machine learning classifier. To keep the notebook lightweight and reproducible, we use logistic regression with team identifiers plus implied-probability features derived from the odds. This is not intended to be a production betting model. It is simply a third benchmark that is data-driven rather than rule-based:

```
features = [
    "HomeTeam",
    "AwayTeam",
    "ImpH",
    "ImpD",
    "ImpA",
    "Overround",
]
preprocess = ColumnTransformer(
    [
        (
            "teams",
            OneHotEncoder(handle_unknown="ignore"),
            ["HomeTeam", "AwayTeam"],
        ),
        (
            "numeric",
            StandardScaler(),
            ["ImpH", "ImpD", "ImpA", "Overround"],
        ),
    ]
)
ml_model = Pipeline(
    [
        ("preprocess", preprocess),
        (
```

```
            "model",
            LogisticRegression(max_iter=2000, solver="lbfgs"),
        ),
    ]
)
ml_model.fit(train_df[features], train_df["FTR"])
valid_df["ml_pick"] = ml_model.predict(valid_df[features])
ml_accuracy = (valid_df["ml_pick"] == valid_df["FTR"]).mean()
ml_confusion = confusion_table(
    valid_df["FTR"], valid_df["ml_pick"],
)
print(f"Machine-learning accuracy: {ml_accuracy:.3f}")
print(ml_confusion)
```

For simplicity, this model uses a relatively small feature set. In particular, it leans on the odds through the implied probabilities, along with basic team identifiers. In Chapters 3 and 8, however, we introduced a broader range of team-level features that can also be incorporated here. If you want to extend this baseline, try adding those features and see whether they improve side selection beyond what the market prices already capture. Here's how we would add more features for comparison:

```
comparison = pd.DataFrame(
    {
        "strategy": [
            "always_home",
            "lowest_odds",
            "machine_learning",
        ],
        "accuracy": [
            home_accuracy,
            lowest_accuracy,
            ml_accuracy,
        ],
    }
).sort_values("accuracy", ascending=False)
print(comparison)
```

Betting Strategies

Once the side-selection rule is fixed, the next question is how much to wager on each match. This is where staking enters the picture. Two bettors can make the same prediction and get very different results if one uses a flat stake while the other sizes bets aggressively.

To keep the logic clear, we will use the same two simple side-selection rules from "Choosing a Side" on page 272 and layer two staking approaches on top of them: fixed-stake betting and the Kelly criterion. That separation is deliberate. Choosing the side and choosing the stake are related, but they are not the same decision.

Fixed-Stake Betting

A *fixed-stake rule* is the simplest staking policy. You decide on one constant amount, such as 100 units, and risk that same amount on every bet. This makes it a useful benchmark because it is transparent, easy to implement, and easy to compare across different side-selection rules. Here's how we implement fixed-stake betting:

```
def fixed_stake_summary(matches, picks, stake=100):
    total_profit = 0.0
    for pick, (_, row) in zip(picks, matches.iterrows()):
        odds = row[f"B365{pick}"]
        profit = (
            stake * (odds - 1) if row["FTR"] == pick else -stake
        )
        total_profit += profit
    total_staked = stake * len(matches)
    roi = total_profit / total_staked
    return total_profit, roi
```

Fixed stakes on the home-team rule

Using the 2024–25 Premier League file, the first 253 matches form the training sample, and the final 127 matches form the holdout set. In this experiment, we stake $100 on every holdout match, so the total planned holdout stake is $12,700.

To establish a baseline, we apply a flat $100 stake to our simplest side-selection rule: always betting on the home team. Because the home team wins less than half the time in the Premier League, and because bookmaker margins are baked into the odds, we expect this naive strategy to lose money over the long run. By passing our validation DataFrame and the `home_pick` column into our summary function, we can quantify exactly how much this strategy bleeds capital over the holdout period:

```
home_fixed_profit, home_fixed_roi = fixed_stake_summary(
    valid_df, valid_df["home_pick"], stake=100,
)
print(
    f"Fixed-stake profit, always home: "
    f"{home_fixed_profit:.2f}"
)
print(f"Fixed-stake ROI, always home: {home_fixed_roi:.3%}")
```

For our time-series plot, return on investment (ROI) is defined as cumulative profit divided by that fixed total holdout stake, not by the amount staked so far. For visual clarity, the x-axis simply indexes the holdout matches from 1 to 127. Under this setup, the always-home rule finishes the holdout period at –$2,048, which corresponds to an ROI of about –16.1%. We use some plotting helpers from `soccerml_figures`. Its implementation details are omitted for brevity.

Here's how we calculate and plot the holdout ROI:

```
import matplotlib.pyplot as plt
from soccerml_figures.plotting import (
    apply_margins,
    create_figure,
    output_path,
    save_figure,
)
home_fixed_total_stake = 100 * len(valid_df)
home_fixed_cumulative_profit = 0.0
home_fixed_roi_path = []
for _, row in valid_df.iterrows():
    profit = (
        100 * (row["B365H"] - 1) if row["FTR"] == "H" else -100
    )
    home_fixed_cumulative_profit += profit
    home_fixed_roi_path.append(
        home_fixed_cumulative_profit / home_fixed_total_stake
    )
games = range(1, len(valid_df) + 1)
fig, ax = create_figure()
ax.plot(
    games, home_fixed_roi_path, linewidth=2, color="#0B5D7A",
)
ax.axhline(0, color="#6B7280", linestyle="--", linewidth=1)
ax.set_title("Holdout ROI: Fixed $100 Stakes, Always Home")
ax.set_xlabel("Holdout Game Number")
ax.set_ylabel("ROI")
ax.grid(alpha=0.25)
apply_margins(fig)
save_figure(fig, output_path(9, "home-fixed-holdout-roi"))
```

Figure 9-3 shows the result. The final profit and ROI are almost always negative, as is expected during most bets. We can also clearly see that the higher the holdout game number is, the lower the ROI becomes.

This example answers a clean question: what happens when we keep the side selection primitive and also keep the stake size constant? The result is easy to interpret because every gain or loss can be traced back to the same decision rule and the same bet size.

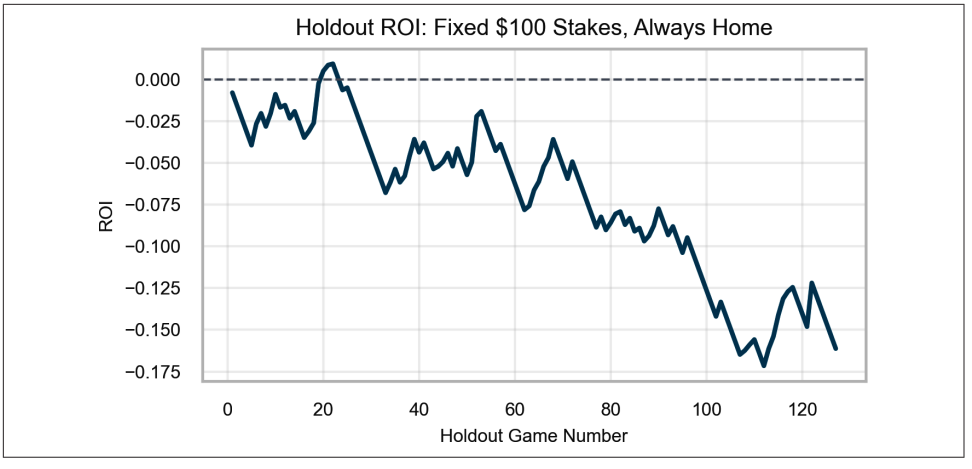


Figure 9-3. Holdout ROI by holdout game number for fixed $100 stakes on the home-team rule

Fixed stakes on the lowest-odds rule

A slightly more sophisticated baseline is to always bet on the bookmaker's favorite—the team with the lowest odds. Favorites win more frequently than home teams do, on average, which means this rule should produce a higher strike rate. However, because the odds are shorter, the payouts on winning bets are smaller. We apply the same $100 fixed stake to the `lowest_odds_pick` column to see whether the increased win rate is enough to overcome the reduced payouts and the bookmaker's overround:

```
lowest_fixed_profit, lowest_fixed_roi = fixed_stake_summary(
    valid_df, valid_df["lowest_odds_pick"], stake=100,
)
print(
    f"Fixed-stake profit, lowest odds: "
    f"{lowest_fixed_profit:.2f}"
)
print(f"Fixed-stake ROI, lowest odds: {lowest_fixed_roi:.3%}")
```

Comparing these two outputs makes an important point. Even before we move to more complex models, changing the side-selection rule can change the payout structure enough to alter the final betting outcome.

Kelly Criterion

The *Kelly criterion* is a widely used staking rule for determining the fraction of the current bankroll to wager. Unlike fixed staking, it adapts bet size to the estimated edge. When the estimated edge is large, Kelly recommends a larger stake. When the edge is small or negative, it recommends a smaller stake or no bet at all.

The formula for the Kelly criterion is:

$$f^* = \frac{bp - q}{b}$$

where:

- f^* represents the fraction of the bankroll to wager.
- p is the estimated probability of winning the bet.
- q (which is $1 - p$) is the probability of losing the bet.
- b is the net decimal payoff (i.e., the decimal odds minus 1).

To keep the example self-contained, we use a deliberately simple probability input for each rule. For the home-team rule, we use the training-sample frequency of home wins. For the lowest-odds rule, we use the training-sample accuracy of that rule. These are crude estimates, not production-quality probability forecasts, but they are good enough to show how Kelly changes the staking logic:

```
    def kelly_fraction(probability, decimal_odds):
        b = decimal_odds - 1
        q = 1 - probability
        return max((b * probability - q) / b, 0)
    def kelly_summary(
        matches, picks, probabilities, starting_bankroll=1000,
    ):
        bankroll = starting_bankroll
        for pick, probability, (_, row) in zip(
            picks, probabilities, matches.iterrows(),
        ):
            odds = row[f"B365{pick}"]
            fraction = min(kelly_fraction(probability, odds), 1)
            stake = bankroll * fraction
            bankroll -= stake
            if row["FTR"] == pick:
                bankroll += stake * odds
        roi = (bankroll - starting_bankroll) / starting_bankroll
        return bankroll, roi
```

Kelly on the home-team rule

When we switch from fixed stakes to the Kelly criterion, the bet size becomes dynamic. To calculate the Kelly fraction, we need an estimate of our probability of winning. For this toy example, we use the simplest possible estimate: the historical frequency of home wins in our training sample. We then set our starting bankroll to match the total planned holdout stake from the fixed-stake example ($12,700).

