2

Introduction to ML via Sklearn

*Present x: The title slide and the Learning Objectives slide. An overview of what we will achieve in this course*

Lesson Objectives:

By the end of this lesson, you will be able to:

* Prepare data for different types of models
* Tune model hyperparameters
* Scale data and tune hyperparameters in a pipeline
* Extract feature importance from tuned model
* Evaluate performance of classification and regression models

Introduction

*Present x: Brief introduction to ML via Scikit-Learn*

Scikit-learn is a free, open-source library built for Python that contains an assortment of supervised and unsupervised machine learning algorithms. Additionally, scikit-learn provides functions for data preprocessing, hyperparameter tuning, and model evaluation. Scikit-learn streamlines the model building process and is easy to install on a wide variety of platforms. Started in 2007 as a Google Summer of Code project by David Corneapeau, and after a series of developments and releases, scikit-learn has evolved into one of the premier tools used by academics and professionals for general purpose machine learning. In this lesson,

First steps with scikit learn

*Present x: First steps with scikit-learn*

To get started with scikit-learn, we will begin by installing the library on our local machine. We will then cover how to preprocess our data’s features by creating dummy variables, scaling the features, and splitting them it into training and testing data sets.

Installation

Scikit-learn is natively installed in the Anaconda distribution of Python as well as some Docker images, but it can also be installed with pip. However, as a prerequisite for Scikit-learn’s installation, NumPy and SciPy must be installed. Once NumPy and SciPy have been installed, scikit-learn can be installed from the terminal using pip install scikit-learn.

Check to make sure scikit-learn is installed using:

import sklearn

print(sklearn.\_\_version\_\_)

Refer to the resultant output:



Figure 2.x: Version of scikit-learn installed

Now that we have successfully installed scikit-learn on our machine, we can begin building machine learning models. In this chapter, we will be learning linear and logistic regression, support vector machines (SVM), decision trees, and random forests. First, we will cover linear and logistic regression.

Introduction to linear and logistic regression

*Present x: Introduction to linear and logistic regression*

In regression, a single dependent, or outcome, variable is predicted using 1 or more independent variables. Use-cases for regression include but are not limited to predicting:

* win percentage given a variety of team statistics
* the risk of heart disease given family history and a number of physical and psychological characteristics
* the likelihood of snowfall given a number of climate measurements

Linear and logistic regression are popular choices for predicting such aforementioned outcomes due to the ease and transparency of interpretability as well as the ability to extrapolate to values not seen in the training data.

The end goal of linear regression is to draw a straight line through the observations that minimizes the absolute distance between the line and the observations (i.e., the line of best fit). Therefore, in linear regression, it is assumed that the relationship between the feature(s) and the continuous dependent variable follows a straight, linear, line. Lines are defined in slope-intercept form (i.e., *y = a + bx*) whereas *a* is the intercept (i.e., the value of *y* when *x* is 0), *b* is the slope, and *x* is the independent variable. There are two types of linear regression: simple linear regression and multiple linear regression.

Simple linear regression

*Present x: Simple linear regression*

Simple linear regression models define the relationship between 1 feature and the continuous outcome variable using *y* = *α* + *βx*. This equation is similar to the slope-intercept form whereas *y* denotes the predicted value of the dependent variable, *α* denotes the intercept, *β* (beta) represents the slope, and *x* is the value of the independent variable. Given *x*, regression models compute the values for *α* and *β* that minimize the absolute difference between predicted y values (i.e., y-hat) and actual y values. For example, if we are predicting the weight of an individual in kg using height in meters as the lone predictor variable, and the simple linear regression model computes 1.5 as the value for *α* and 50 as the coefficient for *β*, this model can be interpreted as for every 1 m increase in height, weight increases by 50 kg. Thus, we can predict that the weight of an individual who is 1.8 m is 91.5 kg using *y* = 1.5 + (50 x 1.8). In the following exercises we will demonstrate conducting simple linear regression using scikit-learn.

Exercise 1: Preparing data for linear regression model

*Exercise 1: Preparing data for linear regression model*

To prepare our data for a simple linear regression model, we will use a random subset of the Weather in Szeged 2006-2016 dataset which consists of hourly weather measurements from April 1, 2006 to September 9, 2016 in Szeged, Hungary. The adapted data consists of 10000 observations of 8 variables:

* Temperature\_c: temperature in Celsius
* Humidity:
* Wind\_Speed\_kmh: wind speed in kilometers per hour
* Wind\_Bearing\_Degrees:
* Visibility\_km: visibility in kilometers
* Pressure\_millibars:
* Rain: rain = 1, snow = 0
* Description: warm, normal, or cold

1. Import the ‘weather.csv’ data set using the following code:

import pandas as pd

df = pd.read\_csv('weather.csv')

1. Explore the data using df.info()

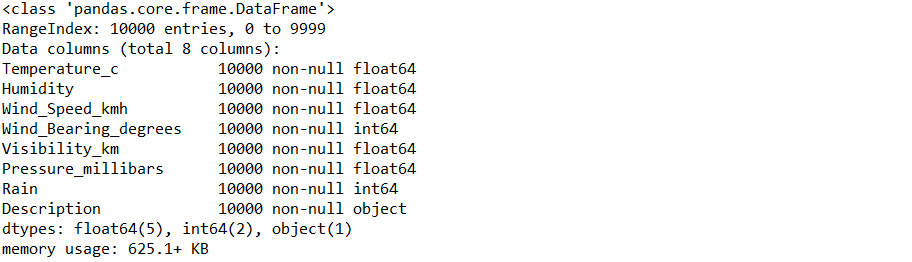


Figure 2.x: Information describing df

1. The ‘Description’ column is the lone categorical variable in df. Check the number of levels in ‘Description’ as follows:

levels = len(pd.value\_counts(df['Description']))

print('There are {} levels in the Description column'.format(levels))



Figure 2.x: Number of levels in the ‘Description’ column

1. The code below shows how to dummy code all categorical variables in 1 step:

import pandas as pd

df\_dummies = pd.get\_dummies(df, drop\_first=True)

1. The original data frame, df, consisted of 8 columns; one of which was nominal with 3 levels. In step 4, we transformed the nominal feature into 2 separate dummy variables and dropped the original feature, ‘Description’. Thus, df\_dummies should now contain 9 columns. Check this out using the following code:

print('There are {} columns in df\_dummies'.format(df\_dummies.shape[1]))



Figure 2.x: Number of columns after dummy coding

1. It is good practice to remove any possible order effects by shuffling the rows in the data before splitting the data into features (X) and outcome (y). To shuffle the rows in df\_dummies, refer to the code below:

from sklearn.utils import shuffle

df\_shuffled = shuffle(df\_dummies, random\_state=42)

1. Now that the data has been shuffled, we will split the rows in our data into features (X) and dependent variable (y). In this exercise, we will pretend that the column ‘Temperature\_c’ (temperature in Celsius) is the dependent variable and that we are preparing data to fit a linear regression model. First, we will split df\_shuffled into X and y as follows:

DV = ‘Temperature\_c’

X = df\_shuffled.drop(DV, axis=1)

y = df\_shuffled[DV]

1. Lastly, we will split X and y into testing and training data using the code below:

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.33, random\_state=42)

Now that the data has been dummy coded, shuffled, split into X and y, and further divided into testing and training data sets it is ready to be used in a linear or logistic regression model.

Note:

It is acknowledged that we did not be scale our features when we prepared our data for linear and logistic regression analysis. However, later in the lesson when we learn about SVM, decision trees, and random forests we will be scaling our features in addition to the preprocessing steps described below.

**Discussion:** Why might we refrain from scaling our data before using it in a linear or logistic regression analysis?

**Answer**: A major reason for using linear or logistic regression is the ease of interpretability. Scaling the data creates an intercept and slope for a line of best fit that applies to the transformed features and not the original features, making interpretability of the original features much more convoluted.

Exercise 2: Fitting a simple linear regression model and determining the intercept and coefficient

*Exercise 2: Fitting a simple linear regression model*

In this exercise, we will continue using the data we prepared in Exercise 1 to fit a simple linear regression model to predict the temperature in Celsius from the humidity.

1. To instantiate a linear regression model refer to the code below:

from sklearn.linear\_model import LinearRegression

model = LinearRegression()

1. Fit the model to the ‘Humidity’ column in the training data using model.fit(X\_train[['Humidity']], y\_train).



Figure 2.x: Output from fitting the simple linear regression model

1. Extract the value for the intercept using the following code:

intercept = model.intercept\_

1. Extract the value for the coefficient as follows:

coefficient = model.coef\_

1. Now, we can print a message with the formula for predicting temperature in Celsius using the code below:

print('Temperature = {0:0.2f} + ({1:0.2f} x Humidity)'.format(intercept, coefficient[0]))



Figure 2.x: Formula to predict temperature in Celsius from humidity using simple linear regression

Great work! According to this simple linear regression model, a day with 0.78 humidity has a predicted temperature in Celsius of 10.56. Now that we are familiar with extracting the intercept and coefficients of our simple linear regression model, it is time to generate predictions and subsequently evaluate how the model performs on unseen, test data.

