4

Dimensionality Reduction and Unsupervised Learning

*Present 1-2: 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:

* Describe the differences between *supervised* and *unsupervised* learning
* Utilize loops for repetitive tasks
* Compare and contrast *Hierarchical Cluster Analysis (HCA)* and *k-Means clustering*
* Conduct a *HCA* and interpret the output
* Tune the number of clusters for *k-Means* clustering
* Select the optimal number of principal components for dimension reduction
* Generate predictions using *Linear Discriminant Function Analysis (LDA)* as a supervised classifier
* Tune hyperparameters using *GridSearchCV*
* Evaluate classifier model performance

Introduction

*Present 3: Brief summary of supervised and unsupervised learning*

In *supervised learning*, clear instruction is provided to *predictive models* concerning the outcome for which to learn. Supervised learning tasks are typically categorized as either *regression* or *classification*. Regression tasks consist of those problems where the dependent variable is *continuous*. Examples of regression tasks include predicting the temperature, rainfall, or weight. Classification tasks, on the other hand, involve those problems where the dependent variable is *categorical*. Examples of classification tasks include predicting if a team will win or if there will be snowfall. Unlike supervised learning models, unsupervised models do not have a dependent variable for which to predict.

Thus, in *unsupervised learning*, *descriptive models* are used for exploratory analysis to uncover patterns in unlabeled data. Examples of unsupervised learning tasks include algorithms for *clustering* and those for *dimension reduction*. In clustering, observations are assigned to groups in which there is high within-group homogeneity and between group heterogeneity. Simply put, observations are put into clusters of samples with other observations that are very similar. Use cases for clustering algorithms are vast. For example, analysts seeking to elevate sales by targeting select customers for marketing ads or promotions will separate customers by their shopping behavior. Additionally, hierarchical clustering has been implemented in academic neuroscience and motor behavior research (<https://www.researchgate.net/profile/Ming-Yang_Cheng/project/The-Effect-of-SMR-Neurofeedback-Training-on-Mental-Representation-and-Golf-Putting-Performance/attachment/57c8419808aeef0362ac36a5/AS:401522300080128@1472741784217/download/Schack+-+2012+-+Measuring+mental+representations.pdf?context=ProjectUpdatesLog>) and k-Means clustering has been used in fraud detection (https://www.semanticscholar.org/paper/Fraud-Detection-in-Credit-Card-by-Clustering-Tech/3e98a9ac78b5b89944720c2b428ebf3e46d9950f).

However, when building predictive or descriptive models, it can be a challenge to determine which features to include in the model to improve the model and which features to exclude because they diminish the model. Too many features can be troublesome because the greater the number of variables in a model the higher the probability of *multicollinearity* and subsequent *overfitting* of the model. Additionally, numerous features expand the complexity of the model and increase the time for model tuning and fitting. This becomes troublesome and aggravating with larger data sets. Fortunately, another use case for unsupervised learning is to reduce the number of features in a data set by creating combinations of the original features. Reducing the number of features in the data helps eliminate multicollinearity and converges on a combination of features to best produce a model that performs well on unseen, test, data.

***Note***

Multicollinearity is a situation in which at least two variables are strongly correlated. Multicollinearity is a problem in linear regression models because it does not allow the isolation of the relationship between each independent variable and the outcome measure. Thus, coefficients and p-values become unstable and less precise.

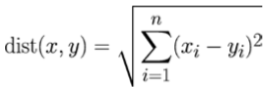
In this lesson we will be covering two widely used unsupervised clustering algorithms: *Hierarchical Cluster Analysis (HCA)* and *k-Means clustering*. Additionally, we will explore dimension reduction using *Principal Component Analysis (PCA)* and observe how reducing dimensionality can improve model performance. Lastly, we will implement *Linear Discriminant Function (LDA)* Analysis, an algorithm that can be used for data reduction and classification, as a supervised classifier. We will tune the LDA model using a grid search and interpret the output.

Hierarchical Cluster Analysis (HCA)

*Present 4-5: Introduction to HCA and interpreting dendrograms*

Hierarchical cluster analysis (HCA) is best implemented when the user does not have an *a priori* number of clusters for which to build. HCA works measuring the *Euclidean distance* between and among all observations in the data and creates links at the distance in which the two points lie.

With the number of features indicated by *n*, Euclidean distance is calculated using the formula:



Equation 4.1. Euclidean Distance

For example, if we compare a subject with 2 cars and 1 boat to a subject with 3 cars and 0 boats, we would determine the distance in Python by, first, importing the math library to access the sqrt function.

***Note***

In Python, \*\*2 is used to square a number.

By wrapping the two squared numbers in the math.sqrt function we are able to get the square root of the sum of those values. To print values to the console, call the print function. To format the number along with the text, place {} in the area the value is supposed to appear. In the example below, the double brackets contain 0:0.2f. The zero prior to the colon references the index of the number to be formatted and the 0.2f rounds the formatted value to 2 decimal places.

import math

euclidean\_distance = math.sqrt = ((2 - 3)\*\*2 + (1 - 0)\*\*2)

print('The Euclidean distance between these points is {0:0.2f}'.format(euclidean\_distance))



After the distance between observations and groups of observations have been calculated, the relationships between and among all observations are displayed using a dendrogram. Dendrograms are tree-like structures displaying horizontal lines at the distance between links. Dr. Thomas Schack (<https://www.researchgate.net/profile/Ming-Yang_Cheng/project/The-Effect-of-SMR-Neurofeedback-Training-on-Mental-Representation-and-Golf-Putting-Performance/attachment/57c8419808aeef0362ac36a5/AS:401522300080128@1472741784217/download/Schack+-+2012+-+Measuring+mental+representations.pdf?context=ProjectUpdatesLog>) relates this structure to the human brain in which each observation is a node and the links between observations are the neurons. This creates a hierarchical structure in which those items that are more closely related are “chunked” together into clusters. An example dendrogram from the scipy.cluster.hierarchy.dendrogram(https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.cluster.hierarchy.dendrogram.html) documentation site is displayed below.

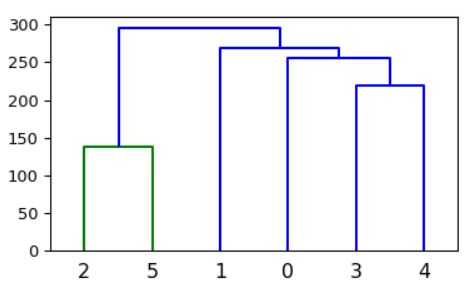


Figure 4.1. Example dendrogram

The y-axis indicates the Euclidean distance while the x-axis indicates the row index for each observation. Horizontal lines denote links between observations with links closer to the x-axis indicating closer distance and subsequent relationship. There appear to be two clusters in this example. The first cluster includes indices 2 and 5 while the second cluster includes indices 1, 0, 3, and 4.

*Discuss situations in which HCA may be of use*

Exercise 1: Segmenting Flower Species using HCA

To demonstrate HCA, we will be using an adapted version of the *glass* dataset from the University of California – Irvine (<http://archive.ics.uci.edu/ml/datasets/Glass+Identification>). This data contains 9 features corresponding to different glass type:

* RI: refractive index
* Na: weight percent in Sodium
* Mg: weight percent in Magnesium
* Al: weight percent in Aluminum
* Si: weight percent in Silicon
* K: weight percent in Potassium
* Ca: weight percent in Calcium
* Ba: weight percent in Barium
* Fe: weight percent in Iron

It is a small data set containing 218 rows and 9 columns. In this exercise, we will be using RI and the various weight percent in each oxide to segment glass types. Until the last section of the chapter where we use LDA as a supervised classifier, we will not include the dependent variable, glass type, in the data. Thus, the glass data set will predominantly be used to demonstrate clustering and data reduction techniques.

1. To get started, we must first import pandas and alias it as pd using import pandas as pd. Then, we will read glass.csv and save it to the data frame object df.

import pandas as pd

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

1. To explore the dimensions of the data and print the information to the console access the number of rows and columns by calling df.shape. This returns a tuple with rows as the first index and columns as the second index. To access the rows use df.shape[0]. To access the columns use df.shape[1]. The number of rows and columns will be saved as the objects n\_rows and n\_columns, respectively. To print these values to the console by concatenating strings and values we will be using {} in both places where we want to insert the values.

***Note***

Since we are not formatting these values, we will not need to specify a series of digits followed by f. Additionally, the values will appear in the brackets in the order in which they are provided in the parenthesis following the format function. To clarify, since n\_rows is listed first in the parenthesis following the format function it will appear in place of the first {}. Since n\_columns appears second in the parenthesis following the format function, it will appear in place of the second {}.

n\_rows = df.shape[0]

n\_columns = df.shape[1]

print('There are {} rows and {} columns in the Glass data set'.format(n\_rows, n\_columns))



1. We can also look for some basic data frame information by printing df.info() to the console.

print(df.info())



From the output, we can see that there are 218 entries (I.e., rows) and 9 columns. We can also see that there are no missing values (I.e., all are non-null) and that the data type for each column is float64. Float numbers are named accordingly because they are floating-point numbers, meaning they are a number that contains a decimal point.

1. It is good practice to remove any possible order effects in the data by shuffling the rows prior to building any models. To shuffle the data, we will import the shuffle function from sklearn.utils using from sklearn.utils import shuffle. The shuffle function takes arrays, lists, data frames or scipy sparse matrices as the first argument, random\_state as another argument, and returns a shuffled copy of the first argument, df\_shuffled.

***Note***

random state is used for initializing the internal random number generator. If it is not specified, then the data will be shuffled differently each time. For reproducibility, it is suggested to set a random\_state. Random state is included as an argument in many Python functions. To keep everything as consistent and reproducible as possible we will always set it to 42.

from sklearn.utils import shuffle

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

In machine learning models, larger numbers carry more weight than smaller ones which leads to erroneous conclusions. To ensure features measured at different scales contribute equally to the analysis, models perform much better when they are standardized. In lesson 3, we learned about preprocessing steps. In this lesson we will be transforming our features into z-scores using Scikit learn’s StandardScaler() function.

