6 Fitting a decision tree and a random forest

This chapter covers

- Decision trees and random forests
- Model interpretation and evaluation
- Mathematical foundations
- Data exploration through grouped bar charts and histograms
- Common data wrangling techniques

In the previous chapter, we solved a classification problem using logistic regression, achieving 87% accuracy in predicting the variety of Turkish raisins based on their morphological features. In this chapter, we will approach a similar classification problem using two powerful modeling techniques: decision trees and random forests.

A *decision tree* is a simple, intuitive model that makes decisions by recursively splitting the data into subsets based on the most significant feature at each step. It operates like a flowchart, where each internal node represents a decision based on a feature, each branch represents the outcome of that decision, and each leaf node represents a class label or a regression value. Decision trees are easy to interpret and visualize, making them a popular choice for both classification and regression tasks.

A *random forest*, on the other hand, is an ensemble learning method that constructs multiple decision trees during training. For classification problems, the random forest outputs the class that is selected by the majority of the trees (a process known as *majority vote*), whereas for regression tasks, it averages the predictions of the individual trees. Random forests are particularly effective because they combine the predictions of many trees, which reduces the risk of overfitting and improves model robustness and accuracy.

6.1 Understanding decision trees and random forests

Building a decision tree involves selecting the best features to split the data at each node. The algorithm evaluates various splits and chooses the one that best separates the classes, typically using metrics such as Gini impurity (which measures misclassification) and information gain (which quantifies reduction in uncertainty) for classification tasks. For regression tasks, the algorithm evaluates splits using criteria such as mean squared error (MSE) and mean absolute error (MAE), both of which measure how well the split reduces the variability in the continuous target variable. The tree continues to grow by adding branches until it reaches a stopping criterion, such as a maximum depth or a minimum number of samples per leaf. The resulting tree is then used to make predictions by following the path from the root to a leaf node.

A random forest takes this process a step further. It is an ensemble technique that constructs multiple decision trees using different subsets of the data, which are randomly selected with replacement through a process called *bagging* (bootstrap aggregating). Each tree is trained on a slightly different data set, leading to variations in the trees. Additionally, random forests introduce randomness by selecting a random subset of features to consider at each split, ensuring that the trees are diverse and not overly correlated. This diversity among the trees strengthens the overall model, making it more resistant to overfitting and enhancing its generalization ability. Furthermore, random forests provide an importance ranking of features based on how often each feature is selected across trees and its effect on reducing impurity, offering valuable insights into which predictors contribute most to the model.

Ensemble techniques

Ensemble techniques, such as combining logistic regression with decision trees or using random forests, use multiple models to enhance prediction accuracy and robustness. By aggregating predictions from diverse models, ensemble methods reduce the risk of overfitting and increase the generalizability of the prediction.

For example, in a random forest, numerous decision trees operate on slightly different subsets of the data and/or features, each contributing a prediction. The final output is typically the majority vote or the average of these predictions, smoothing out anomalies and errors that individual models might make. Similarly, combining fundamentally different models such as logistic regression and decision trees allows the

(continued)

strengths of one model to compensate for the weaknesses of another, offering a more balanced and reliable prediction across various scenarios. This synergy effectively harnesses the unique capabilities of each model type, leading to more accurate and stable outcomes.

For classification tasks, the final prediction of a random forest is determined by a majority vote among the trees, whereas for regression tasks, it is the average prediction of all the trees. This approach results in a model that is typically more accurate and robust than a single decision tree. Furthermore, random forests have the advantage of requiring less preprocessing than regression models, because they are not sensitive to the scale or distribution of the input variables and can naturally handle both numerical and categorical data without requiring normalization or transformation. They are typically faster to fit in high-dimensional settings due to their randomized approach. After training, feature importance can be easily estimated and visualized, providing insights into which variables have the most significant effect on the predictions.

We will first demonstrate how to build a single decision tree, interpret its structure, and evaluate its performance. Following that, we will explore how to construct and use the power of a random forest to solve our classification problem, showing the benefits of random forests and the ease with which they can be implemented and interpreted.

But before exploring the nuts and bolts of decision trees and random forests, we will introduce the classification problem we'll be solving: predicting whether an American football team will successfully convert a fourth-down attempt. To solve this problem, we'll begin by importing a real data set, demonstrating common data wrangling techniques to prepare the data for analysis, and then exploring the data by computing basic statistics and creating graphical content to reveal leading insights.

6.2 Importing, wrangling, and exploring the data

American football is a game of strategy and skill that evolved from English rugby and soccer—although nowadays, nobody would mistake one for the others. Players use their hands to control the ball, not their feet, and teams take turns in possession of the ball. The game is played with 11 players on each side. The team on offense attempts to advance the ball downfield to score 3 points by kicking a field goal between the goal posts or 7 points by running or passing the ball into their opponent's end zone—and then converting the extra point. The team on defense tries to stop them.

Teams must advance the ball against their opponent's defense at least 10 yards at a time over three successive plays, also known as downs, to maintain possession and get a new set of downs. Fourth down is the critical down. If out of range for a field goal attempt, most teams choose to punt the ball and relinquish possession to their opponent. However, depending on the situation, teams sometimes go for it: that is, rather than punting, they call a run or pass play in an attempt to gain the required yardage for a new set of downs. They convert the fourth down or they don't. It's a classic risk-and-reward scenario with binary outcomes: if successful, teams maintain possession of the ball and keep alive their chances for a score, but if unsuccessful, their opponent gains possession where the fourth down failed.

We will attempt to predict whether a professional football team converts a fourth-down play by fitting a decision tree and then a random forest to a data frame containing play-by-play data from the 2023 National Football League (NFL) regular season and postseason. Our data set is a .csv file stored in our working directory. We typically read .csv files into Python by passing the filename (including the .csv extension) to the `pd.read_csv()` method from the `pandas` library. However, in this instance, it makes sense to add the `usecols` parameter to our code to specify which variables, or columns, to import rather than the entire data set. Using `usecols` is often preferred over importing everything and then subsetting, as it reduces memory usage and speeds up data loading, particularly when working with large data sets that contain many irrelevant variables. This approach helps streamline the data import process by bringing in only the columns essential to our analysis:

```
>>> import pandas as pd
>>> nfl_pbp = pd.read_csv('nfl_pbp.csv',
>>>                    usecols = ['QUARTER',
>>>                               'DOWN',
>>>                               'TO_GO',
>>>                               'OFFENSIVE_TEAM_VENUE',
>>>                               'SCORE_DIFFERENTIAL',
>>>                               'PLAY_TYPE',
>>>                               'YARDS_GAINED'])
```

Let's review these variables one by one.

6.2.1 Understanding the data

Even if you're well-acquainted with American football, some of the `nfl_pbp` variables may not be immediately intuitive. Each observation in the data corresponds to a single play:

- `QUARTER` represents the period in which a play occurred. Games are typically divided into four quarters, so `QUARTER` is an integer that, for most observations, equals `1`, `2`, `3`, or `4`. However, some games end in a tie, and then an overtime period is played to determine a winner. For those plays that occurred in overtime, `QUARTER` equals `5`.
- `DOWN` is an integer that represents the play's down number, with a value equal to `1`, `2`, `3`, or `4` corresponding to a first-down, second-down, third-down, or fourth-down play, respectively.
- `TO_GO` is an integer that equals the number of yards remaining for a first down; it therefore equals any whole number equal to or greater than 1.

- OFFENSIVE_TEAM_VENUE is a character string that can be either Road or Home. When the visiting team is in possession of the ball, OFFENSIVE_TEAM_VENUE is set to Road; otherwise, it is set to Home.
- SCORE_DIFFERENTIAL is an integer that equals the number of accumulated points scored by the home team minus the number of accumulated points scored by the road team. It can therefore be a negative number if the road team has accumulated more points, a positive number if the home team has accumulated more points, or zero if both teams have accumulated an equal number of points.
- PLAY_TYPE is a character string that represents the category of play that occurred. For instance, if the team on offense executed a running play, PLAY_TYPE equals Run; if a field goal was attempted, PLAY_TYPE equals Field Goal.
- YARDS_GAINED is an integer that represents the number of yards gained on a play; it can be positive, negative, or zero.

Unfortunately, our data is not plug-and-play ready for the type of problem we intend to solve, which means it requires some wrangling before we can fit a decision tree or a random forest to it.

6.2.2 Wrangling the data

Data wrangling is the process of transforming and preparing raw data into a clean and usable format for analysis. This may involve one or more of the following steps: reshaping the data by gathering columns into rows or spreading rows into columns, extracting observations that meet a logical criterion, selecting variables by name, transforming data types, correcting errors or inconsistencies, handling missing values, or joining data from other sources. The goal of data wrangling is to ensure the data is accurate, complete, and properly formatted to avoid a "garbage in, garbage out" problem. Wrangling data doesn't do much toward demonstrating how to fit a decision tree or a random forest, but it is typically required as a prerequisite to fitting any type of model or performing any sort of analysis.

DATA FILTERING

Data filtering is the process of subsetting a data frame based on specific logical criteria. Our data frame, nfl_pbp, now contains every play from the 2023 NFL season, including the postseason. However, we only need fourth-down plays that involve a run or pass; we don't need plays from other downs, nor do we need fourth-down plays that included a punt or a field-goal attempt. So, we filter nfl_pbp where the variable DOWN equals 4 and where the variable PLAY_TYPE equals Pass or Run:

```
>>> nfl_pbp = nfl_pbp[(nfl_pbp['DOWN'] == 4) & \
>>>                   (nfl_pbp['PLAY_TYPE'].isin(['Pass', 'Run']))]
```

The code creates a pair of Boolean series that each resolve to TRUE when the specified logical criteria are met. The bitwise AND operator combines the two Boolean series to create a final Boolean series that resolves to TRUE only if both conditions are met.

Missing values

The variable YARDS_GAINED contains several NaN values, which is short for "Not a Number." NaN is a special floating-point value that typically represents undefined or unrepresentable numerical results, which makes sense for those plays—a timeout, for instance—where there were no yards to be gained. Although we've stripped those observations from the nfl_pbp data frame, YARDS_GAINED also equals NaN rather than 0 when a passing or running play gained no yards.

This is a minor inconvenience that we can quickly and easily fix. In the following line of code, we call the fillna() method to replace every instance of NaN in YARDS_GAINED with 0:

```
>>> nfl_pbp['YARDS_GAINED'] = nfl_pbp['YARDS_GAINED'].fillna(0)
```

This common fix addresses a frequent problem, thereby preventing errors and inaccuracies when the data is subjected to analysis. Although replacing NaN values with 0 is appropriate in this case, it's important to note that this approach may not be suitable in all situations. Imputing missing data can significantly affect model predictions, potentially leading to unintended patterns, such as an excess of zeros or distorted minima. Carefully consider the implications and context of the data before choosing an imputation method.

Data transformation

Let's say the home team has accumulated 24 points and the road team has accumulated 17 points. The variable SCORE_DIFFERENTIAL therefore equals 7, because it is derived by subtracting the number of points accumulated by the road team from the number of points accumulated by the home team. If the variable OFFENSIVE_TEAM_VENUE equals Home, it indicates that the home team has a 7-point advantage over the road team, which is what we want. But if OFFENSIVE_TEAM_VENUE instead equals Road, SCORE_DIFFERENTIAL should equal –7 rather than 7; after all, the road team is *trailing* by 7 points.

The following line of code inverts the sign of the SCORE_DIFFERENTIAL values where OFFENSIVE_TEAM_VENUE equals Road by multiplying the raw data by -1:

```
>>> nfl_pbp.loc[nfl_pbp['OFFENSIVE_TEAM_VENUE'] == \
>>>              'Road', 'SCORE_DIFFERENTIAL'] *= -1
```

The net effect of this operation is that positive numbers are flipped to negative and negative numbers are flipped to positive, but only when the variable OFFENSIVE_TEAM_VENUE is equal to Road. Where SCORE_DIFFERENTIAL equals 0, the data remains unchanged, of course, because multiplying 0 by any number still results in 0.

Data type conversion

Machine learning models, including decision trees and random forests, require numeric inputs; so, converting categorical string data to numeric data types makes the data frame fully compatible with machine learning algorithms. The following snippets of code convert the values in OFFENSIVE_TEAM_VENUE and PLAY_TYPE from strings to integers (0 and 1):

```
>>> OFFENSIVE_TEAM_VENUE_mapping = {'Road': 0, 'Home': 1}
>>> nfl_pbp['OFFENSIVE_TEAM_VENUE'] = \
>>>     nfl_pbp['OFFENSIVE_TEAM_VENUE'].map(OFFENSIVE_TEAM_VENUE_mapping)

>>> PLAY_TYPE_mapping = {'Run': 0, 'Pass': 1}
>>> nfl_pbp['PLAY_TYPE'] = \
>>>     nfl_pbp['PLAY_TYPE'].map(PLAY_TYPE_mapping)
```

We have simply converted the string values of both variables to integers. For OFFENSIVE_TEAM_VENUE, Road has been mapped to 0 and Home to 1; similarly, for PLAY_TYPE, Run has been mapped to 0 and Pass to 1.

DERIVED VARIABLE

Decision tree and random forest algorithms use a categorical target variable—binary or otherwise—during training to build and optimize each decision tree, whether as part of an ensemble or as a standalone model. It might go without saying, but our data frame does not contain a binary variable indicating whether a professional football team converted a fourth-down play; so, we need to create a new feature, or attribute, from existing nfl_pbp data through a mathematical operation and assign it as our target variable.

From the variable TO_GO, we know the number of yards needed to convert a fourth down and get a new set of downs; and from the variable YARDS_GAINED, we know the number of yards gained on the play. So if YARDS_GAINED is less than TO_GO, then our derived variable, CONVERT, should equal 0, indicating failure; otherwise it should equal 1, indicating success.