**Teaching tip**: Practice calculating temperature at various levels of humidity.

Exercise 3: Generating predictions and evaluating performance of simple linear regression model

*Exercise 3: Generating predictions and evaluating performance of simple linear regression model*

The very purpose of supervised learning is to use existing, labeled data to generate predictions. Thus, this exercise will demonstrate how to generate predictions on the test feature and generate model performance metrics by comparing the predictions to the actual values.

1. Generate predictions on the test data using the following:

predictions = model.predict(X\_test[['Humidity']])

1. A common way to evaluate model performance is to examine the correlation between the predicted and actual values using a scatterplot and a Pearson r correlation coefficient. Refer to the following code:

import matplotlib.pyplot as plt

from scipy.stats import pearsonr

plt.scatter(y\_test, predictions)

plt.xlabel('Y Test (True Values)')

plt.ylabel('Predicted Values')

plt.title('Predicted vs. Actual Values (r = {0:0.2f})'.format(pearsonr(y\_test, predictions)[0], 2))

plt.show()

Below is the resultant output:

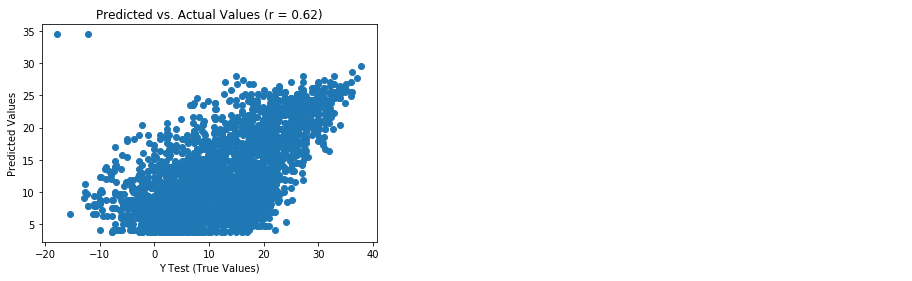


Figure 2.x: Predicted vs actual values from simple linear regression model

1. With a Pearson r value of 0.62, there is a moderate correlation between the predicted and actual values. A perfect model would have all points on the plot in a straight line and an r value of 1.0. Another indicator of model performance is the distribution of the residuals (i.e., the difference between the predicted and actual values). If the model fits the data well, the residuals will be normally distributed. To create a density plot of the residuals, refer to the following code:

import seaborn as sns

from scipy.stats import shapiro

sns.distplot((y\_test - predictions), bins = 50)

plt.xlabel('Residuals')

plt.ylabel('Density')

plt.title('Histogram of Residuals (Shapiro W p-value = {0:0.3f})'.format(shapiro(y\_test - predictions)[1]))

plt.show()

Refer to the resultant output below:

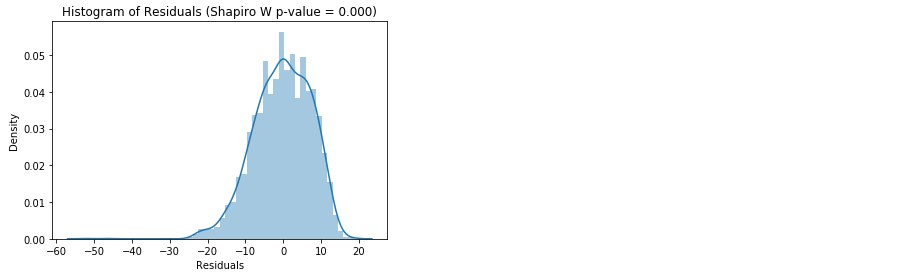


Figure 2.x: Histogram of residuals from simple linear regression model

1. The histogram shows us that the residuals are negatively skewed and the value of the Shapiro W p-value in the title tells us that the distribution is not normal. This gives us further evidence that our model is not so good. Lastly, we will compute metrics for mean absolute error, mean squared error, root mean squared error, and R-squared and put them into a data frame using the code below:

from sklearn import metrics

import numpy as np

metrics\_df = pd.DataFrame({'Metric': ['MAE',

'MSE',

'RMSE',

'R-Squared'],

'Value': [metrics.mean\_absolute\_error(y\_test, predictions),

metrics.mean\_squared\_error(y\_test, predictions),

np.sqrt(metrics.mean\_squared\_error(y\_test, predictions)),

metrics.explained\_variance\_score(y\_test, predictions)]}).round(3)

print(metrics\_df)

Please refer to the resultant output:



Figure 2.x: Model evaluation metrics from simple linear regression model

There we have it. We have successfully used scikit-learn to fit and evaluate a simple linear regression model. This is the first step in a very exciting journey to becoming a machine learning guru. Next, we will continue expanding our knowledge of regression and improving upon this model by exploring multiple linear regression.

**Discussion:** How do we interpret the value associated with R-squared?

**Answer**: We interpret this is as 38.9% of the variance in temperature is explained by humidity.

Multiple linear regression

*Present x: Multiple linear regression*

Multiple linear regression models define the relationship between 2 or more features and the continuous outcome variable using *y* = *α* + *β1xi1 + β2xi2* + … + *βp-1xi,p-1.* Again, *α* represents the intercept and *β* denotes the slope for each feature (*x*) in the model. Thus, if we are predicting the weight of an individual in kg using height in m, total cholesterol in mg/dL, and minutes of cardiovascular exercise per day, and the multiple linear regression model computes 1.5 as the value for *α*, 50 as the coefficient for *β1*, 0.1 as the coefficient for *β2,* and -0.4 as the coefficient for *β3*, this model can be interpreted as for every 1 m increase in height, weight increases by 50 kg, controlling for all other features in the model. Additionally, for every 1 mg/dL increase in total cholesterol, weight increases by 0.1 kg, controlling for all other features in the model. Lastly, for every minute of cardiovascular exercise per day, weight decreases by 0.4 kg, controlling for all other features in the model. Thus, we can predict the weight of an individual who is 1.8 m tall, with total cholesterol of 200 mg/dL, and completes 30 minutes of cardiovascular exercise per day as 99.5 kg using y = 1.5 + (0.1 x 50) + (200 x 0.5) + (30 x -0.4). In the following exercise we will demonstrate conducting multiple linear regression using scikit-learn.

Exercise 4: Fitting a multiple linear regression model and determining the intercept and coefficient

*Exercise 4: Fitting a multiple linear regression model and determining the intercept and coefficient*

In this exercise, we will continue using the data we prepared in Exercise 1 to fit a multiple linear regression model to predict the temperature in Celsius from all of the features in the data.

1. To instantiate a linear regression model refer to the code below:

from sklearn.linear\_model import LinearRegression

model = LinearRegression()

1. Fit the model to the training data using model.fit(X\_train, y\_train).



Figure 2.x: Output from fitting the multiple linear regression model

1. Extract the value for the intercept using the following code:

intercept = model.intercept\_

1. Extract the value for the coefficients as follows:

coefficients = model.coef\_

1. Now, we can print a message with the formula for predicting temperature in Celsius using the code below:

print('Temperature = {0:0.2f} + ({1:0.2f} x Humidity) + ({2:0.2f} x Wind Speed) + ({3:0.2f} x Wind Bearing Degrees) + ({4:0.2f} x Visibility) + ({5:0.2f} x Pressure) + ({6:0.2f} x Rain) + ({7:0.2f} x Normal Weather) + ({8:0.2f} x Warm Weather)'.format(intercept, coefficients[0], coefficients[1], coefficients[2], coefficients[3], coefficients[4], coefficients[5], coefficients[6], coefficients[7]))



Figure 2.x: Formula to predict temperature in Celsius from humidity using multiple linear regression

Nice job! According to this multiple regression model, a day with 0.78 humidity, 5.0 wind speed, 81 wind bearing degrees, 3 km of visibility, 1000 millibars of pressure, no rain, and is described as normal has a predicted temperature in Celsius of 5.72 degrees. Now that we are familiar with extracting the intercept and coefficients of our multiple linear regression model, we can generate predictions and evaluate how the model performs on the test data.

**Discussion**: By what value do I multiply the beta coefficient for ‘Rain’ if it is raining?

**Answer**: If it is raining, we would multiply the beta coefficient for ‘Rain’ by 1 (since it was not raining in our example, we multiplied it by 0).

Activity 1: Generating predictions and evaluating performance of multiple linear regression model

*Activity 1: Generating predictions and evaluating performance of multiple linear regression model*

In Exercise 3, we learned how to generate predictions and evaluate the performance of a simple linear regression model using a variety of methods. To reduce the code redundancy, we will evaluate the performance of our multiple linear regression model using the metrics in step 4 of Exercise 3 and we will determine if the multiple linear regression model performed better or worse in relation to the simple linear regression model.