1. Transform each observation into a z-score using sklearn.preprocessing.StandardScaler. First, import StandardScaler using from sklearn.preprocessing import StandardScaler. Next, instantiate a StandardScaler() and save it as the object scaler. Fit the scaler object to df\_shuffled using scaler.fit(df\_shuffled). Lastly, transform df\_shuffled to z-scores and save it as the object scaled\_features using scaled\_features = scaler.transform(df\_shuffled).

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler() # instantiate scaler object

scaler.fit(df\_shuffled) # fit scaler to the features

scaled\_features = scaler.transform(df\_shuffled) # transform features to scaled version

1. Build the HCA model by importing linkage using from scipy.cluster.hierarchy import linkage. Build our linkage model and save it as the model object using model = linkage(scaled\_features, method=’complete’). By default, linkage uses Euclidean distance as its metric argument, so we do not need to specify it.

from scipy.cluster.hierarchy import linkage

model = linkage(scaled\_features, method='complete')

***Note***

Scikit-Learn also has a function for HCA (*https://scikit-learn.org/stable/modules/generated/sklearn.cluster.AgglomerativeClustering.html#sklearn.cluster.AgglomerativeClustering)*, but it is easier to visualize the dendrogram by using the *scipy* library.

1. To display the dendrogram we will plot the linkage model using the dendrogram function. To plot the linkage model, import matplotlib using import matplotlib.pyplot as plt. Next, import the dendrogram function using from scipy.cluster.hierarchy import dendrogram. Edit the size of our plot using plt.figure(figsize=(10,5)). To title our plot “Dendrogram for Glass Data”, use plt.title(‘Dendrogram’). To plot the dendrogram, rotate the x tick labels 90 degrees, and make the font size 6, use dendrogram(model, leaf\_rotation=90, leaf\_font\_size=6). Finally, to display the plot use plt.show().

import matplotlib.pyplot as plt

from scipy.cluster.hierarchy import dendrogram

plt.figure(figsize=(10,5))

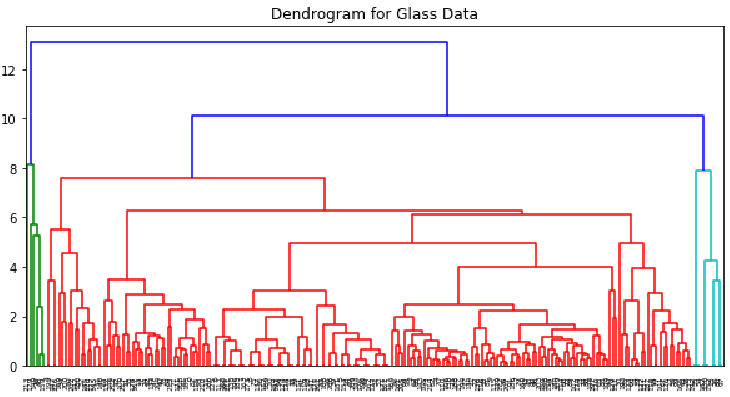
plt.title('Dendrogram for Glass Data')

dendrogram(model,

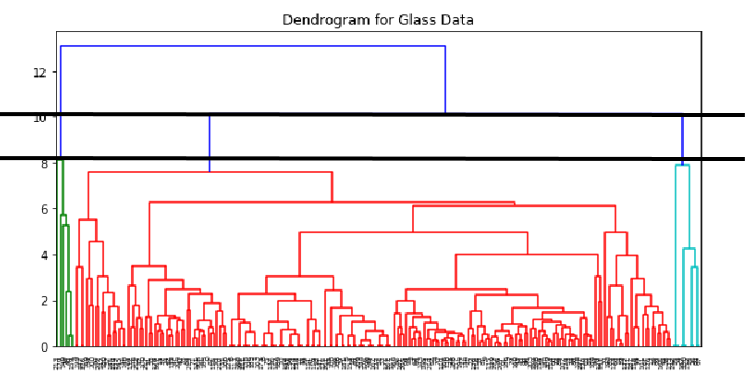
leaf\_rotation=90,

leaf\_font\_size=6)

plt.show()



On the x-axis is the index for each observation, or row, in the data set. On the y-axis is Euclidean distance. Horizontal lines are links between and among observations. By default, scipy will color code the different clusters that it finds. However, a manual method for determining the optimal number of clusters is to find the maximal (vertical) distance between links and count the vertical lines in that area. In this case, it would be three as the maximal distance between links contains 3 vertical lines.



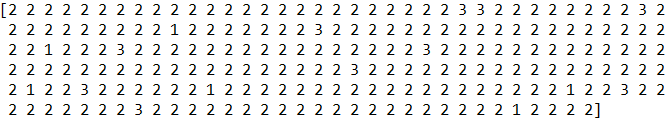
Now that we have the predicted clusters of observations, we can use the fcluster function from scipy.cluster.hierarchylibrary to generate an array of labels that correspond to the rows in df\_shuffled and scaled\_features.

1. To generate the predicted labels of the cluster for which an observation belongs, use the fcluster function from scipy.cluster.hierarchy. To access the fcluster function, use from scipy.cluster.hierarchy import fcluster. Using the fcluster function, use the linkage model as the first argument, 9 for the threshold to apply when forming flat clusters, and ‘distance’ for the criterion argument so that each cluster can have no greater distance than 9, and save them as labels.

from scipy.cluster.hierarchy import fcluster

labels = fcluster(model, t=9, criterion='distance')

print(labels)



***Teaching Tip:***

Experiment with the threshold, t, argument in the fcluster function and see how that affects the labels array.

1. Now that the labels have been generated for each observation, we can add them as a column in either the df\_shuffled or scaled\_features data frames.

***Note***

Remember, we cannot add labels to any data frames generated before being shuffled because labels were generated on the data after being shuffled. Thus, the order of the values of labels do not match with the order of the observations in the data prior to being shuffled.

To add labels as a column in df\_shuffled, assign labels to df\_shuffled[‘Predicted\_Cluster’] using df\_shuffled[‘Predicted\_Cluster’] = labels. This code creates a new column called ‘Predicted\_Cluster’ to df\_shuffled and assigns that column the values of labels.

df\_shuffled['Predicted\_Cluster'] = labels

We have successfully learned the differences between supervised and unsupervised learning, how to format numbers in text, how to import data from sklearn.datasets, save the features as a pandas data frame, data preprocessing (I.e., shuffling and scaling) steps, how to build an HCA model using scipy.cluster.hierarchy.linkage, how to visualize the HCA dendrogram using scipy.cluster.hierarchy.dendrogram and interpret it, and how to assign predicted cluster labels to observations.

Here we have utilized HCA to cluster our data into three groups and matched the observations with their predicted cluster. While HCA models are easy to build, there’s no need to specify the number of clusters in advance, and the visualizations make sense and are easy to interpret it has its drawbacks as well. For example, there is some vagueness in terms of the termination criteria (I.e., when to finalize the number of clusters). Additionally, the algorithm cannot adjust once the clustering decisions have been made. Lastly, it can be very computationally expensive to build HCA models on large data sets with many features.

Next, we will be introducing you to another clustering algorithm, k-Means clustering. This algorithm addresses some of the shortcomings of HCA by having the ability to adjust when the clusters have been initially generated and being less computationally expensive than HCA.

K-Means Clustering

*Present 8-12: Brief explanation of how k-Means works*

K-Means also uses distance to cluster observations. However, rather than linking observations to each other as in HCA, k-Means assigns *n* observations (I.e., number of rows) to *k* (number of) clusters. To determine which observations belong to which clusters, observations are assigned to the cluster in which it’s Euclidean distance is closest to the cluster center. Like the starting weights in artificial neural networks, cluster centers are generated at random which means that the randomness may have a large effect as to the clusters in which observations are assigned. After the cluster centers have been randomly generated there are two phases: 1) assignment phase and 2) updating phase.

***Note***

The cluster centers being randomly generated is an important point to remember and we will be visiting it later in the chapter. Some refer to this random generation of cluster centers as a weakness of the algorithm, but we can turn it into a strength by leveraging the power of loops.

In the assignment phase, observations are assigned to the cluster from which it has the smallest Euclidean distance.

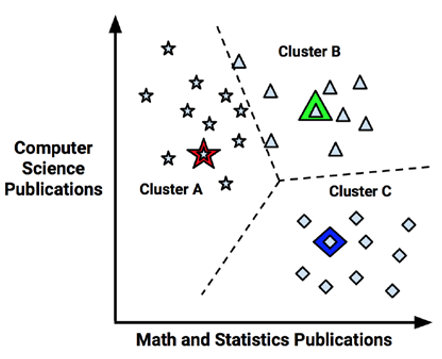


Figure 4.2. A scatterplot of observations and the cluster centers as denoted by the star, triangle, and diamond.

Next, in the updating phase, cluster centers are shifted to the mean position of the points in that cluster. These cluster means are known as the centroids.

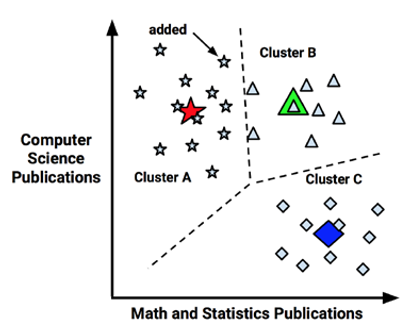


Figure 4.3. Shifting of the cluster centers to the cluster centroid.

However, once the centroids have been calculated, some of the observations are reassigned to a different cluster due to being closer to the new centroid than the previous cluster center. Thus, the model must update its centroids once again.

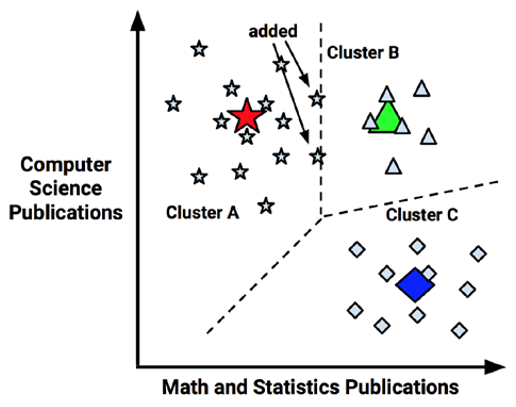


Figure 4.4. Updating of the centroids after observation reassignment.