Because Kelly bets aggressively when it perceives an edge, and because our probability estimate is crude, we expect this approach to produce a highly volatile bankroll trajectory. Here's how we implement the Kelly criterion on the home team:

```
home_rule_probability = (train_df["FTR"] == "H").mean()
home_kelly_starting_bankroll = 100 * len(valid_df)
home_kelly_bankroll, home_kelly_roi = kelly_summary(
    valid_df,
    valid_df["home_pick"],
    [home_rule_probability] * len(valid_df),
    starting_bankroll=home_kelly_starting_bankroll,
)
print(
    f"Kelly final bankroll, always home: "
    f"{home_kelly_bankroll:.2f}"
)
print(f"Kelly ROI, always home: {home_kelly_roi:.3%}")
```

Because the home-team probability estimate is the same for every match in this toy example, the only thing driving Kelly's bet-size variation is the odds. That makes the example simple enough to follow while still illustrating the core idea of bankroll-sensitive staking. To keep the comparison aligned with the fixed-stake example, we start Kelly with the same $12,700 bankroll as the total planned holdout stake. Using the training-sample home-win frequency as the Kelly input, the always-home Kelly strategy finishes the holdout period with about $65.27 remaining, which corresponds to an ROI of about –99.5%. Here's how we calculate the ROI with the Kelly criterion:

```
home_kelly_bankroll = home_kelly_starting_bankroll
home_kelly_roi_path = []
for _, row in valid_df.iterrows():
    fraction = min(
        kelly_fraction(home_rule_probability, row["B365H"]), 1,
    )
    stake = home_kelly_bankroll * fraction
    home_kelly_bankroll -= stake
    if row["FTR"] == "H":
        home_kelly_bankroll += stake * row["B365H"]
    home_kelly_roi_path.append(
        (home_kelly_bankroll - home_kelly_starting_bankroll)
        / home_kelly_starting_bankroll
    )
games = range(1, len(valid_df) + 1)
fig, ax = create_figure()
```

```
ax.plot(
    games, home_kelly_roi_path, linewidth=2, color="#B63A1B",
)
ax.axhline(0, color="#6B7280", linestyle="--", linewidth=1)
ax.set_title("Holdout ROI: Kelly Stakes, Always Home")
ax.set_xlabel("Holdout Game Number")
ax.set_ylabel("ROI")
ax.grid(alpha=0.25)
apply_margins(fig)
save_figure(fig, output_path(9, "home-kelly-holdout-roi"))
```

Figure 9-4 shows the holdout ROI by game number for Kelly stakes on the home-team rule. We can see that the curve sinks rapidly and approximates –100% (meaning all money is lost) by game number ~100.

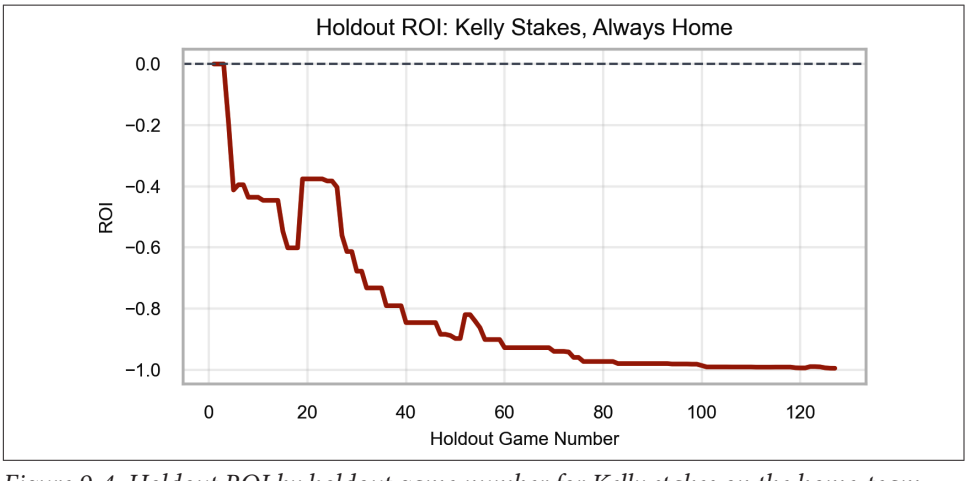


Figure 9-4. Holdout ROI by holdout game number for Kelly stakes on the home-team rule



The Kelly plot is accurate for the ordering used here, but that ordering is still only an approximation. We index the holdout bets by kickoff order, yet some matches overlap in time, so the nth match may begin before the $(n - 1)$th match has fully settled. That matters for Kelly because the stake on bet n depends on the bankroll after bet $n - 1$. So the figure is best read as an illustration of the idea rather than as a fully realistic execution model. In practice, overlapping matches require a more careful treatment of timing, settlement, and bankroll updates.

Kelly on the lowest-odds rule

Finally, we apply the Kelly criterion to the lowest-odds rule. Just as before, we need a probability estimate to feed into the formula. Here, we use the training-sample accuracy of the lowest-odds rule—essentially assuming that the favorite will win at its historical rate. We initialize a smaller starting bankroll of $1,000 for this test to observe how Kelly scales its wagers when dealing with shorter odds and a higher baseline win probability. Because the payouts are smaller, Kelly should theoretically recommend more conservative fractions of the bankroll compared to the home-team rule. Here's how we use the Kelly criterion on the lowest-odds rule:

```
lowest_rule_probability = (
    train_df["lowest_odds_pick"] == train_df["FTR"]
).mean()
lowest_kelly_bankroll, lowest_kelly_roi = kelly_summary(
    valid_df,
    valid_df["lowest_odds_pick"],
    [lowest_rule_probability] * len(valid_df),
    starting_bankroll=1000,
)
print(
    f"Kelly final bankroll, lowest odds: "
    f"{lowest_kelly_bankroll:.2f}"
)
print(f"Kelly ROI, lowest odds: {lowest_kelly_roi:.3%}")
```

This is still a simplified Kelly application, but it clearly makes the conceptual point. The side-selection rule tells us what to bet on. The Kelly rule tells us how aggressively to back it. Those are separate layers of the betting workflow, and both matter.



As an exercise, repeat the fixed-stake and Kelly analysis for other side-selection rules, such as always betting the away team, always betting the draw, or always picking the highest odds. Compare how the confusion matrix, accuracy, and bankroll results change once you alter the rule used to choose the side.

The same structure can later be extended to machine learning predictions as well. Once you have a model that produces match-specific probabilities you trust, those probabilities can replace the crude rule-level estimates used here and feed directly into fixed-stake or Kelly-style staking decisions.

From Predictions to Potential Profit

Soccer betting sits at a fascinating intersection of sports knowledge, statistical analysis, and risk management. Throughout this chapter, we have moved from basic odds and implied probabilities to arbitrage, baseline strategies, bankroll management, and machine learning. As we close, it is worth stepping back to ask what all this really means in practice.

The Reality of Profitability

Let's be honest: consistently profitable soccer betting is extremely difficult. Bookmakers are not naive. They employ analysts, models, and real-time data feeds, and their odds already incorporate enormous amounts of information. When you place a bet, you are effectively claiming that your estimate is better than the market's, at least in that particular situation.

That does not make profit impossible. Edges can exist, especially in less efficient leagues, narrower markets, or situations where you bring specialized knowledge or a better modeling approach. But those edges are usually small, temporary, and difficult to sustain. Backtesting can help you study whether a strategy would have worked in the past, but past performance is never a guarantee of future success. Soccer changes, teams change, and betting markets adapt quickly.

Practical Considerations

Before any strategy can succeed in the real world, it has to survive the practical realities of betting. A profitable idea on paper can still fail if bankroll management is weak, records are poor, or decisions become emotionally driven.

Bankroll management

Your *bankroll* is the total amount of money you set aside for betting. Managing it well is arguably more important than any individual prediction. Even a profitable model can fail if bet sizes are too aggressive.

A common rule of thumb is the *1%–5% rule*: do not risk more than 1%–5% of your bankroll on a single wager. This helps protect you from the inevitable losing streaks that occur even with a positive edge. Consider two bettors with the same model and the same long-run win rate. The one staking too much is far more likely to go broke before the model's advantage has time to play out.

This is the core idea behind *risk of ruin*. The Kelly criterion is one formal attempt to balance growth and survival, but the larger lesson is broader: bankroll preservation is a necessary part of any serious betting workflow.

Record keeping and the closing line value

Serious bettors keep detailed records. At minimum, that means tracking the date, match, prediction, odds, stake, and outcome for every wager. Many also track why the bet was made and what the closing odds were.

One especially useful metric is the *closing line value* (CLV), which compares the odds you received with the odds available right before kickoff. If you consistently beat the closing line, that is often stronger evidence of real value than a short run of wins or losses. It suggests that your process is finding favorable prices, even when short-run variance obscures the results.

Psychology and losing streaks

Betting is as much a psychological challenge as an analytical one. Common pitfalls include confirmation bias, recency bias, and loss aversion. A bettor may overreact to recent matches, search only for evidence that supports a preferred outcome, or increase stake sizes after losses in an attempt to recover quickly.

That is why discipline matters so much. Even with a profitable strategy, losing streaks are inevitable. With a 55% win rate, for example, a run of losses can happen purely by chance. The key is to distinguish variance from genuine model failure and to avoid chasing losses when results turn against you.

Shopping for odds

Different bookmakers offer different prices on the same match, and those differences add up. Even a modest improvement in average odds can have a meaningful effect on long-run profitability. That is why experienced bettors compare prices across bookmakers rather than accepting the first number they see.

The Importance of Discipline

Perhaps the most important lesson from this chapter is not about any single model or strategy. It is about discipline. A sophisticated model is useless if you do not follow clear rules for when to bet, how much to wager, and how to evaluate whether the process is still working.

Successful betting requires treating decisions systematically. You need a repeatable process, honest review, and enough emotional control not to rewrite your strategy after every hot streak or cold streak. In practice, many bettors fail less because their analysis is wrong than because their decision-making becomes inconsistent under pressure.

The Role of Technology

The tools we discussed—Python for analysis, scikit-learn for modeling, and backtesting frameworks for evaluation—have made quantitative betting far more accessible. An individual analyst can now perform work that once would have required a much larger team.

But technology is a double-edged sword. As these tools become more common, the market becomes more efficient. What once looked like an edge may disappear simply because too many people have found and exploited it. That means continuous learning is essential. A strategy that works today may need adaptation tomorrow.

Ethical Considerations

Any serious discussion of betting also needs an ethical dimension. Gambling can be addictive, and problem gambling causes real harm. If someone chooses to bet, they should do so with strict limits and with a clear understanding that losses are possible, even with a strong model. Never bet money you cannot afford to lose, and never chase wins as a way to recover emotionally.

Broader ethical issues exist too. Betting markets create incentives that can distort the sport itself, from match-fixing concerns to misleading advice from social media tipsters who advertise unrealistic win rates while hiding losses. As analysts, transparency matters. If you share betting ideas or model outputs, explain the limitations and be clear about conflicts of interest.

