Module 3. Case study

We are reaching the end of the Data Analytics certificate. In this module, we will apply the many concepts and techniques we have learned to do a case study of solving a real-world problem in sports analytics.

We will build an Expected Goals (xG) model to predict the probability of a shot turning into a goal in football/soccer. Expected goals are a fundamental framework for analyzing player and team performances in football. We have picked this well-known problem because many public models exist, and all are built slightly differently with different datasets, techniques and algorithms.

Unit 3.1 Problem statement

Build an Expected Goal (xG) model in football to predict the goal probability of a shot.

Unit 3.2 How do we go about this?

Here are some early questions we need to answer based on the problem statement.

3.2.1 What are the inputs to the model?

The inputs to the model are information surrounding the "shot" before the shot is taken. This last bit about information "pre-shot" is key because for a model to be predictive, we must predict its outcome right before the shot leaves the foot of the footballer. If we use any "post-shot" information, like if the shot was saved or hit the post, etc., the model ceases to be predictive.

Data companies collect "pre-shot" metadata, such as location, shot type (foot or head), whether the shot has come off a regular play or set piece, and so on. A combination of these will form the model's inputs.

Input features (X): shot location (distance, angle), body part, etc.

Target variable (y): whether the shot resulted in a goal (1) or not (0).

3.2.2 What are the outputs of the model?

The model's output is a number between 0 and 1 that denotes the goal probability. Example: 0.1, 0.45, 0.01 etc.

3.2.3 Knowing the inputs and outputs, which technique or algorithm best answers the question?

If we go back to module 4 of Course 3, we discussed how to pick techniques based on the problem type. To predict the probability of a binary outcome, logistic regression is a good choice to start with to build a baseline. However, logistic regression does not deal with non-linear relationships between the features.

We will use the supervised learning technique of xGBoost Classifier as it is much better suited for this project.

A fun exercise for students is to build a baseline model using logistic regression and compare the results with what we make in this model to see the differences in outputs of the two models.

3.2.4 What data do we need?

It is clear now that we need shot events and metadata around shot events to build this model. Since we use the supervised learning method, we train the model on data with an outcome label (goal or no goal) and then test it on a new data set previously unseen by the model.

How much data do we need to build a predictive model?

We must consider statistical power analysis and effect size to determine the minimum number of shots required for a reliable xG model. We will not go into detail here, as it is off-topic.
Some of the things to consider are the following.

- Expected event rate: in football, the typical shot-to-goal conversion rate is around 10% of shots (0.10).
- Number of features in your model.
- How rare are certain shot types.
- Granularity of predictions.

To satisfy all the above, we need a minimum of 10 000 shots. Ideally, closer to 20 000 shots to build a robust model.

20 000 shots would ensure we have enough coverage of the rare scenarios and enough shots for the "training set-testing set" split.

Not having the recommended data doesn’t mean you can't build a good model. In most real-world cases, there isn’t enough data to create the most robust model possible. Advanced techniques to mitigate sample size issues are beyond this certification's scope.

Sourcing the data. Where and How?

Sourcing data will not be an issue if you work at a football club in most leagues. Clubs have contracts with data providers, giving you access to event data from multiple leagues over multiple seasons.

Football averages 25–30 shots per match, so you need about 800–1000 matches' worth of shot data.

Luckily for us, the football data provider HUDL Statsbomb (https://www.statsbomb.com) has made a lot of free data available for enthusiasts like those taking this course. The data is available in a GitHub repository that you can clone and download the matches to your computer free of charge.

- Here is the link to the Statsbomb free data page.
- Landing page: https://statsbomb.com/what-we-do/hub/free-data/
- Github: https://github.com/statsbomb

This is a typical checklist of questions you go through for every problem. While this makes the initial process seem slow, it will make the subsequent steps faster and more efficient. At this point, you move to the next step, data.

Unit 3.3 Data

We have identified our data source in 2.6 and downloaded it to our computer. The next step is to examine the data and then figure out how to extract the specific subset of data (shots event data) for our project.

3.3.1 Statsbomb data

StatsBomb data are provided in JSON format. To understand this structure, see the Wikipedia entry for JSON (n.d.) at https://en.wikipedia.org/wiki/JSON. In the StatsBomb open-data GitHub repository (https://github.com/statsbomb/open-data/tree/master/data), files are organized by competition, match and data type, which simplifies navigation and analysis.