This process of updating centroids continues until there are no further observation reassignments.

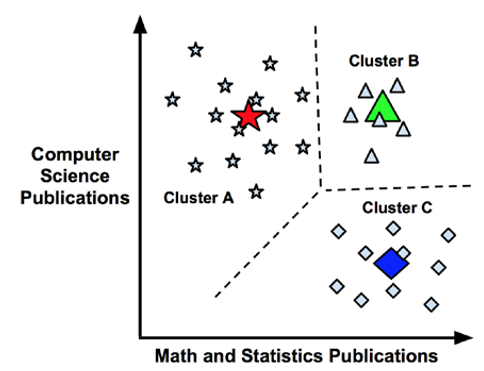


Figure 4.5. Final centroid position and cluster assignments.

Using the same glass data set from Exercise 1, we will fit a k-Means model with a user-defined number of clusters. Next, because of the randomness in which group centroids are chosen, we will increase the confidence in our predictions by building an ensemble of k-Means models with a given number of clusters and assigning each observation to the mode of the predicted clusters. After that, we will tune the optimal number of clusters by monitoring the inertia, or within-cluster homogeneity, by number of clusters and finding the point at which there are diminishing returns in inertia by adding more clusters. Lastly, we will further improve the confidence of our clusters by building an ensemble of k-Means models at each number of clusters and, again, finding the point of diminishing returns.

Exercise 2: Segmenting Iris into 2 Clusters using k-Means

After the glass data set has been imported, shuffled, and standardized (see Exercise 1):

1. Instantiate a KMeans model by, first, importing the KMeans function from sklearn.cluster using from sklearn.cluster import KMeans with, in this case, 2 clusters. We assign the number of clusters a priori by assigning a number to the KMeans argument n\_clusters and assigning it to the object model.

from sklearn.cluster import KMeans

model = KMeans(n\_clusters=2)

1. Fit the model to scaled\_features using model.fit(scaled\_features).

model.fit(scaled\_features)

After the model has been fit, we can see the hyperparameters that the model is using. The algorithm completes the assignment phase and updating phase for us.

***Note***

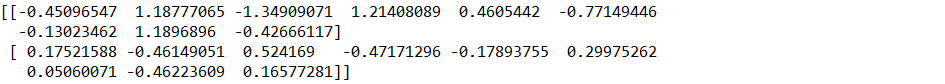
We will be discussing hyperparameters in greater detail later in the chapter when we discuss tuning models using a grid search.



1. Return the coordinates of the group centroids and save them to the object centroids using centroids = model.cluster\_centers\_. Since we told the algorithms that we wanted 2 clusters we see that there are two rows in the centroids array. Additionally, there are 9 features in the data, so we return 9 columns in the centroids array.

centroids = model.cluster\_centers\_

print(centroids)



1. To return the inertia for our model with 2 clusters we use model.inertia\_ and save the value to the object inertia. We also print the formatted inertia value to the console along with text using {0:0.2f} in the area where we want to place our value.

inertia = model.inertia\_

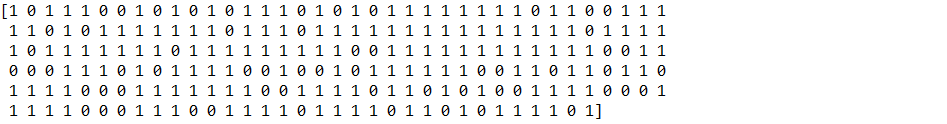
print('The within-group sum of squares (i.e., inertia) with 2 clusters is {0:0.2f}'.format(inertia))



1. To access the cluster labels from our model we use model.labels\_ and assign the value to the labels object. Since we chose 2 clusters, there are 2 possible labels; 0 and 1. If we chose to have 3 clusters, we would have 3 possible labels; 0, 1, and 2.

labels = model.labels\_

print(labels)



1. It is a good idea to generate a frequency table of the labels using the pandas function value\_counts. To examine the frequency of each cluster, import pandas as pd and then create the frequency table using pd.value\_counts(labels). Using 2 clusters, 61 observations were placed into the first cluster and 157 observations were grouped into the second cluster.

import pandas as pd

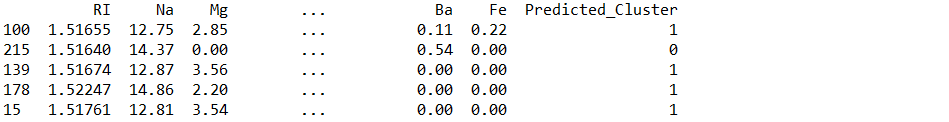
pd.value\_counts(labels)



1. Now that we have the labels generated for each observation, we can add them as a column in either the df\_shuffled or scaled\_features data frames. To add labels as a column in df\_shuffled, we will assign labels to df\_shuffled[‘Predicted\_Cluster’] using df\_shuffled[‘Predicted\_Cluster’] = labels. This code creates a new column called ‘Predicted\_Cluster’ to df\_shuffled and assigns that column the values of labels. We will then use df\_shuffled.head(5) to preview the first 5 rows of df\_shuffled after the ‘Predicted\_Cluster’ column has been created.

df\_shuffled['Predicted\_Cluster'] = labels

print(df\_shuffled.head(5))



*Discuss: How confident are we in these predictions? Why might we lack confidence in these predictions? What can we do to increase the confidence we have in our predictions?*

However, it is important to remember that the initial group centers were chosen at random, so the next time we run the k-Means analysis, even on the exact same data, we will get different results. Thus, building an ensemble of models (I.e., numerous models), and taking an average or mode of those models may be used to generate more stable predictions.

***Teaching Tip***

If students are unfamiliar with loops now is a good time to teach how loops work. In particular, it is important that students are familiar with appending items to lists and data frames using loops.

Activity 1: K-Means Clustering (ensemble)

When algorithms use randomness as part of their method for finding the optimal solution (I.e., in Artificial Neural Networks and k-Means clustering), running back-to-back models may result in very different conclusions. Thus, it is advised to run these types of models many times and generate predictions using a summary measure across all models (I.e., mean, median, mode). In this activity, we will build an ensemble of 100 k-Means clustering models and keep the label for each observation that occurs most frequently (I.e., the mode).

After the glass data set has been imported, shuffled, and standardized (see Exercise 1):

1. Instantiate an empty data frame for which to append the labels for each model and save it as the new data frame object labels\_df.
2. Using a for loop, iterate through 100 models, appending the predicted labels to labels\_df to a new column each time named after the iteration it was generated. This will result in labels\_df containing 100 columns at the end of the loop.
3. Calculate the mode for each row and assign it to the column ‘row\_mode’ in labels\_df.
4. Print a frequency table of labels\_df[‘row\_mode’].

**Solution:**

After the iris data set has been imported, shuffled, and standardized (see Exercise 1):

1. Instantiate an empty data frame for which to append the labels for each model and save it as the new data frame object labels\_df using import pandas as pd to access the pandas library and labels\_df = pd.DataFrame() to create the empty data frame.

import pandas as pd

labels\_df = pd.DataFrame()

1. First, import the KMeans function using from sklearn.cluster import KMeans.

from sklearn.cluster import KMeans

Then, we decide a range through which to iterate. We want to build 100 models, so we could use for i in range(100). This range of values will return numbers 0 through 99. However, after each iteration we want to add a column to labels\_df named after the iteration. For example, for the first iteration we want a column added to labels\_df named Model\_1\_Labels. So, to loop through the values 1 through 100, we must use for i in range(1, 101):.

for i in range(1, 101):

Save a KMeans model object with 2 clusters (decided upon *a priori*) using model = KMeans(n\_clusters=2).

model = KMeans(n\_clusters=2)

Fit the model to scaled\_features using model.fit(scaled\_features).

model.fit(scaled\_features)

Generate the labels and save them to the labels object using labels = model.labels\_.

labels = model.labels\_

Store labels as a column in labels\_df named after the iteration using labels\_df[‘Model\_{}\_Labels’.format(i)] = labels. After each iteration, i will be replaced with the iteration 1 through 100 resulting in 100 columns in the previously empty labels\_df.

labels\_df['Model\_{}\_Labels'.format(i)] = labels

Altogether, this looks like:

from sklearn.cluster import KMeans

for i in range(1, 101):

model = KMeans(n\_clusters=2)

model.fit(scaled\_features) # fit model

labels = model.labels\_ # get predicted labels

labels\_df['Model\_{}\_Labels'.format(i)] = labels # put the labels into the empty df

print('There are {} rows and {} columns in the labels\_df data frame'.format(labels\_df.shape[0], labels\_df.shape[1]))



1. To calculate the mode for each row and assign it to a new column in labels\_df entitled row\_mode, use labels\_df[‘row\_mode’] = labels\_df.mode(axis=1). The argument axis=1 computes across all columns. The default, axis=0 computes down all rows.

labels\_df['row\_mode'] = labels\_df.mode(axis=1)

1. If we wish to generate a frequency table of the row\_mode column inside of labels\_df, we will use the pandas function value\_counts. To examine the frequency of each cluster, import pandas as pd and then create the frequency table using pd.value\_counts(labels\_df[‘row\_mode’). In this case, the frequency of labels after running 100 models is the same as the frequency of labels after running it once (see Exercise 2). However, we can be more confident in our predictions after running our models through hundreds or even thousands of iterations. Additionally, iterating through an ensemble of models and calculating a mean and standard deviation across models allows us to report the variability and confidence levels of our predictions.

import pandas as pd

pd.value\_counts(labels\_df['row\_mode'])



*Discuss: How confident are we in the predictions now that we have computed them from 100 models as opposed to 1? How can we further increase our confidence that we have appropriately segmented this sample?*

If we know beforehand that we want to separate these observations into 2 clusters, then our work here is done.