The following snippet of code creates a new variable called CONVERT and appends it to the nfl_pbp data frame. It compares the values from TO_GO and YARDS_GAINED and assigns a value of 0 or 1 to the new variable:

```
>>> nfl_pbp['CONVERT'] = np.where(nfl_pbp['YARDS_GAINED'] < \
>>>                               nfl_pbp['TO_GO'], 0, 1)
```

When the number of yards gained is less than the number of yards needed to earn a new set of downs, it's assumed the fourth-down attempt failed, and a value of 0 is assigned to the derived variable CONVERT. Alternatively, when the number of yards gained is equal to or greater than the yards needed, we're assuming the fourth-down attempt was successful, and a value of 1 is assigned to CONVERT.

SUMMARY

Now that we have our data frame in order, let's summarize it. The info() method returns a concise summary of the nfl_pbp data frame:

```
>>> print(nfl_pbp.info())
<class 'pandas.core.frame.DataFrame'>
Index: 883 entries, 16 to 52381
```

```
Data columns (total 8 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   QUARTER                883 non-null    int64
 1   DOWN                   883 non-null    float64
 2   TO_GO                  883 non-null    float64
 3   OFFENSIVE_TEAM_VENUE   881 non-null    float64
 4   SCORE_DIFFERENTIAL     883 non-null    int64
 5   PLAY_TYPE              883 non-null    int64
 6   YARDS_GAINED           883 non-null    float64
 7   CONVERT                883 non-null    int64
dtypes: float64(4), int64(4)
memory usage: 62.1 KB
None
```

This tells us the following about our data:

- There are 883 observations (rows) and 8 variables (columns) in `nfl_pbp`. So, during the 2023 NFL regular season and postseason, there were 883 instances where teams ran a play from scrimmage—that is, a running play or a passing play—in an attempt to convert a fourth down.
- There are no remaining null values. We effectively eliminated them by converting every `NaN` in `YARDS_GAINED` to `0`.
- Every variable is now numeric because we converted the character strings of `OFFENSIVE_TEAM_VENUE` and `PLAY_TYPE` (int or float makes no difference).

To get an overview of the `nfl_pbp` data frame, we can call the `head()` method to print the first 10 observations. However, before doing so, we should instruct Python to display all columns by using the `pd.set_option()` method. By default, Python limits the number of columns shown in a data frame's output to avoid overwhelming the display, which can be problematic when there are many columns and we need to see all of them without truncation:

```
>>> pd.set_option('display.max_columns', None)
>>> print(nfl_pbp.head(10))
     QUARTER  DOWN  TO_GO  OFFENSIVE_TEAM_VENUE  SCORE_DIFFERENTIAL  \
16         1   4.0    2.0                   0.0                   0
85         2   4.0   10.0                   0.0                  -7
162        4   4.0    2.0                   0.0                   1
168        4   4.0   25.0                   1.0                  -1
500        4   4.0    1.0                   0.0                   0
536        1   4.0    1.0                   0.0                   0
636        3   4.0    1.0                   0.0                  -9
696        4   4.0   13.0                   0.0                 -16
709        4   4.0    3.0                   0.0                 -16
857        4   4.0    4.0                   0.0                 -13
     PLAY_TYPE  YARDS_GAINED  CONVERT
16           0           3.0        1
85           1           0.0        0
162          1           0.0        0
168          1           0.0        0
500          0           1.0        1
```

```
536          1         -11.0        0
636          0           0.0        0
696          1           0.0        0
709          1          12.0        1
857          1         -13.0        0
```

Seeing the data is one thing, but getting insights from it is another.

6.2.3 Exploring the data

In our exploration of the data, we will be laser-focused on identifying which features might be the most significant for predicting the derived variable `CONVERT`. By examining these relationships, we aim to uncover key indicators and logical splits within the variables that will inform our decision tree model. This analysis is essential for understanding the data's structure and ensuring that our model is built on a solid foundation of relevant features. Along the way, we will demonstrate how to compute basic statistics and how to plot grouped bar charts and paired histograms. We will take a variable-by-variable approach, starting with `QUARTER`.

QUARTER

We already know that `nfl_pbp` contains 883 observations, and we know there is a one-to-one relationship between observations and fourth-down attempts. But we don't yet know the observation counts between the two `CONVERT` class labels. In the snippet of code that follows, the `value_counts()` method counts the number of times each unique value appears in the `CONVERT` column from the `nfl_pbp` data frame:

```
>>> convert_counts = nfl_pbp['CONVERT'].value_counts()
>>> print(convert_counts)
CONVERT
1    462
0    421
Name: count, dtype: int64
```

So, teams were successful more times than not on fourth-down attempts during the 2023 NFL regular season and postseason—although not by much. Let's group these totals by `QUARTER` and plot the results in a Matplotlib grouped bar chart. A grouped bar chart is a type of bar chart that displays multiple bars grouped together for each category, thereby allowing a comparison of subcategories within each main category:

```
>>> import matplotlib.pyplot as plt
>>> grouped_data = (nfl_pbp.groupby(['QUARTER', 'CONVERT']) \
>>>         .size().unstack(fill_value = 0))
>>> grouped_data.plot(kind = 'bar')
```

Library must be imported before running subsequent

Creates a grouped bar chart

Groups the data by QUARTER and CONVERT and then reshapes it so each CONVERT value becomes a column; unstack() enables grouped bar plotting by aligning categories side-by-side; fill_value = 0 passed to unstack() replaces any NaNs

```
>>> for p in ax.patches:
>>>     ax.annotate(str(p.get_height()),
>>>                 (p.get_x() + p.get_width() / 2., p.get_height()),
>>>                 ha = 'center', va = 'bottom',
>>>                 fontweight = 'bold',
>>>                 color = 'black')
>>> plt.title('Observation Counts by Quarter')
>>> plt.xlabel('Quarter')
>>> plt.ylabel('Count')
>>> plt.xticks(rotation = 0)
>>> plt.legend(title = 'CONVERT',
>>>            labels = ['No', 'Yes'])
>>> plt.show()
```

Initiates a loop that iterates over each bar and annotates the height of each bar on top of it

Sets the title

Sets the x-axis label

Sets the y-axis label

Sets the rotation angle of the x-axis tick labels to 0 degrees, effectively keeping them horizontal

Adds a legend with custom labels; default placement is in the upper-right corner of the plot

Displays the plot

As we move forward to visualize the results, it's worth noting that the color scheme for a Matplotlib grouped bar chart defaults to a predefined color cycle if not explicitly set. Although we had the option to customize the colors by adding the color parameter to the `plot()` method, we chose to keep things simple with the default settings. Figure 6.1 shows the grouped bar chart we've generated.

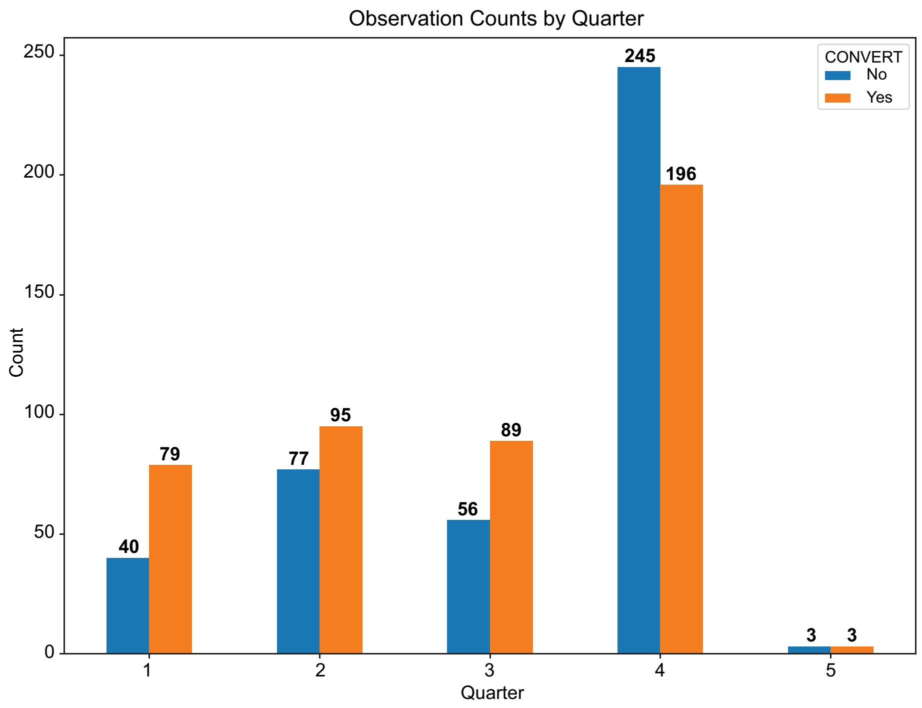

Figure 6.1 A grouped bar chart that displays the `CONVERT` class label counts by `QUARTER`. Teams were more successful than not in converting fourth-down attempts in the first three quarters, but less successful in the fourth quarter.

At least during the 2023 regular season and postseason, teams were more successful than not when attempting fourth-down conversions during the first three quarters, but less successful when doing so in the fourth quarter. This suggests that the quarter in which a play occurs might be an important feature for predicting the likelihood of a fourth-down conversion.

TO_GO

We might hypothesize that teams are more likely to convert a fourth-down attempt when only a few yards are needed and less likely to be successful when more yards are required. Let's quickly test this hypothesis by calling the `mean()` method to compute the `TO_GO` average by each `CONVERT` class label:

```
>>> mean_to_go_by_convert = nfl_pbp.groupby('CONVERT')['TO_GO'].mean()
>>> print(mean_to_go_by_convert)
CONVERT
0    5.947743
1    2.560606
Name: TO_GO, dtype: float64
```

Teams that successfully converted fourth-down attempts needed to gain, on average, 2.56 yards to keep possession and get another set of downs. When unsuccessful, teams needed almost 6 yards, on average, to convert. For those of you not terribly familiar with American football, which is sometimes described as a game of inches, that's a tremendous variance.

Because the mean can be influenced by outliers, it's important to also compute the median—that is, the middle value when the data is sorted from lowest to highest—grouped by the `CONVERT` class labels. Although the median is not completely unaffected by outliers and skewed data, it is less affected than the mean.

To do this, we simply swap out the `mean()` method from the previous snippet of code and replace it with the `median()` method:

```
>>> median_to_go_by_convert = nfl_pbp.groupby('CONVERT')['TO_GO'].median()
>>> print(median_to_go_by_convert)
CONVERT
0    4.0
1    1.0
Name: TO_GO, dtype: float64
```

The median, just like the mean, is a measure of central tendency. Teams that failed on their fourth-down conversion attempts needed a median of 4 yards, whereas teams that succeeded required a median of just 1 yard.

Let's plot the `TO_GO` distributions grouped by the `CONVERT` class labels with a Matplotlib paired histogram. A paired histogram—which we introduced in the previous chapter—is a type of visualization where two histograms are plotted on the same axes to compare the distributions of a single variable across two different groups. This allows for easy visual comparison of how the variable behaves within each group:

```
>>> convert_0_data = nfl_pbp[nfl_pbp['CONVERT'] == \
>>>                            0]['TO_GO']
```

Filters the data frame to extract the values of TO_GO where CONVERT equals 0

```
>>> convert_1_data = nfl_pbp[nfl_pbp['CONVERT'] == \
>>>                          1]['TO_GO']                  <— Filters the data frame to extract the values of TO_GO where CONVERT equals 1
>>> plt.subplot()                                          <— Initializes the plot
>>> plt.hist(convert_0_data, alpha = 0.5,
>>>          label = 'CONVERT = No')                       <— Generates the first histogram where CONVERT equals 0
>>> plt.hist(convert_1_data, alpha = 0.5,
>>>          label = 'CONVERT = Yes')                      <— Generates the second histogram; alpha makes the plots half transparent
>>> plt.title('Yards Needed for First Down by CONVERT')   <— Sets the title
>>> plt.xlabel('Yards Needed for First Down')             <— Sets the x-axis label
>>> plt.ylabel('Frequency')                                <— Sets the y-axis label
>>> plt.legend()                                           <— Adds a legend
>>> plt.show()                                             <— Displays the plot
```

The result is shown in figure 6.2. The distributions are similar in that they are both right-skewed. A right-skewed distribution is a probability distribution where the majority of the data points are concentrated on the left, with the tail extending to the right; it is also known as a positive-skewed distribution because the long tail extends into the positive, rather than the negative, direction.

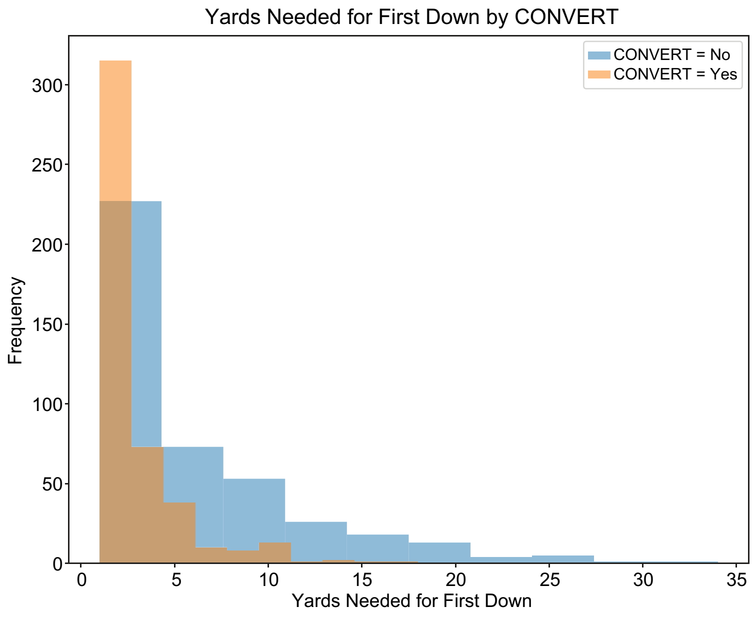

Figure 6.2 Paired histograms that display the distributions of `TO_GO` grouped by the `CONVERT` class labels. When teams succeeded on their fourth-down attempts, they usually needed fewer than 3 yards to convert, and definitely fewer than 5 yards. When teams failed to convert on fourth down, they often needed to gain more than 5 yards and sometimes up to 25 yards or more.