Figure 1. How the data is organized

Source: own elaboration based on Github, n.d.

We are interested in "shot" events, so let us check what is in the "events" folder.

Figure 2. Events folder

Source: own elaboration based on Github, n.d.

Inside the "events" folder, you see a seemingly endless stream of JSON files whose names are numbers, like the one highlighted, "15946.json."

Each file represents the events of a specific match. The number is the unique match_id assigned by Statsbomb to the particular match.

If you open the JSON file in a browser or an IDE like Visual Studio Code, you will slowly understand how the data is structured.

Here is a typical "shot" event in Statsbomb data. Please note that this is not the complete shot event. It runs over two pages. The following image is included here to familiarize you with the data format.

Some fields have been highlighted.

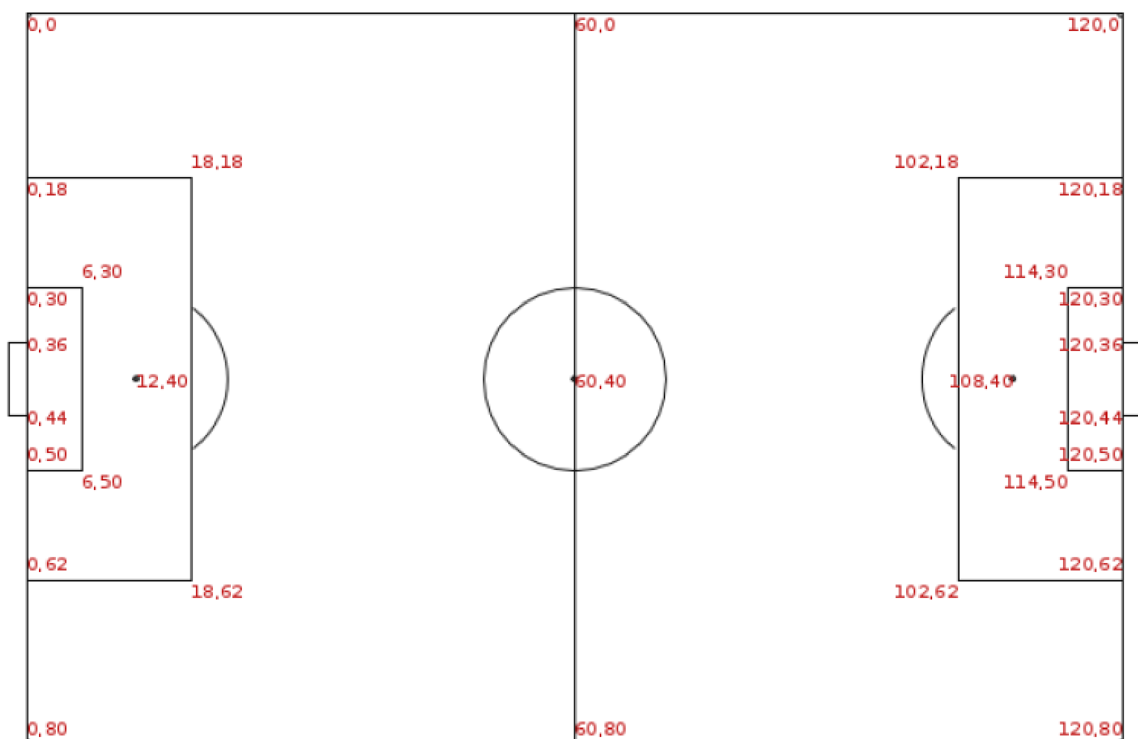
"Name" – name of the event.

"play_pattern" – the specific type of play during which the shot occurred. In this case, it happened off of a throw-in.

"location" – the [x,y] location of the shot on the pitch. Statsbomb uses a 120x80 grid. Here are the specifications of the Statsbomb pitch.

"Body part" – another piece of shot metadata indicating which foot the player used for the shot.

Figure 3. Statsbomb data 1



Source: own elaboration.