Looking Ahead: Building on Top of the Fundamentals

The field of soccer analytics and betting continues to evolve rapidly. Tracking data, richer event streams, more sophisticated machine learning, and in-play betting markets all create new possibilities. At the same time, they also create new challenges: more noise, faster markets, and a smaller margin for error.

The fundamentals, however, remain the same. Long-run success depends on the combination of accurate prediction, careful bet sizing, rigorous evaluation, and honest self-assessment. For most people, betting should remain a form of disciplined analytical exploration or entertainment, not a primary source of income.

Conclusion

This chapter has equipped you with a framework for approaching soccer betting scientifically. You now understand how odds encode probabilities, how value emerges when your estimate differs from the market's, how bankroll strategy shapes outcomes, and how models can support more disciplined decision-making.

But knowledge is only the starting point. The real test is whether you can apply these ideas consistently, with rigor, humility, and realistic expectations. Soccer remains unpredictable, and no model will ever capture every bounce, finish, injury, or refereeing decision that changes a match. That unpredictability is exactly what makes the game compelling and what makes betting on it so difficult.

The goal is not to correctly predict every match. It is to make better decisions than the market often enough, and to size your bets wisely when you believe you have an edge. That is a high standard, but it is the right one. If you approach this domain carefully, analytically, and responsibly, you will be far better equipped than the typical bettor.

Bet responsibly, stay skeptical of easy profit claims, and let disciplined analysis matter more than excitement.

In Chapter 10, we step back from betting as a specific application and return to the broader analytical lessons of the book, along with where soccer analytics may go next.

CHAPTER 10

What's Next for Soccer Analytics?

Throughout this book, the most important thing you have built is not a single model. It is a workflow for reasoning with data. Soccer gave us the setting, but the deeper achievement was learning how to move from a question to a usable analytical result: frame the task, gather data, clean it, explore it, engineer features, train models, evaluate them, and interpret them with care.

That workflow is the real conclusion of this book. Soccer provides a rich and intuitive environment in which to learn, but the larger lesson is about how to reason inside a complex, uncertain system. Matches are noisy. Performance is contextual. Predictions are probabilistic. And useful analysis depends as much on judgment and representation as it does on algorithms.

This final chapter therefore has two jobs. First, it brings the main ideas of the book back together so the overall arc is clear. Second, it points forward. If you want to continue in soccer analytics, you have a great deal more to explore. If you want to transfer these methods to another sport or another domain, the same core logic still applies. In that sense, this chapter is not only about what comes next in soccer analytics, but also about what should remain with you after the final page: a repeatable way of thinking, building, and learning.

What This Book Was Really About

At the surface level, this book taught Python and machine learning through soccer examples. At a deeper level, it taught a way of working. The point of this section is to make that larger structure explicit.

From Questions to Data

The first lesson was that analytics does not begin with a model. It begins with a question. What are we trying to explain? What are we trying to predict? What decision will this analysis support?

Once the question is clear, the data work becomes more focused. We can decide what level of information matters, what the target should be, and what inputs are likely to be useful. Without that framing, even a large dataset can lead to shallow analysis. In other words, a clear question is what turns data work into purposeful analysis.

Data Preparation Is Part of the Analysis

You also saw that raw data is rarely ready for immediate use. Soccer data often includes missing values, inconsistent labels, awkward formats, duplicated records, and variables that need interpretation before they can support modeling.

That preparation step is not a minor technical chore. It is part of the analytical process itself. The choices we make while cleaning and structuring data shape what the model can learn and what the later results will mean. Bad data preparation does not merely slow the workflow. It weakens everything that follows. That is why preparation should be treated as part of the analytical argument, not just as a preliminary task.

Exploration Before Prediction

Another central lesson was that good modeling usually follows good exploration. Exploratory data analysis (EDA) helps us understand the structure of the problem before we formalize it. It reveals patterns, inconsistencies, and plausible signals. It also reminds us that some intuitively appealing variables may be weak, unstable, or misleading.

That matters because machine learning is not magic. If we do not understand the shape of the data, we are much more likely to build models that are fragile, overfit, or difficult to interpret. Exploration gives us the context needed to model with more discipline.

Feature Engineering Is Where Domain Knowledge Enters

One of the strongest themes of the book was the importance of feature engineering. Raw statistics such as shots, possession, or pass counts are rarely enough on their own. Their analytical value increases when we transform them into more meaningful representations: rolling form, venue splits, shot quality, contextual factors, and rating-based summaries of team strength.

The feature-engineering step is the point at which soccer understanding becomes part of the data representation. A model does not know on its own that recent form may matter more than season-long averages, that home and away performance should be separated, or that ten poor shots are not equivalent to three high-quality chances. Those ideas have to be encoded. This is one reason feature engineering remains so central even when more advanced models are available.

Models Support Judgment, They Do Not Replace It

Across the book, we introduced a range of models, from simpler baselines to more advanced machine learning and deep learning approaches. The point was never to suggest that one algorithm solves soccer. The point was to show that models are useful because they help us structure uncertainty.

A predicted win probability is not an oracle. It is an estimate built from data, assumptions, and modeling choices. Injuries, tactics, red cards, officiating, randomness, and match-specific context still matter. Good analytics improves judgment. It does not eliminate the need for it. That balance between evidence and judgment is one of the main themes in this book. Taken together, these lessons form the foundation you can now carry into more ambitious work.

The Next Stage in Soccer Analytics

If the earlier chapters established the foundation, the next question is where you can go from here. There is no single answer because soccer analytics now spans data engineering, statistical modeling, visualization, scouting, coaching support, recruitment, betting, broadcasting, and product development. Still, a few directions stand out as especially important. Each of the following directions extends something we already practiced in earlier chapters.

From Static Datasets to Living Data Pipelines

Most of this book worked with fixed datasets, and that was the right choice for learning. Static data makes it easier to focus on framing questions, cleaning tables, engineering features, and evaluating models without also worrying about infrastructure.

In practice, however, one of the most important next steps is automation. Instead of manually rerunning the same notebook whenever new information arrives, you may want a workflow that refreshes fixtures, lineups, injuries, standings, odds, or match events on a schedule. That schedule might run each day, each hour, or even more frequently, depending on the use case.

This is where APIs become useful. A service such as `API-Football` can provide ongoing access to fixtures, results, standings, team details, player information, and other match-related feeds. The key advantage is not only that the data is newer.

It is that the workflow can now be automated. Once the data source is available programmatically, the same analytical steps can run repeatedly with far less manual effort.

The right provider depends on league coverage, data granularity, historical depth, refresh frequency, pricing, and terms of use. For a first automated workflow, it is usually better to choose one provider, one competition, and one refresh schedule, then build something reliable before expanding.

For many readers, `API-Football` is a reasonable place to begin. It supports many practical football use cases, is accessible to independent analysts, and fits well with daily or higher-frequency refresh workflows.

A practical automation project might look like this:

1. Trigger a job on a schedule such as every morning or every hour.
2. Pull updated match, team, and player data from an API.
3. Store the raw responses and record when each refresh occurred.
4. Rebuild features and recompute forecasts automatically.
5. Save the updated outputs for later reports, dashboards, alerts, or further analysis.

That kind of workflow changes the nature of the project. You are no longer only analyzing a dataset that was prepared once. You are creating a system that updates itself as new data arrives.

That shift introduces a new set of questions worth learning:

- How to schedule jobs reliably
- How to handle authentication, rate limits, retries, and failed runs
- How to store raw data so the pipeline remains reproducible
- How to detect stale data or broken refreshes
- How to cleanly separate ingestion, transformation, and model scoring

This is one of the clearest ways to move from a learning project to real-world practice. Static datasets are excellent for building foundational skills. Automated workflows are what make repeated, up-to-date analytics possible. They are one of the strongest bridges from an exploratory notebook to a reliable analytical process.

From Team-Level Models to Player-Level and Lineup-Level Models

Many of the examples in this book worked at the team level. That is a reasonable and useful starting point because team-level data is widely available and easier to model. But matches are ultimately shaped by players, roles, and lineups. A natural next

step is to move beyond team-level features and incorporate player-level information, lineup composition, substitutions, and tactical roles. These analyses can be added directly to predictive models or developed as standalone tools for understanding how individual players and combinations of players affect match outcomes.

One important next step is therefore to bring player-level information into the pipeline. That might include likely starting 11s, injuries, suspensions, recent player form, positional profiles, or aggregated player metrics such as shot creation, passing progression, pressing intensity, or defensive coverage.

This matters because many team-level averages hide the very details that coaches, scouts, and analysts care about most. Two teams may have nearly identical attacking numbers, but one may depend heavily on a single winger for progression, while the other distributes chance creation across several players. Those differences become crucial when the lineup changes.

Player-level modeling also opens the door to more detailed scenario analysis. What happens if a holding midfielder is unavailable? How much attacking value is lost when a fullback who supplies width is missing? How does a team perform with one center-back pairing compared to another? These are much closer to the questions that clubs and analysts actually face.

Once player information enters the model, the question becomes more realistic. You are no longer asking only whether team A is stronger than team B in the abstract. You are asking how strong team A is under this particular lineup and match context.

Richer Data: Tracking, Video, and Multimodal Analysis

Event data is powerful, but it still captures only selected moments in the game. Tracking data goes further by recording player and ball locations over time. With it, analysts can study spacing, compactness, off-ball movement, press resistance, defensive shape, and pitch control.

That richer view matters because event data records only selected actions. It can tell us that a pass was completed or that a shot was taken, but not how the passing lane opened, how the defense shifted, or how a run away from the ball created the space for the chance. Tracking data helps analysts move from isolated events to the continuous structure of the match.

One especially valuable application is the study of off-ball movement. Some of the most important contributions in soccer are invisible in traditional summaries: a decoy run that drags a defender away, a subtle repositioning that opens a passing lane, or a coordinated press that forces play into a trap. With positional data, those behaviors become measurable. A striker's run can be evaluated not only by whether it receives the ball, but by whether it shifts defenders, opens a lane, or changes the shape

of the attack. That kind of analysis is much closer to the way coaches and scouts think about space creation.

Tracking data is also powerful on the defensive side. It allows analysts to study compactness, line spacing, recovery speed, and pressing structure in far more detail than event data alone. Instead of asking only whether a team pressed, we can begin asking when the press was triggered, how well the block stayed connected, and which spaces became vulnerable when the press was beaten. This brings tactical ideas such as shape, pressure, and rest defense into a more measurable analytical framework.

Computer vision provides another path. Instead of relying entirely on manually labeled events, it can extract structure directly from video. This opens the door to richer representations of the match, especially for clubs or analysts who have access to footage but not elite commercial data feeds.

This is also where pitch control models become especially interesting. Rather than asking only what happened, they ask which team was more likely to control a space at a given moment. That creates a bridge between positional data and tactical understanding: which spaces were dangerous, which options were available, and how pressure changed the quality of a decision. In practice, pitch control models can help evaluate passing options, identify which zones were realistically available, and show how defensive positioning changed the value of the next action. They are a good example of analytics moving beyond event counting and toward representing the continuous geometry of the game.