Yet the distributions are also different. When `CONVERT` equals `1`, there is a concentration of higher frequencies around lower values, with a relatively short tail; alternatively, when `CONVERT` equals `0`, the distribution is more spread and the tail is much longer. This distinction suggests that `TO_GO`, which represents the yards needed for a first down, plays a crucial role in determining the likelihood of a fourth-down conversion. The shorter distances correlate with higher conversion success, indicating that teams are more likely to succeed when the required yardage is minimal. In practical terms, this insight could have significant implications for coaching strategies and decision-making during games. Coaches might prioritize plays that minimize the yardage needed on fourth down or adjust their strategies based on the specific value, rather than relying solely on the game quarter or other factors. This understanding could lead to more informed and effective play-calling, potentially increasing a team's chances of success in critical fourth-down situations.

SCORE_DIFFERENTIAL

A quick reminder: we transformed `SCORE_DIFFERENTIAL` so that it consistently reflects the point margin from the standpoint of the team possessing the ball and therefore attempting the fourth-down conversion. Let’s call the `mean()` and `median()` methods in succession to compute those measures grouped by the `CONVERT` class labels:

```
>>> mean_score_differential_by_convert = \
>>>     nfl_pbp.groupby('CONVERT')['SCORE_DIFFERENTIAL'].mean()
>>> print(mean_score_differential_by_convert)
CONVERT
0   -7.125891
1   -3.688312
Name: SCORE_DIFFERENTIAL, dtype: float64

>>> median_score_differential_by_convert = \
>>>     nfl_pbp.groupby('CONVERT')['SCORE_DIFFERENTIAL'].median()
>>> print(median_score_differential_by_convert)
CONVERT
0   -7.0
1   -4.0
Name: SCORE_DIFFERENTIAL, dtype: float64
```

To begin with, it seems that teams typically opt for a fourth-down conversion when they’re trailing. Our next snippet of code returns the `nfl_pbp` counts where the variable `SCORE_DIFFERENTIAL` is `negative` (team in possession of the ball is trailing), `positive` (the same team is leading), and `zero` (the score is tied):

```
>>> score_differential_counts = (nfl_pbp['SCORE_DIFFERENTIAL'] \
>>>                    .agg({'negative': lambda x: (x < 0).sum(),
>>>                          'positive': lambda x: (x > 0).sum(),
>>>                          'zero': lambda x: (x == 0).sum()}))
>>> print(score_differential_counts)
negative    577
positive    212
zero         94
Name: SCORE_DIFFERENTIAL, dtype: int64
```

The code uses the `agg()` method along with `lambda` functions to apply specific conditions to `SCORE_DIFFERENTIAL`, thereby computing the number of instances remaining in the `nfl_pbp` data frame that are `negative`, `positive`, and `zero`. In this instance, `agg()` aggregates results from different conditions applied to `SCORE_DIFFERENTIAL` into one result set, and the `lambda` functions are used to define custom aggregation functions for specific conditions within the `agg()` method.

During the 2023 NFL regular season and postseason, approximately two-thirds (577 of 883) of fourth-down conversion attempts were made by teams trailing their opponent. These teams tended to be behind by a touchdown when failing to convert and by more than a field goal when successful, as indicated by our previous mean and median calculations.

Another plot of paired histograms should be even more insightful. This time, we're plotting the distribution of `SCORE_DIFFERENTIAL` by the two `CONVERT` class labels:

```
>>> convert_0_data = \
>>>     nfl_pbp[nfl_pbp['CONVERT'] == \
>>>             0]['SCORE_DIFFERENTIAL']
>>> convert_1_data = \
>>>     nfl_pbp[nfl_pbp['CONVERT'] == \
>>>             1]['SCORE_DIFFERENTIAL']
>>> plt.subplot()
>>> plt.hist(convert_0_data, alpha = 0.5,
>>>          label = 'CONVERT = No')
>>> plt.hist(convert_1_data, alpha = 0.5,
>>>          label = 'CONVERT = Yes')
>>> plt.title('Score Differential by CONVERT')
>>> plt.xlabel('Score Differential')
>>> plt.ylabel('Frequency')
>>> plt.legend()
>>> plt.show()
```

Extracts the values of SCORE_DIFFERENTIAL where CONVERT equals 0

Extracts the values of SCORE_DIFFERENTIAL where CONVERT equals 1

Initializes the plot

Generates the first histogram where CONVERT equals 0

Generates the second histogram; alpha makes the plots half transparent

Sets the title

Sets the x-axis label

Sets the y-axis label

Adds a legend

Displays the plot

Although the code is more or less the same—aside from swapping out variables—figure 6.3 shows that the results are not the same as before. The data is normally distributed

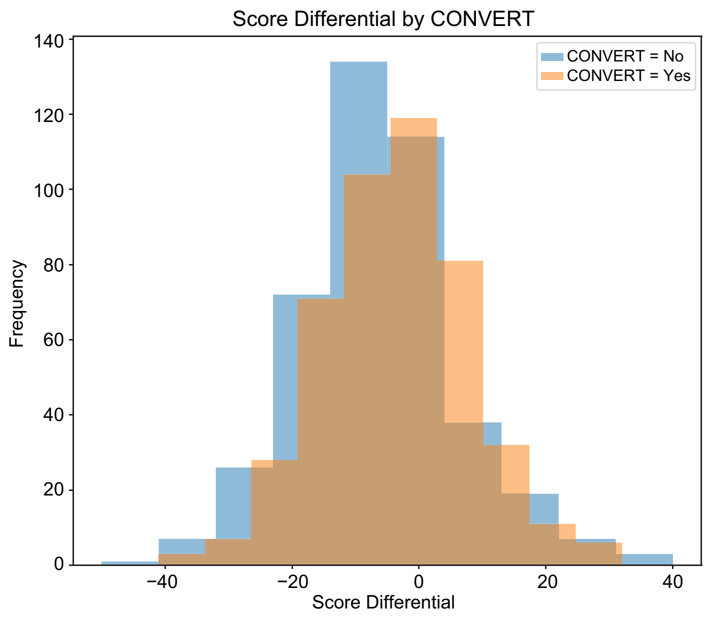

Figure 6.3 Paired histograms that display the distributions of `SCORE_DIFFERENTIAL` grouped by the `CONVERT` class labels. Regardless of the `CONVERT` class label, the distribution is normally distributed about the mean.

about the respective means—around −7 when CONVERT equals 0 and about −3 when CONVERT equals 1. However, there is notably less distinction between the distributions compared to what we observed with the TO_GO plot, which suggests that SCORE_DIFFERENTIAL is most likely less significant than TO_GO when it comes to predicting whether teams will convert on fourth down. It may or may not be more significant than QUARTER.

OFFENSIVE_TEAM_VENUE

OFFENSIVE_TEAM_VENUE indicates which team—Road or Home, mapped and converted to 0 and 1—possesses the ball. The following snippet of code creates a grouped bar chart to show the number of observations for fourth-down conversion attempts, categorized by whether the offensive team is playing at home or on the road and whether they succeeded or failed. This chart helps us compare the distribution of conversion attempts and their outcomes based on the venue:

```
>>> grouped_data = \
>>>     (nfl_pbp.groupby(['OFFENSIVE_TEAM_VENUE', 'CONVERT']) \
>>>      .size().unstack(fill_value = 0))
>>> grouped_data.plot(kind = 'bar')
>>> for p in plt.gca().patches:
>>>     plt.gca().annotate(str(p.get_height()),
>>>                        (p.get_x() + p.get_width() /
>>>                         2., p.get_height()),
>>>                        ha = 'center', va = 'bottom',
>>>                        fontweight = 'bold',
>>>                        color = 'black')
>>> plt.title('Observation Counts by \
>>>             Road vs. Home')
>>> plt.xlabel('Road or Home')
>>> plt.ylabel('Count')
>>> plt.xticks([0, 1], labels = ['Road Team', 'Home Team'],
>>>             rotation = 0)
>>> plt.xticklabels(['Road Team', 'Home Team'])
>>> plt.legend(title = 'CONVERT', labels = ['No', 'Yes'],
           loc = 'upper right')
>>> plt.show()
```

Groups the data by OFFENSIVE_TEAM_VENUE and CONVERT

Creates a grouped bar chart

Initiates a loop that iterates over each bar and annotates the height of each bar on top of it

Sets the title

Sets the x-axis label

Sets the y-axis label

Sets the position of the ticks on the x axis at positions 0 and 1 to match customization; rotates the angle of the x-axis tick labels to 0 degrees

Sets the labels for the ticks on the x axis

Adds a legend with custom labels; places the legend in the upper-right corner of the plot

Displays the plot

Our second grouped bar chart (see figure 6.4) illustrates the distribution of fourth-down conversion attempts and their outcomes, thereby providing a clear comparison between road and home teams. Teams playing at home were successful in about 55% of their fourth-down conversion attempts, with 227 out of 408 being successful. In contrast, visiting teams had a success rate of less than 50%, converting just 233 out of 473 attempts. (Note that a pair of conversion attempts on a neutral field is not shown.) However, whether this distinction is significant enough to predict the outcome of fourth-down conversion attempts remains to be seen.

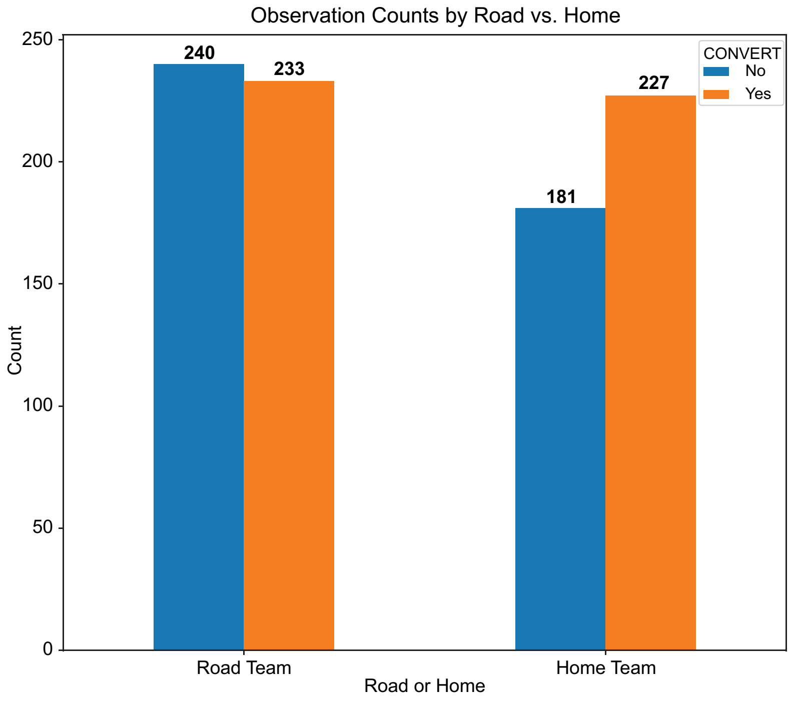

Figure 6.4 Counts of fourth-down conversion attempts categorized by the `OFFENSIVE_TEAM_VENUE` and `CONVERT` class labels. Teams playing at home were more successful than visiting teams in converting fourth-down attempts.

PLAY_TYPE

`PLAY_TYPE` is a categorical variable: a running play is represented by `0` and a passing play by `1`. Because teams often pass when many yards are needed to convert a fourth down and frequently run when fewer yards are required, `PLAY_TYPE` could serve as an effective proxy for `TO_GO`, or at least an extension of it. Our next snippet of code creates a third, and final, grouped bar chart that plots the observation counts of running and passing plays by the `CONVERT` class labels. The code should be familiar by now:

```
>>> grouped_data = \
>>>     (nfl_pbp.groupby(['PLAY_TYPE', 'CONVERT']) \
>>>      .size().unstack(fill_value = 0))
>>> ax = grouped_data.plot(kind = 'bar')
>>> for p in plt.gca().patches:
>>>     plt.gca().annotate(str(p.get_height()),
>>>                        (p.get_x() + p.get_width() /
>>>                         2., p.get_height()),
>>>                         ha = 'center', va = 'bottom',
>>>                         fontweight = 'bold',
>>>                         color = 'black')
>>> plt.title('Observation Counts by \
>>>               Run vs. Pass')
```

Groups the data by PLAY_TYPE and CONVERT

Creates a grouped bar chart

Initiates a loop that iterates over each bar and annotates the height of each bar on top of it

Sets the title

```
>>> plt.xlabel('Running or Passing Play')            <-- Sets the x-axis label
>>> plt.ylabel('Count')                              <-- Sets the y-axis label
>>> plt.xticks([0, 1], labels = ['Run', 'Pass'],
         rotation = 0)                               <-- Sets the position of the ticks on the x
                                                         axis at positions 0 and 1 to match
                                                         customization; rotates the angle of the
                                                         x-axis tick labels to 0 degrees
>>> ax.set_xticklabels(['Run', 'Pass'])              <-- Sets the labels for the ticks on the x axis
>>> plt.legend(title = 'CONVERT',
>>>            labels = ['No', 'Yes'])               <-- Adds a legend with custom labels
>>> plt.show()                                       <-- Displays the plot
```

Figure 6.5 shows the breakdown of running and passing plays from the 2023 NFL regular season and postseason categorized by the CONVERT class labels. Although teams passed the ball on fourth down much more frequently than they ran it (555 times to 328), running plays actually led to more fourth-down conversions than passing plays (233 conversions to 229). This doesn't necessarily suggest that teams should have called more running plays on fourth downs; rather, these results might reflect the number of yards needed for a conversion.

