Dimensionality Reduction and Unsupervised Learning

Lesson Objectives:

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

* Describe the differences between *supervised* and *unsupervised* learning
* 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

In *supervised learning*, clear instruction is provided to *predictive models* concerning the outcome for which to learn. Meanwhile, in *unsupervised learning*, *descriptive models* are used in exploratory analysis to uncover patterns in unlabeled data.

*PowerPoint: Slide 3*

For example, analysts seeking to elevate sales by targeting select customers for marketing ads or promotions will separate customers by their shopping behavior. K-Means clustering has been used in fraud detection (Tech, 2014). Hierarchical clustering has been implemented in neuroscience and motor behavior research (Schack, 2012).

*Source: Tech, M. (2014). Fraud Detection in Credit Card by Clustering Approach.*

*Source: Schack, T. (2012). Measuring mental representations. Measurement in sport and exercise psychology, 8, 203-14.*

*Discuss: Discuss the differences between supervised and unsupervised learning.*

*Discuss: Brainstorm some ways in which clustering can be used in a variety of domains.*

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.

*Discuss: Discuss why data reduction is important for improving model performance*

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.

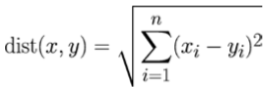
In this chapter we will be covering two widely used unsupervised clustering algorithms: *Hierarchical Cluster Analysis (HCA)* and *k-Means clustering*. Additionally, we will explore dimension reduction using *Principal Component Analysis (PCA)* and observe how reducing dimensionality can improve model performance. Additionally, 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.

Unsupervised Learning

*Section 1: Hierarchical Cluster Analysis (HCA)*

Hierarchical cluster analysis (HCA) is best implemented when the user does not have an *a priori* number of clusters for which to build. HCA measures 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:

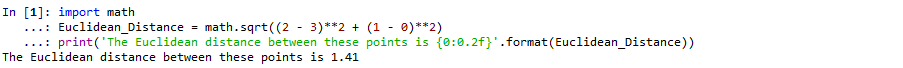


Equation 1. Euclidean Distance

*Source: Lantz, B. (2013). Machine learning with R. Packt Publishing Ltd.*

*PowerPoint: Slide 4*

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:



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. Thomas Schack (2012) 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.dendrogram* documentation site is displayed below.

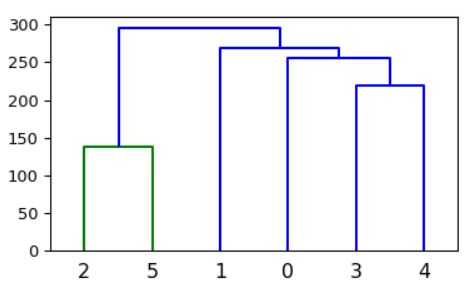


Figure 1. Example dendrogram

*Link to scipy.dendrogram documentation:* [*https://docs.scipy.org/doc/scipy/reference/generated/scipy.cluster.hierarchy.dendrogram.html*](https://docs.scipy.org/doc/scipy/reference/generated/scipy.cluster.hierarchy.dendrogram.html)

*PowerPoint: Slide 5*

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: Discuss dendrogram interpretation*

To demonstrate HCA, we will be using the well-known *Iris* data set from *sklearn.datasets*. This data is widely-used for tutorials on clustering and classification by using 4 flower attributes (sepal length, sepal width, petal length, and petal width) to predict 3 Iris flower species (Iris Setosa, Iris Versicolor, and Iris Virginica). It is a small data set with only 150 rows. Until the last section of the chapter where we use LDA as a supervised classifier, we will not include the dependent variable, flower species, in the data. Thus, the *Iris* data set will predominantly be used to demonstrate clustering and data reduction techniques.

Step 1: To get started, we must first import the Iris utils bundle from sklearn.datasets and save it to the object *iris*.

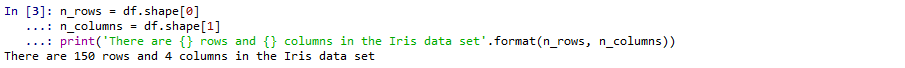
*Source: To access the documentation for the Iris data utils use: https://scikit-learn.org/0.16/modules/generated/sklearn.datasets.load\_iris.html#sklearn.datasets.load\_iris*



Step 2: Now, we can access only the features from the *iris* object and save that numpy array as the data frame, *df*.



Step 3: Explore the dimensions of the data and print the information to the console.



Step 4: We can also look for some basic information from the data frame.



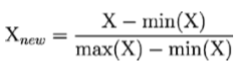
From the output, we can see that there are 150 entries (I.e., rows) and 4 columns. We can also see that there are no missing values (I.e., all are *non-null*) and that the data type for each column is *float64*. Float numbers are named accordingly because they are floating-point numbers, meaning they are a number that contains a decimal point.

Step 5: It is good practice to remove any possible order effects in the data by shuffling the rows prior to building any models.



Here, we use *sklearn.utils.shuffle* to shuffle *df* and create a new, shuffled data frame named *df\_shuffled*. The *random\_state* argument inside of the *shuffle* function is used for reproducibility. By setting the random state to 42 every time I ensure that my data’s rows are shuffled the same way each time.

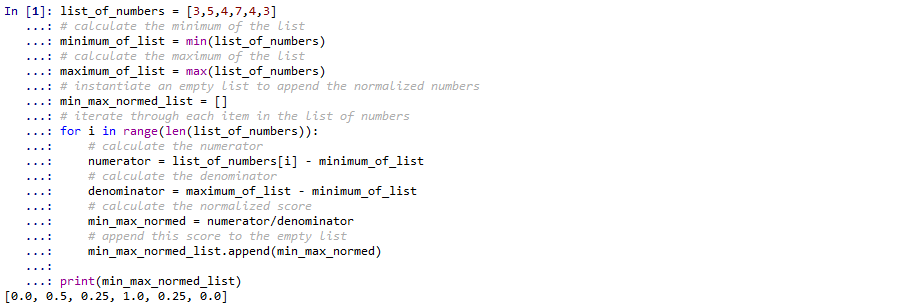
Step 4: To ensure features measured at different scales contribute equally to the analysis, models perform much better when they are standardized. Typically, variables are standardized using min-max normalization where the numbers become a proportion of the range of values. This results in floating point numbers ranging from 0 to 1. The formula for min-max normalization is as follows:



Equation 2. Min-max normalization formula

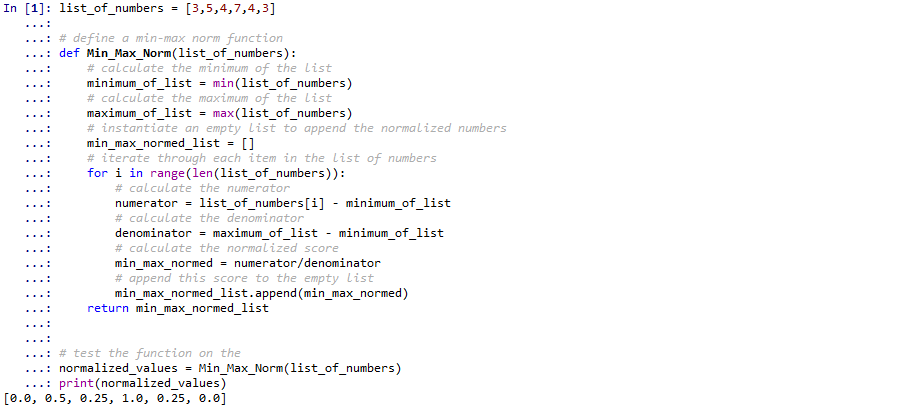
*Source: Lantz, B. (2013). Machine learning with R. Packt Publishing Ltd.*

Using a made-up list of values, calculate this in Python as follows:



Here, we instantiated a list of numbers and converted each to a min-max normalized score by iterating through the list, subtracting the minimum from each item and dividing by the range. Each normalized score is then appended to the empty list (I.e., *min\_max\_normed\_list*) and printed to the console.

