4

Dimensionality Reduction and Unsupervised Learning

Present 1-2: The title slide and the lesson slide. Slide 3 contains the learning objectives.

Lesson Objectives:

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

* 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
* Perform supervised dimension compression using Linear Discriminant Function Analysis (LDA)

This lesson will cover the various concepts that fall under dimensionality reduction and unsupervised learning.

Introduction

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.

Note

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 descriptive or predictive 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 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 for supervised dimensionality reduction

Hierarchical Cluster Analysis (HCA)

Present 4+: Slides 4-5 provides explains 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. Thus, it is a common approach to use HCA as a precursor to other clustering techniques where a predetermined number of clusters is recommended. HCA works by merging observations that are most similar into clusters and continues merging clusters that are closest in proximity until all observations are merged into a single cluster.

HCA determines similarity as the Euclidean distance between and among observations 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:

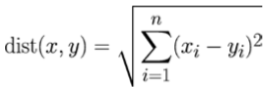


Figure 4.1: Euclidean Distance

After the distance between observations and cluster 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 is displayed below:

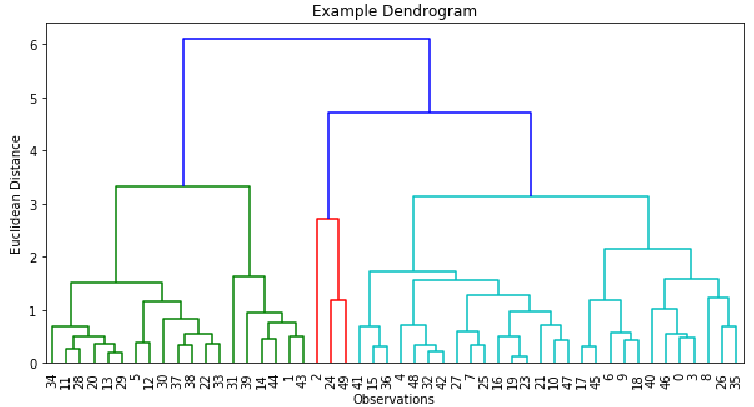


Figure 4.2: Example dendrogram

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 shorter distance and a subsequent closer relationship. In this example, there appears to be 3 clusters. The first cluster includes the observations colored in green, the second cluster includes the observations colored in red, and the third cluster includes the observations colored in turquoise.

Exercise 1: Building HCA Model

Present 6: Building HCA Model

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 218 observations and 9 features corresponding to the percent weight of various oxides found in glass:

* 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

In this exercise, we will be using refractive index (RI) and the various weight percent in each oxide to segment glass type.

1. To get started, we will import pandas and read the glass.csv file using the following code:

import pandas as pd

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

1. Look for some basic data frame information by printing df.info() to the console using print(df.info()).

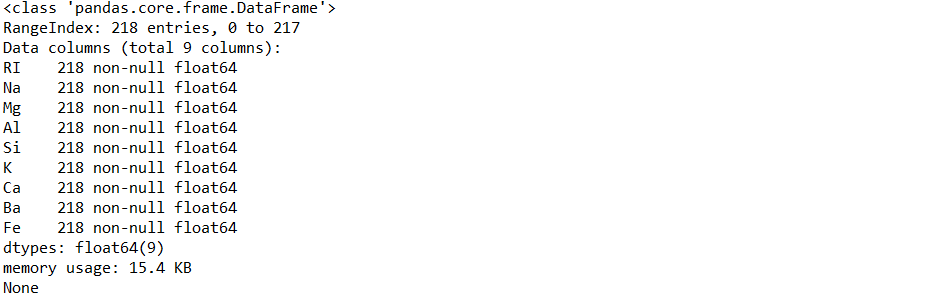


Figure 4.3: DataFrame information

1. To remove any possible order effects in the data, we will shuffle the rows prior to building any models and save it as a new data frame object as follows:

from sklearn.utils import shuffle

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

1. Transform each observation into a z-score by fitting and transforming the shuffled data using:

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

scaled\_features = scaler.fit\_transform(df\_shuffled)

1. Perform hierarchical clustering using the linkage function on scaled\_features. The below code will show you how:

from scipy.cluster.hierarchy import linkage

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

Discuss 7: In the preceding screenshot, how many rows and columns do we have in our data?

Answer: There are 218 entries (I.e., rows) and 9 columns.