Wearable and biometric data can add yet another layer. GPS systems, inertial sensors, heart-rate monitoring, and related performance tools can help analysts connect tactical performance with fatigue, player load, recovery, and injury risk. For elite environments, this means analytics is no longer only about what a team did on the pitch, but also about what players may be physically able to sustain over a training block, a congested fixture list, or a return from injury.

This is especially valuable when the data is tracked over time rather than read as a one-off measurement. A drop in sprint output, a change in acceleration profile, or a growing asymmetry in movement may signal fatigue, overload, or a need for modified training before the problem becomes more serious. In that sense, wearable analytics helps connect performance analysis with sports science and medical decision-making.

In some environments, these signals can also support more immediate choices about substitution timing, recovery planning, or the management of player minutes. Even when the operational details differ across competitions and clubs, the larger point remains the same: richer physical data makes it easier to judge not only what a team is trying to do but also what its players are realistically ready to do.

Over time, the strongest analytical systems will likely combine multiple modalities:

- Structured match and event data
- Player and lineup information
- Tracking or positional data
- Text reports, injury news, or scouting notes
- Odds, market information, or schedule context

This is not only a question of getting more data. It is a question of representing the game more faithfully. As the representation improves, the kinds of questions you can ask also become more ambitious. Instead of asking only who is stronger, you can begin asking who controlled space better, who manipulated pressure more effectively, or which tactical patterns repeatedly created advantages.

Stronger Modeling Directions

The models in this book provide a solid foundation, but many readers will want to go further. The goal here is not to chase complexity for its own sake, but to match the method to the structure of the problem. Several natural directions exist, such as the following:

- More advanced and often more complex deep learning architectures are often a strong next step for tabular prediction problems, especially when feature interactions matter.
- Bayesian models are useful when uncertainty itself is a central part of the analysis.
- Sequence models, such as long short-term memory (LSTM) networks or transformer architectures (which are very successful in natural language processing), can help when the order of events matters, rather than just the final aggregate counts.
- Graph-based models can represent relationships among players or teams.
- Causal methods become important when the question shifts from prediction to intervention: what happens if a tactical choice changes, a player is removed, or a pressing scheme is adjusted?

Some of these directions are useful not only because they may improve accuracy, but because they answer different kinds of questions. Sequence models are helpful when order matters. Bayesian approaches are valuable when uncertainty needs to be quantified with a probabilistic distribution. Causal approaches matter when the analyst wants to estimate the likely effect of a change rather than merely describe historical association.

The key lesson is that model complexity should follow the problem, not the other way around. A more advanced model is worthwhile only when it captures structure that the simpler model cannot. That principle is easy to forget, but it is one of the safest guides for practical work.

From Notebooks to Analytical Products

Another meaningful next step is to think beyond analysis as a notebook and toward analysis as a product. That product might be a dashboard for coaches, a scouting interface for recruitment, a daily match preview for journalists, a betting monitor, or a research tool for your own use.

This shift forces a valuable discipline. It is one thing to produce an interesting result once. It is another to produce reliable outputs repeatedly, explain them clearly, and present them in a form that others can act on. That shift often marks the difference between analysis as an exercise and analysis as something genuinely useful.

This is also where analytics begins to affect more than match prediction. Products built on analytical workflows can support recruitment, tactical preparation, player development, media coverage, fan engagement, and even commercial operations. For coaches, that might mean opponent reports, set-piece tendencies, or match-state dashboards that help frame tactical decisions. For recruitment teams, it might mean tools that identify stylistic fit, flag undervalued players, or compare transfer targets across leagues. For player development staff, it might mean progress monitoring that helps track growth, workload, and role-specific strengths over time.

The same logic extends beyond performance departments. Media teams may use analytical outputs to build clearer match previews or post-match explainers. Broadcasters may use them to make advanced metrics more accessible to fans. Club leadership may use them to support pricing, planning, or longer-term resource decisions. In practice, the value of analytics often depends not just on the model, but on whether the insight can be delivered consistently to the people who need it, in a form they can actually use.

At that stage, software design, communication, and reliability become just as important as model performance.

AI and Large Language Models in the Analytics Workflow

Modern AI can also support this next stage of work. Throughout the book, we mainly treated machine learning as a modeling tool. But AI can also help with the broader analytics workflow itself. This is where AI becomes especially relevant for working analysts.

AI can help in at least two broad ways. First, it can work as an assistant while you build an analysis. Second, it can act more like an interface or agent that helps read information, organize it, and carry out tasks on your behalf.

The first role is AI-assisted analytics development. A large language model (LLM) can help draft code for data ingestion, cleaning, feature engineering, visualization, and predictive modeling. It can explain unfamiliar libraries, suggest debugging steps, summarize exploratory findings, and generate first drafts of reports. It can also help generate hypotheses worth testing, propose validation checks, and accelerate iteration when you are moving through an end-to-end workflow. This is part of why AI-assisted development has become so common. The analyst can move faster but still needs to define the question, judge which ideas are sensible, and verify the results carefully.

The second role is AI as an interface to analytics work itself. LLMs can sit between the user and the data, translating natural-language questions into code, SQL, filters, or visualizations. Instead of always writing every query manually, an analyst might ask, *Which teams have become more vulnerable to counterattacks late in matches?* or *Show me examples of high-xG chances created after a turnover in the final third.* The model can help turn that request into a query, a chart, or a first draft of an explanation.

This role becomes even more powerful when AI is connected to reports, databases, dashboards, and other tools. In that setting, AI is not only helping you write code. It may help read scouting reports, summarize recent match notes, combine structured metrics with unstructured text, or generate a first-pass tactical briefing on an opponent. With a more agentic design, such a system might also monitor multiple data sources, surface anomalies, draft updates, and help coordinate several small tasks across a workflow. Used carefully, this can reduce time spent on repetitive synthesis and free the analyst to focus on interpretation and decision-making.

At the same time, these systems should be treated as assistants, not authorities. Generated code must be checked. Generated hypotheses still need evidence. Generated summaries must be verified. An agent that can do more tasks can also make more mistakes at scale if it is not supervised well. This is the same principle we applied to models throughout the book: useful tools support judgment, but they do not replace it. That caution becomes even more important when the outputs are fluent, persuasive, and easy to trust too quickly.

Challenges and Limits

The future of soccer analytics is exciting, but it is important not to confuse possibility with ease. As the field becomes more sophisticated, the work does not become simpler. In many cases, it becomes more demanding: technically, organizationally, and ethically. If this chapter is meant to responsibly point forward, it should also make clear what still makes the field difficult.

Data Quality and Access

One of the biggest constraints in soccer analytics is not modeling, but data. The most valuable data sources, such as tracking data, biometric measurements, and some detailed event feeds, are often proprietary. That means the most ambitious analytical ideas are not equally available to everyone.

Even when data is available, it is rarely clean or perfectly aligned. Different providers may define the same event differently, use different coordinate systems, or update schemas over time. As a result, a large part of advanced work still depends on careful data validation, standardization, and documentation.

Interpretability and Trust

As models become more complex, they can also become harder to explain. A more accurate model is not automatically a more useful one if coaches, scouts, or decision-makers cannot understand what it is doing or why it is producing a certain result.

This is one reason simpler models often remain valuable in practice. Interpretability is not only a technical preference; it is part of how analytics earns trust inside real organizations. In many settings, the best model is not the most complex one, but the one that balances predictive power with clarity and usability.

Uncertainty Will Not Disappear

No matter how advanced the methods become, soccer will remain a sport with substantial uncertainty. It is low scoring, context sensitive, and heavily influenced by rare events: a deflection, a red card, an injury, a refereeing decision, or a moment of individual brilliance can change an entire match.

That means analytics should aim to improve decisions, not eliminate uncertainty. The goal is not perfect prediction. The goal is to reason better under imperfect information and to understand which conclusions are strong, which are tentative, and which are still too fragile to trust.

Ethics, Privacy, and Responsible Use

As data collection becomes more detailed, ethical questions become more important. Player monitoring, biometric measurement, automated scouting systems, and AI-assisted evaluation all create new opportunities, but they also create new responsibilities.

Analysts and organizations must think carefully about privacy, consent, fairness, and the possible misuse of models. A system that appears efficient can still be harmful if it reinforces bias, over-monitors players, or encourages false certainty in high-stakes decisions. Responsible analytics means paying attention not only to what can be built, but also to how it should be used.

Beyond Soccer

Although this book focused on soccer, the workflow we developed is much more general. The same underlying pattern appears again and again across applied machine learning:

1. Clearly define the task.
2. Gather and clean the data.
3. Represent the problem with useful features.
4. Fit models that match the structure of the task.
5. Evaluate carefully.
6. Interpret the results in context.

That pattern transfers well. In basketball, you might model possessions, lineups, and shot quality. In baseball, you might work with pitch sequences, batter-pitcher matchups, and defensive alignments. In volleyball, you might analyze rally structure, rotation effects, and serve-receive dynamics.

The same logic also extends far beyond sports. Fraud detection, recommendation systems, click prediction, credit risk, and operations forecasting all involve the same broad analytical cycle: messy data, structured representation, probabilistic modeling, and careful decision support.

This transfer matters because it reframes what you have learned. The book did not merely teach a set of soccer-specific metrics. It taught how to approach a prediction problem, how to turn domain knowledge into features, and how to responsibly interpret model outputs. Those skills remain useful even when the subject changes completely.

This broader transfer matters. It means that learning through soccer was never a detour from data science. It was a concrete, engaging way of learning the real work of data science.

A Practical Roadmap After This Book

If you want to keep going, the best next step is not to read endlessly. It is to build. A good post-book roadmap might look like this:

1. Rebuild one complete project from scratch. Take one problem from the book and reproduce it without looking line by line at the earlier chapter. This reveals what you truly understand.
2. Turn a static workflow into a repeatable pipeline. Use an API source such as `API-Football` or another provider to refresh data daily, store it cleanly, and rerun your features and forecasts on a schedule.
3. Add one richer layer of information. That could be player availability, lineup projections, market odds, tracking features, or textual injury reports.
4. Compare a strong baseline with a more advanced model. Do not assume complexity wins. Measure whether the added complexity actually helps.
5. Clearly communicate the result. Build a simple dashboard, report, notebook narrative, or app. An analysis becomes more valuable when it can be understood and reused.

This kind of progression keeps the work grounded. It turns isolated experiments into a durable skill set. It also gives you a practical way to keep improving without needing a perfect long-term plan in advance.