This can be used as a user-defined function as follows:



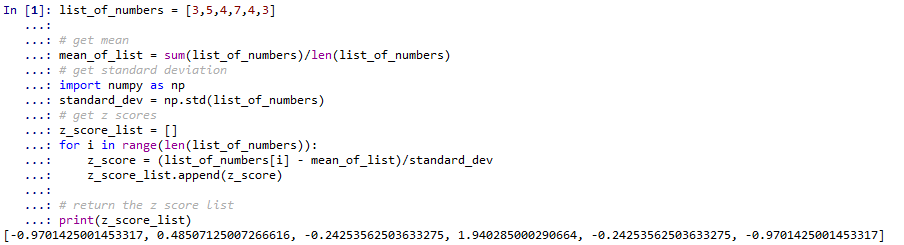
However, we will be scaling our variables into z-scores. That is, each score is represented in terms of the number of standard deviations from the mean. The formula for z-score is as follows:



Equation 3. Z-score standardization formula

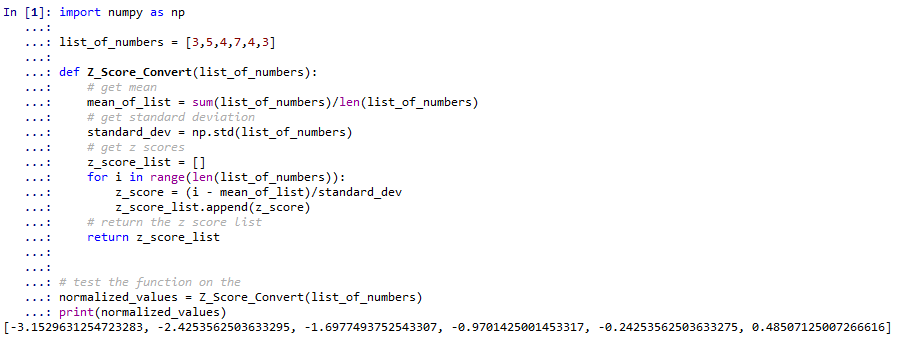
*Source: Lantz, B. (2013). Machine learning with R. Packt Publishing Ltd.*

Using a made-up list of numbers, we can convert each item in this list to a z-score in Python as follows:



Here, we instantiated a list of numbers, calculated the mean and standard deviation of the list, and then converted each value in the list to a z-score by iterating through each value in the list, subtracting the mean, and dividing by the list’s standard deviation. Each z-score was then appended to the empty list (I.e., *z\_score\_list*) and printed to the console.

This can be converted to a user-defined function in python as follows:



This function (I.e., *Z\_Score\_Convert*) takes a list of numbers as an argument and converts each number in that list to a z-score. Luckily, we do not need to define our own preprocessing functions because Scikit Learn provides us with a *StandardScaler* function that converts each value in an array to a z-score.

Step 6: Transform each observation into a z-score using *sklearn.preprocessing.StandardScaler*. This process involves instantiating a *StandardScaler()* model, fitting that model to the shuffled features (I.e., *df\_shuffled*), and using that model to transform the shuffled features into scaled features (I.e., *scaled\_features*).



To use the *StandardScaler* function, we instantiate a StandardScaler object (I.e., *scaler*), fit this object to the shuffled data, and then store the transformed data as a scaled feature data frame (I.e., *scaled\_features*).

*Source: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html*

Now that the features have been scaled, we can build our HCA model.

Step 7: Build the model using the *linkage* function from *scipy.cluster.hierarchy*

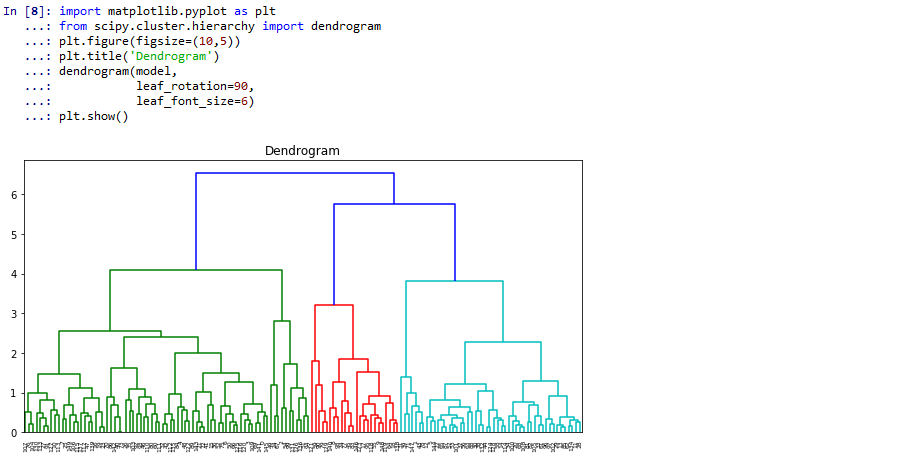


*Source*: https://docs.scipy.org/doc/scipy/reference/generated/scipy.cluster.hierarchy.linkage.html

Note: Scikit-Learn also has a function for hierarchical cluster analysis, but it is easier to visualize the dendrogram by using the *scipy* library.

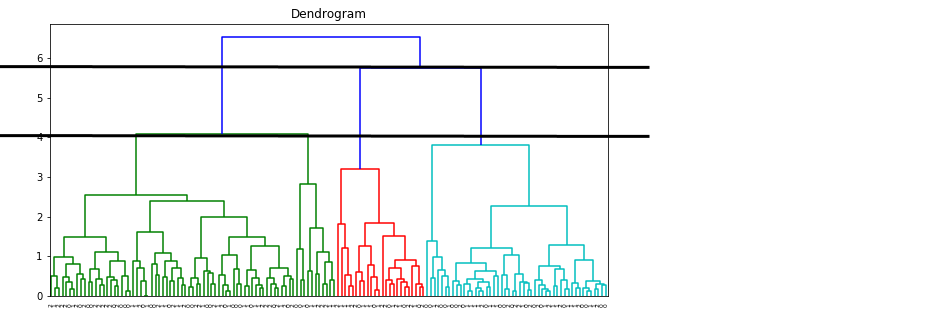
*Link to Scikit-Learn's library for HCA: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.AgglomerativeClustering.html#sklearn.cluster.AgglomerativeClustering*

Step 8: Display the dendrogram by plotting the linkage *model*.



*Link to scipy’s dendrogram function: https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.cluster.hierarchy.dendrogram.html*

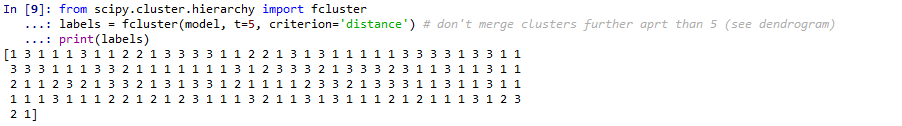
On the x-axis is the index for each observation, or row, in the data set. On the y-axis is Euclidean distance. Horizontal lines are links between and among observations. By default, scipy will color code the different clusters that it finds. However, a manual method for determining the optimal number 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.hierarchy* library to generate an array of labels that correspond to the rows in *df\_shuffled* and *scaled\_features*.

*Source for scipy’s fcluster function: https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.cluster.hierarchy.fcluster.html*

Step 9: Generate the predicted labels.



We can assign the label to each row by adding the *labels* array as a column in *df\_shuffled* or *scaled\_features*. Remember, we cannot add the *labels* array as a column into the data before it was shuffled because they are in different orders and the labels will not necessarily coincide with the row for which it is assigned.

Note: Here, we are using 5 as the argument for *t*. This is because at a Euclidean distance of 5 we have the optimal number of clusters.

Step 10: Assign *labels* array as a column in *df\_shuffled*.



There we go. We have successfully utilized HCA to cluster our data into three groups and matched the observations with their predicted cluster. However, HCA is just one method of clustering data. Another clustering algorithm with which it is important to be familiar is k-Means clustering.