Figure 4. Statsbomb data 2


```
}, {
  "id" : "2258c11c-0916-4976-bfde-a66e89ffeb8d",
  "index" : 191,
  "period" : 1,
  "timestamp" : "00:03:31.114",
  "minute" : 3,
  "second" : 31,
  "type" : {
    "id" : 16,
    "name" : "Shot"
  },
  "possession" : 10,
  "possession_team" : {
    "id" : 207,
    "name" : "Valencia"
  },
  "play_pattern" : {
    "id" : 4,
    "name" : "From Throw In"
  },
  "team" : {
    "id" : 207,
    "name" : "Valencia"
  },
  "player" : {
    "id" : 4367,
    "name" : "Gonçalo Manuel Ganchinho Guedes"
  },
  "position" : {
    "id" : 16,
    "name" : "Left Midfield"
  },
  "location" : [ 101.0, 21.3 ],
  "duration" : 1.020612,
  "related_events" : [ "12f35fe0-b936-439e-8087-0777a7dc8a38" ],
  "shot" : {
    "statsbomb_xg" : 0.008752456,
    "end_location" : [ 117.0, 39.7, 1.6 ],
    "key_pass_id" : "430b01dd-eb13-4bf5-8df1-ecd0b76155a0",
    "body_part" : {
      "id" : 40,
      "name" : "Right Foot"
    },
    "type" : {
      "id" : 87,
      "name" : "Open Play"
    },
```

Source: own elaboration.

JSON format efficiently transmits large amounts of semi-structured data over the internet. However, we need the data in a more structured tabular format to run data analysis efficiently.

Python (and most other programming languages) supports flattening a JSON file into a structured table.

Extracting shot data

A typical soccer match has about 3000 events, including passes, tackles, shots, duels, etc. We are only interested in the shot data. We need to write code to strip the shot data from all the matches in the free Statsbomb database.

We will use the following Python script to do it. I ran this script locally on my laptop from Jupyter Notebook.

1. Import all necessary packages. Since you will reference folders on your computer, you need the Path library.

Figure 5. Import all necessary packages

```
import json
import os
import pandas as pd
from pathlib import Path
```

Source: own elaboration.

2. Add the path where the match files are stored.

Figure 6. Add the path where the match files are stored

```
# Define the correct path
base_path = r"C:\Users\ravi\sbdata\events"
```

Source: own elaboration.

This is where I have my data; you must ensure the code points to the correct folder containing the match files.

3. Code to extract shots from a single match.

Figure 7. Code to extract shots from a single match

```
def extract_shots_from_match(events_data):
    """Extract shot events from a match's events data"""
    shots = []
    for event in events_data:
        if event.get('type', {}).get('name') == 'Shot':
            shots.append(event)
    return shots
```

Source: own elaboration.

This is straightforward: extract all events where "type" = "Shot". Pandas handles the JSON parsing.

4. Now, we add code to iterate through the entire set of match files and extract the shot events. We iterate through each match and call the function "extract_shots_from_match" to extract the shot data from each match. We added some error handling to make sure if the code fails, it will do so gracefully.

Figure 8. Add code to iterate through the entire set of match files and extract the shot events

```
def get_all_shots(events_directory):
    """Process all JSON files in the directory and extract shots"""
    all_shots = []

    # List all files in directory
    print(f"\nListing contents of {events_directory}:")
    try:
        files = os.listdir(events_directory)
        print(f"Found {len(files)} files")

        # Process JSON files
        for filename in files:
            if filename.endswith('.json'):
                file_path = os.path.join(events_directory, filename)

                try:
                    with open(file_path, 'r', encoding='utf-8') as f:
                        match_events = json.load(f)

                    match_shots = extract_shots_from_match(match_events)
                    match_id = filename.replace('.json', '')
                    for shot in match_shots:
                        shot['match_id'] = match_id

                    all_shots.extend(match_shots)
                    print(f"Processed {filename}: Found {len(match_shots)} shots")

                except Exception as e:
                    print(f"Error processing {filename}: {str(e)}")

    except Exception as e:
        print(f"Error accessing directory: {e}")

    return all_shots
```

Source: own elaboration.

5. The next bit is the primary function where we execute the logic we have written.

Figure 8. Execute the logic

```
def main():
    print("Starting to process files...")

    # Get all shots
    shots = get_all_shots(base_path)

    if shots:
        # Convert to DataFrame
        shots_df = pd.json_normalize(shots)

        # Save to CSV in the current directory
        output_file = 'statsbomb_shots.csv'
        shots_df.to_csv(output_file, index=False)
        print(f"\nSuccessfully saved {len(shots_df)} shots to {output_file}")
        print(f"Columns in the dataset: {list(shots_df.columns)}")
    else:
        print("No shots found in the events files")

if __name__ == "__main__":
    main()
```

Source: own elaboration.