If you want a few concrete project directions, these are three especially good next steps:

- A daily match-preview pipeline that refreshes fixtures, form features, and predicted probabilities each morning
- A player-fit or recruitment tool that compares transfer targets by role, style, and expected contribution
- A team-strength dashboard that tracks rolling form, injuries, lineup changes, and market signals across a season

If you want a concrete way to sequence this, a useful pattern is to first reproduce, then extend, then automate, then communicate. Reproduce one idea from the book until you own it. Extend it with a richer data source or a better feature set. Automate it into a repeatable workflow. Then package it in a way that someone else can understand and use.

Final Thoughts

This book was not only about predicting soccer matches. It was about learning how to move from raw observations to structured analysis, from intuition to evidence, and from isolated metrics to decision-relevant insight.

Soccer analytics is ultimately not just about soccer; it's about how to spot important signals in a lot of noise—about seeing reality more clearly, truthfully, and accurately.

Soccer is a beautiful setting for that work because it is rich, dynamic, emotional, and difficult to reduce to simple explanations. The sport is structured enough to reward careful modeling but open enough that no model can capture it completely. There is always more context to consider, more uncertainty to respect, and more room for thoughtful interpretation. That tension is part of what makes soccer analytics worth doing.

Whether your next step is building a stronger soccer forecasting pipeline, developing a player-level scouting model, creating an API-driven daily workflow, applying these ideas to another sport, or using them in a field far outside sports, the core habits remain the same: ask clear questions, respect the data, engineer meaningful signals, choose models thoughtfully, and interpret results with humility.

Whatever you choose to build—even if it's not about soccer—should always follow a desire to learn to see more clearly in a world that rarely gives us certainty. In moments when you might doubt your abilities or when your head might be spinning from facing so much complexity, remember that you can almost always try to see just one percent more clearly.

The book ends here, but the work does not. Every dataset we study, every feature we design, every model we test, and every question we ask is part of a larger practice: learning to see more clearly in a world that rarely gives us certainty. That is the real promise of analytics—a disciplined way to stay curious, to challenge our assumptions, to make better decisions, and to keep improving. In the end, analytics is not only about understanding the game more deeply; it is about learning how to think, decide, and act with greater clarity in an uncertain world.

APPENDIX

References and Further Reading

This appendix serves as a curated guide to the literature that underpins the methods and concepts discussed in this book. It is divided into two main parts: a collection of recommendations for further study, organized by topic, and a list of works cited directly in the text. Each entry in the further reading list includes a brief annotation to help you understand its relevance and what you can expect to gain from it.

Whether you are looking to deepen your understanding of a specific statistical technique, explore the broader field of machine learning, or dive into the nuances of soccer-specific analytics, this guide provides a roadmap for your continued learning journey.

Further Reading by Topic

This section provides a curated list of books for readers who wish to delve deeper into specific areas of statistics, machine learning, and sports analytics. The books are organized by topic to help you find the resources most relevant to your interests.

Here are some suggestions for how to make the most of these resources:

Start with your interests.
: If you are particularly interested in machine learning, begin with the books in that section. If you want to deepen your understanding of regression, start with the foundational statistics books.

Match the level to your background.
: The level ratings are intended to help you choose books that are appropriate for your current knowledge. If you are new to a topic, start with the beginner or intermediate books before moving on to the advanced texts.

Don't feel obligated to read cover to cover.
: Many of these books are comprehensive references that are best used selectively. Identify the chapters or sections that are most relevant to your needs and focus on those.

Combine theory with practice.
: The best way to learn is to combine reading with hands-on practice. As you work through these books, try to apply the methods to your own data and projects.

Foundational Statistics and Regression

These books provide a strong foundation in the statistical concepts that are essential for any data analyst. They are recommended for readers who want to build a solid theoretical understanding of the methods used throughout this book.

Applied Linear Regression by Sanford Weisberg (Wiley, 2013)

: *Level*: Intermediate
This classic text provides a comprehensive and practical introduction to linear regression. It is an excellent resource for readers who want to move beyond the basics of regression and learn how to apply these methods in a rigorous and thoughtful way.

Introduction to Statistics and Data Analysis by Christian Heumann, Michael Schomaker, and Shalabh (Springer, 2023)

: *Level*: Beginner
This book offers a broad introduction to the essential concepts and tools of statistical thinking. With a good balance between theory and application, it is a valuable resource for those who are new to statistics or who want to refresh their knowledge.

Python for Probability, Statistics, and Machine Learning by José Unpingco (Springer, 2022)

: *Level*: Intermediate
This book provides a unique and valuable bridge between the theoretical concepts of probability, statistics, and machine learning and their practical implementation in Python. It is an excellent companion to this book, as it shows how to use Python to apply the statistical methods we have discussed.

The Elements of Statistical Learning by Trevor Hastie, Robert Tibshirani, and Jerome Friedman (Springer, 2009)

> *Level*: Advanced
> Often referred to as the bible of machine learning, this book provides a comprehensive and mathematically rigorous treatment of the field. It covers a wide range of topics, from linear models to neural networks, and is an essential reference for anyone who wants to develop a deep understanding of machine learning.

An Introduction to Categorical Data Analysis by Alan Agresti (Wiley, 2018)

> *Level*: Intermediate
> This book is a classic and highly readable introduction to the analysis of categorical data, covering the most important methods, including logistic regression and log-linear models. It is an excellent resource for readers who want to learn more about the classification methods used throughout this book.

Generalized Linear Models by P. McCullagh and J.A. Nelder (Chapman and Hall, 1989)

> *Level*: Advanced
> This book provides a unified and comprehensive treatment of generalized linear models (GLMs), which are a cornerstone of modern statistics. It is recommended for readers who want to move beyond the basics of regression and learn about a more general and powerful class of models.

Applied Logistic Regression by David W. Hosmer, Stanley Lemeshow, and Rodney X. Sturdivant (Wiley, 2013)

> *Level*: Intermediate to Advanced
> This is the definitive reference on logistic regression, covering both the theory and the practical application of this essential method. It is an excellent resource for readers who want to develop a deep understanding of classification models and how to apply them to real-world problems.

Statistical Inference by George Casella and Roger L. Berger (Chapman and Hall, 2024)

> *Level*: Advanced
> This is a comprehensive and rigorous treatment of the theory of statistical inference, covering all fundamental concepts, from probability theory to hypothesis testing and estimation. It is an essential reference for anyone who wants to master the mathematical foundations of statistics.

Advanced Statistical Methods

These books cover more specialized statistical methods that are useful for advanced applications. They are recommended for readers who want to explore topics beyond the scope of this book.

Generalized Linear Mixed Models: Modern Concepts, Methods and Applications by Walter W. Stroup, Marina Ptukhina, and Julie Garai (Chapman and Hall, 2024)

> *Level*: Advanced
> This book provides a comprehensive treatment of generalized linear mixed models (GLMMs), which extend generalized linear models (GLMs) to handle hierarchical or clustered data—a structure that arises frequently in soccer analytics when modeling players within teams or matches within seasons. It will be most useful to readers who are already comfortable with GLMs and want to extend their knowledge to more complex settings.

Mixed Models: Theory and Applications with R by Eugene Demidenko (Wiley, 2013)

> *Level*: Advanced
> This book provides a thorough treatment of mixed effects models with a focus on applications in R. It is recommended for readers who want to learn how to apply mixed models to data with nested or hierarchical structures, which are common in sports analytics.

Machine Learning and Deep Learning

These books cover the core concepts and methods of machine learning and deep learning, providing both theoretical foundations and practical guidance.

Pattern Recognition and Machine Learning by Christopher M. Bishop (Springer, 2006)

> *Level*: Advanced
> This book provides a comprehensive and mathematically rigorous treatment of machine learning, with a strong focus on Bayesian methods and probabilistic models. It is an excellent resource for readers who want to develop a deep understanding of the theoretical foundations of the field.

Deep Learning by Ian Goodfellow, Yoshua Bengio, and Aaron Courville (MIT Press, 2016)

> *Level*: Advanced
> This is the definitive textbook on deep learning, written by three of the field's leading researchers, covering all essential concepts from the basics of neural networks to advanced topics like convolutional and recurrent architectures. It is an essential reference for anyone who wants to understand the theory and practice of deep learning.

Sports Analytics

These books provide a broader perspective on sports analytics, covering methods and applications across multiple sports. They are recommended for readers who want to see how the techniques used in soccer analytics are applied in other contexts.

Analytic Methods in Sports: Using Mathematics and Statistics to Understand Data from Baseball, Football, Basketball, and Other Sports by Thomas Severini (Chapman and Hall, 2020)

> *Level*: Intermediate
> This book provides a comprehensive overview of the statistical methods used in sports analytics, with examples drawn from a variety of sports. It is an excellent resource for readers who want to see how the methods used in soccer analytics are applied to other sports and to appreciate the common challenges across the field.

Handbook of Statistical Methods and Analyses in Sports, edited by Jim Albert, Mark E. Glickman, Tim B. Swartz, and Ruud H. Koning (Chapman and Hall, 2020)

> *Level*: Intermediate to Advanced
> This comprehensive handbook covers a wide range of statistical methods used in sports analytics, with contributions from leading researchers in the field. It is an invaluable reference for anyone working in sports analytics, providing both theoretical insights and practical guidance on topics such as ranking methods, player evaluation, and game strategy.

Soccer-Specific Analytics

This book focuses specifically on soccer analytics, providing in-depth coverage of the methods and applications that are most relevant to the beautiful game.

Predictive Modelling for Football Analytics by Leonardo Egidi, Dimitris Karlis, and Ioannis Ntzoufras (Chapman and Hall, 2025)

> *Level*: Intermediate to Advanced
> This is the most recent and comprehensive book on soccer analytics, covering a wide range of predictive modeling techniques for match outcomes, player performance, and team success. It is an essential companion to the present text for readers who want a deeper dive into soccer-specific methods and case studies.

Sports Betting and Business Analytics

This book covers the business and betting aspects of sports analytics, providing insights into how analytics is used in commercial applications.

The Business of Sports Betting, edited by Becky Harris, John T. Holden, and Gil Fried (Human Kinetics, 2024)

> *Level*: Intermediate
> This book provides a comprehensive overview of the business side of sports betting, including the role of analytics in setting odds, managing risk, and identifying profitable opportunities. It is recommended for readers who are interested in the commercial applications of sports analytics.

Cited References

The following works were cited directly in the text of this book:

Agresti, Alan. *An Introduction to Categorical Data Analysis*. Wiley, 2006.

Albert, Jim, Mark E. Glickman, Tim B. Swartz, and Ruud H. Koning, eds. *Handbook of Statistical Methods and Analyses in Sports*. Chapman and Hall, 2017.

Bishop, Christopher M. *Pattern Recognition and Machine Learning*. Springer, 2006.

Casella, George, and Roger L. Berger. *Statistical Inference*. Chapman and Hall, 2024.

Demidenko, Eugene. *Mixed Models: Theory and Applications with R*. Wiley, 2013.