*Section 2a: K-means clustering*

K-Means also uses distance to cluster observations. However, rather than linking observations to each other as in HCA, k-Means assigns *n* observations to *k* (number of) clusters. To determine which observations belong to which clusters, observations are assigned to the cluster in which it’s 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; 2) updating phase.

*PowerPoint: Slide 6*

In the assignment phase, observations are assigned to the cluster to which it has the smallest Euclidean distance from the cluster center. 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. After this shift in cluster centers, some observations may find themselves in a different cluster which causes the centroid to shift once again. This cycle continues until there are no reassignments (Lantz, 2013).

*PowerPoint: Slide 7*

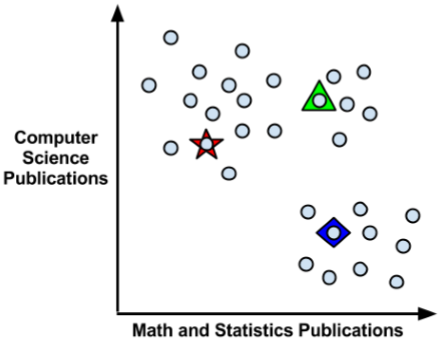


Figure 2. A scatterplot of observations and the group centroids as denoted by the star, triangle, and diamond.

*Source: Lantz, B. (2013). Machine learning with R. Packt Publishing Ltd.*

*Discuss: Discuss the way in which clusters are created using a k-Means cluster analysis. How is it different from the way in which clusters are generated using HCA? What might be some advantages/disadvantages to the group centroids being randomly chosen?*

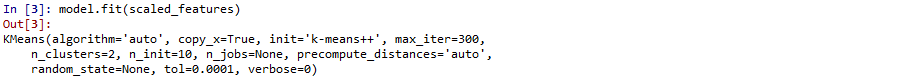
Using the *Iris* data set from *sklearn.datasets*, we will, first, fit a *k-Means* model with a user-defined number of clusters. Next, because of the randomness in which group centroids are chosen, we will increase the confidence in our predictions by building an ensemble of k-Means models with a given 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.

After the *Iris* data set has been imported, shuffled, and standardized, we will build our k-Means clustering model.

Step 2: Instantiate a *KMeans* model with, in this case, 2 clusters.



Step 3: Fit the model to *scaled\_features.*



Step 4: Return the randomly chosen group centroids.

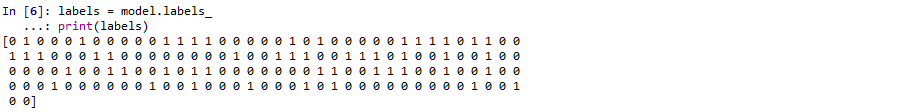


These are the coordinates of cluster centers. There are two rows because we have two clusters and four columns because we have four features in *scaled\_features*. To find the mean sum of squared distances of samples to their closest cluster center we will return the model’s inertia.

Step 5: Return the inertia for *n\_clusters* = 2.



Step 6: Find the predicted labels.



Since we chose 2 clusters, there are two possible labels; 0 and 1.

Step 7: Examine the frequency of each cluster.



Using 2 clusters, 100 observations were placed into the first cluster and 50 observations were grouped into the second cluster. Now, we can match the predicted class, *labels*, to the observations in *df\_shuffled*.

Step 8: Match the label with the observation by adding *label* as a column on the *df\_shuffled* data frame and view it by previewing the first 5 rows.



*Discuss: Discuss why it is important to match the label with the shuffled data or scaled data and not the original data.*

However, it is important to remember that these group centroids 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.

*PowerPoint: Slide 8*

*Discuss: Discuss the reasons behind building an ensemble of k-Means models.*

*Section 2b: 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).

*PowerPoint: Slide 9*

Building an ensemble of models involves iterating through *n* models, saving each model’s predictions, and calculating a prediction for each row by incorporating predictions from each model. To demonstrate model ensembles, we will begin by importing the *Iris* data, shuffling it, and standardizing the features.

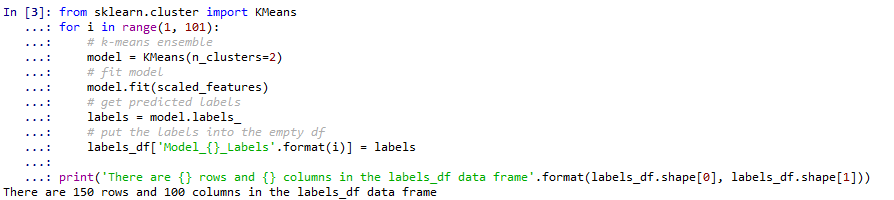
*After* the *Iris* data has been imported, shuffled, and standardized we must create an empty data frame where we will store an array of predicted labels in each column.

Step 2: Instantiate an empty data frame for which to append each model’s labels.



Now that the empty data frame, *labels\_df,* has been instantiated, we will iterate through *n* k-Means models and, after each iteration, append the predicted labels to a column in the data frame. As a result, *labels\_df* will have the same number of rows as *df\_shuffled* (I.e., 150) and 100 columns (one for each array of model predictions).

Step 3: Build 100 models and append these to *labels\_df*



Next, we must decide what metric to use across rows in order to assign a label to each row that incorporates labels from each model. For this example, we will use the mode. That is, the label occurring most frequently for that row across 100 models will be the final prediction.

Step 4: Calculate the mode for each row and that is the cluster for which it will be assigned



Step 5: Again, we can check out the frequency of each cluster.



In this case, the frequency of labels after running 100 models is the same as the frequency of labels after running it once. 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.

*Discuss: Discuss other ways of determining the model label (i.e., taking a mean).*

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 for *n\_clusters*. Thus, to find the best value of *n\_clusters*, we must decide over a range of *n\_clusters*.

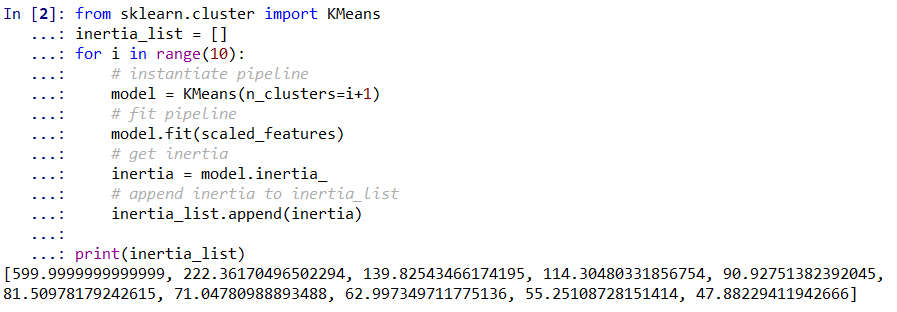
*PowerPoint: Slide 10*

*Section 2c: K-means clustering (tuning n\_clusters)*

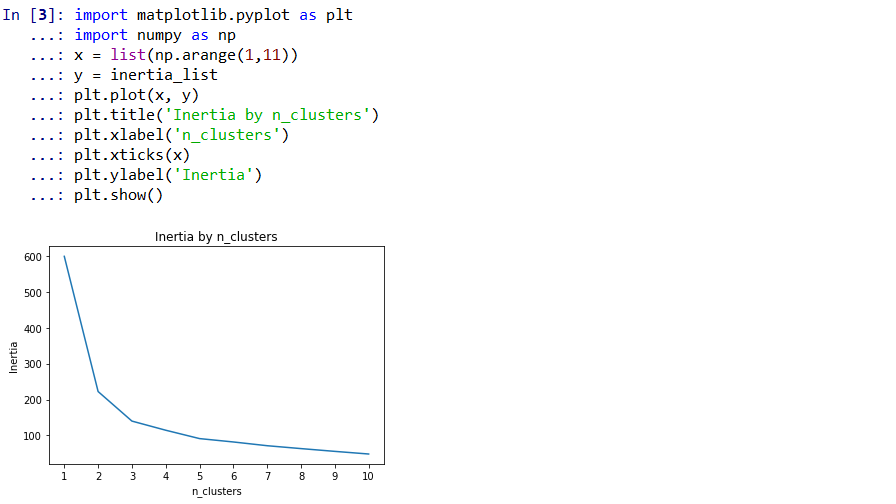
To tune the number of clusters, we must think of an outcome measure that determines how well a k-Means model is performing. In k-Means, model performance is interpreted in terms of sum of squared distances of samples to their closest cluster center, or *inertia*. Thus, to determine which number of clusters 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\_clusters* while 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 *Iris* data.