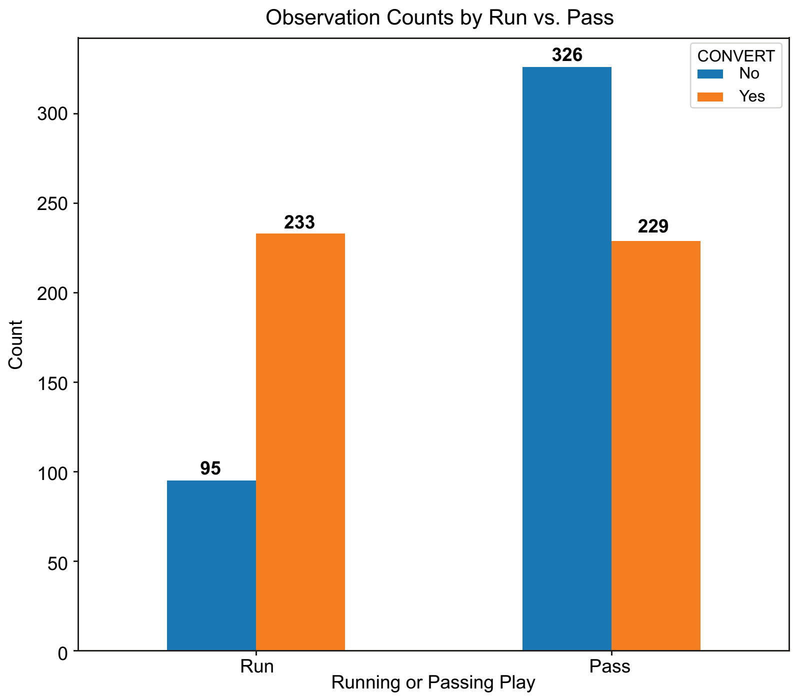

Figure 6.5 Counts of fourth-down conversion attempts categorized by the PLAY_TYPE and CONVERT class labels. Teams that ran the ball on fourth downs were much more successful in converting those attempts compared to teams that passed the ball instead. This doesn't necessarily mean that running is a better strategy than passing; rather, it might simply reflect the yards required for a conversion.

The next code snippet calculates the TO_GO mean and median by the PLAY_TYPE class labels:

```
>>> mean_to_go_by_play_type = \
>>>     nfl_pbp.groupby('PLAY_TYPE')['TO_GO'].mean()
>>> print(mean_to_go_by_play_type)
PLAY_TYPE
0    1.948171
1    5.491892
Name: TO_GO, dtype: float64

>>> median_to_go_by_play_type = \
>>>     nfl_pbp.groupby('PLAY_TYPE')['TO_GO'].median()
>>> print(median_to_go_by_play_type)
PLAY_TYPE
0    1.0
1    4.0
Name: TO_GO, dtype: float64
```

Regardless of the measure used, it's obvious that teams called a running or passing play based on the yards needed to convert a fourth down. However, it will be interesting to observe how PLAY_TYPE influences our decision tree, especially in relation to TO_GO. It's time to find out.

6.3 Fitting a decision tree

Decision trees are a popular and intuitive machine learning model used for both classification and regression tasks. They work by splitting the data into subsets based on the value of input features, creating a tree-like model of decisions. Each internal node represents a test on an attribute, each branch represents the outcome of the test, and each leaf node represents a class label or continuous value.

Decision trees are highly interpretable, as they closely mimic human decision-making processes, making it easy to understand how the model arrives at its conclusions. Additionally, they typically require minimal data preparation and can handle both numerical and categorical data equally well.

Let's look at a very simple and straightforward example of a decision tree for demonstration purposes (see figure 6.6). It illustrates the decision-making process for determining whether a home loan will be approved based on the applicant's age and income.

Starting at the root node, the decision tree first evaluates whether the applicant's age is 30 years or fewer. If the answer is yes, we move to the left side of the tree. Here, the tree examines the applicant's income at the first internal node, checking whether it is $50,000 or less. If the income is indeed $50,000 or less, the decision at the leaf node is "No," meaning the applicant is not approved for the loan. Conversely, if the applicant's income exceeds $50,000, the decision at the corresponding leaf node is "Yes," and the loan is approved.

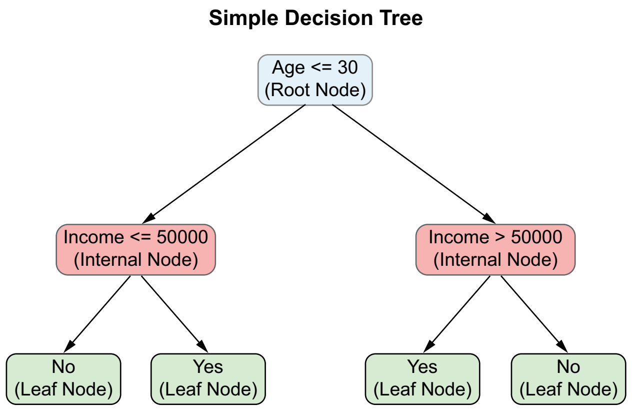

Figure 6.6 A decision tree demonstrating the home loan approval process based on the applicant's age and income. The tree is interpreted from top to bottom, starting at the root node and progressing to the leaf nodes. The left branch is followed when conditions are true, and the right branch is followed when conditions are false.

On the right side of the tree, where the applicant's age is greater than 30, the tree evaluates whether the applicant's income is more than $50,000. If the income exceeds $50,000, the decision at the left leaf node is "Yes," indicating that the loan is approved. However, if the income is $50,000 or less, the tree directs us to a "No" decision at the right leaf node, indicating that the loan will not be approved.

We will next fit a decision tree model to the `nfl_pbp` data set, aiming to predict the outcome of fourth-down conversion attempts based on various game features. Through this analysis, we seek to identify the most significant factors influencing the outcome of these attempts.

Decision trees are fit by following these steps:

1. Split the data into mutually exclusive subsets for training and testing.
2. Fit the model on the training subset.
3. Predict responses, or outcomes, on the testing subset.
4. Evaluate the model.
5. Plot the decision tree.

We will start by defining our feature (independent) variables and the target (dependent) variable, and then we will split the data into two parts. For now, we will concentrate on demonstrating how to fit a decision tree in Python; afterward, we will explain the mechanics and mathematical foundations.

6.3.1 Splitting the data

Our first snippet of code creates a new data frame called `X` by extracting the variables from `nfl_pbp` that we just analyzed. This is a common approach for defining the feature set to fit a decision tree:

```
>>> X = nfl_pbp[['QUARTER', 'TO_GO', 'OFFENSIVE_TEAM_VENUE',
>>>                  'SCORE_DIFFERENTIAL', 'PLAY_TYPE']]
```

The numeric variable `YARDS_GAINED` would have been an appropriate target variable if we were set on solving a regression task, but it was imported solely to be combined with TO_GO to derive the binary target variable CONVERT to set up our classification problem.

Next we extract CONVERT from the nfl_pbp data frame and assign it to the variable y. This, too, is a common step in preparing data for machine learning, where y typically represents the target variable or the label that the model will predict:

```
>>> y = nfl_pbp['CONVERT']
```

In a decision tree model, it is crucial to split the data into training and test sets to evaluate the model's performance effectively. The training set is used to build and fit the model, allowing it to learn the patterns and relationships within the data. The test set, which the model has not seen before, is then used to assess how well the model generalizes to new, unseen data. This split helps in detecting overfitting and ensures that the model's predictions are robust and reliable. Here, we split the features (X) and target variable (y) into training and test sets by calling the train_test_split() method from scikit-learn (sklearn), a machine learning library. The train_test_split() method requires a minimum of three parameters: the selected features, the target variable, and the proportion of the data that should be assigned to the test set. We will train our decision tree on 70% of the data and then test it on the remaining 30%. Passing random_state as a fourth parameter to train_test_split() ensures that the split is reproducible. Every time our code runs, the data will be split in the same way, which is crucial for reproducibility in experiments:

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = \
>>>     train_test_split(X, y, test_size = 0.3, random_state = 0)
```

The shape attribute returns the dimensions, or shape—that is, the number of rows and columns—of the data frame it is called on. This is a quick and easy way to confirm the 70/30 split of our data:

```
>>> print(X_train.shape)
(618, 5)
```

So, X_train contains 618 rows, which is equal to 70% of 883, and 5 columns. We should therefore expect X_test to contain 265 rows (derived by subtracting 618 from 883) and 5 columns, and we should expect y_train and y_test to contain 618 and 265 rows, respectively:

```
>>> print(X_test.shape)
(265, 5)

>>> print(y_train.shape)
(618,)

>>> print(y_test.shape)
(265,)
```

We've completed feature selection and data splitting; we're now ready to build our decision tree.

6.3.2 Fitting the model

Building a decision tree in Python is a two-step process. First, we create a decision tree classifier object using the `DecisionTreeClassifier()` method from `sklearn`:

```
>>> from sklearn.tree import DecisionTreeClassifier
>>> clf = DecisionTreeClassifier(criterion = 'gini', \
>>>                               max_depth = 3, random_state = 0)
```

The preceding code snippet naturally requires some explanation:

- `DecisionTreeClassifier()` initializes a decision tree classifier object named `clf`.
- In the context of decision trees, `criterion` refers to the function used to measure the quality of a split—`gini` is the default, but discretionary; `entropy` is also an option.
  - Gini impurity (`criterion = 'gini'`) measures how often a randomly chosen element from the set would be incorrectly labeled if it were randomly labeled according to the distribution of the labels in the set. It is by far the most common criterion method. Much more on Gini impurity later.
  - Entropy (`criterion = 'entropy'`) measures the level of impurity or disorder in a set of data. It is based on the concept of information gain and calculates the reduction in entropy after a data set is split.
  - Both methods are used to evaluate the quality of a split, but they may lead to different results in practice. Gini impurity tends to compute faster and is typically the preferred method; by contrast, entropy may be more sensitive to changes in class probabilities.
- `max_depth = 3` is a discretionary parameter that sets the maximum depth of the decision tree, which is to say we are choosing to prune the tree while building it, rather than performing these two tasks sequentially. Pruning is a technique used to prevent overfitting in decision trees by trimming parts of the tree that are not statistically significant or do not contribute significantly to its predictive accuracy. It involves removing branches of the tree that have little effect on its performance, thereby simplifying the model and improving its generalization to unseen data.
- `random_state = 0` is yet another discretionary parameter that sets the random seed to ensure reproducible results. When the same seed is used, the random splitting of data during construction remains consistent across multiple runs.

That was the first step. The second step involves training the classifier using the `fit()` method with the training data, `X_train` and `y_train`. In other words, `fit()` builds the decision tree by fitting it to the 70% training split we created earlier:

```
>>> clf = clf.fit(X_train, y_train)
```

Now that we've fit the model to the training subset, we'll next make predictions on the 30% test subset the model hasn't yet seen.

6.3.3 Predicting responses

To predict the CONVERT class labels, we utilize the trained decision tree classifier, clf, to make predictions on the test data set (X_test) and assign the predicted values to an object called y_pred:

```
>>> y_pred = clf.predict(X_test)
```

This is a NumPy array that includes 265 predictions, the size of the 30% test set, encoded to 0 and 1. These predictions can then be cross-referenced with the actual CONVERT class labels, also encoded as 0 or 1, to evaluate the predictive power of our decision tree. This just so happens to be our next task.

6.3.4 Evaluating the model

The metrics.accuracy_score() method is used to calculate the accuracy of a classification model. It requires two arguments: y_test, which contains the true labels of the test set, and y_pred, which contains the predicted labels of the test set. Our code multiplies the result by 100 to convert the accuracy to a percentage.

The print() method displays the result rounded up or down to the nearest whole number:

```
>>> from sklearn import metrics
>>> clf_accuracy = metrics.accuracy_score(y_test, y_pred) * 100
>>> print(f'Accuracy: {round(clf_accuracy, 0)}%')
Accuracy: 61.0%
```

The accuracy of our decision tree classifier on the test set is 61%. This means the model correctly predicted the outcome of 61% of the fourth-down conversion attempts in the test set. In other words, out of all the instances in the test set, the model's predictions matched the actual outcomes 61% of the time. Although this is better than random guessing, there is room for improvement. This result suggests that our model captures some patterns in the data, but it may not be fully capturing all the complexities or may need further tuning or more features to improve its predictive performance.

Confusion matrix

A confusion matrix, introduced in chapter 5, is a table used to evaluate the performance of a classification model. It provides a detailed breakdown of the model's performance compared to the actual values, allowing for a more granular analysis of the model's performance. The confusion matrix contains four key components for binary classification:

- *True positives (tp):* The number of correctly predicted positive observations (true and predicted labels equal 1)
- *True negatives (tn):* The number of correctly predicted negative observations (true and predicted labels equal 0)
- *False positives (fp):* The number of incorrectly predicted positive observations (true labels equal 0, predicted labels equal 1)

- *False negatives (fn):* The number of incorrectly predicted negative observations (true labels equal 1, predicted labels equal 0)

We have already calculated the accuracy rate of our decision tree. The confusion matrix will provide deeper insight into the model's performance by breaking down the results based on class labels. This will help us determine whether our model performs consistently across all classes or if there are disparities:

```
>>> from sklearn.metrics import confusion_matrix,
>>> conf_matrix = confusion_matrix(y_test, y_pred)
>>> print(conf_matrix)
[[71 67]
 [36 91]]
```

The final results are as follows:

- $tp = 91$
- $tn = 71$
- $fp = 67$
- $fn = 36$

Using Python as a calculator, we can apply these figures to a pair of arithmetic operations to compute the accuracy rate by class. The accuracy rate when the true label is 0 is

$$\text{True Label}_0 = \left(\frac{tn}{tn + fp}\right) 100$$

And when the true label is 1, the accuracy rate is

$$\text{True Label}_1 = \left(\frac{tp}{tp + fn}\right) 100$$

Thus,

```
>>> accuracy_rate_0 = 71 / (71 + 67) * 100
>>> print(f'Accuracy: {round(accuracy_rate_0, 0)}%')
Accuracy: 51.0%

>>> accuracy_rate_1 = 91 / (91 + 36) * 100
>>> print(f'Accuracy: {round(accuracy_rate_1, 0)}%')
Accuracy: 72.0 %
```

Our decision tree's performance was subpar when the true label was 0, with a 51% accuracy rate, which is no better than random guessing. However, it performed significantly better when the true label was 1, achieving a 72% accuracy rate. This disparity suggests that the model is better at predicting one class over the other—successful over unsuccessful fourth-down conversion attempts—possibly due to imbalanced data or differing patterns in the features for each class. An imbalanced data set occurs when one class label appears more frequently than the other, which can lead the model to favor the majority class and underperform on the minority class.

6.3.5 Plotting the decision tree

The following snippet of code generates a visual representation of the decision tree classifier, `clf`, using the `plot_tree()` method from the `sklearn.tree` module (see figure 6.7). It specifies the feature names and class names to be displayed in the tree plot. The `filled = True` parameter instructs Python to fill the decision tree nodes with colors to represent the majority class in each node; `rounded = True` rounds the edges of the nodes for aesthetic value; and `fontsize = 12` increases the font from the default setting to make the tree and its attributes more readable:

```
>>> from sklearn import tree
>>> plt.figure()
>>> tree.plot_tree(clf, feature_names = ['QUARTER',
>>>                                      'TO_GO',
>>>                                      'OFFENSIVE_TEAM_VENUE',
>>>                                      'SCORE_DIFFERENTIAL',
>>>                                      'PLAY_TYPE'],
>>>                class_names = ['0', '1'],
>>>                filled = True,
>>>                rounded = True
>>>                fontsize = 12)
>>> plt.tight_layout()
>>> plt.show()
```

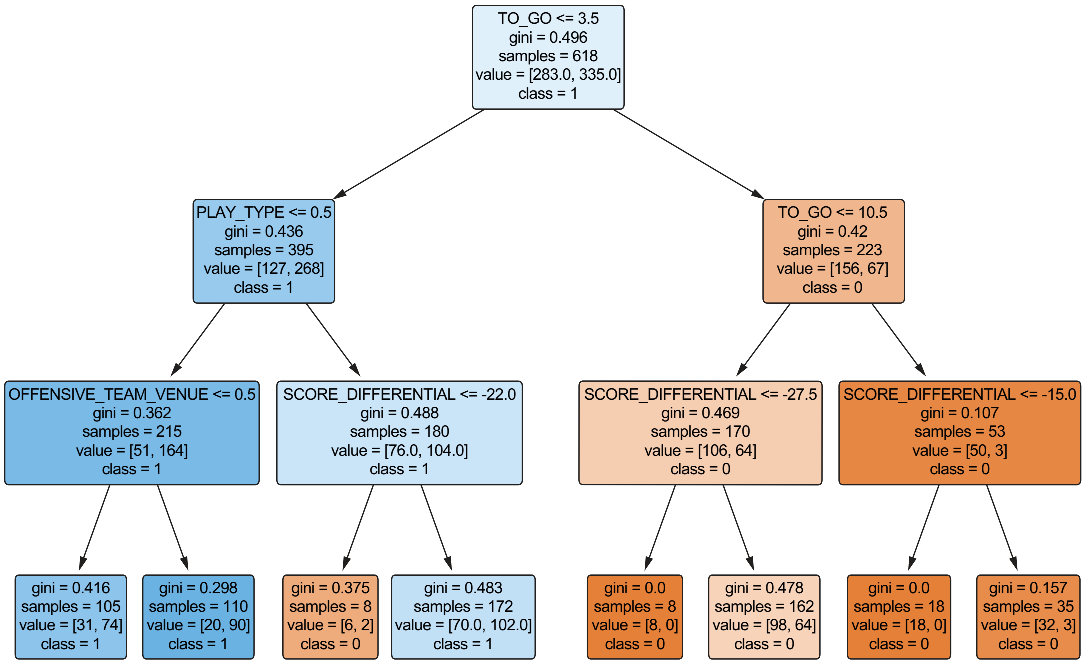

Figure 6.7 A plotted decision tree that represents the model's decision-making process, showing how features are used to split the data and make predictions. This tree was pruned during construction, so it contains fewer splits and even fewer (less significant) features than otherwise. Python automatically adds color shading to each node based on the predicted class label and purity of the split, thereby making the tree easier to interpret visually.

The tree has already been pruned, meaning it has fewer splits and utilizes fewer features than it otherwise would. You might notice that some splits, such as those on `TO_GO`, are made on non-integer values (e.g., `TO_GO <= 3.5`), even though `TO_GO` is an integer variable. This is a standard outcome when using decision trees in Python, as the algorithm doesn't automatically restrict splits to integer thresholds. Although it might look unusual, this approach does not affect the model's predictive accuracy and is commonly accepted in practice. For now, take a moment to observe the structure, but don't worry about interpreting the details just yet—this will be explained in great detail momentarily.

Decision trees, especially after pruning, are relatively easy to interpret because they visually represent the decision-making process through a series of straightforward splits based on feature values. However, despite their intuitive nature and resemblance to human decision-making, they require a thorough walkthrough to fully grasp their structure. The underlying mechanics and mathematical foundations of decision trees, including concepts like impurity and pruning criteria, are not straightforward and can be complex to understand without a solid background in machine learning.

6.3.6 Interpreting and understanding decision trees

The structure of a decision tree consists of several key components: the root node, internal nodes, and leaf nodes. The *root node* is the topmost node of the tree and represents the initial decision point from which all subsequent branches originate. *Internal nodes*, located between the root and the leaves, represent intermediate decision points that split based on specific feature values, leading to further branches. *Leaf nodes*, also known as *terminal nodes*, are the endpoints of the tree and represent the final class labels or predictions. In a plotted decision tree, the root node is easily identified as it is positioned at the top and has arrows pointing to the first set of internal nodes. Internal nodes have both incoming and outgoing arrows, whereas leaf nodes only have incoming arrows, indicating they do not split further.

To interpret a decision tree, start at the root node and follow the branches based on the conditions provided at each node. These conditions are typically in the form of "if-else" statements that test specific feature values. For example, at the root node, we see the condition `TO_GO` <= 3.5, which dictates the path to follow depending on whether the condition is true or false. Continue down the tree, moving from node to node, following the branches that match the conditions. Each internal node will guide you through further splits until you reach a leaf node, which provides the final prediction. The path taken from the root to a leaf represents a series of decisions based on the feature values, leading to a classification outcome (or a regression value if we were instead solving a regression problem). This step-by-step approach ensures a clear understanding of how the model makes its predictions, reflecting the logical flow of decision-making within the tree. In fact, decision trees are constructed in the same way they should be interpreted—beginning at the top with the root node.

Root node

The first step in constructing a decision tree, whether in Python, another software application, or manually, is to determine which feature to assign to the root node. This is achieved by identifying the decision or split that best separates the class labels, which is done by calculating the weighted average Gini impurity for each feature. Getting to this measure is relatively straightforward when working with categorical variables such as `PLAY_TYPE` but not so much when working with numeric variables like `TO_GO`. Understanding how to compute Gini impurities and the weighted average Gini impurity for both categorical and numeric variables is essential for determining which feature and which split should be assigned to the root node.

Gini impurity is a measure of the impurity or diversity of a single feature within a data set. Specifically, it evaluates how well, or not so well, a feature can split the data into different classes, aiming to minimize the impurity at each node in the decision tree. Gini impurity ranges from 0 (perfectly pure, where all elements are of a single class) to 0.5 (maximum impurity, where elements are equally distributed across classes):

It is calculated using the following equation:

$$\text{Gini impurity} = 1 - (\text{Probability of “Yes”})^2 - (\text{Probability of “No”})^2$$

At the decision nodes—that is, the root node and the internal nodes—Gini impurity should be calculated separately for each side of a condition, assuming an imbalance in row counts; these values are then combined to obtain the weighted average Gini impurity, which represents the total impurity for a single feature.

Let’s use the categorical variable `PLAY_TYPE` as an example for demonstration purposes. We previously computed the counts of fourth-down conversion attempts categorized by the `PLAY_TYPE` and `CONVERT` class labels—but we ran that query against the entire `nfl_pbp` data set. Because our decision tree was fit on the `X_train` and `y_train` subsets from `nfl_pbp`, we need to join `X_train` and `y_train` into a single object by calling the `pd.concat()` method and then compute the counts for each `PLAY_TYPE` and `CONVERT` combination using `groupby()` and `size()`:

```
>>> train_data = pd.concat([X_train, y_train], axis = 1)
>>> counts = train_data.groupby(['PLAY_TYPE', 'CONVERT']).size()
>>> print(counts)
PLAY_TYPE  CONVERT
0          0           64
           1          175
1          0          219
           1          160
dtype: int64
```

Before we plug these figures into the Gini impurity equation, it’s useful to refresh our understanding from chapter 2 of how to calculate empirical probabilities, or probabilities derived from trials or real-world observations:

$$\text{Probability(Event)} = \frac{\text{number of successes observed}}{\text{number of observations made}}$$

It's also important to remember conditional probabilities from chapter 3, because awareness of specific circumstances can narrow the sample space, leading to more precise and relevant probabilities.

When the condition `PLAY_TYPE` <= 0.5 from our decision tree is true, it indicates that a running play (encoded as `0`) was called on fourth down. This resulted in 175 successful conversions and 64 failed attempts, leading to a loss of possession. Thus, our denominator equals the sum of 175 and 64, or 239. The numerator is therefore 175 for the probability of a successful conversion and 64 for the probability of a failed attempt. So, when `PLAY_TYPE` <= 0.5, the Gini impurity is

$$\text{Gini impurity(True)} = 1 - \left(\frac{175}{175+64}\right)^2 - \left(\frac{64}{175+64}\right)^2 = 0.3922$$

When the same condition is false, it means a passing play (encoded as `1`) was called on fourth down, which resulted in 160 successful and 219 unsuccessful conversion attempts. Our denominator is therefore the sum of 160 and 219, or 379. The numerator is 160 for the probability of a successful conversion and 219 for the probability of a failed attempt. Thus, when `PLAY_TYPE` is > 0.5, the Gini impurity is

$$\text{Gini impurity(False)} = 1 - \left(\frac{160}{160+219}\right)^2 - \left(\frac{219}{160+219}\right)^2 = 0.4879$$

The total Gini impurity, or the weighted average Gini impurity, is derived by multiplying the Gini impurities by the observation percentages—38.67% of the `train_data` observations are running plays and 61.33% are passing plays—and then summing these products, like so:

$$\text{Weighted Average Gini impurity} = 0.3922\,(0.3867) + 0.4879\,(0.6133) = 0.4509$$

Thus, the total Gini impurity for `PLAY_TYPE` equals 0.4509.

This must be greater than the Gini impurity for `TO_GO` because `TO_GO`, not `PLAY_TYPE`, is the feature assigned to the root node. We'll verify this assumption by explaining and demonstrating how the Gini impurity is derived for a numerical variable. It's a three-step process:

1. Sort the data in ascending order.
2. Calculate the averages between adjacent values.
3. Calculate the impurity values for each average. The split should occur where the Gini impurity is the lowest. For `TO_GO`, this point is at the average of 3 and 4.

Our next chunk of code splits `train_data` into two parts based on whether `TO_GO` is less than or equal to 3.5 or greater than 3.5. It then groups each subset by the `CONVERT` class labels and returns the row counts for each:

```
>>> to_go_less_than_3_5 = \
>>>     train_data[train_data['TO_GO'] <= 3.5]
>>> to_go_greater_than_3_5 = \
>>>     train_data[train_data['TO_GO'] > 3.5]
>>> counts_less_than_3_5 = (to_go_less_than_3_5 \
>>>             .groupby('CONVERT').size())
>>> print('Counts for TO_GO <= 3.5:')
>>> print(counts_less_than_3_5)
>>> counts_greater_than_3_5 = (to_go_greater_than_3_5 \
>>>             .groupby('CONVERT').size())
>>> print('Counts for TO_GO > 3.5:')
>>> print(counts_greater_than_3_5)
Counts for TO_GO <= 3.5:
CONVERT
0    127
1    268
dtype: int64
Counts for TO_GO > 3.5:
CONVERT
0    156
1     67
dtype: int64
```

Creates one data frame where TO_GO is less than or equal to 3.5

Creates another data frame where TO_GO is greater than 3.5

Computes the counts from the first data frame by the CONVERT class labels

Prints a header for the results

Prints the results

Computes the counts from the second data frame by the CONVERT class labels

Prints a header for the results

Prints the results

From these figures, we can compute the impurity values for both sides of the TO_GO <= 3.5 split:

$$\text{Gini impurity(True)} = 1 - \left(\frac{268}{268+127}\right)^2 - \left(\frac{127}{268+127}\right)^2 = 0.4363$$

$$\text{Gini impurity(False)} = 1 - \left(\frac{67}{67+156}\right)^2 - \left(\frac{156}{67+156}\right)^2 = 0.4204$$

If the data were equally divided across both sides of the TO_GO split, we could obtain the weighted average Gini impurity by simply adding 0.4603 and 0.3004 together. However, 63.92% of the data is on the left (true) side of the TO_GO <= 3.5 split, and the remaining 36.08% is on the right (false) side. Thus, we get the weighted impurity this way:

$$\text{Weighted Average Gini impurity} = 0.4363\,(0.6392) + 0.4204\,(0.3608) = 0.4305$$

Not only is this the lowest Gini impurity within TO_GO, but it is also the lowest impurity value across the feature set; as a result, TO_GO is assigned to the root node. Therefore, *it is the most significant variable for predicting the final class labels.* Now that we've explained how a feature is assigned to the root node, we will deconstruct the rest of the tree.

Node attributes

One advantage of decision trees—we'll review the pros and cons of decision trees before shifting our focus to random forests—is their alignment between construction and interpretation methodologies. Decision trees are constructed from the top down, beginning with the root node and then branching out into internal (or decision) nodes and leaf nodes. This structure mirrors the process of decision-making, where we start with overarching considerations (the root node) and gradually refine our choices based on specific criteria (internal nodes) until we arrive at a final decision (leaf nodes). Interpreting a decision tree involves understanding this hierarchical structure and the criteria used at each node to guide the decision-making process.

The information contained within every node of a decision tree—be it root, internal, or leaf—is typically referred to as the *node attributes* or occasionally as the *node properties* (see figure 6.8). Prioritizing the explanation of these attributes before adopting a broader perspective of our decision tree is crucial, as understanding the former undoubtedly enhances our comprehension of the latter.

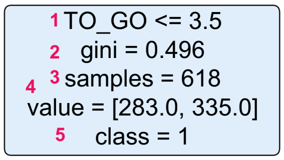

Figure 6.8 A close look at the root node attributes from the plotted decision tree. The root node and the internal nodes all contain these same attributes; leaf nodes have these attributes, too, minus the condition statement at the top.

Let's examine the note attributes in figure 6.8 line by line:

- *Line 1* shows the *condition statement*, often referred to as the *decision* or *split*. If the condition evaluates to true—for instance, if `TO_GO` is <= 3.5 or if `PLAY_TYPE` is <= 0.5—proceed to the left node; otherwise, proceed to the right node. Condition statements apply to the root node and internal nodes only.
- *Line 2* displays the *unweighted* Gini impurity. Another, potentially more practical approach to conceptualizing this measure is to consider it as representing the probability of incorrect classification if chosen randomly. Therefore, the closer the Gini impurity is to zero, the better the node is at classifying the data.
- *Line 3* shows the number of records from the training set that satisfy the condition of the node. The root node contains 618 observations, or samples—395 sent to the left of the tree and the other 223 sent to the right. The highest level of internal nodes—the nodes directly beneath the root node—also contains 618 samples when their individual samples are added together, and so forth. Decision trees don't drop any records. Pruning removes features and splits that don't provide significant differentiation, thereby simplifying the tree and making it easier to interpret and use; however, the records remain and, if necessary, are reallocated to other internal nodes.
- *Line 4* is an array that represents the distribution of samples among the target feature classes *at that node*. For example, at the root node, our training data includes 283 instances where `CONVERT` equals `0` and 335 instances where `CONVERT` equals `1`. Similarly, when the data is further subset to 395 records based

on PLAY_TYPE, there are 127 instances where CONVERT equals 0 and 268 additional instances where CONVERT equals 1.

- *Line 5* shows the classification at that stage of the tree. For instance, if we were to stop the tree at the root node, we should classify CONVERT as a successful fourth-down conversion (encoded as 1), because at this level, there is a higher probability of converting on fourth down than not. Of course, the classification becomes more accurate as we move down the tree.

As we move forward to examine the remainder of our decision tree and demonstrate how to interpret it, it's important to emphasize one key point: pruning the tree has made this analysis far more manageable. Without pruning, the tree could have had three to four times as many nodes, making it much more complex; instead, it contains just two levels of internal nodes. Although this wasn't our primary objective, we prioritized simplicity and usability to avoid the risk of overfitting.

Internal and leaf nodes

A decision tree is evaluated from the root node down to the leaf nodes, similar to how it is constructed. There are numerous permutations of "if-else" conditions along the way, thereby creating many unique paths from start to finish. Let's start by examining the two internal nodes directly beneath the root node (see figure 6.9).

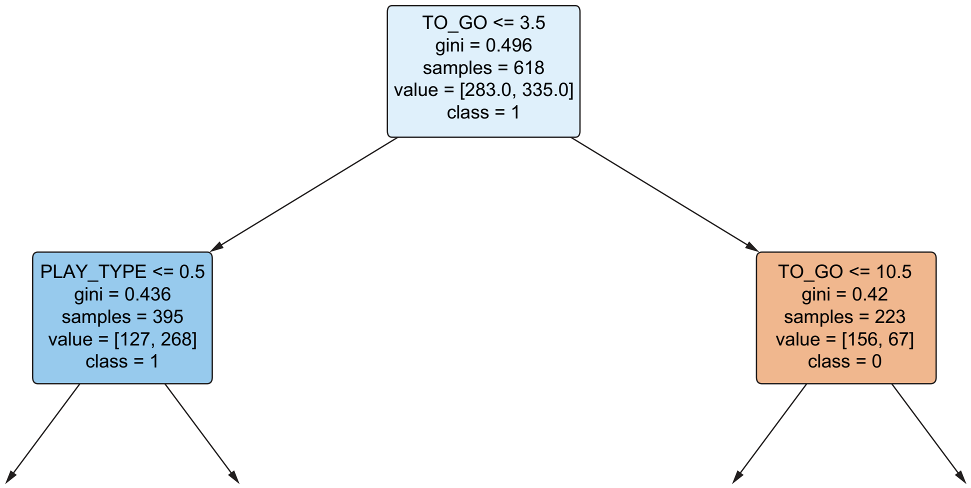

Figure 6.9 A close look at the very top of our decision tree: the root node and the first level of internal nodes. The root node has arrows pointing away from it, whereas internal nodes have arrows pointing toward and away from them.

When a condition is true, we always move to the left side of the tree; otherwise, we move to the right. Thus, when TO_GO is less than or equal to 3.5, we proceed to PLAY_TYPE; if TO_GO is greater than 3.5, we proceed instead to another TO_GO condition.

This particular split is especially meaningful because if the tree stopped at the first level of internal nodes, we would have opposite classifications for our target variable (and these internal nodes would instead be leaf nodes).

Before we proceed further down the tree, several points need to be addressed:

- The decision tree algorithm recursively splits the data at each node. After determining the root node, the data is then divided into subsets based on the best feature and threshold for that node.
- For each subset created by the root node split, the algorithm evaluates all possible features and thresholds to determine the best split. The split is the one that minimizes impurity (or, alternatively, information gain if we had selected entropy as our criterion instead of Gini) for that particular subset.
- Each split is optimized locally within the subset of data, which means different features or different thresholds of the same feature may be chosen for different branches.
- Numeric variables often have complex relationships with the target variable that can't be fully captured by a single split. Additional splits on the same numeric variable allow the tree to model these relationships more accurately.
- After the initial split at the root node, the resulting subsets might have different distributions and characteristics. A numeric variable that provided a useful split at the root might still be useful within these subsets, albeit with different thresholds.
- The hierarchical structure of decision trees means each node independently selects the best feature and threshold from the data available at that node. This allows the same numeric variable to be used multiple times with different conditions.
- During the construction of the tree, each node is evaluated independently. If a numeric variable has lower impurity at an unused threshold compared to other features and conditions, it will be selected again for further splits.

In short, the decision tree algorithm recursively splits data at each node by selecting the best remaining feature and threshold combination that minimizes impurity, optimizing each split locally within subsets. Numeric variables—which are split various ways depending on the number of unique values in the set—may be used multiple times with different thresholds to capture complex relationships with the target variable, as each node independently evaluates and selects the optimal feature and condition from top to bottom.

The left subtree is first divided by the categorical feature `PLAY_TYPE`, followed by subsequent splits at `OFFENSIVE_TEAM_VENUE` (categorical) and `SCORE_DIFFERENTIAL` (numeric). Because `PLAY_TYPE` is the topmost internal node, it is the most significant feature on this side of the tree. The left branch of the tree shows a strong propensity toward predicting a class label of 1, particularly when `PLAY_TYPE` assumes a value of 0. In instances where `PLAY_TYPE` equals 1, the predictive outcome is then determined by the split at SCORE_DIFFERENTIAL, which implies that OFFENSIVE_TEAM_VENUE holds relatively little significance: despite its role in reducing Gini impurity, it consistently predicts the majority class regardless of the condition (see figure 6.10).

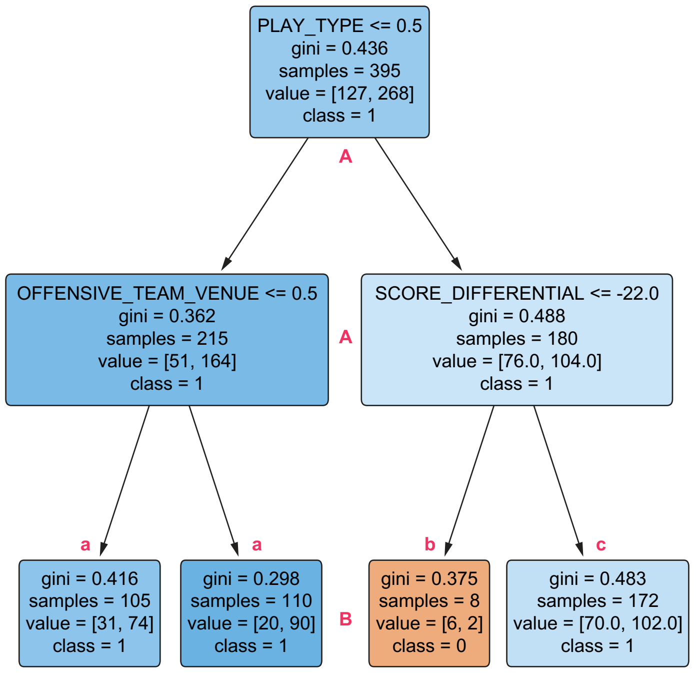

Figure 6.10 A closer look at the left subtree: two levels of internal nodes (A) and, at the bottom, four leaf nodes (B). When PLAY_TYPE is 0, thereby indicating a running play, the decision tree predicts a successful fourth-down conversion attempt (a); it doesn't matter if the team on offense is the road or home team. Alternatively, when PLAY_TYPE is 1, indicating a passing play, the decision tree evaluates SCORE_DIFFERENTIAL before making a prediction. If the team on offense is trailing by 22 points or more, the tree predicts a failed fourth-down conversion attempt (b); but if the team on offense is trailing by fewer than 22 points, the decision tree predicts a successful conversion (c).

The right subtree of the decision tree begins with the split on TO_GO <= 3.5 at the root. This split effectively divides the data, leading to a significant reduction in impurity. The subsequent split on TO_GO <= 10.5 at the top of the subtree further refines this separation, establishing a clear pathway for the classification process. SCORE_DIFFERENTIAL then plays a pivotal role, introducing additional conditions that further dissect the data into smaller, more homogenous subsets. Despite these multiple splits, the consistency in class predictions (class 0) throughout the leaf nodes indicates that the initial TO_GO split is highly effective in distinguishing the target class. The SCORE_DIFFERENTIAL conditions primarily serve to reduce the Gini impurity, enhancing the purity of the nodes without altering the overall class prediction. This consistent prediction pattern underscores the strong influence of the initial TO_GO split and highlights how subsequent conditions refine the data to confirm the class rather than change it (see figure 6.11). The right subtree therefore effectively uses both TO_GO and SCORE_DIFFERENTIAL to create a robust and consistent classification pathway down to the leaf nodes.

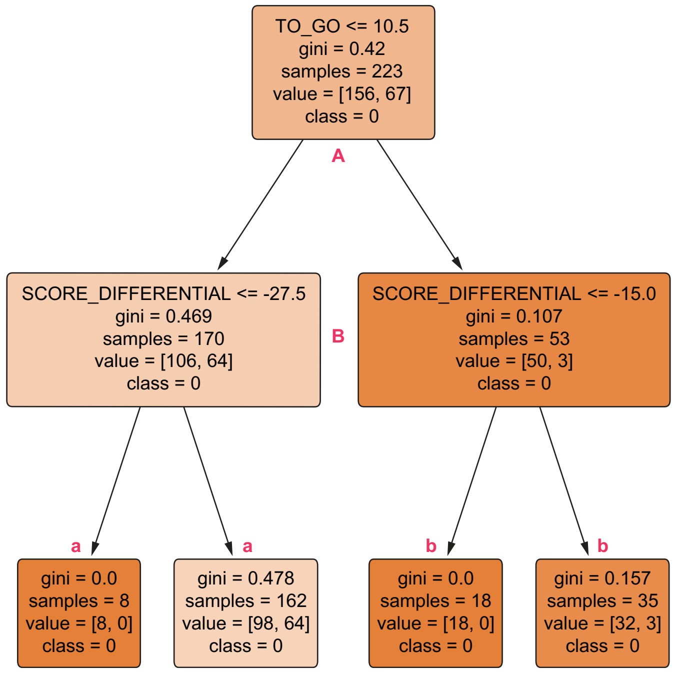

Figure 6.11 The right branch of the decision tree uses internal nodes TO_GO (A) and SCORE_DIFFERENTIAL (B) to establish a classification pathway, with the initial TO_GO split playing a crucial role in distinguishing the target class, followed by subsequent refinements that confirm the predicted class labels (a and b).

Although it was pruned, the decision tree algorithm strategically removed the feature QUARTER because it complicated the model more than it improved the model's predictive power. Incorporating additional features often boosts predictive accuracy, but sometimes the added complexity may not yield proportional benefits, particularly in decision trees where more features can lead to a proliferation of nodes and branches, potentially diminishing the clarity and usability of the outcomes.

On taking a broader perspective and considering the sequence of introduced features in the tree, with the understanding that feature significance diminishes from top to bottom, our decision tree strongly correlates with the preliminary analysis conducted prior to its construction. It was anticipated that TO_GO and PLAY_TYPE would hold the greatest predictive weight in determining the CONVERT class labels, a notion then validated by the structure of the tree. Before we transition from discussing a single decision tree to exploring a forest of trees, let's outline the advantages and disadvantages of decision trees as a machine learning approach compared to other models.

6.3.7 Advantages and disadvantages of decision trees

We have periodically referenced the pros and cons of decision trees, but let's take this opportunity to succinctly summarize their advantages and disadvantages. All model types come with inherent strengths and weaknesses, and decision trees are no exception. Understanding these points will help you determine whether decision trees are the best fit for your specific problem-solving needs.

Advantages:

- Decision trees are easy to understand and interpret. Their visual representation makes them accessible to non-experts, and they clearly show the decision-making process.
- Decision trees do not require normalization or scaling of data, making preprocessing simpler.
- They capture nonlinear relationships between features and the target variable without the need for complex transformations.
- Decision trees handle numeric and categorical data equally well.
- They provide insights into feature importance, indicating which features are most influential in predicting the target variable.
- They are relatively robust to outliers, as splits are based on the majority of the data.
- Decision trees can be used to solve both classification and regression problems.

Disadvantages:

- Decision trees are prone to overfitting, especially if left unpruned.
- Small changes in the data can result in a completely different tree structure, thereby making them unstable.
- They can be biased if one class is dominant. They may also perform poorly on imbalanced data sets without proper tuning.
- Large trees can become complex and therefore difficult to interpret, negating one of their advantages.
- Most decision trees create splits that are perpendicular to feature axes, which may not capture more complex relationships effectively.
- A single decision tree might lack the predictive power achievable through other methods.
- Decision trees use a greedy algorithm for splitting, meaning they make the best immediate decision at each step (splitting the data based on the most significant feature at that moment) without considering the overall effect on the entire tree.

Many of these disadvantages can be mitigated by fitting a random forest instead of a single decision tree.

6.4 Fitting a random forest

Random forest models are an ensemble learning technique that builds on the principles of decision trees to enhance predictive performance and robustness. Unlike a single decision tree, which can be prone to overfitting and instability, a random forest constructs multiple decision trees during training and merges their outputs to produce more accurate and stable predictions.

In a random forest, multiple decision trees are used to make a final prediction, each contributing to the overall decision. For example, consider two simple trees in the forest:

- Tree 1:
  - Root node: Age ≤ 30
  - Internal node (left): Income ≤ 50,000
  - Leaf node (left): No (loan not approved)
  - Leaf node (right): Yes (loan approved)
- Tree 2:
  - Root node: Income > 40,000
  - Internal node (right): Age > 30
  - Leaf node (left): Yes (loan approved)
  - Leaf node (right): No (loan not approved)

Each tree independently classifies the data, and the final prediction is determined by the majority vote of all the trees in the random forest.

Each tree in the forest is trained on a different subset of the data, and features are randomly selected at each split, which helps to reduce variance and prevent overfitting. Although decision trees are easy to interpret and visualize, random forests, being aggregations of many trees, offer greater predictive power and resilience to noisy data at the expense of interpretability. This makes random forests a powerful tool for classification and regression tasks where accuracy and generalization are prioritized.

Random forests and decision trees are both fundamental tools in machine learning, but they differ significantly in methodology and performance. A decision tree operates by recursively partitioning the data based on the most significant features at each node, ultimately forming a tree-like structure of decisions. Although decision trees are easy to interpret and visualize, they are susceptible to overfitting, especially when they grow too deep. Overfitting occurs because the model captures noise and minor fluctuations in the training data, resulting in poor generalization to new, unseen data. Additionally, decision trees can be unstable, so small changes in the data can lead to a completely different tree structure.

In contrast, random forests mitigate these issues by employing an ensemble approach, combining the predictions of multiple decision trees to produce a more robust and accurate model. Each tree in a random forest is trained on a random subset of the data (with replacement, known as *bootstrap sampling*) and a random subset of features at each split. This randomness introduces diversity among the trees, thereby reducing the likelihood of overfitting. The final prediction is typically made by averaging the predictions (for regression) or taking a majority vote (for classification) from all the trees in the forest. Although this ensemble method enhances predictive performance and stability, it does so at the cost of interpretability. Unlike a single decision tree, which provides a clear decision path, the aggregated nature of random forests makes it more challenging to understand the individual decision-making process. However, the trade-off is often worthwhile in practice, as random forests generally offer superior accuracy and resilience to noisy data compared to individual decision trees.

This specific sampling technique—random subsets of data and features—is used to reduce correlation between individual trees. If every tree saw the same data and features, they would likely make similar mistakes. By introducing randomness, each tree learns different patterns, which increases the overall diversity of the forest and leads to better generalization when making predictions on unseen data.

Fitting a random forest in Python is very similar to fitting a decision tree, as both use straightforward commands from the scikit-learn library. However, there is a lot more happening behind the scenes with random forests, including the training of multiple trees on different data subsets and feature selections, to create a robust ensemble model.

6.4.1 Fitting the model

Typically, we start by importing a data set, identifying meaningful relationships between features and the target variable, and splitting our data frame into training and testing subsets. However, those steps are firmly in our rearview mirror. Because we are fitting a random forest to the data we've been working with all along, we can skip to model fitting, using the training subset previously derived from the `nfl_pbp` data frame.

Whereas the `DecisionTreeClassifier()` method initializes a decision tree classifier object, the `RandomForestClassifier()` method, also from Python's scikit-learn library, initializes a random forest classifier object named `rf`. The first of four parameters passed to `RandomForestClassifier()` is `n_estimators = 50`, which sets the number of trees in the random forest to `50`.

Although there are no strict rules dictating the number of trees to fit in a random forest—it all depends on factors such as the dimensionality of the data, computational resources available, and desired model performance—there are some general guidelines:

- Increasing the number of trees in the forest can improve the model's performance, up to a certain point. Adding more trees can help reduce overfitting and improve the generalization ability of the model.
- However, there are diminishing returns associated with adding more trees. After a certain point, the improvement in model performance achieved by adding additional trees becomes marginal, and the computational cost increases.
- The computational resources required to fit a large number of trees are not incidental; it is therefore essential to consider factors like processing power and memory when deciding on the number of trees to fit.

- Although there is no one-size-fits-all rule, a common heuristic is to start with a moderate number of trees (e.g., 50) and adjust based on performance results. In practice, the optimal number of trees may vary depending on the task and the data.

The remaining three parameters are identical to those we previously passed to the `DecisionTreeClassifier()` method:

- `criterion = 'gini'` specifies the criterion used to measure the quality of a split at each node. The Gini impurity criterion measures the probability of incorrectly classifying a randomly selected element if it were randomly labeled according to the distribution of samples in the node.
- `max_depth = 3` restricts the maximum depth of each decision tree in the forest to the root node, two levels of internal nodes, and the leaf nodes. Although pruning prevents overfitting and improves interpretability, it may also hinder the model's ability to capture complex patterns in the data, thereby sacrificing some of its complexity and expressiveness.
- `random_state = 0` sets the random seed to ensure reproducible results if and when the same seed is used in the future.

The `RandomForestClassifier` class from the `ensemble` module of the scikit-learn library must be imported first. Although the code structure resembles that of fitting a decision tree classifier, it originates from a separate module:

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> rf = RandomForestClassifier(n_estimators = 50,
>>>                             criterion = 'gini',
>>>                             max_depth = 3,
>>>                             random_state = 0)
```