Egidi, Leonardo, Dimitris Karlis, and Ioannis Ntzoufras. *Predictive Modelling for Football Analytics*. Chapman and Hall, 2025.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

Harris, Becky, John T. Holden, and Gil Fried, eds. *The Business of Sports Betting*. Human Kinetics, 2024.

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer, 2009.

Heumann, Christian, Michael Schomaker, and Shalabh. *Introduction to Statistics and Data Analysis*. Springer, 2023.

Hosmer, David W., Jr., Stanley Lemeshow, and Rodney X. Sturdivant. *Applied Logistic Regression*. Wiley, 2013.

McCullagh, P., and J.A. Nelder. *Generalized Linear Models*. Chapman and Hall, 1989.

Severini, Thomas A. *Analytic Methods in Sports: Using Mathematics and Statistics to Understand Data from Baseball, Football, Basketball, and Other Sports*. Chapman and Hall, 2020.

Stroup, Walter W., Marina Ptukhina, and Julie Garai. *Generalized Linear Mixed Models: Modern Concepts, Methods and Applications*. Chapman and Hall, 2024.

Unpingco, José. *Python for Probability, Statistics, and Machine Learning*. Springer, 2022.

Weisberg, Sanford. *Applied Linear Regression*. Wiley, 2013.

Glossary

activation function
: A mathematical function in a neural network that determines the output of a node given a set of inputs. Common activation functions include ReLU (rectified linear unit), sigmoid, and tanh (hyperbolic tangent).

algorithm
: A set of rules or instructions to be followed in calculations or other problem-solving operations, especially by a computer. In machine learning, an algorithm is used to learn patterns from data.

application programming interface (API)
: A set of rules and protocols that allows different software applications to communicate with each other. In soccer analytics, APIs are often used to access data from providers like StatsBomb or Opta.

arbitrage
: In sports betting, an arbitrage opportunity exists when a bettor can place proportional bets on every possible outcome of an event across different bookmakers to guarantee a profit, regardless of the outcome.

area under the curve (AUC)
: A metric used to evaluate the performance of a binary classification model. It represents the area under the receiver operating characteristic (ROC) curve. A higher AUC value, closer to 1, indicates a better-performing model.

array
: A data structure that stores a collection of items, typically of the same data type. In Python, NumPy arrays are commonly used for numerical operations in data analysis.

backpropagation
: An algorithm used to train neural networks. It works by calculating the gradient of the loss function with respect to the network's weights, allowing the weights to be adjusted to minimize the loss.

bankroll management
: The practice of managing the funds used for betting to minimize risk and sustain betting activities over the long term. It involves strategies for determining bet sizes relative to the total bankroll.

bias-variance trade-off
: A fundamental concept in machine learning that describes the trade-off between a model's ability to fit the training data (low bias) and its ability to generalize to new, unseen data (low variance). Overly simple models tend to have high bias, while overly complex models tend to have high variance.

binary classification
: A type of classification task where the goal is to predict one of two possible outcomes. For example, predicting whether a soccer match will result in a win or not a win (i.e., a loss or a draw).

causality
: The relationship between cause and effect. In analytics, establishing causality is challenging but crucial for understanding whether a change in one variable directly causes a change in another, as opposed to being merely correlated.

class imbalance
: A situation in classification tasks where the number of observations in one class is significantly different from the number of observations in other classes. This can bias the model's performance toward the majority class.

coefficient
: A numerical factor in a term of a mathematical expression. In linear regression, coefficients represent the change in the dependent variable for a one-unit change in the corresponding independent variable.

Colley matrix
: A method for ranking teams or players based on a system of linear equations. It considers the number of wins and losses and the strength of the schedule.

confusion matrix
: A table used to evaluate the performance of a classification model. The table shows the number of true positives, true negatives, false positives, and false negatives.

cross-validation
: A technique for assessing how the results of a statistical analysis will generalize to an independent dataset. It involves partitioning the data into subsets, training the model on some subsets, and testing it on the remaining subset.

data cleaning
: The process of detecting and correcting (or removing) corrupt or inaccurate records from a record set, table, or database. It is a crucial step in the data analysis pipeline to ensure the quality of the data used for modeling.

DataFrame
: A two-dimensional labeled data structure with columns of potentially different types. In Python, the pandas DataFrame is a primary tool for data manipulation and analysis.

decision boundary
: In a classification model, a boundary that separates classes. The model predicts the class of a new data point based on which side of the boundary it falls.

decision tree
: A tree-like model used for both classification and regression tasks. It works by splitting the data into smaller and smaller subsets based on the values of the input features, creating a flowchart-like structure.

deep learning
: A subfield of machine learning based on artificial neural networks with many layers (deep neural networks). It has been successful in a wide range of tasks, including image recognition and natural language processing.

defensive actions
: In soccer, actions taken by a player to disrupt the opponent's attack. This can include tackles, interceptions, blocks, and clearances.

Elo rating
: A method for calculating the relative skill levels of players in zero-sum games. In this self-correcting system, the winner of a game takes points from the loser. The number of points gained or lost depends on the expected outcome of the game.

ensemble learning
: A machine learning technique where multiple models (often called *weak learners*) are trained to solve the same problem and combined to get better results. Random forests and gradient boosting are examples of ensemble methods.

epoch
: In the training of a neural network, a complete pass through the entire training dataset.

event-level data
: Detailed data that captures every on-the-ball event that occurs during a soccer match, such as passes, shots, tackles, and dribbles. Each event is typically timestamped and associated with a player and location on the pitch.

expected goals (xG)
: A metric that measures the quality of a shot by assigning it a probability of being scored. The probability is calculated based on various factors, such as the location of the shot, the type of assist, and the game state.

exploratory data analysis (EDA)
: The process of analyzing and visualizing datasets to summarize their main characteristics, often with visual methods. EDA is used to understand the data, discover patterns, and formulate hypotheses.

extreme gradient boosting (XGBoost)
: An optimized distributed gradient boosting library designed to be highly efficient, flexible, and portable. It implements machine learning algorithms under the Gradient Boosting framework.

F1 score
: A metric for evaluating a classification model's performance, calculated as the harmonic mean of precision and recall. It provides a balance between these two metrics.

feature engineering
: The process of using domain knowledge to create new features from existing data in order to improve the performance of machine learning models. This can involve combining features, creating interaction terms, or transforming features.

feature importance
: A technique used to determine the contribution of each feature to the prediction of a machine learning model. It helps in understanding which features are most influential in the model's decisions.

feature scaling
: The process of transforming the range of features in a dataset. This is often done to improve the performance of machine learning algorithms that are sensitive to the scale of the input features, such as K-nearest neighbors (KNN) and support vector machines.

feature selection
: The process of selecting a subset of relevant features from a dataset to use in a machine learning model. This can help improve model performance, reduce overfitting, and decrease training time.

goodness-of-fit
: A statistical measure of how well a model fits a set of observations. It summarizes the discrepancy between observed values and the values expected under the model in question.

gradient boosting
: An ensemble learning technique that builds a strong predictive model by sequentially adding weak learners (typically decision trees). Each new tree is trained to correct the errors of the previous ones.

heterogeneity
: The quality or state of being diverse in character or content. In soccer analytics, it refers to the fact that not all players or

teams are the same, and models should account for these differences.

hyperparameter
: A parameter whose value is set before the learning process begins. Examples include the learning rate in a neural network or the number of trees in a random forest. Hyperparameter tuning is the process of finding the optimal set of hyperparameters for a model.

implied probability
: The probability of an outcome occurring as implied by the odds offered by a bookmaker. This probability is calculated by converting the odds into a percentage.

interaction features
: Features created by combining two or more existing features. They can capture the interaction effects between variables, which may not be captured by the individual features alone.

intercept
: In a linear regression model, the predicted value of the dependent variable when all independent variables are equal to zero.

Jupyter Notebook
: An open source web application that allows you to create and share documents containing live code, equations, visualizations, and narrative text. This application is widely used in data science for exploratory analysis and modeling.

Kelly criterion
: A mathematical formula used to determine the optimal size of a series of bets to maximize the long-term growth of a bankroll. This formula takes into account the probability of winning and the odds of the bet.

K-nearest neighbors (KNN)
: A nonparametric machine learning algorithm used for both classification and regression. It predicts the class or value of a new data point based on the majority class or average value of its KNN in the feature space.

lagged features
: Features created from past values of a time series. They are used in time series forecasting to capture the temporal dependencies in the data.

linear regression
: A statistical method for modeling the relationship between a dependent variable and one or more independent variables by fitting a linear equation to the observed data.

logistic regression
: A statistical model used for binary classification. It models the probability of a binary outcome by fitting the data to a logistic function.

Loss function
: A function that measures the difference between the predicted output of a model and the actual output. The goal of training a machine learning model is to minimize this function.

match-level data
: Data that summarizes the events of a soccer match at a high level. This can include the final score, the teams involved, and the date of the match.

Matplotlib
: A popular Python library for creating static, animated, and interactive visualizations in Python. It is often used for creating plots and charts in data analysis.

matrix
: A two-dimensional array of numbers. In machine learning, matrices are used to represent data and perform mathematical operations.

mean squared error (MSE)
: A metric used to measure the average squared difference between the estimated values and the actual value. It is a common loss function for regression models.

missing data
: Values that are not available for a variable in a dataset. Handling missing data is an important step in data preprocessing.

model
: A mathematical representation of a real-world process. In machine learning, a model is trained on data to make predictions or decisions.

mplsoccer
: A Python library for plotting soccer data. This library provides a set of tools for creating soccer-specific visualizations, such as pitch maps and pass networks.

multiclass classification
: A classification task where the goal is to predict one of three or more possible outcomes. For example, predicting whether a soccer match will result in a win, loss, or draw.

multicollinearity
: A situation in which two or more independent variables in a regression model are highly correlated. This can make it difficult to determine the individual effect of each variable on the dependent variable.

neural network
: A machine learning model inspired by the structure and function of the human brain. This model consists of interconnected nodes (neurons) organized in layers. Each connection has a weight that is adjusted during training.

normalization
: A feature-scaling technique that scales the values of a feature to a fixed range, typically from 0 to 1.

NumPy
: A fundamental package for scientific computing in Python. NumPy provides support for large, multidimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays.

odds
: A way of expressing the probability of an event occurring. In betting, odds represent the ratio of the amount of money that can be won to the amount of money wagered.

one-hot encoding
: A technique for converting categorical data into a numerical format that can be used by machine learning algorithms. It creates a binary column for each category, with a 1 indicating the presence of that category and a 0 otherwise.

optimizer
: An algorithm used to update the weights of a neural network during training to minimize the loss function. Examples include stochastic gradient descent (SGD), Adam, and RMSprop.

out-of-bag (OOB) error
: A method for estimating the prediction error of a random forest model. It is calculated by testing each tree on the data points that were not included in its bootstrap sample.

overdispersion
: In a Poisson regression model, a situation where the variance of the data is greater than the mean. This can indicate that the model is not a good fit for the data.

overfitting
: A situation in which a machine learning model performs well on the training data but poorly on new, unseen data. This occurs when the model learns the noise in the training data instead of the underlying patterns.

overround
: In sports betting, the bookmaker's profit margin. It is the sum of the implied probabilities of all possible outcomes of an event, which is typically greater than 100%.