Step 2: Instantiate an empty list (I.e., *inertia\_list*) for which the inertia values will be stored after every iteration. Build and fit 10 separate k-Means models with *n\_clusters* equaling the number of iterations (I.e., 1-10). After each iteration, return the model’s *inertia\_* argument and append it to the empty *inertia\_list*.



Step 3: Now that we have a list of inertia values for each value of *n\_clusters*, we can plot inertia by number of clusters.



For obvious reasons, the distance 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 drastic and the drop 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: Discuss the elbow method in choosing n\_clusters. Could one make a case for selecting 2 clusters? How about 5 clusters?*

*Discuss: How confident are we that we have chosen the best number of clusters for this data? If we have a great degree of confidence, why? If we feel we could improve our confidence in our choice for n\_clusters how would we go about doing that?*

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.

*PowerPoint: Slide 11*

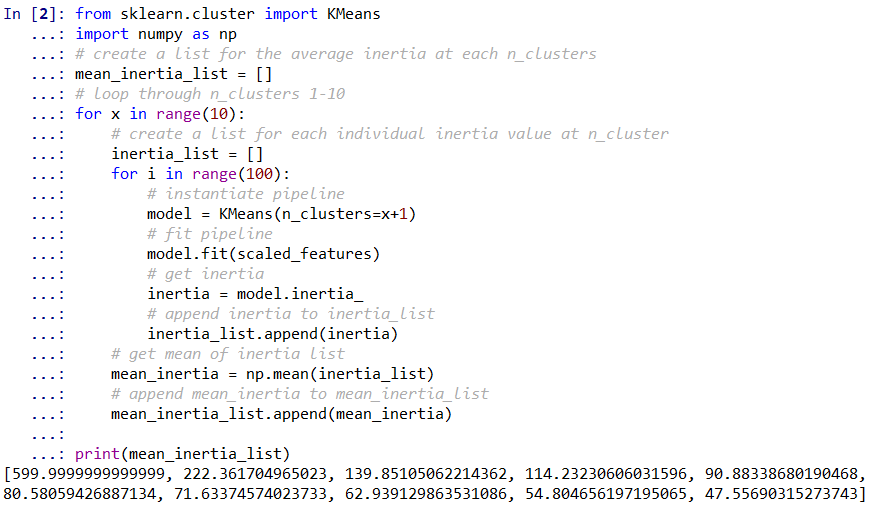
*Section 2d: K-means clustering (tuning n\_clusters using ensembles)*

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 it is done so with random centroids 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 iterate through 10 possible values for *n\_clusters* and 100 models will be built for each number of clusters.

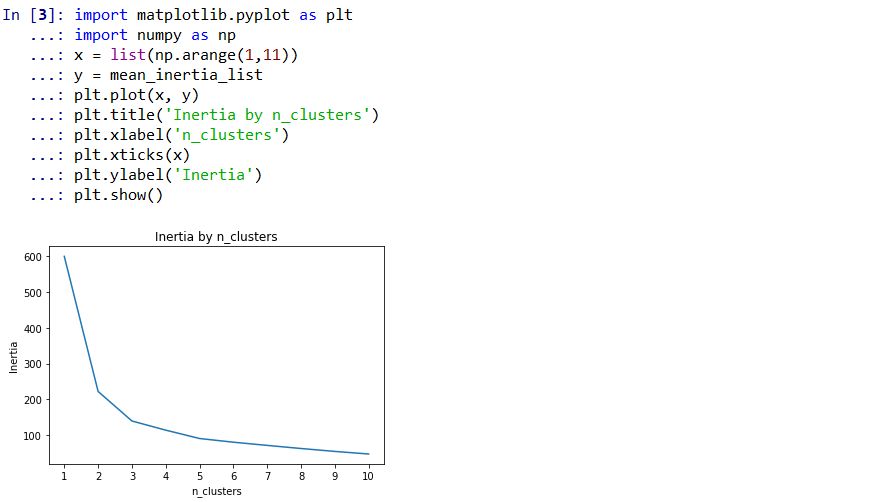
This technique requires a nested loop. The inner loop will run the same model 100 times while the outer loop will determine the number clusters for which the models are run. Thus, 10 different models will be ran 100 times each, resulting in 1000 models.

So, after importing, shuffling, and standardizing the data:

Step 2: Instantiate an empty list *inertia\_list* in the inner-loop so inertia values for each of the 100 models can be appended to it. After each time the inner loop has ran 100 times, the mean inertia value in *inertia\_list* is calculated and appended to *mean\_inertia\_list*; an empty list in the outer loop to contain the mean of 100 models for each value of *n*\_*clusters*.



Step 3: Plot the mean inertia by *n\_clusters*. When we plot these, they will be similar to the values with only one model (Section 2c), but our confidence in them is much higher because we have run each model 100 times for a total of 1000 models.

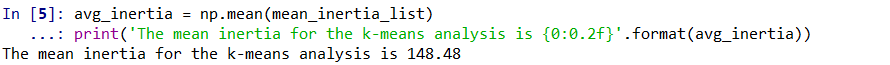


After the nested loop, 3 seems to, again, be the elbow of the plot.

Step 4: Let’s look at the inertia values for 3 and print them to the console.



Step 5: Let’s also look at the mean inertia for this k-means analysis and print it to the console.



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.

*Source: https://scikit-learn.org/stable/modules/clustering.html#k-means*

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.

*Discuss: Discuss the reasons for reducing the number of dimensions in a data set.*

*PowerPoint: Slide 12*

Section 3: Dimensionality Reduction

*Section 3a: Principal component analysis (PCA)*

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.

*PowerPoint: Slide 13*

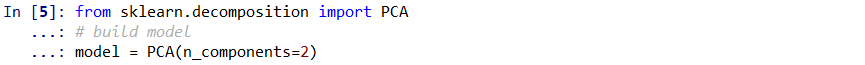
*Sklearn PCA documentation: https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html*

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

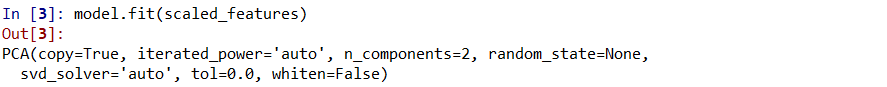
After the *Iris* data has been imported, shuffled, and standardized:

Step 2: Instantiate a *PCA* model with, in this case, 2 components (i.e., *n\_components*). Note that this number can be whatever you want it to be between 1 and the number of features in the data.

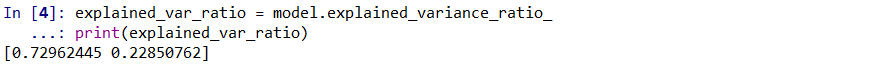


There are as many available *n\_components* as there are features in the data (I.e., in this case, 4). However, by setting *n\_components* to 2, we opt to only return the first 2 principal components. If we do not specify a value for *n\_components* then the maximum number of components is returned.

Step 3: Fit the *PCA* model to *scaled\_features.*

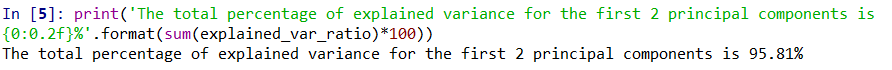


Step 4: Get the proportion of explained variance in the data for each of the 2 components.



From this we can see that the first principal component explains 72.89% of the variance in the data while the second principal component explains 22.85% of the variance in the data.

Step 5: Calculate the total explained variance in the data from the first 2 principal components.



Step 10: Transform the *scaled\_features* data into the first 2 principal components.



The original data with 4 features has been condensed into 2 features which combine to explain 95.81% of the variance in the data. 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.