However, in unsupervised learning, it is probable that we do not know how many clusters for which to choose. Thus, to find the best value of n\_clusters, we must decide over a range of n\_clusters.

Exercise 3: k-Means clustering: Tuning n\_clusters

To tune the number of clusters, we must think of an outcome measure that determines how well a k-Means model is performing. In k-Means, model performance is interpreted in terms of sum of squared distances of samples to their closest cluster centroid, or *inertia*. Thus, to determine which number of clusters produces the best model we need to run a different model for each number of n\_clusters in a given range and see which value of n\_clusters results in the smallest inertia, right? No, we want the smallest number of n\_clusterswhile still minimizing the value for inertia. Therefore, we determine the value for n\_clusters by running a separate model for each value of n\_clusters and plotting inertia. The number of clusters where the decrease in inertia begins to level-off (I.e., there are diminishing returns with more n\_clusters) will be considered the best choice for n\_clusters.

To demonstrate tuning the number of n\_clusters, first, import, shuffle, and standardize the glass data (see Exercise 1).

1. Import the KMeans function using from sklearn.cluster import KMeans.

from sklearn.cluster import KMeans

Instantiate an empty list for which the inertia values will be stored after every iteration entitled inertia\_list.

inertia\_list = []

Next, instantiate a for loop to iterate through the each integer in the range 1-10 using for i in range(1, 11):.

for i in range(1, 11):

Build a KMeans model with the argument n\_clusters equaling the value for i using model = KMeans(n\_clusters=i).

model = KMeans(n\_clusters=i)

Fit the model to scaled\_features using model.fit(scaled\_features).

model.fit(scaled\_features)

Return the inertia and store the value in the object inertia using inertia = model.inertia\_.

inertia = model.inertia\_

Append inertia it to inertia\_list using inertia\_list.append(inertia).

inertia\_list.append(inertia)

Altogether, this looks like:

from sklearn.cluster import KMeans

inertia\_list = []

for i in range(1, 11):

model = KMeans(n\_clusters=i) # instantiate model

model.fit(scaled\_features) # fit model

inertia = model.inertia\_ # get inertia

inertia\_list.append(inertia) # append inertia to inertia\_list

print(inertia\_list)



1. Now that we have a list of inertia values for each value of n\_clusters, we can plot inertia by number of clusters by, first, importing matplotlib using import matplotlib.pyplot as plt.

import matplotlib.pyplot as plt

Create a list of numbers and save it as the object x so we can plot it on the x-axis using x = list(range(1,11)).

x = list(range(1,11))

Save inertia\_list as the object y using y = inertia\_list.

y = inertia\_list

Plot inertia by number of clusters using plt.plot(x, y).

plt.plot(x, y)

Set the plot title to read ‘Inertia by n\_cluster’ using plt.title(‘Inertia by n\_clusters’).

plt.title('Inertia by n\_clusters')

Label the x-axis ‘n\_clusters’ using plt.xlabel(‘n\_clusters’).

plt.xlabel('n\_clusters')

Set the tick labels on the x-axis as the values in x use plt.xticks(x).

plt.xticks(x)

Set the y-axis label to ‘Inertia’ using plt.ylabel(‘Inertia’).

plt.ylabel('Inertia')

Display the plot use plt.show().

plt.show()

Altogether, this looks like:

import matplotlib.pyplot as plt

x = list(range(1,11))

y = inertia\_list

plt.plot(x, y)

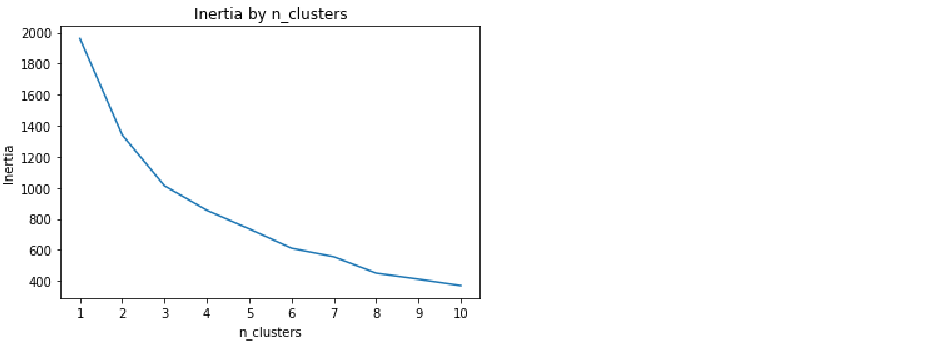
plt.title('Inertia by n\_clusters')

plt.xlabel('n\_clusters')

plt.xticks(x)

plt.ylabel('Inertia')

plt.show()



The distance (I.e., inertia) between observations and cluster centroids will always decrease as n\_clusters increase. However, we want the smallest number of clusters possible with the smallest inertia possible. Therefore, we will use the “elbow” method for determining n\_clusters. The elbow method involves finding the point where there are diminishing returns from selecting more clusters. For example, using the plot above, when we increase n\_clusters from 1 to 2 we see a significant drop in inertia. When we increase n\_clusters from 2 to 3 there is still a pronounced decrease in inertia. However, after 3 clusters the decreases in inertia are less pronounced and the decrease in inertia after 3 clusters is not worth the added complexity from adding more clusters. Thus, the number of clusters we would choose in this situation is 3.

*Discuss: How confident are we that n\_clusters = 3 is the best number of n\_clusters for this sample? Why might we lack confidence in this number of n\_clusters? What can we do to increase the confidence we have in the number of n\_clusters?*

However, it is important to remember that each time one of these models was fit it was done so using random cluster centers. So, it is possible that if we ran these 10 models again that we could get a different number for the optimal value of clusters. To increase our confidence that 3 is the optimal number of clusters we will run each of the same 10 models 100 times and take a summary statistic over the 100 predicted labels.

Exercise 4: Tuning n\_clusters using Ensembles

Previously, when we built an ensemble of models, we chose one value for n\_clusters and iterated over it 100 times to generate more stable predictions. After that, we looked to tune the number of clusters by iterating over a range of values for n\_clusters. But only 1 model was built for each possible value of n\_clusters, and knowing that each time a k-Means model is fit with random cluster centers, we had some question marks regarding our confidence in the optimal number of n\_clusters. Thus, to improve our confidence in the tuned number of clusters for our k-Means model we will expand our skillset and knowledge of loops by combining both of our previous loops into one, nested loop. A nested loop is a loop occurring inside of another loop. We will use a nested loop to find the best number of clusters by iterating through 10 possible values for n\_clusters and building 100 models for each value of n\_clusters.

After the glass data set has been imported, shuffled, and standardized (see Exercise 1):

1. Import the packages we need outside of the loop using from sklearn.cluster import KMeans and import numpy as np.

***Note***

We import our packages outside the loop so we do not waste time and computational resources importing the packages at every iteration when we do not need to.

from sklearn.cluster import KMeans

import numpy as np

It is easier to build and comprehend nested loops by working from the inside-out. First, instantiate an empty list, inertia\_list, for which we will be appending inertia values after each iteration of the inside loop using inertia\_list = [].

inertia\_list = []

In the inside loop, we will iterate through 100 models using for i in range(100):.

for i in range(100):

Build KMeans model with n\_clusters=x using model = KMeans(n\_clusters=x).

model = KMeans(n\_clusters=x)

***Note***

The value for x is determined by the outer loop, which we have not covered yet, but we will cover in detail very shortly.

Fit the model to scaled\_features using model.fit(scaled\_features).

model.fit(scaled\_features)

Get the inertia value and save it to the object inertia using inertia = model.inertia\_.

inertia = model.inertia\_

Append inertia it to inertia\_list using inertia\_list.append(inertia).

inertia\_list.append(inertia)

***Note***

After all 100 iterations of n\_clusters=x have concluded, inertia\_list will contain 100 inertia values; one for each model.

Moving to the outside loop, instantiate another empty list to store the average inertia values using mean\_inertia\_list = [].

mean\_inertia\_list = []

Since we want to check the average inertia over 100 models for n\_clusters equaling 1 through 10, instantiate the outer loop using for x in range(1, 11):. The value for x will dictate the value for n\_clusters and since it is on the outside, it will not move to the next value of x until the inner loop has iterated through its 100 loops. The number of clusters cannot equal 0, thus the reason for iterating through the range 1 through 11 as opposed to 0 through 10.

for x in range(1, 11):

After the inside loop has run through its 100 iterations, and the inertia value for each of the 100 models have been appended to inertia\_list, compute the mean of this list and save the object as mean\_inertia using mean\_inertia = np.mean(inertia\_list).

mean\_inertia = np.mean(inertia\_list)

Append mean\_inertia to mean\_inertia\_list using mean\_inertia\_list.append(mean\_inertia).

mean\_inertia\_list.append(mean\_inertia)

After 100 iterations have been completed 10 times for a total of 1000 iterations, mean\_inertia\_list contains 10 values that are the average inertia values for each value of n\_clusters. This allows us to plot mean\_inertia by n\_clusters and choose the optimal number of n\_clusters using the, previously described, “elbow” method.