1. Generate predictions on the test data using all of the features
2. Plot predictions vs actual using a scatterplot
3. Plot the distribution of the residuals
4. Calculate the metrics for mean absolute error, mean squared error, root mean squared error, and R-squared and put them into a data frame
5. Determine if the multiple linear regression model performed better or worse in relation to the simple linear regression model

**Solution:**

1. Generate predictions on the test data using the following:

predictions = model.predict(X\_test)

1. Plot predicted versus actual values on a scatterplot using the following code:

import matplotlib.pyplot as plt

from scipy.stats import pearsonr

plt.scatter(y\_test, predictions)

plt.xlabel('Y Test (True Values)')

plt.ylabel('Predicted Values')

plt.title('Predicted vs. Actual Values (r = {0:0.2f})'.format(pearsonr(y\_test, predictions)[0], 2))

plt.show()

Refer to the resultant output below:

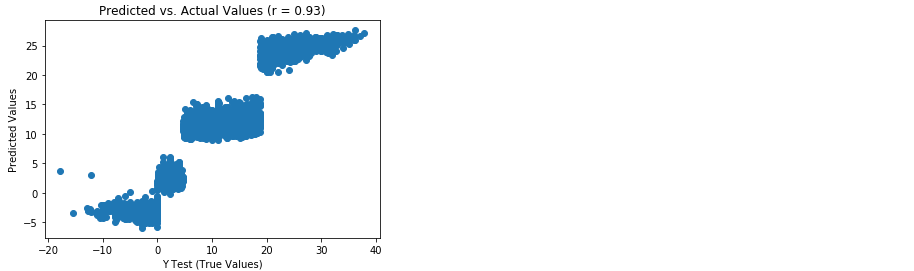


Figure 2.x: Scatterplot of predicted versus actual values from multiple linear regression model

1. There is a much stronger linear correlation between the predicted and actual values in the multiple linear regression model (r = 0.93) relative to the simple linear regression model (r = 0.62). To plot the distribution of the residuals refer to the code below:

import seaborn as sns

from scipy.stats import shapiro

sns.distplot((y\_test - predictions), bins = 50)

plt.xlabel('Residuals')

plt.ylabel('Density')

plt.title('Histogram of Residuals (Shapiro W p-value = {0:0.3f})'.format(shapiro(y\_test - predictions)[1]))

plt.show()

Refer to the resultant output below:

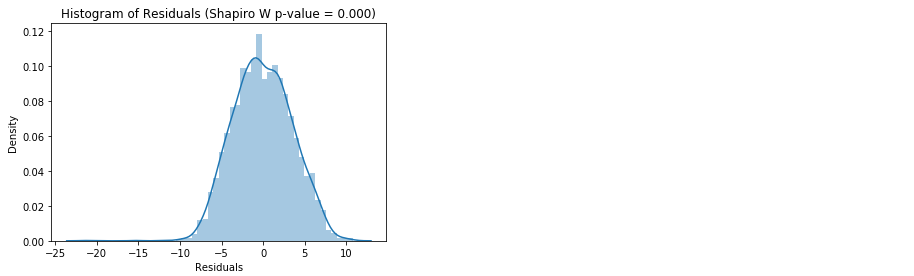


Figure 2.x: Distribution of the residuals from multiple linear regression model

1. Our residuals are negatively skewed and non-normal, but it is less skewed than the simple linear model. Calculate the metrics for mean absolute error, mean squared error, root mean squared error, and R-squared and put them into a data frame as follows:

from sklearn import metrics

import numpy as np

metrics\_df = pd.DataFrame({'Metric': ['MAE',

'MSE',

'RMSE',

'R-Squared'],

'Value': [metrics.mean\_absolute\_error(y\_test, predictions),

metrics.mean\_squared\_error(y\_test, predictions),

np.sqrt(metrics.mean\_squared\_error(y\_test, predictions)),

metrics.explained\_variance\_score(y\_test, predictions)]}).round(3)

print(metrics\_df)

Please refer to the resultant output:



Figure 2.x: Model evaluation metrics from multiple linear regression model

1. The multiple linear regression model performed better in every metric relative to the simple linear regression model. Most notably, in the simple linear regression, only 38.9% of the variance in Temperature was described by the model. Whereas, in the multiple linear regression model, 86.6% of the variance in Temperature was explained by the combination of features.

The transparent nature of the intercept and beta coefficients make linear regression models very easy to interpret. In business, it is commonly requested that data scientists explain the effect of a certain feature on an outcome. Thus, linear regression provides metrics allowing a reasonable response to the business inquiry above. However, much of the time a problem requires the data scientist to predict an outcome measure that is not continuous, but categorical. For example, in insurance, given certain features of an insured what is the probability that this customer will not renew their policy? In this case, there is not a linear relationship between the features in the data and the outcome variable, so linear regression will falter. A viable option for conducting regression analysis on a categorical dependent variable is logistic regression.

**Discussion:** Is logistic regression used for regression or classification problems?

**Answer**: Logistic regression is used to solve classification problems.

Logistic regression

*Present x: Logistic regression*

Logistic regression uses categorical and continuous variables to predict a categorical outcome. When the dependent variable of choice has two categorical outcomes the analysis is termed binary logistic regression. However, if the outcome variable consists of more than 2 levels the analysis is referred to as multinomial logistic regression. For the purposes of this lesson we will focus our learning on the former.

When predicting a binary outcome, we do not have a linear relationship between the features and the outcome variable; an assumption of linear regression. Thus, to express a nonlinear relationship in a linear way we must transform the data using the logarithmic transformation. As a result, logistic regression allows us to predict the probability of the binary outcome occurring given the feature(s) in the model.

In logistic regression with 1 predictor, the logistic regression equation is shown below:

Equation 2.x: Logistic regression formula with 1 predictor

In equation 2.x, *P(Y)* is the probability of the outcome occurring, *e* is the base of natural logarithms, *α* is the intercept, *β* is the beta coefficient, and *x* is the value of the predictor. This equation can be extended to multiple predictors using the formula below:

Equation 2.x: Logistic regression formula with more than 1 predictor

Thus, using logistic regression to model the probability of an event occurring is the same as fitting a linear regression model except the continuous outcome variable has been replaced by the log odds of success for a binary outcome variable. In linear regression, we assumed a linear relationship between the predictor variable(s) and the outcome variable. Logistic regression, on the other hand, assumes a linear relationship between the predictor variable(s) and the natural log of *p*/(1-*p*), where *p* is the probability of the event occurring.

In the following exercise we will use the ‘weather.csv’ data set to demonstrate building a logistic regression model to predict the probability of rain using all the features in our data.

Exercise 5: Fitting a logistic regression model and determining the intercept and coefficient

*Exercise 5: Fitting a logistic regression model and determining the intercept and coefficient*

To model the probability of rain (as opposed to snow) using all of the features in our data, we will use the ‘weather.csv’ file and store the dichotomous variable ‘Rain’ as the outcome measure.

1. Import data using the following code:

import pandas as pd

df = pd.read\_csv('weather.csv')

Dummy code the ‘Summary’ variable as follows:

import pandas as pd

df\_dummies = pd.get\_dummies(df, drop\_first=True)

Shuffle df\_dummies using the code below:

from sklearn.utils import shuffle

df\_shuffled = shuffle(df\_dummies, random\_state=42)

Split the features and outcome into X and y, respectively as follows:

DV = 'Rain'

X = df\_shuffled.drop(DV, axis=1)

y = df\_shuffled[DV]

Split the features and outcome into training and testing data using the code below:

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.33, random\_state=42)

1. Instantiate a logistic regression model using:

from sklearn.linear\_model import LogisticRegression

model = LogisticRegression()

1. Fit the logistic regression model to the training data using model.fit(X\_train, y\_train)

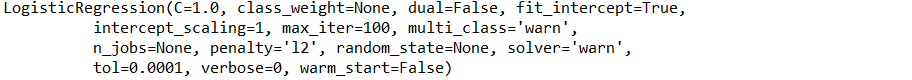


Figure 2.x: Output from fitting logistic regression model

1. Get the intercept using intercept = model.intercept\_
2. Extract the coefficients using coefficients = model.coef\_
3. Place the coefficients into a list using coef\_list = list(coefficients[0,:])
4. Match features to their coefficients, place them in a data frame, and print the data frame to the console as follows:

coef\_df = pd.DataFrame({'Feature': list(X\_train.columns),

'Coefficient': coef\_list})

print(coef\_df)

Refer to the resultant output below:



Figure 2.x: Features and their coefficients from logistic regression model

The coefficient for temperature can be interpreted as for every 1 degree increase in temperature, the log odds of rain increase by 5.69, controlling for all other features in the model. To generate predictions, we could convert the log odds to odds and odds to probability. However, scikit-learn has functionality to generate predicted probability as well as predicted classes.

**Discussion:** Why did we not scale the data prior to using it in our model?

**Answer**: We opted not to scale our data to improve the ease of interpretability provided by the model’s beta coefficients.

Exercise 6: Generating predictions and evaluating performance of logistic regression model

*Exercise 6: Generating predictions and evaluating performance of multiple linear regression model*

In exercise 5 we learned how to fit a logistic regression model and extract the elements necessary to generate predictions. However, scikit-learn makes our lives much easier by providing us with functions to predict the probability of an outcome as well as the classes of an outcome. In this exercise, we will learn to generate predicted probabilities and classes as well as evaluating model performance using a confusion matrix and a classification report.