PageRank
: An algorithm originally used by Google to rank websites in its search engine results. PageRank can be adapted to rank teams or players in a network based on their connections and the importance of those connections.

pandas
: A popular Python library for data manipulation and analysis. It provides data structures like the DataFrame and Series, which are essential for working with structured data.

pass completion rate
: The percentage of a player's or team's passes that are successfully completed.

passes per defensive action (PPDA)
: A metric used to measure the intensity of a team's pressing. It is calculated by dividing the number of passes allowed by the opponent by the number of defensive actions (tackles, interceptions, and fouls) in the opponent's half.

pipeline
: In machine learning, a way to automate the workflow of a machine learning model. It can include steps for data preprocessing, feature engineering, and model training.

player-level data
: Data that focuses on the performance of individual players. This can include their physical attributes, their on-field actions, and their overall contribution to the team.

Poisson regression
: A regression model used to model count data. Poisson regression is often used in soccer analytics to predict the number of goals that will be scored in a match.

polynomial features
: Features created by raising existing features to a power. They can be used to capture nonlinear relationships in the data.

possession
: The amount of time that a team has control of the ball during a match. Possession is often expressed as a percentage.

precision
: A metric for evaluating a classification model's performance. Precision is the proportion of true positives among all the instances that were predicted as positive.

pressing intensity
: A measure of how aggressively a team tries to win back the ball when that team is not in possession. Pressing intensity can be quantified using metrics like PPDA.

progressive pass
: A pass that moves the ball significantly closer to the opponent's goal. It is a key metric for identifying players who are effective at advancing the ball up the field.

PyTorch
: An open source machine learning library based on the Torch library. PyTorch is known for its flexibility and ease of use, and is widely used for research and production in deep learning.

random forest
: An ensemble learning method that constructs a multitude of decision trees at training time and outputs the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees.

recall
: A metric for evaluating a classification model's performance. Recall is the proportion of true positives that were correctly identified. Also known as *sensitivity* or *TP rate*.

receiver operating characteristic (ROC) curve
: A graphical plot that illustrates the diagnostic ability of a binary classifier system as its discrimination threshold is varied. The curve is created by plotting the TP rate against the FP rate at various threshold settings.

regularization
: A technique used to prevent overfitting in machine learning models by adding a penalty term to the loss function. This penalty discourages the model from becoming too complex.

residual
: The difference between the observed value of the dependent variable and the value predicted by the regression model.

residual analysis
: The process of examining the residuals of a regression model to check for patterns or anomalies. Residual analysis is used to assess the validity of the model's assumptions.

return on investment (ROI)
: A performance measure used to evaluate the efficiency of an investment or compare the efficiency of multiple investments. In betting, ROI is the profit or loss made on an investment, expressed as a percentage of the total amount staked.

root mean squared error (RMSE)
: The square root of the mean squared error. It is a common metric for measuring the differences between values predicted by a model and the values observed.

R-squared
: A statistical measure of how well the data fits the regression model. It represents the proportion of the variance in the dependent variable that is predictable from the independent variable(s).

scikit-learn
: A free software machine learning library for the Python programming language. This library features various classification, regression, and clustering algorithms including support vector machines, random forests, gradient boosting, k-means, and DBSCAN, and is designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy.

Seaborn
: A Python data visualization library based on Matplotlib. Seaborn provides a high-level interface for drawing attractive and informative statistical graphics.

Shapley additive explanations (SHAP)
: A game-theoretic approach to explain the output of any machine learning model. It connects optimal credit allocation with local explanations, using the classic Shapley values from game theory and their related extensions.

shot quality
: A measure of the likelihood of a shot resulting in a goal. Shot quality is often based on the same factors as expected goals (xG), such as shot location and type of assist.

standardization
: A technique that rescales features so that they have the properties of a standard normal distribution with a mean of 0 and a standard deviation of 1.

TensorFlow
: An open source software library for high-performance numerical computation. Its flexible architecture allows easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices.

test set
: A subset of the data used to assess the performance of a final, trained machine learning model. A test set is used to provide an unbiased evaluation of the model's ability to generalize to new, unseen data.

threshold
: In a classification model, a value that is used to distinguish between classes. For example, in a logistic regression model, a threshold of 0.5 is often used to classify an instance as positive if the predicted probability is greater than 0.5.

training set
: A subset of the data used to train a machine learning model. The model learns the patterns and relationships in this data.

validation set
: A subset of the data used to tune the hyperparameters of a machine learning model. A validation set is used to select the best-performing model from a set of candidate models.

vector
: A one-dimensional array of numbers. In machine learning, vectors are used to represent data points and perform mathematical operations.

virtual environment
: A self-contained directory tree that contains a Python installation for a particular version of Python, plus a number of additional packages. A virtual environment allows you to work on a specific project without affecting other projects.

XGBoost
: See extreme gradient boosting.

zero inflation
: In a Poisson regression model, a situation where there are more zeros in the data than would be expected under a standard Poisson distribution. This can occur in soccer analytics when modeling goals, as many matches have zero goals.

Index

Symbols

1%-5% rule, 282
1X2 (match result), 260

A

activation function, 185-188
Adam optimizer, 196
AI in sports analytics, 295
Akaike information criterion (AIC), 108
analytical products, 294
API-Football, 289
arbitrage, 266-269
  execution risk, 269
  optimal stakes, 267
  rarity, 269
  restrictions, 269
  speed, 269
area under the curve (AUC), 136
ARIMA (autoregressive integrated moving average), 124
arithmetic operators, 25
assists, 230
association versus causation, 12
AUC (area under the curve), 136
autoregressive integrated moving average (ARIMA), 124

B

backpropagation, 190
bagging, 170
bankroll, 282
bankroll management, 282
bar charts, 60-61
Bayesian information criterion (BIC), 108
betting, 4, 259-285
betting market data, 11
bias, 186
bias-variance trade-off, 144
BIC (Bayesian information criterion), 108
binary classification, 128-138, 194
biometric data, 292
Bitbucket, 22
bookmaker, 259
bootstrap aggregating, 170
both teams to score (BTTS), 261
boxplots, 63-65
Brier score, 138
BTTS (both teams to score), 261

C

calibration, 138
camel case, 27
case studies, 75-80, 109-116, 144-148, 255-256
categorical features, 49-50, 117
causal inference, 124
class imbalance, 149, 179
  class weighting, 179
  class weights, 149
  evaluation metrics, 179
  oversampling, 149
  resampling, 179
  threshold tuning, 179
  undersampling, 149
classification, 127-160
classification tree (see decision trees)

closing line value (CLV), 283
coefficients, 99
Colley matrix, 240-241
comma-separated values (CSV), 39
common pitfalls, 120-122, 149-151
  data quality, 121
  model selection, 121
  multicollinearity, 120
  overfitting, 120
comparison operators, 25
computer vision, 292
conditional statements, 32-33
confusion matrix, 133-135, 199, 272
  false negatives (FN), 133
  false positives (FP), 133
  true negatives (TN), 133
  true positives (TP), 133
contextual comparison, 12
contextual features, 236-239
correct score bet, 261
cost functions (see loss functions)
cross-entropy loss (see log loss)
cross-validation, 109, 143
CSV (comma-separated values), 39

D

data cleaning, 57-59
data leakage, 119, 224
data pipelines, 289
DataFrame, 43
decimal odds, 262
decision boundaries, 99
decision threshold, 151-153, 179
decision trees, 164-169
  leaf, 164
  root, 164
  tuning, 167
deep learning, 183-220
defensive actions, 231
  interceptions, 231
  recoveries, 231
  tackles, 231
descriptive analytics, 9
dictionaries, 30-32
  key, 30
domain-specific features, 119
  defensive pressure, 119
  fatigue, 119
  motivation, 119

E

early stopping, 202
EDA (see exploratory data analysis)
Elo ratings, 245-247
English Premier League (EPL), 142
ensemble features, 249
ensemble methods, 123, 159, 176
  voting, 159
EPL (English Premier League), 142
epoch, 196
ethical considerations, 160
EV (expected value), 256, 260
event data, 10
event-level data, 41
expected assists (xA), 8
expected goals (xG), 7-8, 177, 226
  cumulative xG, 53
  model, 130
expected threat (xT), 8
expected value (EV), 256, 260
exploratory data analysis (EDA), 39-83
  aggregating, 55
  building DataFrames, 53
  filtering, 54
  frequency counting, 54
  grouping, 54

F

F1 score, 152
feature engineering, 116-119, 180, 221-257
  aggregated features, 180
  categorical encoding, 180
  spatial features, 180
  temporal features, 180
feature engineering pipeline, 251-255
feature importance, 171
feature scaling, 96, 156-158
feature selection, 150
features, 81
FIFA Women's World Cup (WWC), 42
fixed-stake betting, 275-277
fixture congestion, 236
for loops, 33-34
form, 237

forward propagation, 185, 188
fractional odds, 262
full-time home goals (FTHG), 200
full-time result (FTR), 197
functions, 36
  docstrings, 36

G

gegenpressing, 234
generalized linear model (GLM), 92
Git, 23
GitHub, 22
GitLab, 22
GLM (generalized linear model), 92
goodness-of-fit, 107
  deviance test, 107
  Pearson chi-square test, 107
Google Colab, 17
Google Gemini, 54
gradient boosting, 173
gradient descent, 191-192
  batch gradient descent, 191
  mini-batch gradient descent, 191
  stochastic gradient descent (SGD), 191

H

heatmaps, 68-70
  hexbin, 69
  kernel density estimate (KDE) plot, 73
  pass location, 68
heterogeneity, 12
heteroscedasticity, 107
histograms, 61-63
home advantage, 224, 272
  (see also venue splits)
hurdle models, 105-106
hyperbolic tangent (tanh), 188
hyperparameter, 150, 176

I

imbalanced data (see class imbalance)
implied probability, 264
influential observations, 108
  Cook's distance, 108
  leverage, 108
interaction features, 116

J

JavaScript Object Notation (JSON), 39
Jupyter Notebook, 20-22
  blended content, 21
  cells, 20
  extensibility, 21
  interactive computing, 20
  visualization, 21

K

k-nearest neighbors (KNN), 95-97, 141-144
  choosing k, 97, 142
  Euclidean distance, 142
Kelly criterion, 278-281
Keras, 209
key passes, 230
KNN (see k-nearest neighbors)