The `RandomForestClassifier()` method does the following:

- Initializes a random forest classifier object with 50 trees
- Limits the maximum depth of each tree to three levels
- Uses the Gini impurity criterion for splitting nodes
- Guarantees reproducibility, as long as the same seed is used in subsequent runs

Next a call is made to the `fit()` method, which trains the random forest classifier, `rf`, using the training data (`X_train`) and corresponding labels (`y_train`) previously cut from the `nfl_pbp` data frame. This step enables the model to learn patterns in the data, preparing it for making predictions on new data:

```
>>> rf.fit(X_train, y_train)
```

This is *the* critical step in the process of training a random forest classifier using the scikit-learn library. Here, `rf` represents an instance of the `RandomForestClassifier` class, which was just configured, of course, with parameters such as the number of trees, maximum depth, and impurity criterion. The `fit()` method is called on this instance, taking `X_train` and `y_train` as required arguments. `X_train` is a 2D array or data frame containing the training data's feature variables, and `y_train` is a 1D array or series containing the corresponding target labels. By invoking the `fit()` method, the classifier learns from the provided data, building multiple decision trees on different subsets of the data and features, and combining their predictions to form an ensemble model.

During the training process, each decision tree in the random forest is trained on a bootstrap sample (a random sample with replacement) of the training data. At each split in the tree, a random subset of features is considered to determine the best split, introducing randomness that helps reduce overfitting and improve generalization. This method ensures that the model captures diverse patterns and relationships within the data. Once the training is complete, the random forest model is equipped with the decision rules derived from the training data, ready to make predictions on new, unseen data.

6.4.2 Predicting responses

Predictions are then made on the test data (`X_test`) using the random forest classifier trained on `X_train` and `y_train`. The predicted class labels are stored in an object called `y_pred`:

```
<<< y_pred = rf.predict(X_test)
```

Each tree in the forest makes a series of predictions; the final predictions in a classification tree are determined by majority voting among the trees.

6.4.3 Evaluating the model

Because random forests consist of an ensemble of decision trees, they are evaluated using similar methods. The `metrics_accuracy_score()` method was previously called to calculate the accuracy rate of our decision tree; we will call it again to compute this for our random forest. Using similar methods to calculate standard metrics across various models, such as decision trees versus random forests or different random forest variants, simplifies the model selection process.

The `metrics_accuracy_score()` method compares the true labels from the test set to the predicted labels and computes an accuracy rate; that figure is then multiplied by `100` to convert that measure to a percentage. The `print()` method displays the result, which has been rounded to the nearest whole number:

```
>>> rf_accuracy = metrics.accuracy_score(y_test, y_pred) * 100
>>> print(f'Accuracy: {round(rf_accuracy, 0)}%')
Accuracy: 62.0%
```

The marginal increase in accuracy from 61% with a decision tree to 62% with a random forest on the same data set may seem underwhelming at first glance. However, this difference in performance reflects deeper insights into the nature of the data and the models themselves. Although the random forest's improvement is modest, it's crucial to recognize that its architecture inherently offers advantages over a single decision tree. Random forests operate by aggregating predictions from multiple decision trees, each trained on different subsets of the data with random feature selection. This ensemble approach typically results in a more robust and generalized model, capable of capturing complex patterns and reducing the risk of overfitting compared to a standalone decision tree.

Several factors could be at play in holding back the random forest's performance. First, the data may lack distinguishing features or exhibit high levels of noise, limiting the model's ability to discern meaningful patterns. Additionally, the hyperparameters of the random forest, such as the number of trees and maximum depth, may not have been optimized for this specific data set, potentially constraining its predictive power. Furthermore, the data set's characteristics, such as class imbalance or the presence of outliers, could pose challenges that affect model performance. However, even with these limitations, the random forest is likely to be more robust than a single decision tree, as it uses the collective wisdom of multiple trees, making it better equipped to handle variations in the data and more likely to generalize well to new, unseen instances.

CONFUSION MATRIX

The 62% accuracy rate of our model is an overall measure across both classes. By generating a confusion matrix, we can obtain more detailed insights—specifically, we can evaluate the model's performance for each `CONVERT` class label:

```
>>> conf_matrix = confusion_matrix(y_test, y_pred)
>>> print(conf_matrix)
[[68 70]
 [32 95]]
```

The confusion matrix displays the model's performance metrics. In the lower-right quadrant, we find the true positives, which represent instances where the model correctly predicted the positive class (95). True negatives, indicating correct predictions of the negative class, are located in the upper-right quadrant (70). Conversely, false positives, indicating incorrect predictions of the positive class, are in the upper-left quadrant (68), whereas false negatives, representing incorrect predictions of the negative class, reside in the lower-left quadrant (32). These figures can be plugged into a pair of arithmetic operations to get the accuracy rates per class. The accuracy rate when the true label is 0 is

$$\text{True Label}_0 = \left(\frac{tn}{tn + fp}\right)100$$

where

- tn represents the true negatives.
- fp represents the false positives.

And the accuracy rate when the true label is 1 is

$$\text{True Label}_1 = \left(\frac{tp}{tp + fp}\right) 100$$

where

- tp represents the true positives.
- fn represents the false negatives.

Using Python again as a calculator, we get the accuracy rates for each class label:

```
>>> accuracy_rate_0 = 70 / (70 + 68) * 100
>>> print(f'Accuracy: {round(accuracy_rate_0, 0)}%')
Accuracy: 51.0%

>>> accuracy_rate_1 = 95 / (95 + 32) * 100
>>> pr print(f'Accuracy: {round(accuracy_rate_1, 0)}%')
Accuracy: 75.0%
```

The random forest performed similarly to the decision tree across classes: it struggled when the true label was 0 but performed quite well (better than the decision tree, in fact) when the true label was 1.

6.4.4 Feature importance

Understanding the importance of each feature in a random forest model is crucial for interpreting its predictions and gaining insights into the underlying data patterns. In a decision tree, we can easily discern the importance of features by plotting the standalone tree and observing the splits. However, in a random forest, which consists of multiple decision trees (50 in this case), plotting each individual tree is impractical. Instead, we need to take a different approach to evaluate feature importance across the entire forest. By analyzing feature importance, we can identify which features have the most significant effect on the model's decisions, enabling better understanding and transparency and potentially guiding further data preprocessing or feature engineering steps.