1. Generate predicted probabilities using predicted\_prob = model.predict\_proba(X\_test)[:,1]
2. Generate predicted classes using predicted\_class = model.predict(X\_test)
3. Evaluate a performance using a confusion matrix as follows:

from sklearn.metrics import confusion\_matrix

import numpy as np

cm = pd.DataFrame(confusion\_matrix(y\_test, predicted\_class))

cm['Total'] = np.sum(cm, axis=1)

cm = cm.append(np.sum(cm, axis=0), ignore\_index=True)

cm.columns = ['Predicted No', 'Predicted Yes', 'Total']

cm = cm.set\_index([['Actual No', 'Actual Yes', 'Total']])

print(cm)

Refer to the resultant output below:

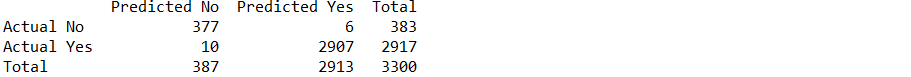


Figure 2.x: Confusion matrix from logistic regression model

From the confusion matrix, we can see that of the 383 observations that were not classified as rainy 377 of them were correctly classified and of the 2917 observations that were classified as rainy 2907 of them were correctly classified. To further inspect our model’s performance using metrics such as precision, recall, and f1-score we will generate a classification report.

1. Generate a classification report using the following code:

from sklearn.metrics import classification\_report

print(classification\_report(y\_test, predicted\_class))

Refer to the resultant output:

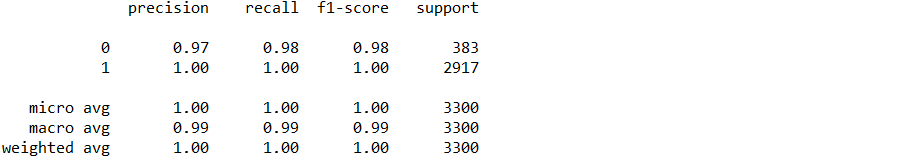


Figure 2.x: Classification report generated from logistic regression model

As we can see from our confusion matrix and classification report, our model is performing very well and may be difficult to improve upon. However, machine learning models including logistic regression consist of numerous hyperparameters that can be adjusted to further improve model performance. In the next exercise, we will learn to find the optimal combination of hyperparameters to maximize model performance.

**Discussion:** How many false positives and false negatives did we predict from this model?

**Answer**: There were 6 false positives and 10 false negatives.

Exercise 7: Tuning hyperparameters of logistic regression model

*Exercise 7: Tuning hyperparameters of multiple logistic regression model*

In step 3 of Exercise 5, we fit a logistic regression model and the subsequent output from that model is displayed in Figure 2.x. Each of those arguments inside the LogisticRegression() function is set to a default hyperparameter. To tune the model, we will use scikit-learn’s grid search function which fits a model for every combination of possible hyperparameter values and determines the value for each hyperparameter resulting in the “best” model. In this exercise, we will learn how to use grid search to tune models.

Continuing from Exercise 6:

1. The data has already been prepared for us (see Exercise 6), thus we can jump right into instantiating a grid of possible hyperparameter values as follows:

import numpy as np

grid = {'penalty': ['l1', 'l2'],

'C': np.linspace(1, 10, 10),

'solver': ['liblinear']}

1. Instantiate a grid search model to find the model with the greatest f1 score (i.e., the harmonic average of precision and recall) as follows:

from sklearn.model\_selection import GridSearchCV

from sklearn.linear\_model import LogisticRegression

model = GridSearchCV(LogisticRegression(), grid, scoring='f1', cv=5)

1. Fit the model on the training using model.fit(X\_train, y\_train) (keep in mind, this may take a while) and find the resultant output below:

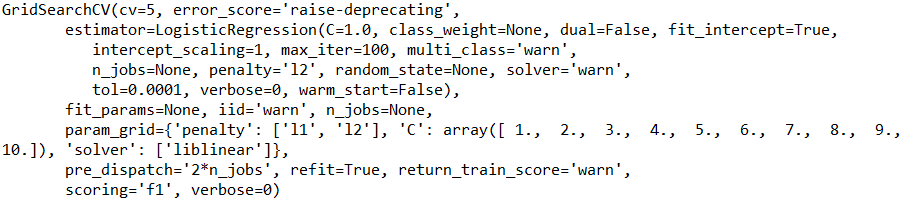


Figure 2.x: Output from logistic regression grid search model

1. We can return the optimal combination of hyperparameters as a dictionary as follows:

best\_parameters = model.best\_params\_

print(best\_parameters)

Refer to the resultant output below:



Figure 2.x: Tuned hyperparameters from logistic regression grid search model

We have found the combination of hyperparameters that maximize f1 score. Remember, simply using the default hyperparameters in Exercise 5 resulted in a model that performed very well on the test data. Thus, in the following activity, we will evaluate how the model with tuned hyperparameters performed on the test data.

**Discussion**: Why is not every hyperparameter listed in the grid?

**Answer**: When not every hyperparameter is listed in the grid because the user has decided to use the default hyperparameter in that case. Additionally, some hyperparameters will not work when other hyperparameters are selected simultaneously, so it is important to always refer and pay close attention to the algorithm’s documentation when setting up a grid for tuning.

Activity 2: Generating predictions and evaluating performance of tuned logistic regression model

*Activity 2: Evaluating tuned model performance*

Once the best combination of hyperparameters has been converged upon, we need to evaluate model performance much like we did in Exercise 5.

Continuing from Exercise 7:

1. Generate the predicted probabilities of rain
2. Generate the predicted class of rain
3. Evaluate performance with a confusion matrix and store it as a data frame
4. Print a classification report

**Solution:**

1. Generate predicted probabilities of rain using the following code:

predicted\_prob = model.predict\_proba(X\_test)[:,1]

1. Generate predicted class of rain using predicted\_class = model.predict(X\_test)
2. Evaluate performance using a confusion matrix and save it as a data frame using the following code:

from sklearn.metrics import confusion\_matrix

import numpy as np

cm = pd.DataFrame(confusion\_matrix(y\_test, predicted\_class))

cm['Total'] = np.sum(cm, axis=1)

cm = cm.append(np.sum(cm, axis=0), ignore\_index=True)

cm.columns = ['Predicted No', 'Predicted Yes', 'Total']

cm = cm.set\_index([['Actual No', 'Actual Yes', 'Total']])

print(cm)

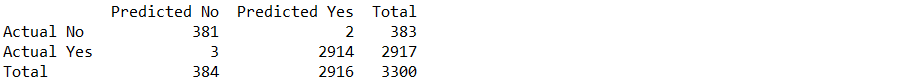


Figure 2.x: Confusion matrix from logistic regression grid search model

1. Nice! We have decreased our number of false positives from 6 to 2. Additionally, our false negatives were lowered from 10 to 3 (see Exercise 6). For further evaluation, print a classification report as follows:

from sklearn.metrics import classification\_report

print(classification\_report(y\_test, predicted\_class))

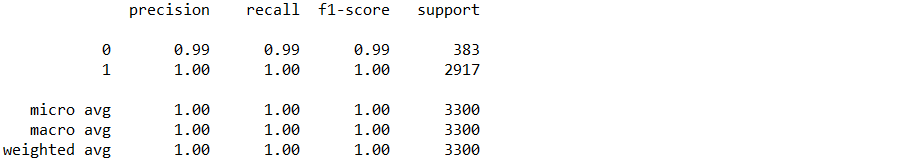


Figure 2.x: Classification report from logistic regression grid search model

By tuning the hyperparameters of the logistic regression model, we were able to improve upon a logistic regression model that was already performing very well. We will continue to expand upon tuning different types of models in the following exercises and activities.

Classification using support vector machines (SVM)

*Present x: Max-Margin classification using support vector machines*

SVM is an algorithm for supervised learning that solves both classification and regression problems. However, SVM is most commonly used in classification problems, so, for the purposes of this chapter and for the sake of simplicity, we will focus on SVM as a binary classifier. The goal of SVM is to determine the best location of a hyperplane that create a class boundary between data points plotted on a multidimensional space. To help clarify this concept, refer to Figure 2.x.

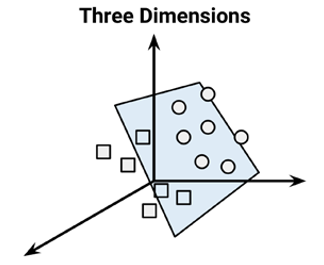


Figure 2.x: Hyperplane (blue) separating the circles from the squares in 3 dimensions

In Figure 2.x, the squares and circles are observations in the same data frame that represent different classes. In this figure, the hyperplane is depicted by a semi-transparent blue boundary lying between the circles and squares that separate the observations into 2 distinct classes. In this example, the observations are said to be linearly separable.