L

lagged features, 118
large language model (LLM), 295
levels of aggregation, 11-12
  league level, 11
  match level, 11
  player level, 11
  team level, 11
linear regression, 86-91
  intercept, 89
  line of best fit, 89
  slope, 89
list comprehensions, 35
lists, 27-28
  slicing, 28
LLM (large language model), 295
log loss, 138
log odds, 130
logistic regression, 129-133
logit, 195
loss functions, 138, 185, 189
  binary cross-entropy (BCE), 189
  categorical cross-entropy, 189
  mean absolute error (MAE), 98
  mean squared error (MSE), 190
  Poisson negative log-likelihood, 202
  Poisson regression loss, 190

M

MAE (mean absolute error), 98
market value, 86-102
match result (1X2), 260
match-level data, 40
Matplotlib, 42
mean absolute error (MAE), 98
mean squared error (MSE), 98, 190
min-max scaling, 157
missing values, 57-58, 158
  dropping, 57
  imputation, 57
  interpreting, 57
  missing indicator, 158
  predictive imputation, 158
MLP (multilayer perceptron), 195
model comparison, 102, 153-156, 176
model selection, 108
momentum (see form)
moneyline odds, 262
mplsoccer, 42, 68-74
MSE (mean squared error), 98, 190
MTL (see multitask learning)
multiclass classification, 139-141, 197
multicollinearity, 120
multilayer perceptron (MLP), 195
multinomial logistic regression, 139
multitask learning (MTL), 204-209
  accuracy, 204
  architecture, 204
  efficiency, 204
  robustness, 204
  shared layers, 204
  task-specific heads, 204

N

name standardization, 58-59
NaN type, 58
negative binomial regression, 104
neural networks, 183-193
  hidden layers, 185
  input layer, 185
  neurons, 186
  output layer, 185
normalization, 157
numerical features, 45-49
  continuous, 45
  discrete, 45
NumPy, 41

O

OBV (on-ball value), 11
odds, 4, 262-266
off-ball movement, 291
on-ball value (OBV), 11
one-hot encoding, 117
ordinal features, 51-52
out-of-bag (OOB) error, 172
over/under bets, 261
overdispersion, 103
overfitting, 98, 150, 168, 192
overround, 265-266

P

PageRank, 242-244
pandas, 41
pass accuracy, 55
passes per defensive action (PPDA), 8, 232
passing networks, 70-71
performance features, 222-228
physiological and performance data, 10
pip, 21
pipeline, 81
pitch control, 292
pitch visualizations, 68-74
player-level data, 40
player-level features, 249-251
playmaker, 230
Poisson goal models, 5
Poisson regression, 92-94, 200
polynomial features, 117
possession, 228
PPDA (passes per defensive action), 8, 232
practical applications, 177-180
precision, 135
predictive analytics, 9
predictors (see features)
prescriptive analytics, 10
pressing intensity, 2, 232
primitive data types, 24-25
  Booleans, 24
  floats, 24
  integers, 24
  strings, 24

progressive pass, 230
Python, 15-37
PyTorch, 193-209

R

R-squared, 90
random forest, 169-173
  bagging, 170
  feature randomness, 170
  limitations, 173
  tuning, 172
ranking features, 239-249
recall, 135
recruitment, 3, 11, 294
rectified linear unit (ReLU), 187
regression, 85-125
regularization, 150, 193
  dropout, 193
  L1 (Lasso), 193
  L2 (Ridge), 193
ReLU (rectified linear unit), 187
repositories, 22-23
  cloning, 23
  downloading a ZIP file, 23
residual analysis, 106
  Anscombe residuals, 107
  deviance residuals, 106
  Pearson residuals, 106
residuals, 91
return on investment (ROI), 275
risk of ruin, 282
RMSE (root mean squared error), 98
ROC curve, 136-137
ROI (return on investment), 275
rolling standard deviation, 224
rolling windows, 223
root mean squared error (RMSE), 98

S

scatterplots, 66-68
scikit-learn, 88
scouting, 3, 11, 294
Seaborn, 42
sensitivity, 135
set pieces, 234-236
sets, 29-30
Shapley additive explanations (SHAP), 174
shot maps, 72
shot quality, 56, 226
sigmoid function, 129, 188
SMOTE (synthetic minority oversampling technique), 179
snake case, 27
softmax, 212
softmax regression, 139
softplus activation, 201
standardization, 157
StatsBomb, 8, 42
strength of schedule, 239
synthetic minority oversampling technique (SMOTE), 179

T

tactical features, 228-236
tanh (hyperbolic tangent), 188
target variable, 81
TensorFlow, 209-219
test set, 99
time series plots, 65-66
totals, 261
tracking data, 10, 291
training set, 98
tree-based models, 163-181
tuples, 28-29

U

UEFA Champions League (UCL), 142

V

VAEP (valuing actions by estimating probabilities), 11
validation set, 98, 142
value bet, 264
valuing actions by estimating probabilities (VAEP), 11
vanishing gradient, 188
variable assignment, 26-27
variance inflation factor (VIF), 120
venue splits, 224, 288
venv, 18
version control, 23
VIF (variance inflation factor), 120
virtual environments, 17-19

isolation, 18
project specificity, 18
reproducibility, 18
visualizations, 59-68

W

wearable data, 292
weights, 186
while loops, 34
WWC (FIFA Women's World Cup), 42

X

xA (expected assists), 8
xG (see expected goals)
XGBoost, 173-177
tuning, 176
xT (expected threat), 8

Z

zero inflation, 104-105

About the Authors

Haipeng Gao is a data science and machine learning expert with extensive industry experience building and optimizing large-scale machine learning systems at leading technology companies, including LinkedIn, TikTok, and PayPal. He holds a PhD in statistics and operations research from the University of North Carolina at Chapel Hill, has taught statistics and probability at UNC Chapel Hill and San Jose State University, and holds multiple AI/ML patents granted by the U.S. Patent and Trademark Office.

Ari Joury is a data scientist and entrepreneur working at the intersection of machine learning, causal inference, and applied analytics. He is the founder and CEO of Wangari Global, where he builds AI systems for decision support in finance, insurance, and sustainability, with a focus on interpretable models and real-world impact. Trained originally as a theoretical particle physicist, he later transitioned into applied data science and quantitative modeling. Ari holds a PhD in physics and an MBA.

Weining Shen is an associate professor of statistics at the University of California, Irvine. His research focuses on machine learning, Bayesian statistics, sports analytics, and large language models. Dr. Shen has published more than 70 papers in leading statistics journals and at premier machine learning conferences. He earned his PhD in statistics from North Carolina State University.

Guanyu Hu is an associate professor at Michigan State University. His research focuses on spatial statistics, Bayesian nonparametric methods, and sports analytics. He has led multiple NSF-funded projects and published more than 50 papers in leading statistics journals and machine learning conferences. Dr. Hu serves as an associate editor for several prominent journals, was chair of the American Statistical Association's Statistics in Sports Section in 2024, and is one of the organizers of the American Soccer Insights Summit. He received his PhD in statistics from Florida State University.

Colophon

The animal on the cover of *Soccer Analytics with Machine Learning* is the Kentish plover (*Anarhynchus alexandrinus*), a small shorebird with a wide geographic range.

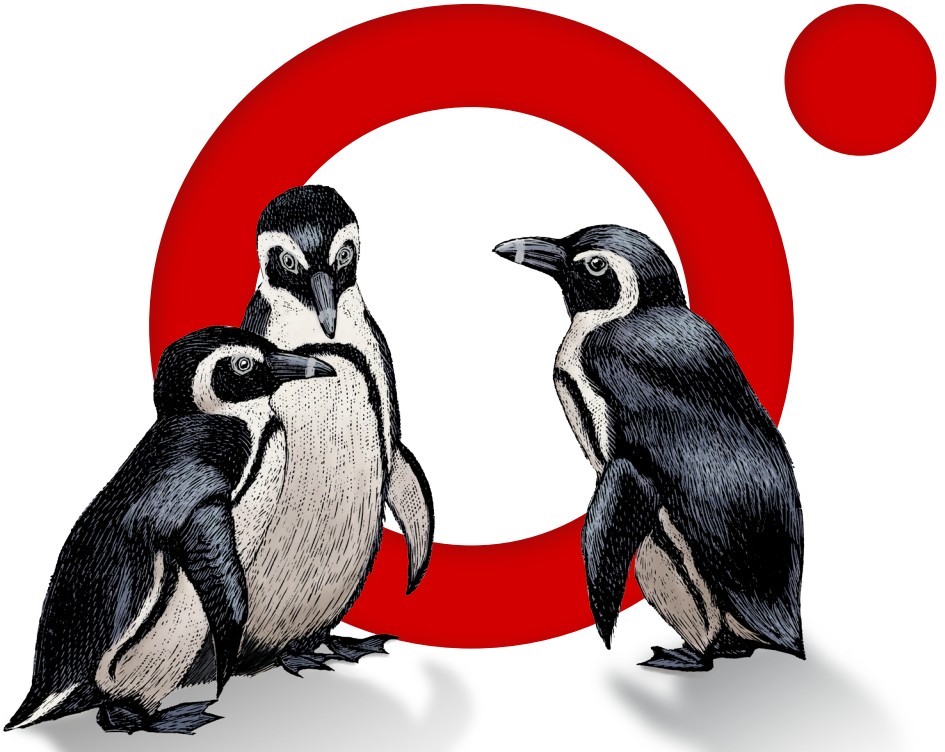
Though named for the English county of Kent, Kentish plovers can be found in northern and western Africa, the Arabian Peninsula, East Asia, and Western Europe. They prefer sea coasts, lake shores, and wetlands. They nest on the ground in small, shallow scrapes, which the males prepare as a form of courtship.

Kentish plovers eat mostly insects, along with aquatic creatures such as molluscs, worms, and crustaceans. Their proportionally long legs are well suited for their technique of running short distances and pausing to scan the area for food. The males change in appearance during breeding season, developing black markings on the head, neck, and chest, while females and nonbreeding males retain the more subdued look of a brownish-gray back and a white belly.

When they pair off to breed, both parents share the duty of incubating the eggs. After the chicks hatch, it is common for either the male or female parent to desert the nest and the other parent to stay behind. If a nesting parent thinks that its eggs or chicks are in danger, it will pretend to be injured in order to draw a predator's attention away from the nest.

Although the global population is in decline, Kentish plovers are not considered vulnerable or endangered. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Monica Kamsvaag, based on an antique line engraving from *British Bird*. The series design is by Edie Freedman, Ellie Volckhausen, and Karen Montgomery. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.



O'REILLY®

Learn from experts. Become one yourself.

60,000+ titles | Live events with experts | Role-based courses
Interactive learning | Certification preparation | Verifiable skills

Try the O'Reilly learning platform free for 10 days.

©2026 O'Reilly Media, Inc. O'Reilly is a registered trademark of O'Reilly Media, Inc. 1035600_7x9.1875