Altogether, this looks like:

from sklearn.cluster import KMeans

import numpy as np

mean\_inertia\_list = [] # create a list for the average inertia at each n\_clusters

for x in range(1, 11): # loop through n\_clusters 1-10

inertia\_list = [] # create a list for each individual inertia value at n\_cluster

for i in range(100):

model = KMeans(n\_clusters=x) # instantiate model

model.fit(scaled\_features) # fit model

inertia = model.inertia\_ # get inertia

inertia\_list.append(inertia) # append inertia to inertia\_list

# moving to the outside loop

mean\_inertia = np.mean(inertia\_list) # get mean of inertia list

mean\_inertia\_list.append(mean\_inertia) # append mean\_inertia to mean\_inertia\_list

print(mean\_inertia\_list)



1. Plot mean\_inertia by number of clusters by, first, importing matplotlib using import matplotlib.pyplot as plt.

import matplotlib.pyplot as plt

Create a list of numbers and save it as the object x so we can plot it on the x-axis using x = list(range(1,len(mean\_inertia\_list)+1)).

x = list(range(1, len(mean\_inertia\_list)+1))

Save mean\_inertia\_list as the object y using y = mean\_inertia\_list.

y = mean\_inertia\_list

Plot mean inertia by number of clusters using plt.plot(x, y).

plt.plot(x, y)

Set the plot title to read ‘Mean Inertia by n\_clusters’ using plt.title(‘Mean Inertia by n\_clusters’).

plt.title(‘Mean Inertia by n\_clusters’)

Label the x-axis ‘n\_clusters’ using plt.xlabel(‘n\_clusters’).

plt.xlabel(‘n\_clusters’)

Set the tick labels on the x-axis as the values in x using plt.xticks(x).

plt.xticks(x)

Set the y-axis label to ‘Mean Inertia’ using plt.ylabel(‘Mean Inertia’).

plt.ylabel(‘Mean Inertia’)

Display the plot use plt.show().

plt.show()

Altogether, this looks like:

import matplotlib.pyplot as plt

x = list(range(1, len(mean\_inertia\_list)+1))

y = mean\_inertia\_list

plt.plot(x, y)

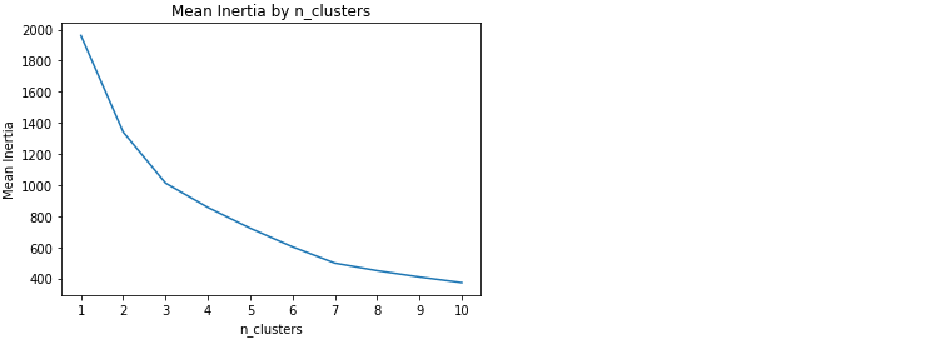
plt.title('Mean Inertia by n\_clusters')

plt.xlabel('n\_clusters')

plt.xticks(x)

plt.ylabel('Mean Inertia')

plt.show()



***Note***

Again, using the elbow method, 3 is the best value for n\_clusters. But this time we can make this claim because we have 1000 models to base this on rather than 10.

*Discuss: How confident are we that n\_clusters = 3 now that we have concluded this from 1000 models as opposed to 10? Now that we know the best value for n\_clusters, how might we improve upon this model?*

However, if the data has too many dimensions, the k-Means algorithm can fall subject to the *curse of dimensionality* by inflated Euclidean distances and subsequent erroneous results. Thus, before fitting a k-Means model, using a dimension reduction strategy is encouraged.

Reducing the number of dimensions helps to eliminate multicollinearity and decreases the time to fit the model. *Principal component analysis (PCA)* is a common method to reduce the number of dimensions by discovering a set of underlying linear variables in the data.

Principal Component Analysis (PCA)

*Present 18-19: Brief intorduction of Principal Component Analysis*

At a high level, PCA is a technique for creating uncorrelated linear combinations from the original features termed *components*. Of the principal components, the first component explains the greatest proportion of variance in the data while the following components account for progressively less variance.

To demonstrate PCA, we will:

* Extract a user-defined number of principal components
* Tune the number of principal components by setting a threshold of explained variance to remain in the data
* Fit those components to a k-Means cluster analysis and compare k-Means performance before and after the PCA transformation

Exercise 5: PCA with 2 Principal Components

After the glass data set has been imported, shuffled, and standardized (see Exercise 1):

1. Import the PCA function using from sklearn.decomposition import PCA. Then, instantiate a PCA model with, in this case, 2 principal components using model = PCA(n\_components=2).

from sklearn.decomposition import PCA

model = PCA(n\_components=2)

***Note***

The number for n\_components can be whatever you want it to be between 1 and the number of features in the data. However, by setting n\_components to 2, we opt to only return the first 2 principal components. If we do not specify a value for n\_componentsthen the maximum number of components is returned.

1. Fit the PCA model to scaled\_features using model.fit(scaled\_features).

model.fit(scaled\_features)



1. Get the proportion of explained variance in the data for each of the 2 components and save the array as the object explained\_var\_ratio using explained\_var\_ratio = model.explained\_variance\_ratio\_. We can see that the first principal component explains 35.31% of the variance in the data while the second principal component explains 25.05% of the variance in the data.

explained\_var\_ratio = model.explained\_variance\_ratio\_

print(explained\_var\_ratio)



1. To print the message ‘The total percentage of explained variance for the first 2 principal components is 60.37%’ programmatically, that is if the total percentage of explained variance for n\_components = 2 changes, use print(‘The total percentage of explained variance for the first 2 principal components is {0:0.2f}%’.format(sum(explained\_var\_ratio)\*100)). As mentioned previously in the chapter, by placing 0:0.2f inside of {}, we will format the value to 2 decimal places. The value we are formatting is the sum of the explained\_var\_ratio list which is calculated using sum(explained\_var\_ratio). However, this value will return a proportion and we want to print a percentage, so we will convert it to a percentage using sum(explained\_var\_ratio)\*100.

print('The total percentage of explained variance for the first 2 principal components is {0:0.2f}%'.format(sum(explained\_var\_ratio)\*100))



1. If we are going to be fitting a model using the first 2 principal components, then we need to transform scaled\_features into the 2 principal components and save it as a new array object. We can do this using df\_pca = model.transform(scaled\_features). The original data with 9 features and 218 observations has been condensed into 2 features and 218 observations which combine to explain 60.37% of the variance in the data.

df\_pca = model.transform(scaled\_features)

***Note***

Before we use these 2 principal components as features in our k-Means model we want to make sure it is the optimal number of principal components by tuning the argument n\_components to prevent underfitting or overfitting. To tune the number of principal components we will return all possible principal components and keep only those components which explain 95% of the variance in the data (95% is a user-defined threshold and can be set to whatever we want). Thus, only the first *n* components that have a cumulative sum of greater than or equal to 95% explained variance will be used as features in our clustering model.

*Discuss: How might we decide how many principal components?*

Exercise 6: PCA - Tuning n\_components

The objective in this exercise is to return all possible principal components (I.e., the same number of features in our data frame) and programmatically determine which ones will combine to explain no less than 95% of the explained variance in the data.

After the glass data set has been imported, shuffled, and standardized (see Exercise 1):

1. To return all possible principal components, import PCA from sklearn.decomposition using from sklearn.decomposition import PCA. Next, instantiate a PCA model with no arguments for n\_components using model = PCA().

from sklearn.decomposition import PCA

model = PCA()

1. Fit the PCA model to scaled\_features using model.fit(scaled\_features).

model.fit(scaled\_features)

1. Transform scaled\_features into the 9 principal components and save it as a new array object. We can do this using df\_pca = model.transform(scaled\_features).

df\_pca = model.transform(scaled\_features)

***Note***

The original data with 9 features and 218 observations has been transformed into 9 principal components and 218 observations which combine to explain 100% of the variance in the data which will be apparent in the next step.

1. Since we want to see the principal component at which 95% of the variance in the data is explained we will calculate the cumulative sum of explained variance by principal component using numpy’s cumsum function. First, import numpy as np. Then, calculate the cumulative sum of the array returned from model.explained\_variance\_ratio\_ and save it as an array using cum\_sum\_explained\_var = np.cumsum(model.explained\_variance\_ratio\_).

import numpy as np

cum\_sum\_explained\_var = np.cumsum(model.explained\_variance\_ratio\_)

print(cum\_sum\_explained\_var)



***Note***

We can easily see that at the sixth principal component we have a value greater than or equal to 0.95. However, this may be difficult with more features, introduces human error in the decision-making process, and makes the analysis less automated. Using some logic, the best number of principal components can be programmatically determined in Python.

*Discuss: Why might we want to set up logic so a threshold is checked programmatically asopposed to checking manally?*

1. We can set our threshold for percent of variance to keep in the data as 95% using threshold = .95.

threshold = .95

Using this threshold, we will loop through the list of cumulative explained variance and see where they explain no less than 95% of the variance in the data. Since we will be looping through the indices of cum\_sum\_explained\_var, we will instantiate our loop using for i in range(len(cum\_sum\_explained\_var)):.

for i in range(len(cum\_sum\_explained\_var)):

Check to see if the item in cum\_sum\_explained\_var is greater than or equal to 0.95 using if cum\_sum\_explained\_var[i] >= threshold:.

if cum\_sum\_explained\_var[i] >= threshold:

If that logic is met, then we, will add 1 to that index (because we cannot have 0 principal components), save the value as an object, and break the loop. To do this we will use best\_n\_components = i+1 inside of the if statement and break in the next line.

best\_n\_components = i+1

Break

The last two lines in the if statement else: and pass tell the loop not to do anything if the logic is not met.

else:

Pass

Altogether, this looks like:

threshold = .95

for i in range(len(cum\_sum\_explained\_var)):

if cum\_sum\_explained\_var[i] >= threshold:

best\_n\_components = i+1

break

else:

pass

print('The best n\_components is {}'.format(best\_n\_components))



1. To display this visually, we will plot the proportion of explained variance by the number of principal components. First, import matplotlib.pyplot using import matplotlib.pyplot as plt.

import matplotlib.pyplot as plt

Create a list of numbers ranging from 1 to the number of principal components and save it as x using x = list(range(1, len(cum\_sum\_explained\_var)+1)).

x = list(range(1, len(cum\_sum\_explained\_var)+1))

Save cum\_sum\_explained\_var as y using y = cum\_sum\_explained\_var.

y = cum\_sum\_explained\_var

Create our plot using plt.plot(x, y, color=’blue’, label=’Explained Variance’).

plt.plot(x, y, color=’blue’, label=’Explained Variance’)

***Note***

The arguments x and y are for the number of principal components and cum\_sum\_explained\_var, respectively. Using ‘blue’ as the argument for color makes the line blue and using ‘Explained Variance’ as the argument for label displays Explained Variance in the plot legend.