Note: 95% is a user-defined threshold.

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: Discuss why we would opt to keep 95% of the variance and not 100%*

*Section 3b: Principal component analysis (Tuning n\_components)*

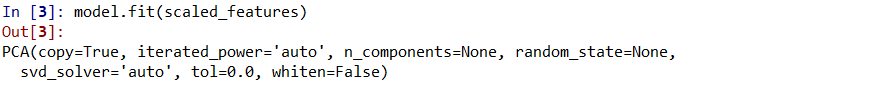
Once the *Iris* data has been imported, shuffled, and standardized we need to return all possible principal components and determine which ones will combine to explain no less than 95% of the explained variance in the data.

*PowerPoint: Slide 14*

Step 2: Instantiate a *PCA* model with no arguments for *n\_components* given. This will return all possible components (i.e., number of features in original data).



Step 3: Fit the model to *scaled\_features* data frame.

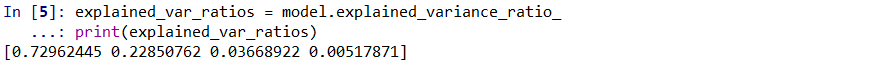


Step 4: Transform the *scaled\_features* into a data frame containing the principal components



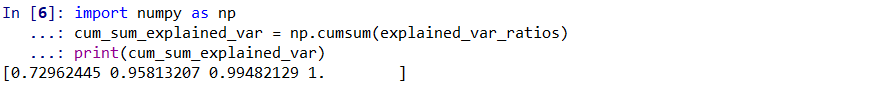
This new data frame, *df\_pca* contains 150 rows and 4 features (I.e., principal components).

Step 5: Return the explained variance ratios for each principal component. Together, these values sum to equal 1.0, or 100% of the explained variance in the data.



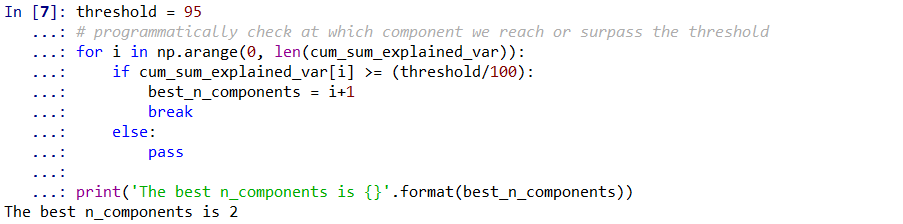
We can demonstrate this by taking a cumulative sum of *explained\_var\_ratios,* saving it is a new list (I.e., *cum\_sum\_explained\_var*), and printing it to the console.

Step 6: Calculate the cumulative sum of explained variance by principal component.



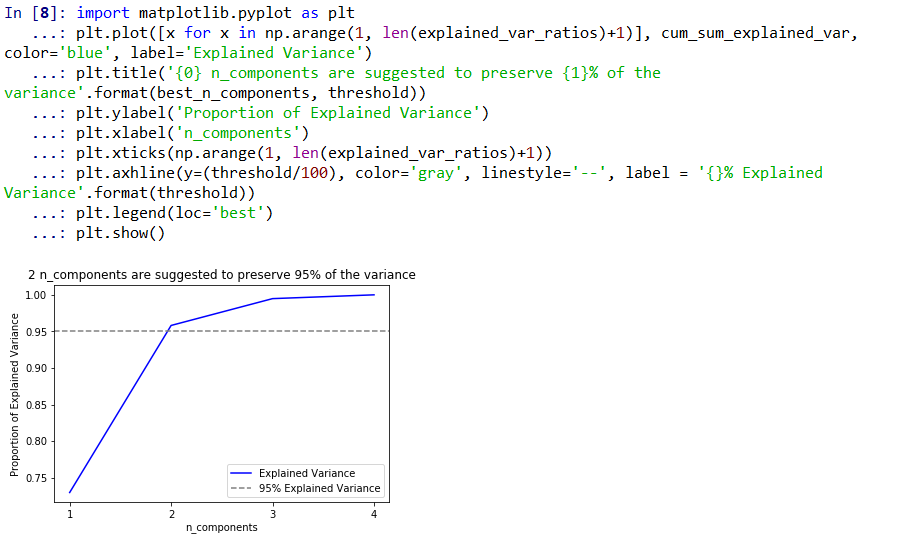
We can easily see that at the second principal component we have a value greater than or equal to 0.95. However, this may be difficult with more features and can be programmatically determined by setting up some logic in python.

Step 7: Setting the threshold to 95%, 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.



Note: we have to add 1 to “i” when we save the object *best\_n\_components* because indexes start at zero. If we did not add 1 to the index, we would return an error because *n\_components* cannot equal 0. Additionally, say we do not encounter an error, our value for *best\_n\_components* will be 1 lower than the actual value.

Step 8: Display this visually by plotting the proportion of explained variance by *n\_components* and include a dotted line for the 95% explained variance threshold to make deciphering the number of components easier.



Note: Notice that the title is programmatic and explains to the user the important information that they will gain from the visualization.

*Discuss: Discuss the importance of a proper title in data visualization. Also discuss how the addition of the dotted threshold line makes interpreting the visualization easier.*

Now, we can refit another *PCA* model with *n\_components* = 2, 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.

*Section 3c: Principal component analysis (evaluating model performance prior-to and after PCA transformation)*

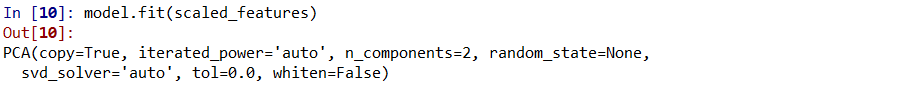
Once the best number of principal components has been determined (I.e., in this case, 2) and saved as an object (I.e., *best\_n\_components*), we can proceed with fitting a new PCA model with *best\_n\_components* as the argument for *n\_components*.