Shot data is saved to a CSV file on my computer.

At this point, we have all the shots extracted from all the matches that we downloaded. The next step is data preparation – explore, format, clean, and prepare the data for analysis

Data preparation

1. **Explore the data**

Table 1. Explore the data

location	duration	match_id	type.id	type.name	play_pattern.name	possession_team.n
[111.5, 52.9]	1.075902	15946	16	Shot	From Throw In	Barcelona
[113.9, 26.4]	0.807592	15946	16	Shot	Regular Play	Barcelona
[93.7, 34.7]	0.979318	15946	16	Shot	From Keeper	Barcelona
[109.2, 39.1]	0.312149	15946	16	Shot	Regular Play	Deportivo AlavÃ©s
[107.8, 24.7]	0.937618	15946	16	Shot	From Corner	Barcelona
[108.6, 27.8]	2.555973	15946	16	Shot	From Free Kick	Barcelona
[112.5, 41.7]	1.166493	15946	16	Shot	From Corner	Barcelona
[96.5, 52.6]	1.228019	15946	16	Shot	From Free Kick	Barcelona

id	play_pattern.name	under_pressure
index	team.id	shot.aerial_won
period	position.id	out
timestamp	position.name	shot.one_on_one
minute	shot.statsbomb_xg	shot.saved_to_post
second	shot.end_location	shot.deflected
possession	shot.key_pass_id	shot.saved_off_target
location	shot.outcome.id	shot.open_goal
duration	shot.outcome.name	shot.follows_dribble
related_events	shot.first_time	shot.redirect
match_id	shot.technique.id	off_camera
type.id	shot.technique.name	shot.kick_off
type.name	shot.body_part.id	shot.freeze_frame
possession_team.id	shot.body_part.name	team.name
possession_team.name	shot.type.id	player.id
play_pattern.id	shot.type.name	player.name

Source: own elaboration

Summary

- There are a total of 87000+ shots. This is excellent news because this is a lot more than the minimum 20000 shots we needed to build a robust model.
- Each shot has 48 data columns. I've highlighted the most relevant ones for this project.
- The green highlighted ones signify the key features + identity of the player, team, etc.
    - Player name, team name.
    - Location, type name, shot technique, shot outcome play pattern, shot type.
- The blue highlighted ones are different true/false flags that can be used as features in the model.
    - If the shot was under pressure, one-on-one with the keeper, open goal, shot follows a dribble, deflected, shot after winning an aerial duel, redirected shot.
- The shot.statsbomb_xg column is the xG of the shot according to Statsbombs internal model. We will compare our final output to theirs at the end of the project.

- Freeze Frame data provides additional context to the shots, giving the number of teammates and defenders between the GK and the shooter. However, we are not going to use this in this project.

This gives us a good sense of what we have regarding different columns and facets in the data we can use in our model.

3.3.2 Filter and trim the data

If you look at the different teams in the *possession_team.name* column, there is a wide variety. The data spans multiple seasons and competitions, including men's and women's competitions and club and national team competitions. National team and youth national team competitions tend to be very different from club competitions as they are infrequent, and the level of competition varies a lot. Since we have abundant data samples, We've excluded national team data from the model.
When I filtered them out from the CSV, I was left with about 74 000 shots.

Penalties: penalties are unique in football. They are high-probability goal events where there is practically no other context apart from the keeper and the penalty taker. We will exclude these from our analysis as they have a standard conversion rate (around 0.78). We will still be left with 73 000+ shots for our analysis. You will see that I excluded them in the analysis code later.

This process can be automated and handled during the extraction phase, but we needed to explore the data before determining what to trim or filter.

3.3.3 Data cleaning and formatting

The next step is data cleaning and formatting to have it in the right shape for analysis. Some of the most common gotchas

- Look for mismatched datatypes in columns – like text in a date column
- Empty or Null values.

If you look at the shots CSV file, you will see a lot of empty values for the True/False columns like a shot.one_on_one, under_pressure, etc.

Figure 9. Shots CSV file

AJ	AK	AL
under_pressure	shot.aerial_won	out
TRUE	TRUE	
		TRUE
TRUE	TRUE	

Source: own elaboration.

These flags are relatively rare and are "FALSE" by default. They are only marked when they are "TRUE" and left empty otherwise. We can address this in two ways.