The location of the hyperplane is determined by finding the position that creates the maximum separation (i.e., margin) between the two classes. Thus, this is referred to as the Maximum Margin Hyperplane (MMH) and improves the likelihood that the points will remain on the correct side of the hyperplane boundary. It is possible to express the MMH using the points from each class that are closest to the MMH. These points are termed Support Vectors and each class has at least 1. Figure 2.x visually depicts the support vectors in relation to the MMH in 2 dimensions.

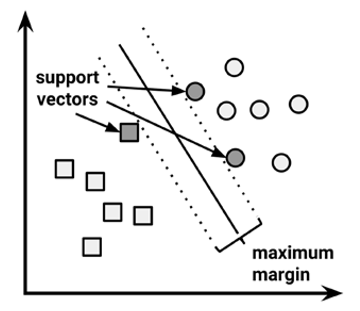


Figure 2.x: Support vectors in relation to the MMH

In reality, most data is not linearly separable. In this case, SVM makes use of a slack variable which creates a soft margin (as opposed to a maximum margin) allowing some observations to fall on the incorrect side of the line (see Figure 2.x below).

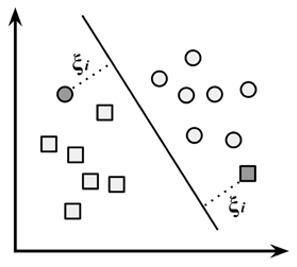


Figure 2.x: 2 observations (as denoted with grey shading and the Greek letter *Χ*i) fall on the incorrect side of the soft margin line.

A cost value is applied to the misclassified data points and, instead of finding the maximum margin, the algorithm minimizes the total cost. As the cost parameter increases, the harder SVM optimization will go for 100 percent separation and may over fit to the training data. Conversely, lower cost parameters emphasize a wider margin and may under fit to the training data. Thus, to create SVM models that perform well on the test data it is important to determine a cost parameter that balances over fitting and under fitting.

Additionally, data that is not linearly separable can be transformed into a higher dimension space using the Kernel trick. After this mapping to a higher dimensional space, a nonlinear relationship can appear linear. By transforming the original data, SVM can discover associations not explicitly apparent in the original features. Scikit-learn uses the Gaussian RBF kernel by default but comes equipped common kernels such as linear, polynomial, and sigmoid as well. In order to maximize the performance of an SVM classifier model, the optimal combination of kernel and cost function must be determined. Luckily, this can be easily achieved using grid search hyperparameter tuning as introduced in Exercise 7. In the following exercises and activities we will learn how this feat is accomplished.

Exercise 8: Preparing data for support vector classifier

*Exercise 8: Preparing data for support vector classifier*

Before fitting an SVM classifier model, we must prepare our data. Since SVM is a black box, meaning the processes between input and output are not explicit, we do not need to worry about interpretability. Thus, we will transform the features in our data into z-scores prior to fitting the model. The following steps will show how to do this.

1. Import ‘weather.csv’ using the following code:

import pandas as pd

df = pd.read\_csv('weather.csv')

1. Dummy code the categorical feature, 'Summary' as follows:

import pandas as pd

df\_dummies = pd.get\_dummies(df, drop\_first=True)

1. Shuffle df\_dummies to remove any ordering effects using the code below:

from sklearn.utils import shuffle

df\_shuffled = shuffle(df\_dummies, random\_state=42)

1. Slit df\_shuffled into X and y using the following code:

DV = 'Rain'

X = df\_shuffled.drop(DV, axis=1)

y = df\_shuffled[DV]

1. Scale the features, X, by transforming them to z-scores as follows:

from sklearn.preprocessing import StandardScaler

model = StandardScaler()

X\_scaled = model.fit\_transform(X)

1. Split X\_scaled and y into testing and training data using the code below:

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_scaled, y, test\_size=0.33, random\_state=42)

Now that our data has been properly divided into features and outcome variable, scaled, and split into testing and training data, we can tune the hyperparameters of our SVC model using a grid search.

Exercise 9: Tuning SVC model using grid search

*Exercise 9: Tuning SVC model using grid search*

Previously, we discussed the importance of determining the optimal cost function and kernel for SVM classifier models. In Exercise 7, we learned how to find the optimal combination of hyperparameters using scikit-learn’s grid search function. In this exercise, we will demonstrate using grid search to find the best combination of the cost function and kernel.

1. Instantiate grid for which to search using the following code:

import numpy as np

grid = {'C': np.linspace(1, 10, 10),

'kernel': ['linear', 'poly', 'rbf', 'sigmoid'],

'shrinking': [True, False],

'tol': np.linspace(0.001, 1, 10)}

1. Instantiate GridSearchCV model with the gamma hyperparameter set to ‘auto’ to avoid warnings and probability to ‘True’ so we can extract probability of rain as follows:

from sklearn.model\_selection import GridSearchCV

from sklearn.svm import SVC

model = GridSearchCV(SVC(gamma='auto', probability=True), grid, scoring='f1', cv=5)

1. Fit the grid search model using model.fit(X\_train, y\_train)

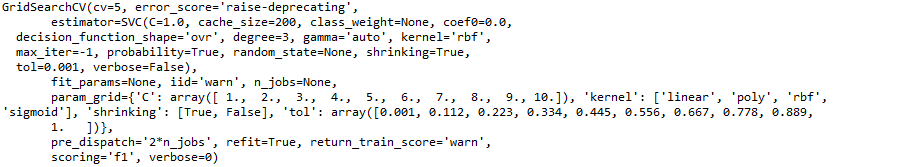


Figure 2.x: Output from fitting SVC grid search model

1. Print the best parameters using the code below:

best\_parameters = model.best\_params\_

print(best\_parameters)

See the resultant output below:



Figure 2.x: Tuned hyperparameters for SVC grid search model

Once the optimal combination of hyperparameters have been determined it is time to generate predictions and subsequently evaluate how our model performed on the unseen, test, data.

**Discussion:** What happens when more possible values of hyperparameters are provided in the grid object?

Answer: More models would be fit and the time to fit would increase.

Activity 3: Generating predictions and evaluating performance of grid search SVC model

*Activity 3: Generating predictions and evaluating performance of grid search SVC model*

In previous exercises/activities, we have learned to generate predictions and evaluate classifier model performance. In this activity we will, again, evaluate the performance of our model by generating predictions, creating a confusion matrix, and printing a classification report.

1. Generate predicted probabilities
2. Extract predicted classes
3. Create and print a confusion matrix
4. Generate and print a classification report

**Solution:**

1. Generate predicted probabilities of rain using predicted\_prob = model.predict\_proba(X\_test)[:,1]
2. Extract predicted classes of rain using predicted\_class = model.predict(X\_test)
3. Create and print a confusion matrix using the code below:

from sklearn.metrics import confusion\_matrix

import numpy as np

cm = pd.DataFrame(confusion\_matrix(y\_test, predicted\_class))

cm['Total'] = np.sum(cm, axis=1)

cm = cm.append(np.sum(cm, axis=0), ignore\_index=True)

cm.columns = ['Predicted No', 'Predicted Yes', 'Total']

cm = cm.set\_index([['Actual No', 'Actual Yes', 'Total']])

print(cm)

See the resultant output below:



Figure 2.x: Confusion matrix from SVC grid search model

1. Generate and print a classification report as follows:

from sklearn.metrics import classification\_report

print(classification\_report(y\_test, predicted\_class))

See the resultant output below:



Figure 2.x: Classification report from SVC grid search model

Here, we demonstrated how to tune the hyperparameters of an SVC model using a grid search. After tuning the SVC model it did not perform as well as the tuned logistic regression model in predicting rain/snow. Additionally, SVC models are a “black box” and do not provide insight into the contribution of features on the outcome measure. In the upcoming section, we will introduce a different algorithm known as a decision tree, which uses a “divide and conquer” approach to generate predictions and offers a feature importance attribute for determining the importance of each feature on the outcome.

Decision trees

*Present x: Decision trees*

Imagine we are considering changing jobs. We are weighing the pros and cons of prospective job opportunities and, after a few years of being in our current positon, we start to realize the things that are important to us. However, not all aspects of a career are of equal importance. In fact, after being in the job force for a few years we decide that the most important aspect of a position is the interest in projects we will be doing, followed by compensation, then work-related stress, trailed by commute time, and, lastly, benefits. We have just created the scaffolding of a cognitive decision tree. We can go further into detail by saying that we want a job where we are very interested in the allocated projects, paying at least $55k/year, with low work-related stress, a commute under 30 minutes, and good dental insurance. Creating mental decision trees is a decision making process we all utilize by nature and is one of the reasons why decision trees are 1 of the most widely-used machine learning algorithms used today.

In machine learning, decision trees use either ‘gini’ impurity or ‘entropy’ information gain as the criterion to measure the quality of a split. First, the decision tree algorithm determines the feature that maximizes the value indicating quality of a split. This becomes referred to as the root node as it is the most important feature in the data. In the job seeking offer mentioned above, being very interested in the prospective projects would be considered the root node. Taking into consideration the root node, the job opportunities are divided into those with very interesting projects and those without very interesting projects. Next, each of these two categories would be divided into the next most important feature given the previous feature(s), and so on and so forth until the potential jobs are identified as of interest or not.