Create a programmatic title explaining the main information from the plot using plt.title(‘{0} n\_components are suggested to preserve {1} of the variance’.format(best\_n\_components, threshold)). This yields a title that reads ‘2 n\_components are suggested to preserve 0.95 of the variance.’

plt.title('{0} n\_components are suggested to preserve {1} of the variance'.format(best\_n\_components, threshold))

***Note***

The 0 in the first {} refers to the first argument in the format function, best\_n\_components, and the 1 in the second {} refers to the second argument in the format function, threshold.

Name the y-axis ‘Proportion of Explained Variance’ using plt.ylabel(‘Proportion of Explained Variance’).

plt.ylabel(‘Proportion of Explained Variance’)

Name the x-axis ‘n\_components’ using plt.xlabel(‘n\_components’).

plt.xlabel(‘n\_components’)

Make sure the x-axis is numbered 1 through 4 (I.e., the maximal number of principal components) using plt.xticks(range(1, len(cum\_sum\_explained\_var +1)).

plt.xticks(range(1, len(cum\_sum\_explained\_var +1))

Make the visualization easier to comprehend by placing a horizontal, dotted gray line at the level of threshold using plt.axhline(y=threshold, color=’gray’, linestyle=’--’, label = ‘{} Explained Variance’.format(threshold)).

plt.axhline(y=threshold, color=’gray’, linestyle=’--’, label = ‘{} Explained Variance’.format(threshold))

***Note***

The y argument tells Python at which value to place the horizontal line. Using ‘gray’ as the value for the color argument tells Python that the line should be gray. Using ‘--’ as the value for the linestyle argument makes sure the line is dotted. The label argument tells Python what to label the line in the legend. Our label is programmatic so if the threshold changes, we would not need to adjust our code.

The line plt.legend(loc=’best’) simply creates a plot legend and puts it in the best location to not interfere with the lines in the plot.

plt.legend(loc=’best’)

Display the plot using plt.show().

plt.show()

Altogether, this looks like:

import matplotlib.pyplot as plt

x = list(range(1, len(cum\_sum\_explained\_var)+1))

y = cum\_sum\_explained\_var

plt.plot(x, y, color='blue', label='Explained Variance')

plt.title('{0} n\_components are suggested to preserve {1} of the variance'.format(best\_n\_components, threshold))

plt.ylabel('Proportion of Explained Variance')

plt.xlabel('n\_components')

plt.xticks(range(1, len(cum\_sum\_explained\_var)+1))

plt.axhline(y=threshold, color='gray', linestyle='--', label = '{} Explained Variance'.format(threshold))

plt.legend(loc='best')

plt.show()

***Note***

Code adapted from (https://towardsdatascience.com/an-approach-to-choosing-the-number-of-components-in-a-principal-component-analysis-pca-3b9f3d6e73fe)

Now, we can refit another PCA model with n\_components = 6, transform the data into the principal components, and use these components in a new k-Meansmodel to lower the inertia values. Additionally, we can compare the inertia values across n\_clusters values for the models built using PCA transformed data to those using data that was not PCA transformed.

Activity 2: Evaluating mean inertia by cluster after PCA transformation

Now that we know the best number of principal components (I.e., 6), how to transform our features into principal components, and how to tune the optimal number of clusters for k-Means clustering with a nested loop, we will put them all together in this activity. We will also compare the inertia by n\_clusters prior to the PCA transformation and after the PCA transformation to see the effect of PCA transformation on k-Means model performance.

After the glass data set has been imported, shuffled, and standardized (see Exercise 1):

1. Instantiate a PCA model with the value for the n\_components argument equal to best\_n\_components (I.e., remember, best\_n\_components = 2).
2. Fit the model to scaled\_features.
3. Transform scaled\_features into its first 2 principal components and store it is the array object, df\_pca.
4. Using a nested loop, tune the number of n\_clusters by calculating the mean inertia over 100 models at values 1 through 10 for n\_clusters (see Exercise 4).
5. Visually compare mean inertia by n\_clusters using scaled\_features to mean inertia by n\_clusters using df\_pca by plotting both on the same plot.

***Note***

For step 6 to work, mean\_inertia\_list from Exercise 4 must still be in our environment.

**Solution:**

1. Import the PCA function using from sklearn.decomposition import PCA. Instantiate a PCA model with the value for the n\_components argument equal to best\_n\_components (I.e., remember, best\_n\_components = 2) using model = PCA(n\_components=best\_n\_components).

***Note***

If we have cleared our environment since exercise 6, we can use model = PCA(n\_components=6).

from sklearn.decomposition import PCA

model = PCA(n\_components=best\_n\_components)

1. Fit the model to scaled\_features using model.fit(scaled\_features).

model.fit(scaled\_features)



1. Transform scaled\_features into its first 6 principal components and store it is the array object, df\_pca, using df\_pca = model.transform(scaled\_features).

df\_pca = model.transform(scaled\_features)

1. Tune n\_clusters by importing the KMeans function using from sklearn.cluster import KMeans and import numpy using import numpy as np.

from sklearn.cluster import KMeans

import numpy as np

***Note***

Remember, it is easier to build and comprehend nested loops by working from the inside-out.

Instantiate an empty list, inertia\_list, for which we will be appending inertia values after each iteration of the inside loop using inertia\_list = [].

inertia\_list = []

In the inside loop, we will iterate through 100 models using for i in range(100):.

for i in range(100):

Build our KMeans model with n\_clusters=x using model = KMeans(n\_clusters=x).

model = KMeans(n\_clusters=x)

***Note***

The value for x will be dictated by the outer loop which is covered in detail below.

Fit the model to df\_pca using model.fit(df\_pca).

model.fit(df\_pca)

Get the inertia value and save it to the object inertia use inertia = model.inertia\_.

inertia = model.inertia\_

Append inertia to inertia\_list using inertia\_list.append(inertia).

inertia\_list.append(inertia)

***Note***

After the 100 iterations of n\_clusters=x has concluded, inertia\_list will contain 100 inertia values; one for each model.

Moving to the outside loop, instantiate another empty list to store the average inertia values using mean\_inertia\_list\_PCA = [].

mean\_inertia\_list\_PCA = []

Since we want to check the average inertia over 100 models for n\_clusters 1 through 10 we will instantiate the outer loop using for x in range(1, 11):.

for x in range(1, 11):

***Note***

The value for x will dictate the value for n\_clusters and since it is on the outside, it will not move to the next value of x until the inner loop has iterated through its 100 loops.

After the inside loop has run through its 100 iterations, and the inertia value for each of the 100 models have been appended to inertia\_list:

Compute the mean of this list and save the object as mean\_inertia using mean\_inertia = np.mean(inertia\_list).

mean\_inertia = np.mean(inertia\_list)

Append mean\_inertia to mean\_inertia\_list\_PCA using mean\_inertia\_list\_PCA.append(mean\_inertia).

mean\_inertia\_list\_PCA.append(mean\_inertia)

***Note***

After 100 iterations have been completed 10 times for a total of 1000 iterations, mean\_inertia\_list\_PCA contains 10 values that are the average inertia values for each value of n\_clusters. This allows us to plot mean\_inertia by n\_clusters and choose the optimal number of n\_clusters after the PCA transformation using the, previously described, “elbow” method.

Altogether, this looks like:

from sklearn.cluster import KMeans

import numpy as np

mean\_inertia\_list\_PCA = [] # create a list for the average inertia at each n\_clusters

for x in range(1, 11): # loop through n\_clusters 1-10

inertia\_list = [] # create a list for each individual inertia value at n\_cluster

for i in range(100):

model = KMeans(n\_clusters=x) # instantiate model

model.fit(df\_pca) # fit model

inertia = model.inertia\_ # get inertia

inertia\_list.append(inertia) # append inertia to inertia\_list

# moving to the outside loop

mean\_inertia = np.mean(inertia\_list) # get mean of inertia list

mean\_inertia\_list\_PCA.append(mean\_inertia) # append mean\_inertia to mean\_inertia\_list

print(mean\_inertia\_list\_PCA)



1. Visually compare average inertia by n\_clusters after PCA transformation and before PCA transformation by plotting both lines on the same plot. This plot differs from the plot in Exercise 4 in that it has two lines; one for mean inertia before PCA (I.e., mean\_inertia\_list) and one after PCA transformation (I.e., mean\_inertia\_list\_PCA). To display these on the same plot:

Import matplotlib using import matplotlib.pyplot as plt.

import matplotlib.pyplot as plt

Create a list of numbers and save it as the object x so we can plot it on the x-axis using x = list(range(1,len(mean\_inertia\_list\_PCA)+1)).

x = list(range(1,len(mean\_inertia\_list\_PCA)+1))

Save mean\_inertia\_list\_PCA as the object y using y = mean\_inertia\_list\_PCA.

y = mean\_inertia\_list\_PCA

Save mean\_inertia\_list as the object y2 using y2 = mean\_inertia\_list.

y2 = mean\_inertia\_list

Plot mean inertia after PCA transformation by number of clusters.

plt.plot(x, y, label='PCA')

Add our second line of mean inertia before PCA transformation by number of clusters using plt.plot(x, y2).

plt.plot(x, y2, label='No PCA')

Set the plot title to read ‘Mean Inertia by n\_clusters for Original Features and PCA Transformed Features’ using plt.title(‘Mean Inertia by n\_clusters for Original Features and PCA Transformed Features’).

plt.title(‘Mean Inertia by n\_clusters for Original Features and PCA Transformed Features’)

Label the x-axis ‘n\_clusters’ using plt.xlabel(‘n\_clusters’).

plt.xlabel(‘n\_clusters’)

Set the tick labels on the x-axis as the values in x using plt.xticks(x).

plt.xticks(x)

Set the y-axis label to ‘Mean Inertia’ using plt.ylabel(‘Mean Inertia’).