*PowerPoint: Slide 15*

Step 9: Instantiate a new *PCA* model with 2 principal components



Step 10: Fit the model to the *scaled\_features*

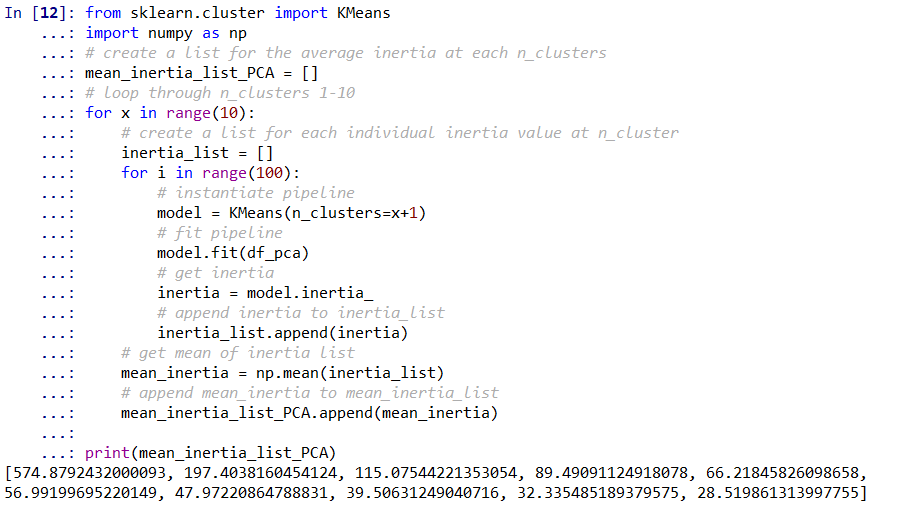


Step 11: Transform the *scaled\_features* into its principal components

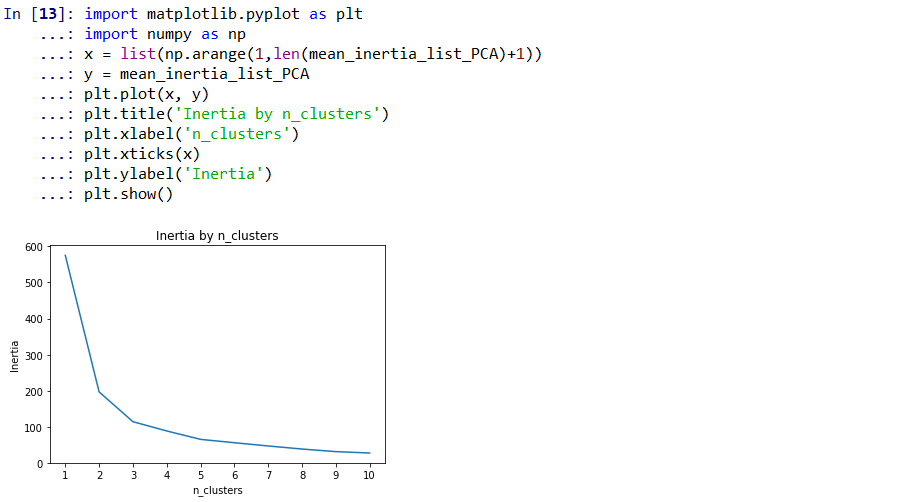


Now, we will use a nested loop as we did in Section 2d to get the mean inertia over 100 k-Means models for each value of *n\_clusters*.

Step 12: Tune *n\_clusters* by fitting 100 models at each possible value of *n\_clusters* from 1 to 10, getting the average inertia over the 100 iterations of each *n\_components*, and appending it to a list.

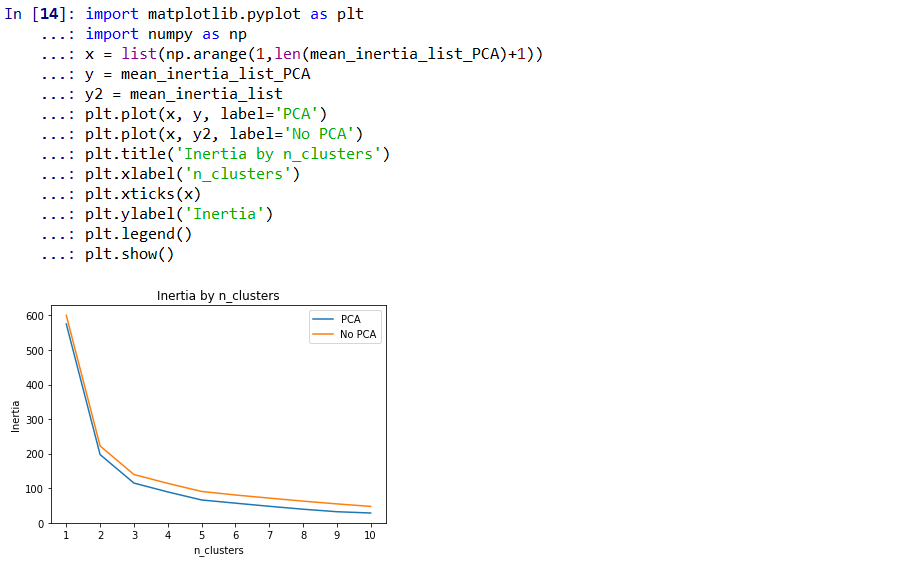


Step 13: Plot average inertia by number of clusters



Again, the elbow of this plot seems to be at *n\_clusters* = 3. However, at 3 clusters, the average inertia is 115.08 after the PCA transformation and was 139.85 before the PCA transformation (see Section 2d). Thus, by reducing our data from 4 features to 3 principal components and reducing the amount of explained variance in our data from 100% to 95%, we were able to produce a k-Means model that better fits our data.

We can visually compare average inertia by *n\_clusters* after PCA transformation and before PCA transformation by plotting both lines on the same plot as follows.



From the plot we can see that inertia is lower in the model using the PCA transformation at every number of clusters. 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 decrease the within-cluster sum of squares (i.e., inertia).

*Discuss: Discuss the impact of a simple, powerful plot like the multi-line plot above. What would you do to improve this plot?*

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.

Section 4: Supervised Linear Discriminant Function Analysis (LDA)

Linear Discriminant Function Analysis (LDA) is described by Field (2013) as the reverse of Multivariate Analysis of Variance (MANOVA). In MANOVA, we are trying to identify the linear variates best differentiating the groups. In LDA, the linear variates are the boundaries used to discern one group from another (Field, 2013).

*PowerPoint: Slide 16*

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

*Discuss: Discuss the relationship between MANOVA and LDA*

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 progressively 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
* Fit our model using a grid search inside of a pipeline

*PowerPoint: Slide 17*

Section 4a: Fitting LDA model with default hyperparameters

To fit the model as a supervised learner using the default parameters of the LDA algorithm we must first import the data and the dependent variable. In the previous sections we did not need to import the dependent variable because the learners were unsupervised.

Step 1: Import the *iris* utils bunch and save it as the object *iris*.



Step 2: Save the features (I.e., *iris.data*) as the data frame *df* and then save the dependent variable (i.e., *iris.target*) as the *target* column in *df*.



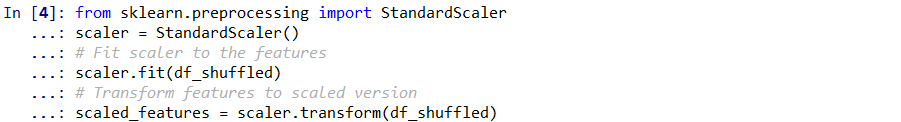
Now that the data is imported, we must preprocess it and get it ready for our model.

Step 3: Shuffle the data to remove any ordering effects.



Step 4: Scale the shuffled features by transforming them into z-scores using *sklearn.preprocessing.StandardScaler*.

*Source: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html*



Step 5: Save *target* as *DV* (I.e., dependent variable) to make the code more elegant and flexible.



Step 6: Split the shuffled data into *X* and *y.*



Note: By saving *target* as *DV* we do not need to write *target* 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 *target* 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.

Step 7: Split *X* and *y* into testing and training data.



Now that the data has been split into testing and training data, we are ready to instantiate our LDA model.

*LDA documentation: https://scikit-learn.org/stable/modules/generated/sklearn.discriminant\_analysis.LinearDiscriminantAnalysis.html*

Step 8: Instantiate the LDA model



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 settings within an algorithm that are set by the user which influence how the model fits and performs on the data. Hyperparameters and their available values are listed in the documentation of every algorithm.

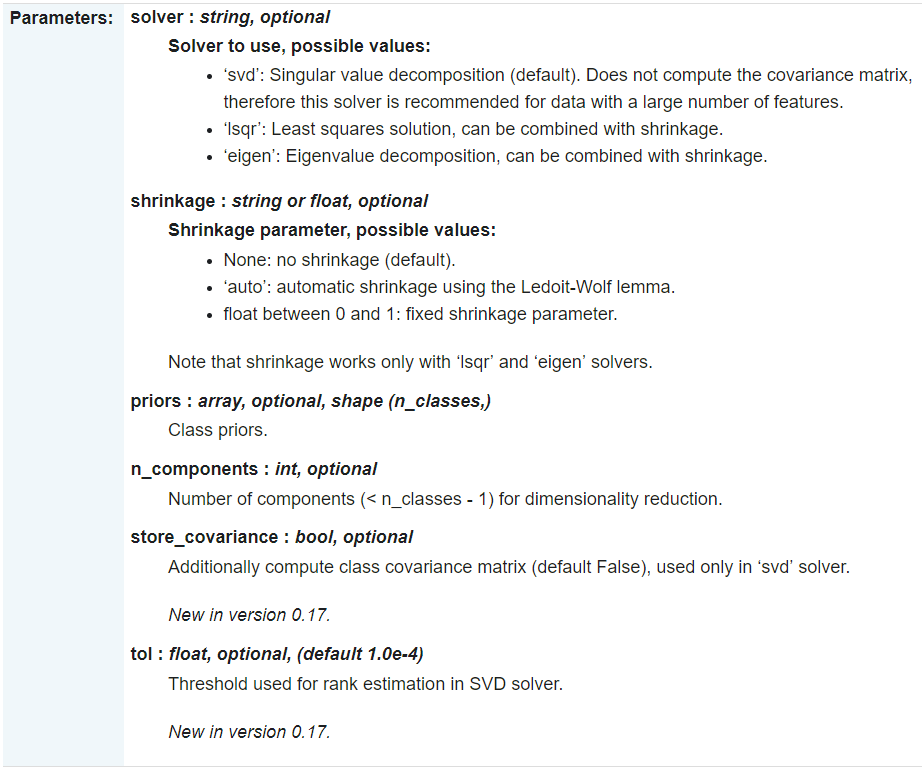


Figure 3. Hyperparameters for *sklearn.discriminant\_analysis.LinearDiscriminantAnalysis*

*PowerPoint: Slide 18*

But how do we know what the default hyperparameters for an algorithm are? Often, 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).