This approach is termed recursive partitioning, or divide and conquer because it continues the process of splitting and subsetting the data until the data determines the subsets in the data as sufficiently homogenous, or:

* Nearly all of the observations at the corresponding node have the same class (i.e., purity)
* There are no further features in the data for which to split
* The tree has reached the size limit decided upon *a priori*

For example, if purity is determined by entropy we must understand that entropy is a measure of randomness within a set of values. Decision trees operate by choosing the splits that minimize entropy (randomness) and, in turn, maximize information gain. Information gain is calculated as the difference in entropy between the split and all other following splits. Total entropy is then computed by taking the sum of the entropy in each partition weighted by the proportion of observations in the partition. Luckily, scikit-learn provides us with a function that does all of this for us. In the following exercises and activities we will implement the decision tree classifier model to predict whether it is raining or snowing using the familiar ‘weather.csv’ data set.

Activity 4: Prepare data for decision tree classifier pipeline

*Activity 4: Prepare data for decision tree classifier pipeline*

In this activity we will prepare our data for a decision tree classifier model by:

1. Importing ‘weather.csv’ and store it as a data frame
2. Dummy code the multi-level, categorical feature ‘Summary’
3. Shuffle the data to remove any possible order effects
4. Split the data into features and outcome
5. Further divide the features and outcome into testing and training data

**Solution:**

1. Import ‘weather.csv’ and store it as a data frame using the following:

import pandas as pd

df = pd.read\_csv('weather.csv')

1. Dummy code the 'Summary' column as follows:

import pandas as pd

df\_dummies = pd.get\_dummies(df, drop\_first=True)

1. Shuffle df\_dummies using the code below:

from sklearn.utils import shuffle

df\_shuffled = shuffle(df\_dummies, random\_state=42)

1. Split df\_shuffled into X and y as follows:

DV = 'Rain'

X = df\_shuffled.drop(DV, axis=1)

y = df\_shuffled[DV]

1. Split X and y into testing and training data

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.33, random\_state=42)

In previous exercises and activities, we have demonstrated how to scale our data prior to tuning the hyperparameters. In the following exercise, we will learn to scale our data as a step in a pipeline.

**Discussion:** Is there a time where we might refrain from scaling our features prior to fitting a decision tree model?

**Answer**: Yes, if we are planning to visualize the tree and interpret the splits we would not want to scale the features.

Exercise 10: Scaling features and tuning decision tree classifier using grid search in pipeline:

*Exercise 10: Scaling features and tuning decision tree classifier using grid search in pipeline*

Up until this point, we have scaled our features as part of data preprocessing. In the current exercise, we will scale our features and tune our hyperparameters in the steps of a pipeline.

1. Set up the steps for a pipeline

from sklearn.preprocessing import StandardScaler

from sklearn.tree import DecisionTreeClassifier

steps = [('scaler', StandardScaler()), ('Tree', DecisionTreeClassifier())]

1. Setup the pipeline

from sklearn.pipeline import Pipeline

pipeline = Pipeline(steps)

1. Specify the hyperparameter space

import numpy as np

parameters = {'Tree\_\_criterion': ['gini', 'entropy'],

'Tree\_\_min\_weight\_fraction\_leaf': np.linspace(0.0, 0.5, 10),

'Tree\_\_min\_impurity\_decrease': np.linspace(0.0, 1.0, 10),

'Tree\_\_class\_weight': [None, 'balanced'],

'Tree\_\_presort': [True, False]}

1. Instantiate the GridSearchCV model

from sklearn.model\_selection import GridSearchCV

model = GridSearchCV(pipeline, parameters, scoring='f1', cv=5)

1. Fit to the training set

model.fit(X\_train, y\_train)

See the resultant output displayed below:

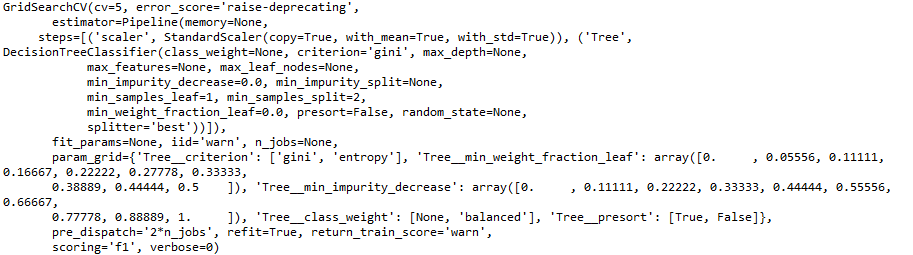


Figure 2.x: Output from fitting a decision tree classifier grid search model using a pipeline

1. Print the tuned parameters

best\_parameters = model.best\_params\_

print(best\_parameters)

See the resultant output below:



Figure 2.x: Tuned hyperparameters for decision tree classifier grid search model

We can see from Figure 2.x that it used ‘gini’ impurity as the criterion to measure the quality of a split. Further explanations of the hyperparameters are being the scope of this chapter but can be found in the algorithm’s scikit-learn documentation.

Remember, in practice, it is common for decision makers to ask how various features are affecting the predictions. In linear and logistic regression, the intercept and coefficient(s) make model predictions very transparent.

**Note:**

Decision trees can also be very easily to interpret as we can see where the decisions were made, but this requires an installation and proper configuration of Graphviz as well as unscaled features. We will not be discussing using Graphviz in this chapter, thus, we will be scaling our features.

Instead of plotting the tree in the following exercise, we will explore an attribute found in scitkit-learn’s tree-based model algorithms, ‘feature\_importances\_’, which returns an array containing values of relative feature importance for each feature. It is important to note that this attribute is unavailable from a grid search model. As a result, in the next exercise, we will learn to programmatically extract values from the best\_parameters dictionary and re-fit the tuned decision tree model; allowing us to access the attributes provided by the decision tree classifier function.

**Discussion:** When we examine the features in the data, which would we predict to be the most important variable in predicting if it is raining/snowing?

**Answer**: We would expect that the most important feature would be temperature as the freezing point of water is 0 degrees celsius.

Exercise 11: Programmatically extracting tuned hyperparameters from decision tree classifier grid search model

*Exercise 11: Programmatically extracting tuned hyperparameters*

In Exercise 10, we saved the tuned hyperparameters as key value pairs in the best\_parameters dictionary. This allows us to programmatically access the values and assign them to the appropriate hyperparameters of a decision tree classifier model. By fitting the tuned decision tree model, we will be able to access the attributes made available from the scikit-learn decision tree classifier function.

1. Print the best\_parameters dictionary using print(best\_parameters)

See the resultant output below:



Figure 2.x: best\_parameters dictionary

1. Prove that we can access the value for ‘Tree\_criterion’ using print(best\_parameters['Tree\_\_criterion'])

See the resultant output below:



Figure 2.x: Value assigned to the ‘Tree\_criterion’ key in the best\_parameters dictionary

1. Instantiate decision tree classifier model and assign the values to the corresponding hyperparameters as follows:

from sklearn.tree import DecisionTreeClassifier

model = DecisionTreeClassifier(class\_weight=best\_parameters['Tree\_\_class\_weight'],

criterion=best\_parameters['Tree\_\_criterion'],

min\_impurity\_decrease=best\_parameters['Tree\_\_min\_impurity\_decrease'],

min\_weight\_fraction\_leaf=best\_parameters['Tree\_\_min\_weight\_fraction\_leaf'],

presort=best\_parameters['Tree\_\_presort'])

1. Fit the model to the training data using model.fit(X\_train, y\_train)

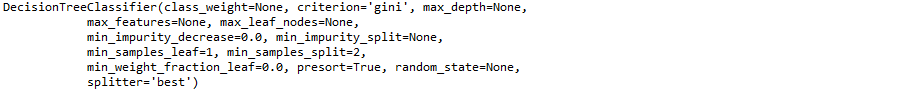


Figure 2.x: Output from fitting the decision tree classifier model with tuned hyperparameters

1. Extract feature\_importances attribute using print(model.feature\_importances\_)

The resultant output is shown below:



Figure 2.x: Array of feature importance from tuned decision tree classifier model

1. From the array in Figure 2.x, we can see that the first feature completely dominated the other variables in terms of feature importance. Visualize this using the code below:

import pandas as pd

import matplotlib.pyplot as plt

df\_imp = pd.DataFrame({'Importance': list(model.feature\_importances\_)}, index=X.columns)

df\_imp\_sorted = df\_imp.sort\_values(by=('Importance'), ascending=True)

df\_imp\_sorted.plot.barh(figsize=(5,5))

plt.title('Relative Feature Importance')

plt.xlabel('Relative Importance')

plt.ylabel('Variable')

plt.legend(loc=4)

plt.show()

See the resultant output below:

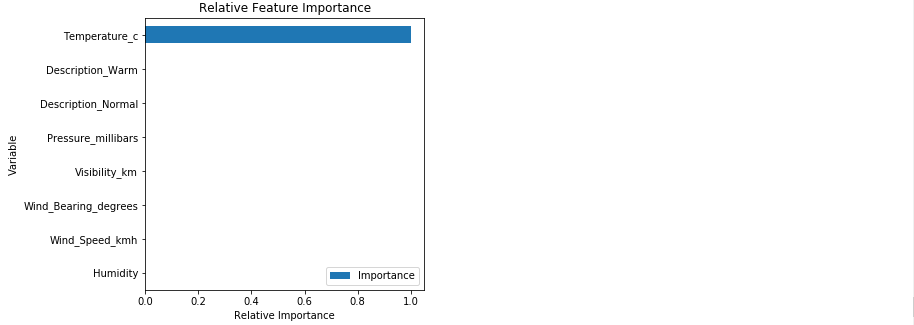


Figure 2.x: Feature importance from tuned decision tree classifier model

It looks like temperature in Celsius was the sole driver in this classification problem. With the outcome measure being rain (‘Rain’=1) or snow (‘Rain’=0) and the way in which decision trees make split decisions via “divide and conquer”, it makes sense that the algorithm used temperature to determine if there was rainfall or snowfall at the time of measurement. In the upcoming activity, we will evaluate how the model performed.