plt.ylabel(‘Mean Inertia’)

Display a legend using plt.legend().

plt.legend()

Display the plot use plt.show().

plt.show()

Altogether, this looks like:

import matplotlib.pyplot as plt

x = list(range(1, len(mean\_inertia\_list)+1))

y = mean\_inertia\_list

plt.plot(x, y)

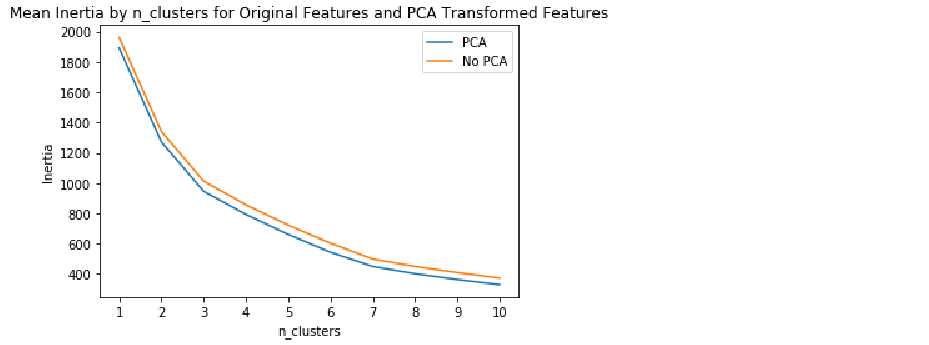
plt.title('Mean Inertia by n\_clusters')

plt.xlabel('n\_clusters')

plt.xticks(x)

plt.ylabel('Mean Inertia')

plt.show()



From the plot we can see that inertia is lower at every number of clusters in the model using the PCA transformation. This indicates that there was less distance between the group centroids and the observations in each cluster after the PCA transformation relative to before the transformation. Thus, by using a PCAtransformation on the original features, we were able to decrease the number of features and simultaneously improve our model by decreasing the within-cluster sum of squares (i.e., inertia).

*Discuss: Is there another step we can take to make sure the model with the 6 principal components is the best possible model? Hint: how would we determine mean inertia by number of clusters over a range of thresholds or a number of principal components?*

HCA and k-Means clustering are two widely-used unsupervised learning techniques used for segmentation. PCA can be used to help reduce the number of dimensions in our data and improve models. Linear discriminant function analysis (LDA) can do both. For the purposes of this chapter, we will learn how to use LDA as a supervised learner for classification.

Supervised Linear Discriminant Function Analysis (LDA)

*Present 26-27: Brief intorduction of Linear Discriminant Function Analysis*

Linear Discriminant Function Analysis (LDA) is related to MANOVA in that we use MANOVA to identify the linear variates best differentiating the groups. In LDA, the linear variates are the boundaries, or functions, used to discern one group from another (Field, 2013).

*Source: Field, A. (2013). Discovering statistics using IBM SPSS statistics. sage.*

In this example, we will walk through how to use LDA as a classification model via supervised learning. That is, we will be tuning and fitting the LDA model on the training data, testing it on the testing data, and evaluating model performance.

To demonstrate how to use LDA as a classifier for supervised learning we will:

* Fit an LDA model using the algorithm’s default hyperparameter settings
* Tune our LDA hyperparameters using a grid search
* Access LDA attributes by refitting a programmatically tuned LDA model

Exercise 7: Fitting LDA Model with Default Hyperparameters

To fit the model as a supervised learner using the default parameters of the LDA algorithm we will be using a slightly different glass data set, glass\_w\_outcome.csv. This data set contains the same 9 features as glass, but also an outcome variable, Type, corresponding to the type of glass. Type is labeled 1, 2, and 3 for building windows float processed, building windows non float process, and headlamps, respectively.

***Note***

Remember, we need an outcome variable in this exercise is because we are fitting a supervised, predictive model to learn an outcome (I.e., glass type). This is contrary to unsupervised, descriptive models where there is no outcome for which to learn.

1. Import the glass\_w\_outcome.csv file and save it as the object df.

import pandas as pd

df = pd.read\_csv('glass\_w\_outcome.csv')

1. Shuffle the data to remove any ordering effects and save it as the data frame df\_shuffled using from sklearn.utils import shuffle and df\_shuffled = shuffle(df, random\_state=42).

from sklearn.utils import shuffle

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

1. Scale df\_shuffled by, first, instantiating a StandardScaler() object named scaler using scaler = StandardScaler(). Fit scaler to df\_shuffled using scaler.fit(df\_shuffled). Transform the features to the scaled features and save them as scaled\_features using scaled\_features = scaler.transform(df\_shuffled).

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler() # create StandardScaler() object

scaler.fit(df\_shuffled) # fit scaler to the features

scaled\_features = scaler.transform(df\_shuffled) # transform features to scaled version

1. Save 'Type’ as DV (I.e., dependent variable) to make the code more elegant and flexible using DV = ‘Type’.

DV = 'Type'

1. Split the shuffled data into X and yusing X = df\_shuffled.drop(DV, axis=1) and y = df\_shuffled[DV], respectively*.*

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

y = df\_shuffled[DV]

***Note***

By saving *target* as *DV* we do not need to write Type in the two places where it refers to DV in step 6. This allows the code to be flexible to different data sets. By saving Type as DV ahead of time the dependent variable just needs to be changed in one place rather than several.

Unlike the unsupervised learning examples in previous sections, we will need to split the features (I.e., X) and the target (I.e., y) into testing and training data. For our convenience, Scikit Learn provides us with a function for this named train\_test\_split. In this example, we are placing 66% of the data into training and 33% in testing.

1. Split X and y into testing and training data using from sklearn.model\_selection import train\_test\_split and X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.33, random\_state=42).

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 the LDA model and save it as model. Import the function using from sklearn.discriminant\_analysis import LinearDiscriminantAnalysis and model = LinearDiscriminantAnalysis().

from sklearn.discriminant\_analysis import LinearDiscriminantAnalysis

model = LinearDiscriminantAnalysis()

***Note***

By instantiating a model with nothing between the trailing parenthesis, we are telling Python that we want to use the default hyperparameters. What are hyperparameters? Hyperparameters are user-defined settings within an algorithm which influence how the model fits and, subsequently performs on the data. Hyperparameters and their available values are listed in the documentation of every algorithm.

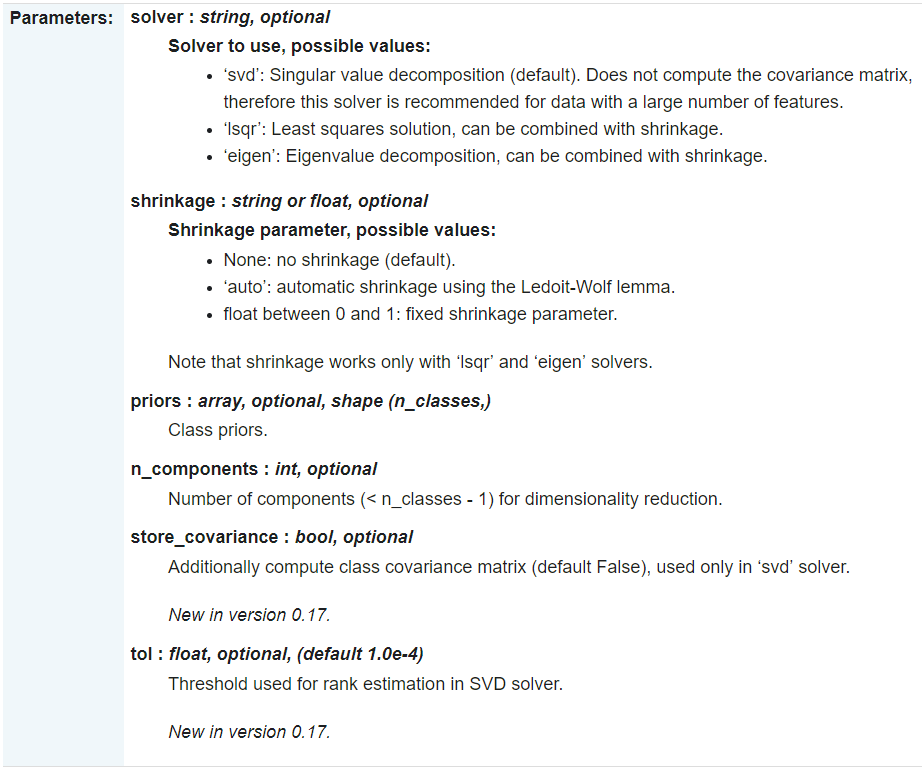


Figure 4.6. Hyperparameters for sklearn.discriminant\_analysis.LinearDiscriminantAnalysis

But how do we know the default hyperparameters for an algorithm? In Scikit-Learn, the *default* hyperparameter will be specified in the parameters section of the documentation table. However, if we scroll to the top of the page, we can see the default hyperparameter settings (see Figure 4.7).



Figure 4.7. Default hyperparameters for LDA

Thus, by failing to specify any hyperparameters in our LDA model we are telling Python that we want the ‘svd’ solver, no shrinkage, no priors, no n\_components, no covariance matrix, and a tolerance of 0.0001. For detailed descriptions of each hyperparameter, see Figure 4.6.

Now that we are aware of how to make sense of the documentation and the default hyperparameters:

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

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



***Note***

In the above screenshot the specified hyperparameters are the same as those in Figure 4.7.

1. Generate predictions on X\_test and save them as the array, predictions. This will return the predicted class for each observation. Once we have predictions on the test data, we can see how the model performed by comparing predictions to y\_test (I.e., actual values).

predictions = model.predict(X\_test)

print(predictions)



***Note***

Scikit-Learn provides for us functions making model performance evaluation very simple. For classification problems we can evaluate model performance using a confusion matrix which compares predicted values to actual values.