Feature importance in a random forest is typically measured by how much each feature contributes to reducing impurity—features that consistently lead to better splits across many trees are assigned higher importance scores. Feature importance can be visualized either by printing it in a table format or by plotting it as a simple bar chart using methods from the `matplotlib` library. We are doing the latter:

Retrieves the importance scores of each feature used in the trained random forest model

Extracts the column names of the feature matrix X, representing the feature names used in the model

```
>>> importances = rf.feature_importances_
>>> features = X.columns
>>> indices = np.argsort(importances)[::-1]
```

Sorts the indices in the importances array in descending order, so the most important features are listed first

```
>>> sorted_features = features[indices]
>>> sorted_importances = importances[indices]
>>> plt.figure()
>>> plt.bar(sorted_features, sorted_importances)
>>> plt.xlabel('Feature')
>>> plt.ylabel('Feature Importance')
>>> plt.title('Feature Importance in \
>>>           RandomForestClassifier')
>>> plt.tight_layout()
>>> plt.show()
```

Reorders the features array according to the sorted indices

Reorders the importances array according to the sorted indices

Initializes the plot

Creates a bar chart where sorted_features is on the x axis and sorted_importances on the y axis

Sets the x-axis label

Sets the y-axis label

Sets the title

Prevents the x-axis labels from overlapping

Displays the plot

In a feature importance plot generated from a `RandomForestClassifier`, the values represent the relative importance of each feature in predicting the target variable (see figure 6.12). A feature importance value of 0.6 for one feature indicates that this feature contributes significantly to the model's predictions, suggesting it has a strong

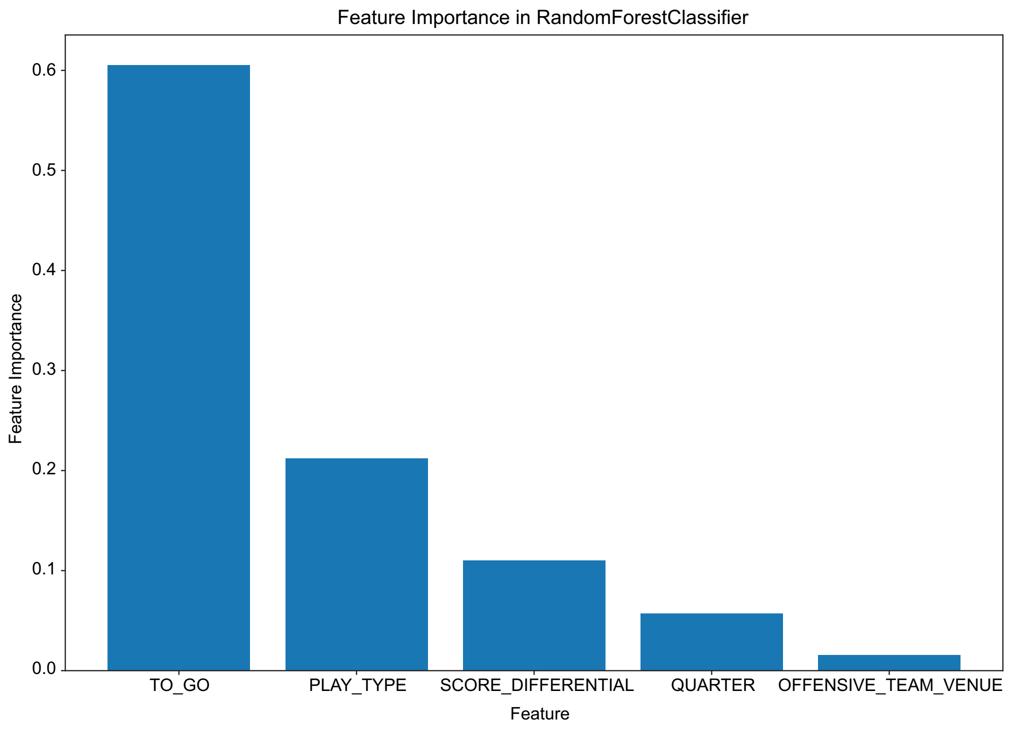

Figure 6.12 A feature importance plot generated from the `RandomForestClassifier`. It displays the relative importance of each feature to the final predicted class labels, where, if stacked, a single bar would equal 1. The features are sorted, from left to right, in descending order of relative importance, with `TO_GO` and `PLAY_TYPE` accounting for approximately 80% of the model's predictive power.

influence on the target variable. Conversely, a feature importance value of 0.1 for another feature implies that this feature has less effect on the predictions compared to the first feature, but it still contributes to the model's overall performance. Therefore, higher values indicate more important features, whereas lower values indicate less influential features.

These findings accurately reflect the analysis conducted prior to fitting our decision tree. It appears that the random forest, in contrast to the decision tree, was able to detect discrepancies in CONVERT class labels between quarters 1 through 3 versus quarter 4.

6.4.5 Extracting random trees

Although it's not practical to plot or display 50 trees, we can extract a small random sample (see figures 6.13 and 6.14). By presenting a subset of randomly extracted trees from the random forest ensemble, the diversity inherent in the model is revealed. Each of these individual trees is trained on distinct subsets of the data and features, providing insight into the varied decision-making processes that contribute to the overall predictive capability of the random forest. Visualizing these trees allows us to analyze differences in feature importance, node splitting criteria, and the overall structure of the ensemble, demonstrating the robustness and adaptability of the random forest algorithm:

```
>>> random_trees = \
>>>     random.sample(rf.estimators_, 2)
>>> for i, tree in enumerate(random_trees):
>>>     plt.figure()
>>>     plot_tree(tree,
>>>               feature_names = X_train.columns,
>>>               filled = True,
>>>               rounded = True,
>>>               fontsize = 12)
>>>     plt.title(f'Tree {i+1}')
>>>     plt.show()
```

Extracts a random sample of two trees from the random forest ensemble rfestimators

Iterates over each tree in random_trees and assigns an index to each tree for enumeration

Sets the dimension of the plot

Generates the decision tree; specifies the feature names from the training data for node labeling

Applies a color scheme to the nodes

Rounds the edges of the nodes for aesthetic purposes

Sets the size of the font

Sets the title of each plot by incrementing the tree index by 1

Displays the plot

The second random tree (figure 6.14) has a very different structure from the first (figure 6.13). In contrast to decision trees, random forest trees lack a direct attribute displaying the classification at each node. Despite the challenge of identifying the randomly selected features for each tree, owing to potential pruning effects, it remains intriguing to observe the diversity among trees. Particularly notable are variations in sample sizes, as well as the assignment of features and splits to the root node. Although examining two trees offers nothing definitive in terms of the final class predictions, analyzing a pair of randomly extracted trees provides valuable insights into the ensemble's inner workings, shedding light on the model's robustness and the variability in feature importance and decision-making processes across different trees.

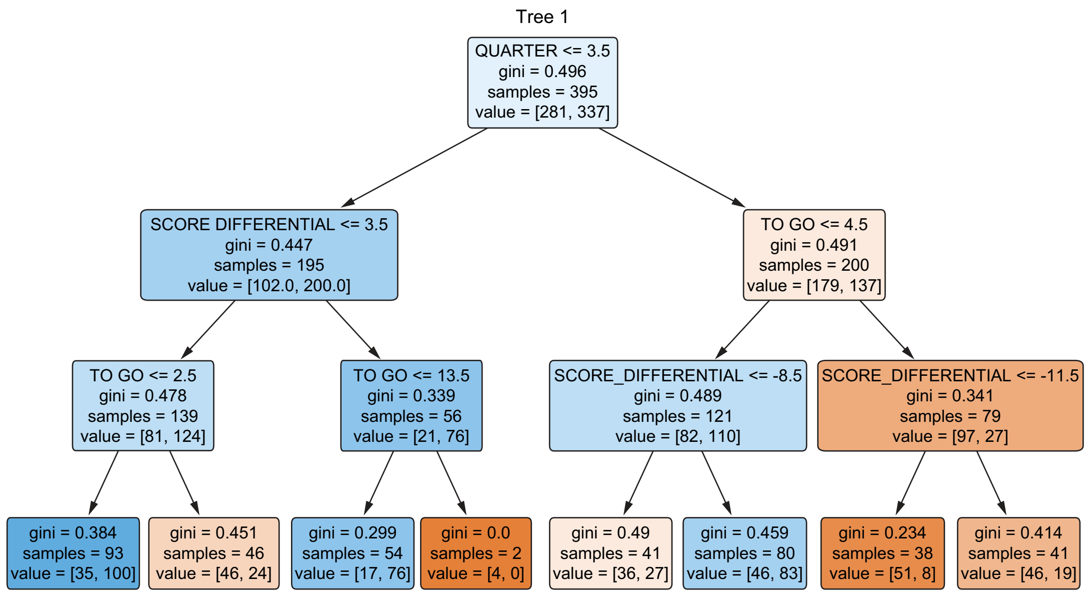

Figure 6.13 The first of two random trees from a random forest model containing 50 trees. Notice that `QUARTER` is at the root node; thus, based on this one random split of the data, `QUARTER`, which didn't factor into our decision tree model, is the most significant variable in this subset for predicting the final class labels.

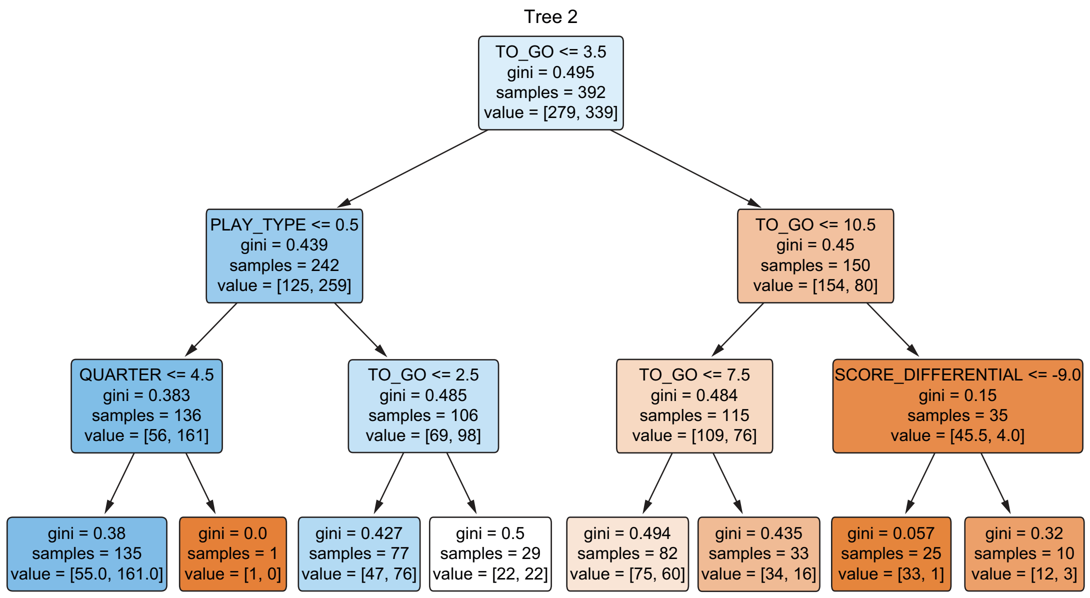

Figure 6.14 The second of two random trees from the same random forest model. The features and splits to construct one tree versus another can, and will, vary significantly.

In this chapter and the two that preceded it, we covered the foundations and applications of linear and logistic regressions, examined the intricacies of decision trees, and explored the ensemble learning technique of random forests. These models have provided us with a diverse set of tools to address a wide range of data types and predictive challenges. In the next chapter, we will advance to fitting time series models, focusing on their unique temporal dependencies and autocorrelations. This will involve techniques such as ARIMA and exponential smoothing, providing a comprehensive approach to forecasting and analyzing sequential data.

Summary

- A decision tree is a supervised machine learning model used for classification and regression tasks. It splits the data into subsets based on feature values, creating a tree structure with decision nodes and leaf nodes representing predictions. Decision trees are easy to interpret and visualize.
- Training a decision tree involves splitting the data based on features to minimize node impurities. It uses the training set for learning and the test set for evaluation, ensuring unbiased accuracy and generalization assessment.
- Evaluating a decision tree includes computing overall accuracy by comparing predicted and actual labels in the test set. A confusion matrix provides a detailed performance breakdown, showing true/false positives and negatives for each class, helping to identify areas for improvement.
- Building a decision tree involves recursive splitting based on significant features to maximize class separation, continuing until the subsets are pure or meet stopping criteria. Interpretation follows the path from root to leaves, ensuring consistent model construction and understanding.
- A random forest is a supervised model for classification and regression, consisting of multiple decision trees trained on different data and feature subsets. This ensemble method typically enhances accuracy and reduces overfitting by aggregating predictions from multiple trees, providing robust performance.
- Evaluating a random forest involves steps similar to a decision tree, including computing overall accuracy and using a confusion matrix for granular performance insights. The aggregated results from multiple trees offer a more robust evaluation and reliable performance across classes.
- Feature importance in random forests measures each feature's contribution to predictive power by assessing impurity reduction across all trees. This provides a comprehensive view of influential features, helping to identify and prioritize significant variables for accurate predictions.
- Although this chapter focuses on decision trees and random forests, other powerful tree-based methods like XGBoost and Gradient Boosting are also popular for tackling complex classification and regression problems. These models build on the strengths of decision trees, using advanced techniques to boost accuracy and handle challenging data patterns and offering further options for sophisticated analysis beyond what we covered here.