- Fill them all with FALSE.
- Or handle them in the analysis/model code.

For simplicity of the model code, I filled all these empty cells with FALSE in the CSV.

At this point, we have our data ready for the most fun part – model building and analysis.

If you have been wondering, "I thought building predictive models was cool and fun", here is a little joke to cheer you up.

Why do data scientists make terrible writers? Because they spend 80% of their time cleaning up data and only 20% telling the story, they call that a "good" ratio!

It's funny because it's painfully true, most data scientists dream of building complex models and discovering groundbreaking insights but end up spending most of their time dealing with missing values, standardizing formats, and figuring out why there's an emoji in what should be a numeric column.

You can see the above manifest in our project so far.

3.3.4 Model building

We have our data set ready. Now, we want to start building the model. The first step is identifying the features. You can do this via a discovery process by running a regression to see which features available in the dataset correlate strongly with goal/no goal.

However, as data scientists and analysts have some expertise in the subject matter—in this case, the game of football—we can list some basic features and discover the secondary features through the programmatic route.

1. **Features**

It is common sense that a shot needs to be taken close to the goal and preferably in a central location concerning the goal mouth for it to be a goal.

- That is two basic features – Distance to the goal and angle to the goal mouth.
- Others include the play pattern, shot type, and the rest of the true/false flags.

Derived features

The angle to the goal mouth – the angle to the center of the goal and the angle to two goal posts are not readily available. However, we can derive them based on the shot location and the coordinates of the goalposts. These angles are a proxy for a more intuitive feature, "visible goal area".

At this point, we have enough features to start coding the model.

2. **Model code**

Figure 10. Import the requisite packages

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import roc_auc_score, brier_score_loss, log_loss
import math
from sklearn.ensemble import GradientBoostingClassifier
import ast
```

Source: own elaboration.

The helper method extracts the [x,y] coordinates from the column. Error checking ensures that null values do not go downstream.

Figure 11. Extract coordinates

```python
def extract_coordinates(loc_str):

    try:
        if isinstance(loc_str, str):
            coords = ast.literal_eval(loc_str.strip())
            return coords[0], coords[1]
        elif isinstance(loc_str, list):
            return loc_str[0], loc_str[1]
        return np.nan, np.nan
    except:
        return np.nan, np.nan
```

Source: own elaboration.

Calculate the derived features.

- Shot distance.
- Shot angle.
- Distance to the center line of the goal.

Figure 12. Calculate the derived features

```python
def calculate_shot_features(x, y):
    """Calculate derived geometric features from shot location"""
    try:
        # Goal posts are at (120, 36) and (120, 44) - center at (120, 40)
        goal_center = (120, 40)
        left_post = (120, 36)
        right_post = (120, 44)

        # Distance to goal
        distance = math.sqrt((goal_center[0] - x)**2 + (goal_center[1] - y)**2)

        # Angle to goal (in radians)
        angle_to_posts = [math.atan2(post[1] - y, post[0] - x) for post in [left_post, right_post]]
        shot_angle = abs(angle_to_posts[1] - angle_to_posts[0])

        # Distance to center line of goal
        distance_to_center_line = abs(y - 40)

        return {
            'distance': distance,
            'shot_angle': shot_angle,
            'distance_to_center_line': distance_to_center_line
        }
    except:
        return {k: np.nan for k in ['distance', 'shot_angle', 'distance_to_center_line']}
```

Source: own elaboration.

Preprocessing data

- Extract location coordinates.

- Calculate derived features from the shot location.
- Process all the binary features (the true and false).
    - Shot.first_time, under_pressure, shot.follows_dribble etc.
- Create “Is assisted” feature.
    - Use the column “shot.key_pass_id” to determine if the shot is assisted or unassisted.
        - If shot.key_pass_id is non-empty, then the shot is assisted. Else, it is unassisted.
- Create dummy variables for categorical features.
    - Shot.body_part.name
    - Shot.type.name
    - Play_pattern.name
- We need dummy variables for categorical features because most models and algorithms can only work with numerical values.
- Lastly, we combine all the features and return them along the target variable, whether the shot outcome is “goal” or “no goal.”

Figure 13. Preprocessing data