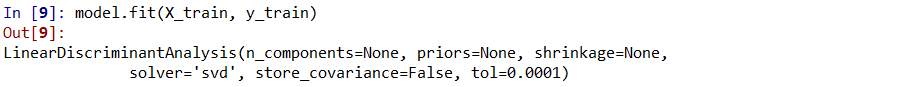


Figure 4. Default hyperparameters for LDA

Thus, by not specifying any hyperparameters in our LDA model we are telling python that we want the ‘svd’ solver, no shrinkage, no priors, no n\_components, do not store the covariance matrix, and a tolerance of 0.0001.

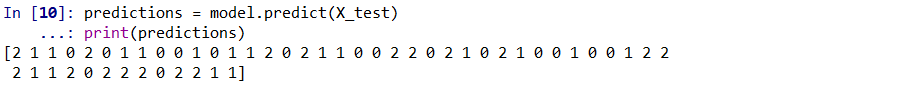
Now that we are aware of how to make sense of the documentation and the default hyperparameters, we can fit our model to the training data.

Step 9: Fit the model to the training data.



Notice in the *out* area of the above screenshot that the specified hyperparameters are the same as those in figure 4.

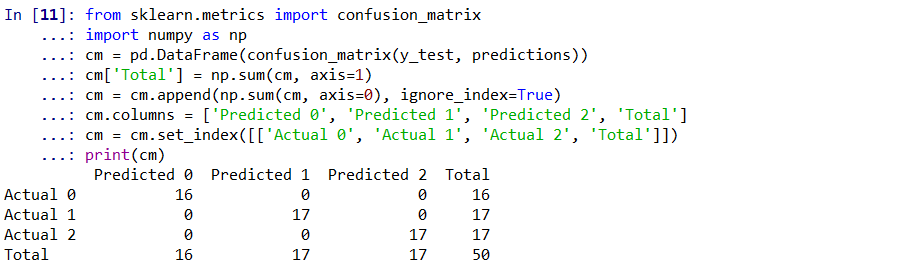
Step 10: Once the model is fit on the training data, we can generate predictions on the test data (I.e., *X\_test*). This will return an array of values that are the predicted class for each observation. Once we have these predictions on the test data, we can see how the model performed by comparing *predictions* to *y\_test*.



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

*Documentation: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion\_matrix.html*

Step 11: Evaluate model performance by comparing the *predictions* to *y\_test*.



Note: The *confusion\_matrix* function, in this case, returns a 3x3 numpy 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 indices. It is easier to interpret this way and can be saved as a csv or excel file with a single line of code.

To interpret this output we need to think in terms of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN). A TP is an observation that was predicted to be in a certain class and is actually in that class (I.e., a correct prediction). A FP is an observation that was predicted to be in a class but is actually not in that class (I.e., an incorrect prediction). A TN is an observation that is predicted not to be in a certain class and is actually not in that class (I.e., a correct prediction). Lastly, a FN is an observation that is predicted not to be in a certain class and is actually in that class (I.e., an incorrect prediction). TP, FP, TN, and FN are necessary components for calculating more evaluation metrics that will be discussed later.

From the 3x3 confusion matrix, we can see that of the 16 observations that were in class *0*, all 16 were predicted to be in class *0*. Of the 17 observations that were in class *1*, all 17 were predicted to be in class *0*. Of the 17 observations that were in class *2*, all 17 were predicted to be in class *2*.

LDA’s default settings predicted with 100% accuracy on this data. However, this would be almost unheard of in real life and, often, the best combination of model hyperparameters must be found. This is referred to as tuning the model. Scikit learn provides an outstanding library for tuning models called *GridSearchCV*.

*Documentation: https://scikit-learn.org/stable/modules/generated/sklearn.model\_selection.GridSearchCV.html*

Section 4b: Tuning LDA hyperparameters using GridSearchCV

What is a grid search? Grid search builds a model for all combinations of hyperparameters and evaluates each model’s performance. 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.

*PowerPoint: Slide 19*

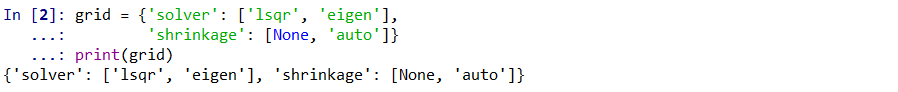
*Discuss: Discuss why tuning a model is important to model performance*

*Discuss: Discuss the pros and cons of using a grid search to tune hyperparameters.*

Alright, let’s tune our LDA model using the *GridSearchCV* library.

After the data has been imported, shuffled, scaled, and split into testing and training data:

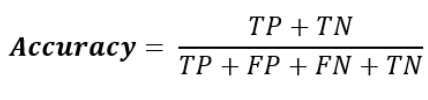
Step 2: Instantiate a grid with the possible values for each hyperparameter as key value pairs in a dictionary.



Here, we are providing 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.

*Documentation: https://scikit-learn.org/stable/modules/model\_evaluation.html#scoring-parameter*

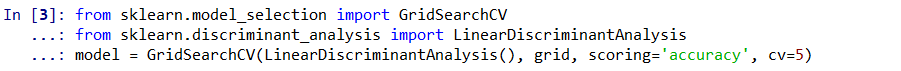
Here, we have specified that we want the model with the combination of hyperparameters that will optimize accuracy. Accuracy is defined by:



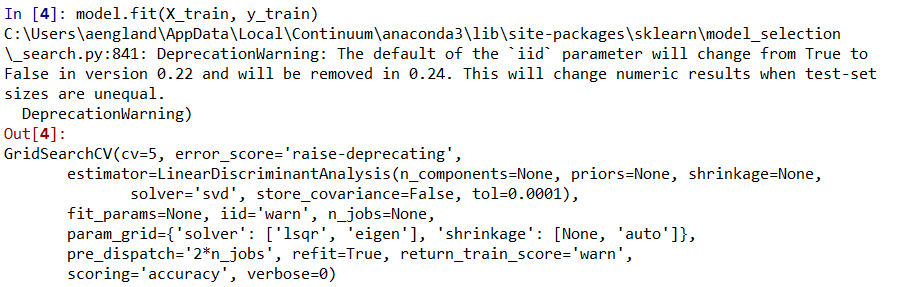
Equation 4. Accuracy.

Thus, accuracy is the proportion of predictions that were correctly classified.

Step 3: Instantiate a *GridSearchCV* model with *accuracy* as the scoring outcome for which we will be maximizing.

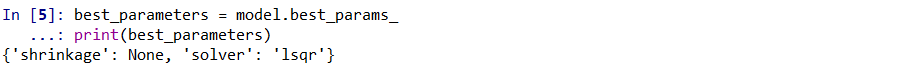


Step 4: Now that the GridSearchCV model has been instantiated, we must fit the model to the training data.

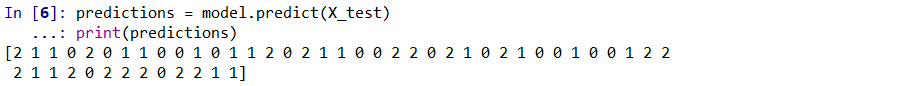


From the grid search, we can see that the hyperparameters that optimized accuracy were no shrinkage and an *lsqr* solver. However, there is an easier way to find these by using the *best\_params\_* attribute from *GridSearchCV*.