Activity 5: Generating predictions and evaluating performance of decision tree classifier model

*Activity 5: Generating predictions and evaluating performance of decision tree classifier model*

We have generated predictions and evaluated model performance in previous exercises and activities. We will be taking the same approach in this activity to evaluate the performance of our tuned decision tree classifier model.

1. Generate predicted probabilities of rain
2. Generate predicted classes of rain
3. Generate and print a confusion matrix
4. Print a classification report

**Solution:**

1. Generate predicted probabilities of rain using predicted\_prob = model.predict\_proba(X\_test)[:,1]
2. Generate predicted classes of rain using predicted\_class = model.predict(X\_test)
3. Generate and print a confusion matrix with the code below:

from sklearn.metrics import confusion\_matrix

import numpy as np

cm = pd.DataFrame(confusion\_matrix(y\_test, predicted\_class))

cm['Total'] = np.sum(cm, axis=1)

cm = cm.append(np.sum(cm, axis=0), ignore\_index=True)

cm.columns = ['Predicted No', 'Predicted Yes', 'Total']

cm = cm.set\_index([['Actual No', 'Actual Yes', 'Total']])

print(cm)

Refer to the resultant output below:



Figure 2.x: Confusion matrix from tuned decision tree classifier model

1. Print a classification report as follows:

from sklearn.metrics import classification\_report

print(classification\_report(y\_test, predicted\_class))

Refer to the resultant output below:



Figure 2.x: Classification report from tuned decision tree classifier model

By tuning a decision tree classifier model on our ‘weather.csv’ dataset, we were able to predict rain (or snow) almost perfectly. We are able to see that the sole driving feature was temperature in celsius. This makes sense due to the way in which decision trees use recursive partitioning to make predictions.

Sometimes, after evaluation, a single model is a weak learner and does not perform well. However, by combining weak learners, we create a stronger learner. The approach of combining numerous weak learners to create a stronger learner is termed ensemble. Random forest models combine numerous decision tree models to create a stronger, ensemble model. Random forests can be used for classification or regression problems.

Random forests

*Present x: Random forests*

As briefly mentioned above, random forests are ensembles of decision trees that can be used to solve classification or regression problems. Random forests use a small portion of the data to fit each tree, so they can handle very large data sets, and they are less prone to the “curse of dimensionality” relative to other algorithms. Predictions of the random forest are then determined by combining the predictions of each tree. Like SVM, random forests are a black box with inputs and outputs which cannot be interpreted.

In the upcoming exercises and activities, we will scale our features and tune a random forest regressor in a pipeline to predict temperature in Celsius. Then we will evaluate the performance of the model much like we did in Exercise 3 and Activity 1.

Exercise 12: Preparing data for random forest regressor pipeline

*Exercise 12: Preparing data for random forest regressor pipeline*

First, we will prepare the data for the pipeline with ‘Temperature\_c’ as the dependent variable just as we did in Exercise 1

1. Import ‘weather.csv’ and save it as df using the following code:

import pandas as pd

df = pd.read\_csv('weather.csv')

1. Dummy code the categorical variables as follows:

import pandas as pd

df\_dummies = pd.get\_dummies(df, drop\_first=True)

1. Remove any possible ordering effects by shuffling df\_dummies using the code below:

from sklearn.utils import shuffle

df\_shuffled = shuffle(df\_dummies, random\_state=42)

1. Split df\_shuffled into X and y using the following code:

DV = 'Temperature\_c'

X = df\_shuffled.drop(DV, axis=1)

y = df\_shuffled[DV]

1. Split X and y into testing and training data as follows:

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.33, random\_state=42)

Now that we have imported, shuffled, separated our data into features (X) and dependent variable (y), and split X and y into testing and training data we will scale X and tune a random forest regressor model in a pipeline.

**Discussion:** If there is a value in the test data larger than any value seen in the training data, how would a random forest handle that value as opposed to linear regression?

**Answer**: Since linear regression returns a coefficient it will attempt to extrapolate out to the large value. A random forest, on the other hand, would place it in a bin with smaller values.

Activity 6: Scaling features and tuning a random forest regressor in a pipeline

*Activity 6: Scaling features and tuning a random forest regressor in a pipeline*

The data has been prepared for inclusion in a random forest regressor grid search pipeline. Now, we must set up the steps for the pipeline and find the optimal combination of hyperparamters using a grid search.

1. Set up the steps for a pipeline
2. Set up the pipeline
3. Specify the hyperparameter space
4. Instantiate the GridSearchCV model optimizing the explained variance
5. Fit the grid search model to the training set
6. Print the tuned parameters

**Solution:**

1. Set up the steps for a pipeline using the following code:

from sklearn.preprocessing import StandardScaler

from sklearn.ensemble import RandomForestRegressor

steps = [('scaler', StandardScaler()), ('Forest', RandomForestRegressor(n\_estimators=10))]

1. Setup the pipeline using the code below:

from sklearn.pipeline import Pipeline

pipeline = Pipeline(steps)

1. Specify the hyperparameter space as follows:

import numpy as np

parameters = {'Forest\_\_criterion': ['mse','mae'],

'Forest\_\_max\_features': ['auto', 'sqrt', 'log2', None],

'Forest\_\_min\_impurity\_decrease': np.linspace(0.0, 1.0, 10),

'Forest\_\_bootstrap': [True, False],

'Forest\_\_warm\_start': [True, False]}

1. Instantiate the GridSearchCV model optimizing the explained variance using the following code:

from sklearn.model\_selection import GridSearchCV

model = GridSearchCV(pipeline, parameters, scoring='explained\_variance', cv=5)

1. Fit the grid search model to the training set using model.fit(X\_train, y\_train)
2. Print the tuned parameters as follows:

best\_parameters = model.best\_params\_

print(best\_parameters)

See the resultant output below:



Figure 2.x: Tuned hyperparameters from random forest regressor grid search model

After performing a grid search of our random forest regressor hyperparameters, we need to fit a random forest regressor model with the tuned hyperparameters. We will programmatically extract the values in the best\_parameters dictionary and assign them to the corresponding hyperparameter in the random forest regressor function, so we can access the attributes from the random forest regressor function.

Exercise 13: Programmatically extracting tuned hyperparameters and determining feature importance from random forest regressor grid search model

*Exercise 13: Programmatically extracting tuned hyperparameters from random forest regressor grid search model*

By extracting the value from the key-value pairs in the best\_parameters dictionary, we eliminate the possibility of manual errors as well as make our code more automated. In this exercise, we will replicate the steps from Exercise 11, but adapt our code for the random forest regressor model.