```python
def preprocess_data(data):
    # Make a copy of the data
    data = data.copy()
    print(f"Initial shape: {data.shape}")
    # Extract coordinates
    coords = data['location'].apply(extract_coordinates)
    data['x'] = coords.apply(lambda x: x[0])
    data['y'] = coords.apply(lambda x: x[1])
    # Calculate geometric features
    shot_features = data.apply(lambda row: calculate_shot_features(row['x'], row['y']), axis=1)
    for key in ['distance', 'shot_angle', 'distance_to_center_line']:
        data[key] = shot_features.apply(lambda x: x[key])

    # Process binary features
    binary_features = [
        'shot.first_time',
        'under_pressure',
        'shot.follows_dribble',
        'shot.open_goal',
        'shot.one_on_one',
        'shot.aerial_won',
        'shot.redirect'
    ]
    # Process binary features - these should all be True/False
    for col in binary_features:
        data[col] = data[col].astype(int)
        print(f"{col}: {data[col].sum()} True values ({data[col].mean()*100:.2f}%)")

    # Create assisted feature from key_pass_id
    data['is_assisted'] = (~data['shot.key_pass_id'].isna()).astype(int)
    print(f"is_assisted: {data['is_assisted'].sum()} True values ({data['is_assisted'].mean()*100:.2f}%)")
    # Create dummy variables for categorical features
    categorical_features = [
        'shot.body_part.name',
        'shot.type.name',
        'play_pattern.name'
    ]
    dummies = pd.get_dummies(data[categorical_features])
    # Combine all features
    features = pd.concat([
        data[[
            'distance',
            'shot_angle',
            'distance_to_center_line',
            'shot.first_time',
            'under_pressure',
            'shot.follows_dribble',
            'shot.open_goal',
            'shot.one_on_one',
            'shot.aerial_won',
            'shot.redirect',
            'is_assisted'
        ]],
        dummies
    ], axis=1)
    # Create target
    target = (data['shot.outcome.id'] == 97).astype(int)

    return features, target, data
```

Source: own elaboration.

The next step is to train the model.

- Split the data into training and test set. (80% data for training, 20% for testing).
- Train the model using the GradientBoostingClassifier.

- Make predictions for training and test data sets.
- Calculate metrics – brier score and roc_auc scores to evaluate the model performance.
- Rank the features by their importance.

Figure 14. Train the model

```python
def train_model(features, target):
    """Train the xG model"""
    print("Starting model training...")
    print(f"Total samples: {len(features)}")
    print(f"Goal distribution: \n{target.value_counts(normalize=True)}")

    # Split data
    X_train, X_test, y_train, y_test = train_test_split(
        features, target, test_size=0.2, random_state=42, stratify=target
    )

    # Scale features
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    # Train model
    model = GradientBoostingClassifier(
        n_estimators=200,
        learning_rate=0.1,
        max_depth=3,
        random_state=42
    )

    model.fit(X_train_scaled, y_train)

    # Make predictions
    train_probs = model.predict_proba(X_train_scaled)[:, 1]
    test_probs = model.predict_proba(X_test_scaled)[:, 1]

    # Calculate metrics
    metrics = {
        'train_roc_auc': roc_auc_score(y_train, train_probs),
        'test_roc_auc': roc_auc_score(y_test, test_probs),
        'train_brier': brier_score_loss(y_train, train_probs),
        'test_brier': brier_score_loss(y_test, test_probs),
        'train_log_loss': log_loss(y_train, train_probs),
        'test_log_loss': log_loss(y_test, test_probs)
    }

    # Feature importance
    feature_importance = pd.DataFrame({
        'feature': features.columns,
        'importance': model.feature_importances_
    }).sort_values('importance', ascending=False)

    print("\nTop 10 Most Important Features:")
    print(feature_importance.head(10))

    return model, scaler, metrics, feature_importance
```

Source: own elaboration.

Save the predictions along with all the columns in the input CSV.

Figure 15. Save the predictions

```python
def save_predictions(original_data, features, model, scaler):
    """Generate and save predictions"""
    print("Generating predictions...")