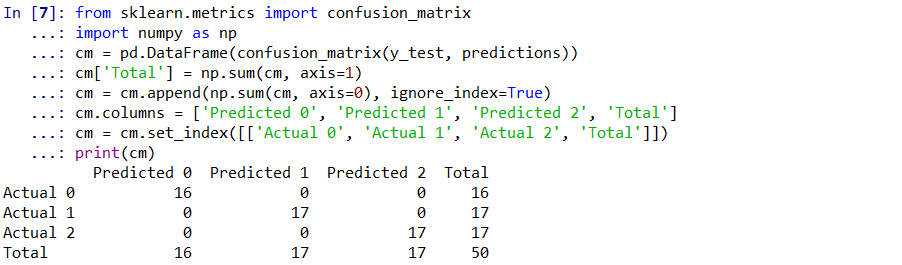
Step 5: Save the best parameters as a dictionary and print them to the console.



Step 6: Generate predictions on the test data.



Step 7: Evaluate performance by comparing the predictions to *y\_test* using a confusion matrix.



When we tuned the model, we were returned the model with the best combination of hyperparameters to maximize the accuracy of the model. To do this, we had to instantiate a grid dictionary and fit a *GridSearchCV* model. In the next example we will look at how to tune a model using a pipeline which helps condense a couple steps. It is important to note that when using a GridSearchCV model, predictions can be generated but some of the attributes associated with the estimator cannot. Thus, we will discover how to programmatically access the hyperparameters from the *best\_params\_* attribute and use them in our estimator, allowing the user to access all attributes the algorithm returns.

Note: Fitting a model using GridSearchCV can be extremely time consuming if the data contains a large number of 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.

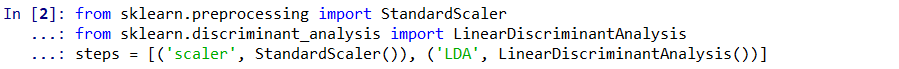
Section 4c: Tuning models in a pipeline

To make our code more elegant and concise by condensing some of the steps (I.e., imputing, scaling, and tuning) we can use a pipeline.

*PowerPoint: Slide 20*

After importing, shuffling, and splitting into testing and training data:

Step 2: Set up the *steps* for a pipeline. Notice we skipped the scaling step from previous pages and will instead opt to scale inside of the pipeline.



Step 3: Set-up the pipeline.



Step 4: Specify the hyperparameter space.

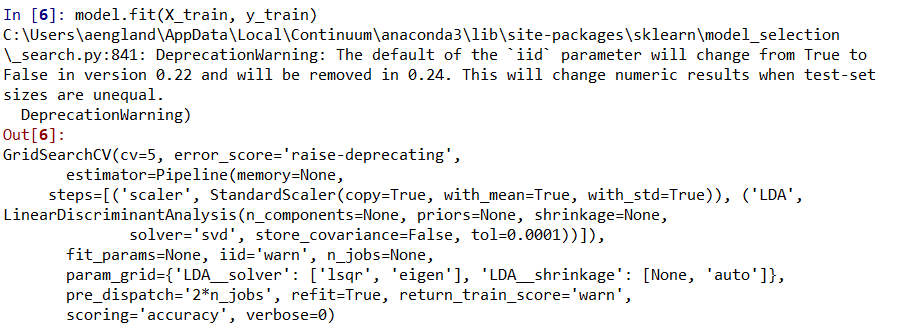


Note: When tuning a model using a pipeline, the estimator name from step 2 (*LDA,* in this case) must be followed by 2 underscores followed by the hyperparameter name.

Step 5: Instantiate a GridSearchCV model.

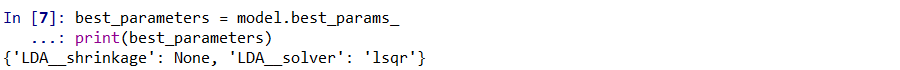


Step 6: Fit the model on the training data.



Note: This model was fit using GridSearchCV, so it is not an LDA model. Thus, not all attribute offered by LDA can be returned. To access the attributes offered by LDA, we must fit an LDA model. But how do we make sure the best hyperparameters are chosen for this model? One option is to copy and paste the hyperparameters inside the parenthesis following the name of the estimator (I.e., *LinearDiscriminantAnalysis*). A better option is to copy the value from the key stored in a dictionary from the GridSearchCV attribute *best\_params\_*.

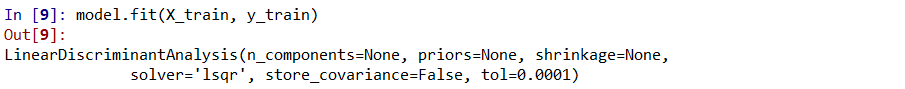
Step 7: Return the best hyperparameters and save it as a dictionary.



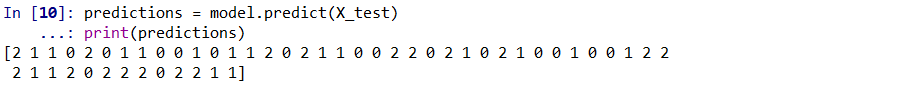
Step 8: Instantiate a *LinearDiscriminantAnalysis* model with the hyperparameters specified by the key value pairs stored in the *best\_paramters* object.



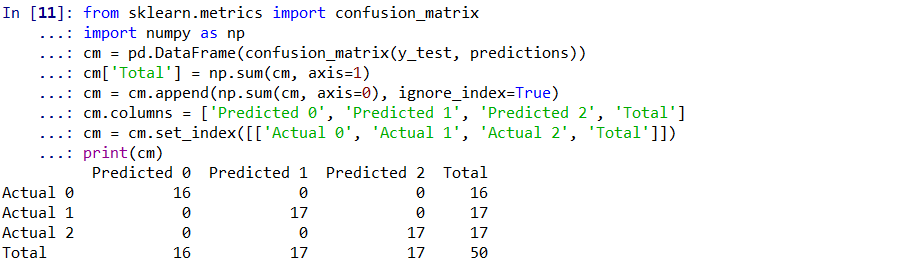
Step 9: Fit the model on the training data.



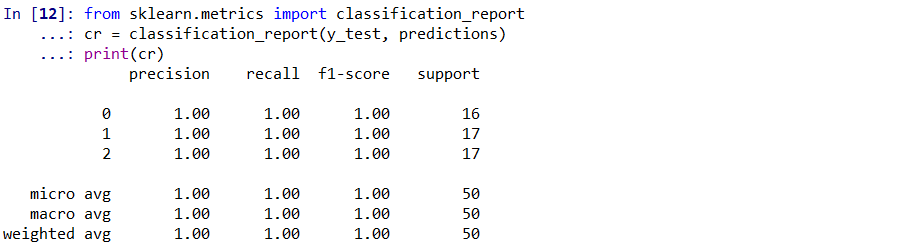
Step 10: Generate predictions from the model.



Step 11: Evaluate model performance using a confusion matrix.

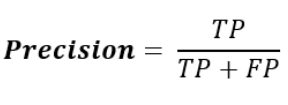


Step 12: Evaluate model performance with a classification report.

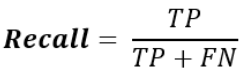


Classification reports provide the metrics for *precision*, *recall*, and *f1-score.*

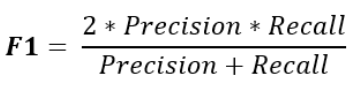
The formulas for precision, recall, and f1-score are as follows:



Equation 5. Precision



Equation 6. Recall



Equation 7. F1-score

Conclusion

This chapter introduced the reader to two widely-used unsupervised, clustering algorithms, *HCA* and *k-Means clustering*. Additionally, the reader was walked through an example of *PCA* for dimension reduction and improving model performance. Lastly, *LDA* was introduced as a supervised learning algorithm and the reader was taught how to tune algorithms using *GridSearchCV* in and out of a pipeline.