1. Instantiate a random forest regressor model with the values for each key from the best\_parameters disctionary assigned to the corresponding hyperparameter.

from sklearn.ensemble import RandomForestRegressor

model = RandomForestRegressor(criterion=best\_parameters['Forest\_\_criterion'],

max\_features=best\_parameters['Forest\_\_max\_features'],

min\_impurity\_decrease=best\_parameters['Forest\_\_min\_impurity\_decrease'],

bootstrap=best\_parameters['Forest\_\_bootstrap'],

warm\_start=best\_parameters['Forest\_\_warm\_start'])

1. Fit the model on the training data using model.fit(X\_train, y\_train)

Find the resultant output below:

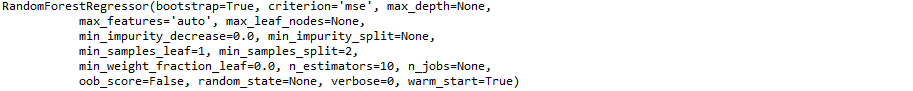


Figure 2.x: Output from fitting the random forest regressor model with tuned hyperparameters

1. Plot feature importance in descending order using the following code:

import pandas as pd

import matplotlib.pyplot as plt

df\_imp = pd.DataFrame({'Importance': list(model.feature\_importances\_)}, index=X.columns)

df\_imp\_sorted = df\_imp.sort\_values(by=('Importance'), ascending=True)

df\_imp\_sorted.plot.barh(figsize=(5,5))

plt.title('Relative Feature Importance')

plt.xlabel('Relative Importance')

plt.ylabel('Variable')

plt.legend(loc=4)

plt.show()

See the resultant output below:

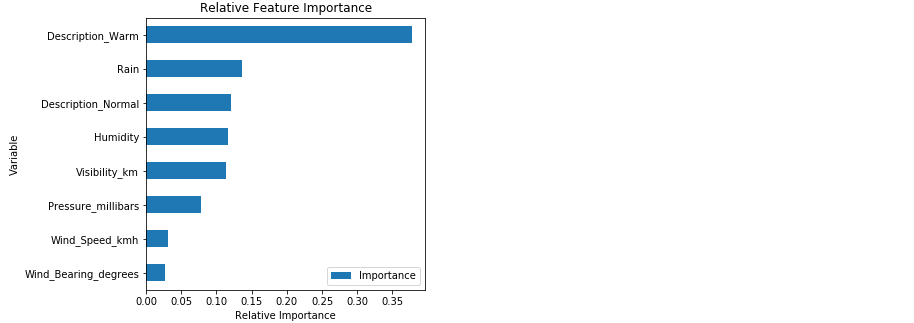


Figure 2.x: Feature importance from random forest regressor model with tuned hyperparameters

From Figure 2.x, we can see that the dummy variable ‘Description\_Warm’ is the main driver of temperature in Celsius. This makes sense as someone describing the day as warm is referring to the perceived temperature. Additionally, the variable ‘Rain’ pertains to temperature because the ‘Rain’ feature determines only if it was raining (‘Rain’ = 1) or snowing (‘Rain’ = 0); both of which are closely related to temperature. Meanwhile, ‘Wind\_Bearing\_Degrees’ and ‘Wind\_Speed\_km’ have a small effect on the temperature. Let’s now check to see how our model performs on the test data.

Activity 7: Generating predictions and evaluating performance of tuned random forest regressor model

*Present x: Generating predictions and evaluating performance of tuned random forest regressor model*

In Exercise 3 and Activity 1, we learned to generate predictions and evaluate the performance of regression models that predict a continuous outcome. In this activity, we will be taking the same approach to evaluate the performance of our random forest regressor model to predict temperature in Celsius.

Continuing from Exercise 13:

1. Generate predictions on the test data
2. Plot correlation of predicted and actual values
3. Plot distribution of residuals
4. Compute metrics, place them in a dataframe, and print it

**Solution:**

1. Generate predictions on the test data using predictions = model.predict(X\_test)
2. Plot correlation of predicted and actual values using the following code:

import matplotlib.pyplot as plt

from scipy.stats import pearsonr

plt.scatter(y\_test, predictions)

plt.xlabel('Y Test (True Values)')

plt.ylabel('Predicted Values')

plt.title('Predicted vs. Actual Values (r = {0:0.2f})'.format(pearsonr(y\_test, predictions)[0], 2))

plt.show()

Refer to the resultant output below:

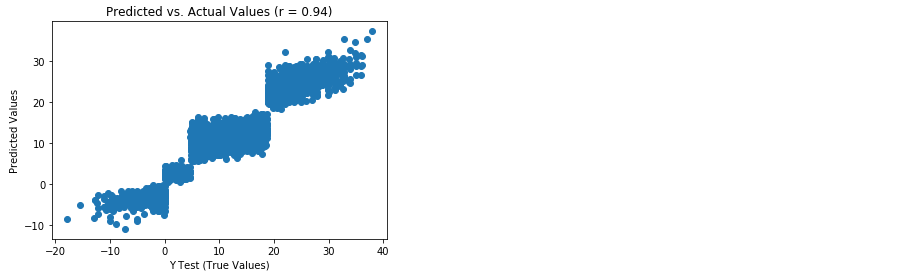


Figure 2.x: Scatterplot of predicted and actual values from random forest regression model with tuned hyperparameters

1. Plot distribution of residuals as follows:

import seaborn as sns

from scipy.stats import shapiro

sns.distplot((y\_test - predictions), bins = 50)

plt.xlabel('Residuals')

plt.ylabel('Density')

plt.title('Histogram of Residuals (Shapiro W p-value = {0:0.3f})'.format(shapiro(y\_test - predictions)[1]))

plt.show()

Refer to the resultant output below:

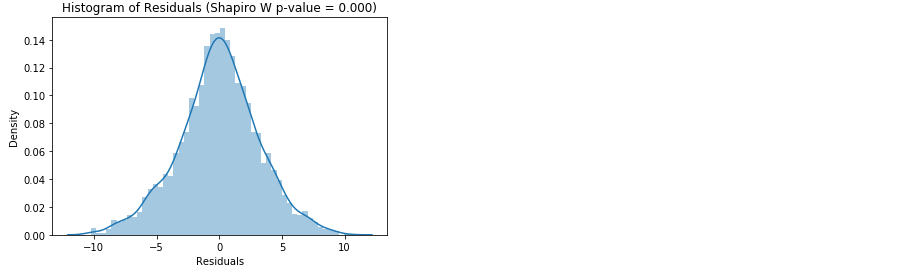


Figure 2.x: Histogram of residuals from random forest regression model with tuned hyperparameters

1. Compute metrics, place them in a dataframe, and print it using the code below:

from sklearn import metrics

import numpy as np

metrics\_df = pd.DataFrame({'Metric': ['MAE',

'MSE',

'RMSE',

'R-Squared'],

'Value': [metrics.mean\_absolute\_error(y\_test, predictions),

metrics.mean\_squared\_error(y\_test, predictions),

np.sqrt(metrics.mean\_squared\_error(y\_test, predictions)),

metrics.explained\_variance\_score(y\_test, predictions)]}).round(3)

print(metrics\_df)

Find the resultant output below:



Figure 2.x: Model evaluation metrics from random forest regression model with tuned hyperparameters

The random forest regressor model seems to outperform the multiple linear regression as evidenced by lower MAE, MSE, and RMSE as well as greater explained variance. Additionally, there was a stronger correlation between the predicted and actual values and the residuals were almost normally distributed. Thus, by leveraging ensemble methods using a random forest regressor, we constructed a model that explains 88.9% of the variance in temperature and predicts temperature in Celsius +/- 2.476 degrees.

**Discussion:** If a model generated perfect predictions what would the scatterplot of the predicted vs. actual values look like?

**Answer**: The scatterplot of predicted vs. actual values of a perfect model would be a straight, diagonal line.

Summary

*Present x: Summary*

In this chapter we were introduced to the open-source machine learning library for Python, scikit-learn. We learned to preprocess data and fit a few different regression and classification algorithms. Furthermore, we learned to scale the features and find the optimal combination of hyperparameters in a pipeline. Lastly, we learned how to quickly and effectively evaluate the performance of classification and regression models. This was a very comprehensive introduction to the scikit-learn library and the strategies employed here can be applied to building numerous additional algorithms provided by scikit-learn.

Assessment Questions:

1. True/False: Scikit-learn is a free, open-source library built for Python that contains an assortment of machine learning algorithms, none of which are unsupervised.
   1. True
   2. False
2. True/False: Scikit-learn contains a vast number of functions to make building machine learning models very user friendly.
   1. True
   2. False
3. Fill in the blanks: In addition to machine learning algorithms, scikit-learn contains functions for data \_\_\_\_\_\_\_\_\_\_\_\_ and model \_\_\_\_\_\_\_\_\_\_.
   1. exploration, deployment
   2. preprocessing, evaluation
   3. splitting, learning
   4. None of the above
4. Fill in the blanks: Linear regression is used for \_\_\_\_\_\_\_\_\_\_ and logistic regression is used for \_\_\_\_\_\_\_\_\_\_\_\_\_\_.
   1. classification, regression
   2. regression, classification
   3. clustering, classification
   4. regression, clustering
5. True/False: The Kernel trick is used when data is not linearly separable.
   1. True
   2. False
6. Fill in the blanks: In classification models we want to \_\_\_\_\_ the number of false positives and \_\_\_\_\_\_\_\_\_\_ the number of false negatives.
   1. minimize, minimize
   2. maximize, minimize
   3. minimize, maximize
   4. multiply, square root
7. In evaluating regression models we can:
   1. examine the relationship between predicted and actual values
   2. plot the distribution of residuals
   3. compute metrics such as MAE, MSE, RMSE, and R-squared
   4. All of the above
8. True/False: The R-squared value tells us how much variance in the dependent variable is explained by the feature(s) in the model.
   1. True
   2. False
9. When deciding upon which model to use we must consider model \_\_\_\_\_\_\_\_\_\_\_\_.
   1. name
   2. contributors
   3. interpretability
   4. None of the above
10. True/False: Once the hyperparameters have been determined using a grid search we must manually copy and paste the values to the corresponding hyperparameters and fit a new model to access the attributes of that model.
    1. True
    2. False