    # Scale features and generate predictions
    features_scaled = scaler.transform(features)
    xg_predictions = model.predict_proba(features_scaled)[:, 1]

    # Create output dataframe with all original columns
    output = original_data.copy()
    output['predicted_xg'] = xg_predictions
    output['actual_goal'] = (output['shot.outcome.id'] == 97).astype(int)

    print("Saving predictions to CSV...")
    output.to_csv('xg_predictions.csv', index=False)
    print("Predictions saved to xg_predictions.csv")

    return output
```

Source: own elaboration.

Lastly, we run the model on our shot data in the primary function.

Figure 16. Run the model

```python
def main():
    print("Reading data...")
    data = pd.read_csv('statsbomb_shots_cleaned.csv', low_memory=False)

    # Preprocess data
    features, target, processed_data = preprocess_data(data)

    # Train model
    model, scaler, metrics, feature_importance = train_model(features, target)

    # Print metrics
    print("\nModel Performance Metrics:")
    for key, value in metrics.items():
        print(f"{key}: {value:.3f}")

    # Save predictions
    predictions = save_predictions(processed_data, features, model, scaler)

    return model, metrics, predictions, feature_importance

if __name__ == "__main__":
    model, metrics, predictions, feature_importance = main()
```

Source: own elaboration.

Woohoo! We built our first model from scratch.

3.3.5 Model outputs and analysis

Let us look at the model's outputs and analyze our xG model.

Figure 17. Summary statistics

```
Final feature shape: (74639, 28)
Number of goals: 8267
Goal percentage: 11.08%
```

Source: own elaboration.

Figure 18. Features

```
Features included: ['distance', 'shot_angle', 'distance_to_center_line', 'shot.first_time', 'under_pressure', 'shot.follows_dri
bble', 'shot.open_goal', 'shot.one_on_one', 'shot.aerial_won', 'shot.redirect', 'is_assisted', 'shot.body_part.name_Head', 'sho
t.body_part.name_Left Foot', 'shot.body_part.name_Other', 'shot.body_part.name_Right Foot', 'shot.type.name_Corner', 'shot.typ
e.name_Free Kick', 'shot.type.name_Open Play', 'shot.type.name_Penalty', 'play_pattern.name_From Corner', 'play_pattern.name_Fr
om Counter', 'play_pattern.name_From Free Kick', 'play_pattern.name_From Goal Kick', 'play_pattern.name_From Keeper', 'play_pat
tern.name_From Kick Off', 'play_pattern.name_From Throw In', 'play_pattern.name_Other', 'play_pattern.name_Regular Play']
```

Source: own elaboration.

Figure 19. Top 10 most important features

	feature	importance
1	shot_angle	0.378568
0	distance	0.136643
11	shot.body_part.name_Head	0.122275
18	shot.type.name_Penalty	0.095074
6	shot.open_goal	0.089160
7	shot.one_on_one	0.047604
17	shot.type.name_Open Play	0.039841
8	shot.aerial_won	0.029142
19	play_pattern.name_From Corner	0.023067
2	distance_to_center_line	0.016806

Source: own elaboration.

Figure 20. Model performance statistics

```
train_roc_auc: 0.818
test_roc_auc: 0.801
train_brier: 0.077
test_brier: 0.080
train_log_loss: 0.268
test_log_loss: 0.278
```

Source: own elaboration.

Model performance analysis

ROC AUC: 0.801 on the test set (0.818 on train)

- Strong discriminative ability.
- A small gap between train/test indicates good generalization.

- Solid performance for football xG models.

Brier score: 0.080 on test set

- Good calibration of probabilities.
- Low value indicates reliable probability estimates.
- Small train/test gap (0.077 vs 0.080) shows stable predictions.

Log loss: 0.278 on test set

- Reasonable performance, given the imbalanced nature of the data.
- Small gap with train (0.268) confirms good generalization.

Key overall insights

- Geometric features (angle and distance) are crucial predictors.
- Situation-specific features (penalties, open goals) have high importance when they occur.
- Body part (headers) significantly affects conversion probability.
- The model seems to learn well from rare but key events (open goals, penalties).
- Play patterns have relatively lower importance compared to shot characteristics.

Figure 21. How does our model compare with the Statsbomb xG model?