1. To generate a confusion matrix, begin by importing the function confusion\_matrix using from sklearn.metrics import confusion\_matrix. Then, save the confusion matrix into a 3x3 array object using conf\_matrix = confusion\_matrix(y\_test, predictions).

from sklearn.metrics import confusion\_matrix

conf\_matrix = confusion\_matrix(y\_test, predictions)

print(conf\_matrix)



***Note***

This 3x3 array can be difficult to comprehend without guidance. The first, second, and third columns represent predicted 1, predicted 2, and predicted 3, respectively. The first, second, and third rows represent actual 1, actual 2, and actual 3, respectively.

1. Make conf\_matrix less confusing by adding some styling. First, convert conf\_matrix into a pandas data frame named cm

import pandas as pd

cm = pd.DataFrame(conf\_matrix)

Create a new column named ‘Total’ which contains the row totals

import numpy as np

cm[‘Total’] = np.sum(cm, axis=1)

Compute a row containing the column total

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

Assign names to the columns in cm

cm.columns = ['Predicted 1', 'Predicted 2', 'Predicted 3', 'Total']

Assign the row names (I.e., indices)

cm = cm.set\_index([['Actual 1', 'Actual 2', 'Actual 3', 'Total']]).

Altogether, this looks like:

import pandas as pd

cm = pd.DataFrame(conf\_matrix)

import numpy as np

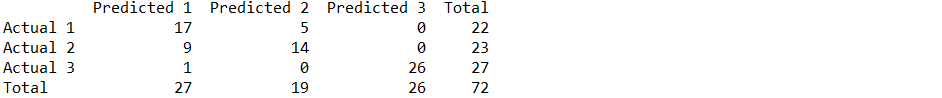
cm['Total'] = np.sum(cm, axis=1)

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

cm.columns = ['Predicted 1', 'Predicted 2', 'Predicted 3', 'Total']

cm = cm.set\_index([['Actual 1', 'Actual 2', 'Actual 3', 'Total']])

print(cm)



***Note***

The confusion\_matrix function, in this case, returns a 3x3 array with no labels. Here, we have converted it into a pandas data frame, created a *Total* column, created a *Total* row, named the columns, and named the row indices. It is easier to interpret this way and can be saved as a csv or excel file with a single line of code using the to\_csv function.

To interpret this output, think in terms of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN):

* TP: observation predicted to be in a certain class and is in that class (I.e., a correct prediction).
* FP: observation predicted to be in a class but is not in that class (I.e., an incorrect prediction).
* TN: observation predicted not to be in a certain class and is not in that class (I.e., a correct prediction).
* FN: observation predicted not to be in a certain class and is in that class (I.e., an incorrect prediction).

From the 3x3 confusion matrix, we can see that of the 22 observations that were in class 1, 17 were predicted to be in class 1. Of the 23 observations that were in class 2, 14 were predicted to be in class 2. Of the 27 observations that were in class 3, 26 were predicted to be in class 3.

LDA’s default settings predicted with 79.17% accuracy on this data. However, this model may be improved using a different combination of hyperparameters. This is referred to as tuning the model. Scikit-learn provides an outstanding library for tuning models called GridSearchCV*.*

Exercise 8: Tuning LDA Hyperparameters using GridSearchCV

Grid search builds a model for all combinations of hyperparameters, evaluates each model’s performance, and picks the ‘best’ combination of hyperparameters according to a scoring metric set by the user inside of the GridSearchCV function. Algorithms with a large number of hyperparameters and/or ranges of values can take a long time to perform a grid search; especially on larger data sets. Luckily, in the following example we are using a small data set with few hyperparameters and each hyperparameter has few possible values.

After the data has been imported, shuffled, scaled, and split into testing and training data (see Exercise 7):

1. Instantiate a grid with the possible values for each hyperparameter as key value pairs in a dictionary.

import numpy as np

grid = {'solver': ['lsqr', 'eigen'],

'shrinkage': [None, 'auto']}

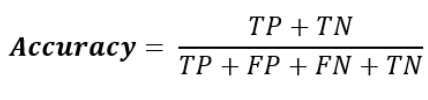
print(grid)



***Note***

Here, we provide 2 hyperparameters for which to tune and each hyperparameter has 2 possible values. In the GridSearchCV function, we provide arguments for estimator (I.e., LinearDiscriminantAnalysis()), param\_grid(I.e., *grid*), scoring (I.e., *accuracy*), and cv(I.e., *5*). The scoring argument has several possible values that are available on the documentation.

Here, we have specified that we want the model with the combination of hyperparameters that will optimize accuracy. Accuracy is the proportion of predictions that were correctly classified and is defined:



Equation 4.3. Accuracy.

1. Instantiate a GridSearchCV model with accuracy as the scoring outcome for which we will be maximizing. To import GridSearchCV use from sklearn.model\_selection import GridSearchCV and to import LinearDiscriminantAnalysis use from sklearn.discriminant\_analysis import LinearDiscriminantAnalysis. To instantiate the GridSearchCV model use model = GridSearchCV(LinearDiscriminantAnalysis(), grid, scoring='accuracy', cv=5).

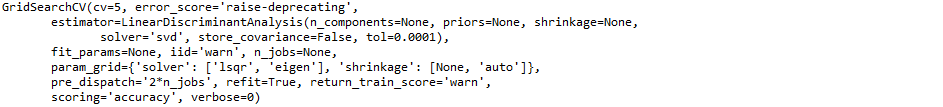
from sklearn.model\_selection import GridSearchCV

from sklearn.discriminant\_analysis import LinearDiscriminantAnalysis

model = GridSearchCV(LinearDiscriminantAnalysis(), grid, scoring='accuracy', cv=5)

1. Fit the model to the training data.

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



1. Save the best parameters as a dictionary.

best\_parameters = model.best\_params\_

print(best\_parameters)



***Note***

From the grid search, we can see that the hyperparameters that optimized accuracy were no shrinkage and a lsqr solver.

1. Generate predictions on X\_test.

predictions = model.predict(X\_test)

print(predictions)



1. Evaluate performance by comparing the predictions to y\_testwith a styled confusion matrix (for instructions, see Exercise 7, part 11).

from sklearn.metrics import confusion\_matrix

import pandas as pd

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

import numpy as np

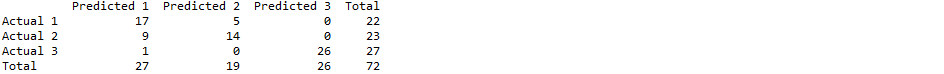
cm['Total'] = np.sum(cm, axis=1)

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

cm.columns = ['Predicted 1', 'Predicted 2', 'Predicted 3', 'Total']

cm = cm.set\_index([['Actual 1', 'Actual 2', 'Actual 3', 'Total']])

print(cm)



In this case, tuning the model’s hyperparameters did not change its predictions. However, this is rarely the case and model hyperparameters must be tuned in order to maximize the performance of our model.

***Note***

GridSearchCV contains numerous useful and interesting attributes, a drawback of using a GridSearchCV model is that some of the attributes we may want to access from our LinearDiscriminantAnalysis model are not accessible when we fit a GridSearchCV model. However, we can access those attributes by refitting a new LinearDiscriminantAnalysis model with the tuned hyperparameters. This can be done by copying and pasting the values from best\_parameters, with their coinciding hyperparameter. However, that requires manual intervention and introduces room for possible errors. To automate the process, we can programmatically access the values from best\_parameters and store those values as the hyperparameters. This method becomes increasingly useful when tuning models in an ensemble with RandomizedSearchCV.

The best\_params\_ attribute from GridSearchCV returns a dictionary with hyperparameters as keys and the tuned value for that hyperparameter as the value. In step 4, we saved this dictionary as best\_parameters. Thus, to access the tuned value for the solver hyperparameter we would use best\_parameters['solver'] which returns ‘lsqr’.

***Note***

In this case, the tuned hyperparameter for shrinkage is None, so if we run best\_parameters['shrinkage'] it does not return anything.

1. Import the LinearDiscriminantAnalysis function using from sklearn.discriminant\_analysis import LinearDiscriminantAnalysis. Instantiate the new LinearDiscriminantAnalysis model with programmatically tuned hyperparameters using model = LinearDiscriminantAnalysis(solver=best\_parameters['solver'], shrinkage=best\_parameters['shrinkage']).

from sklearn.discriminant\_analysis import LinearDiscriminantAnalysis

model = LinearDiscriminantAnalysis(solver=best\_parameters['solver'],

shrinkage=best\_parameters['shrinkage'])

1. Fit the model

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



***Note***

Notice from the output that the tuned hyperparameters are set to the values from our best\_parameters dictionary.

Now, we not only have access to the predict attribute and others from the GridSearchCV function, but also attributes such as coef\_, intercept\_, covariance\_, means\_, priors\_, and classes\_.

***Note***

Fitting a model using GridSearchCV can be extremely time consuming if the data contains many features and/or large ranges of hyperparameters. Additionally, the scoring argument can be adjusted for supervised learning models such as classification and regression as well as unsupervised models (i.e., clustering). It is important to become familiar with the documentation for GridSearchCV and RandomizedSearchCV to make model-tuning more efficient and maximize model performance.

*Discuss: Use cases for programmatically selecting tuned hyperparameters and refitting tuned parameters? Hint: feature performance in ensembles of XGBoost (trees), Decision Trees, and Random Forests using RandomizedSearchCV.*

Conclusion

This chapter introduced the reader to two widely-used unsupervised, clustering algorithms, *HCA* and *k-Means clustering*. While learning about *k-Means clustering*, we leveraged the power of loops to create ensembles of models for tuning the number of clusters and to gain more confidence in our predictions. During the *PCA* section, we determined the number of principal components for dimensionality reduction and fit the components to a *k-Means* model. Additionally, we compared the differences in *k-Means* model performance before and after *PCA* transformation. Lastly, we were introduced to an algorithm, *LDA*, which can be used for dimension reduction or classification. We then tune LDA hyperparameters using *GridSearchCV*. Along the way, we learned to evaluate classifier model performance using a confusion matrix. This was a comprehensive chapter aiming to teach dimensionality reduction and unsupervised learning in progressively challenging and creative ways. By the end of the chapter, students should understand clustering techniques, the benefits of dimensionality reduction, and methods for hyperparameter tuning.