```
Mean Difference: 0.0065
Median Difference: 0.0051
Std of Difference: 0.0709
Mean Absolute Error: 0.0384
RMSE: 0.0712

Correlation: 0.8765

Overall xG Comparison:
Our Model Average xG: 0.1109
Statsbomb Average xG: 0.1044
```

```
xG by Actual Outcome:

Goals:
Count: 8267
Our Model: 0.2944
Statsbomb: 0.2979

Non-Goals:
Count: 66372
Our Model: 0.0881
Statsbomb: 0.0803
```

```
xG by Shot Type:

Open Play:
Count: 70041
Our Model: 0.1057
Statsbomb: 0.0997
Correlation: 0.8486

Free Kick:
Count: 3738
Our Model: 0.0602
Statsbomb: 0.0400
Correlation: 0.5928

Penalty:
Count: 840
Our Model: 0.7715
Statsbomb: 0.7835
Correlation: nan

Corner:
Count: 20
Our Model: 0.2854
Statsbomb: 0.0002
Correlation: nan
```

Source: own elaboration.

Analysis

Even though Statsbomb uses an entirely different model to predict xG and incorporates different features, including freeze frame data, there is a strong alignment between the two models.

Overall:

- High correlation (0.8765) shows strong general agreement.
- Small mean difference (0.0065) indicates our model slightly overestimates compared to Statsbomb.

- RMSE of 0.0712 and MAE of 0.0384 suggest reasonably close predictions.
- Overall average xG: Ours (0.1109) vs Statsbomb (0.1044).

By shot type:

- **Open play shots (94% of shots):**
    - Good correlation (0.8486).
    - Slight overestimation (0.1057 vs. 0.0997).
    - Most reliable predictions.
- **Free kicks:**
    - Moderate correlation (0.5928).
    - Significant overestimation (0.0602 vs. 0.0400).
    - Model needs improvement for free kicks.
- **Penalties:**
    - Similar averages (0.7715 vs. 0.7835).
    - NaN correlation might be due to constant values.
    - Slightly underestimates compared to Statsbomb.
- **Corner shots:**
    - Major discrepancy (0.2854 vs 0.0002).
    - Tiny sample size (20 shots).
    - Needs investigation or might need to be merged with another category.

By outcome:

- **Goals:**
    - Very close average xG (0.2944 vs 0.2979).
    - Slight underestimation of actual goals.
- **Non-goals:**

    - Slight overestimation (0.0881 vs 0.0803).
    - Consistent with overall pattern.

All xG calculations are approximations and depend significantly on the modeling techniques and features. As such, there is no absolute "ground truth". If we do consider Statsbomb_xg as the "ground truth" here, here are the things we need to improve in our model.

- Free kick modeling.
- Corner shots need special handling or reclassification as it is such a small sample.
- Probably should separate models for different shot types, especially for penalties.

Following these steps, implementing your xG model with different features and/or techniques, and comparing the results with our model or the Statsbomb_xg model is a great exercise.

Unit 3.4 Next steps: fine-tuning, operationalization and maintenance

3.4.1 Fine-tuning

Now that we have an xG model that works well for the most part, most of our work is done, but not all of it. We can try to make the improvements mentioned in the model evaluation analysis above. This will make the model more complicated but better handle the edge cases.

This is another area you can explore on your own.

3.4.2 Operationalization

Once we are thoroughly satisfied with our xG model, we need to deploy it into daily use. This is usually the task of the data engineers. This is not trivial because:

- New matches are played almost daily, so new data needs to be processed. This needs to be done in an automated way. We run the xG update process every night at 2 a.m.

- Create downstream visuals and easy ways for data scientists and analysts to build reports and further analyze this data.

3.4.3 Maintenance

On a longer horizon (once every 6 months or so), re-run model evaluation processes to see if the model needs an update or if new edge cases need to be handled.

Unit 3.5 Conclusion

In this module, we started by building a predictive model and went through every step of creating one using the example of an Expected Goals Model (xG) in football. Regarding learning objectives, we wanted to give you a flavor of what it is like to work as a data professional in the industry.

We strongly encourage you to pick a problem or a question and use this module as a guide to explore the fascinating world of building mathematical models. You will learn and have so much fun.

Further reading

Beazley, D., and Jones, B. K. (2022). *Python Cookbook: Recipes for Mastering Python* 3 (3° ed.). O'Reilly Media.

Bishop, C. M. (2006). *Pattern recognition and machine learning*. Springer.

Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The elements of statistical learning: Data mining, inference, and prediction* (2° ed.). Springer.

Leskovec, J., Rajaraman, A., and Ullman, J. D. (2020). *Mining of massive datasets* (3° ed.). Cambridge University Press.

Python Software Foundation. (2024). Python documentation. https://docs.python.org/3/

VanderPlas, J. (2023). *Python data science handbook: Essential tools for working with data* (2° ed.). O'Reilly Media.