Exercise 2: Plotting HCA Model and Assigning Predictions

Present 8: Plotting HCA Model and Assigning Predictions

Now that the HCA model has been built, we will continue with the analysis by visualizing the clusters using a dendrogram and using the visualization to generate predictions.

1. Display the dendrogram by plotting the linkage model as follows:

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()

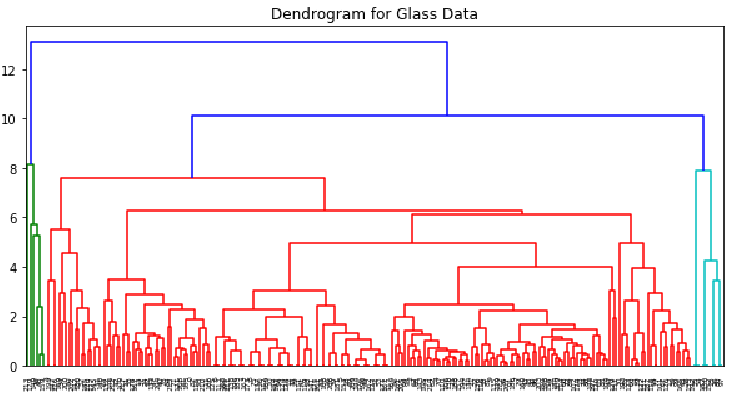


Figure 4.4: Dendogram for glass data

Note:

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.

Now that we have the predicted clusters of observations, we can use the fcluster function to generate an array of labels that correspond to the rows in df\_shuffled.

1. Generate the predicted labels of the cluster for which an observation belongs using the following code:

from scipy.cluster.hierarchy import fcluster

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

Instructor Note:

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

1. Add the labels array as a column in the shuffled data and preview the first 5 rows using the following code

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

print(df\_shuffled.head(5))

1. Check the output in the following figure.

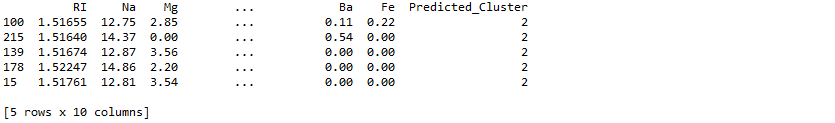


Figure 4.5: First 5 rows of df\_shuffled after predictions have been matched to observations.

Discuss 9: Why must we refrain from adding labels as a column to any data frames generated before being shuffled?

Answer: We will not 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.

We have successfully learned the differences between supervised and unsupervised learning, how to build an HCA model, how to visualize and interpret the HCA dendrogram, and how to assign the predicted cluster label to the appropriate observation.

Here we have utilized HCA to cluster our data into three groups and matched the observations with their predicted cluster. Some strengths of HCA models include:

* Easy to build
* No need to specify the number of clusters in advance
* Visualizations are easy to interpret

However, some drawbacks of HCA include:

* Vagueness in terms of the termination criteria (I.e., when to finalize the number of clusters)
* The algorithm cannot adjust once the clustering decisions have been made.
* 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 more computationally frugal than HCA.

K-Means Clustering

Present 10+: Slides 10-11 provide a brief explanation on how k-Means works

Like HCA, K-Means also uses distance to assign observations into clusters not labeled in the data. However, rather than linking observations to each other as in HCA, k-Means assigns observations to *k* (user-defined number of) clusters.

To determine the cluster to which each observation belongs, *k* cluster centers are randomly generated, and observations are assigned to the cluster in which its Euclidean distance is closest to the cluster center. Like the starting weights in artificial neural networks, cluster centers are initialized at random. After the cluster centers have been randomly generated there are two phases:

* Assignment phase
* 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 because results vary between fitting the same model on the same data and it is not guaranteed to assign observations into the appropriate cluster, 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, as shown in the following figure.

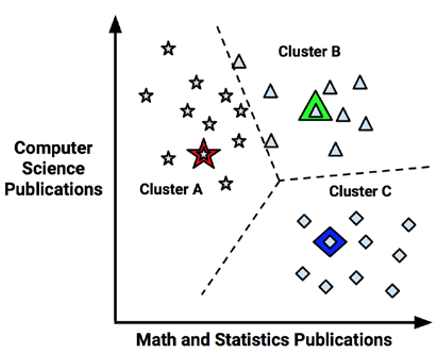


Figure 4.6: A scatterplot of observations and the cluster centers as denoted by the star, triangle, and diamond.

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, as shown in the following figure.

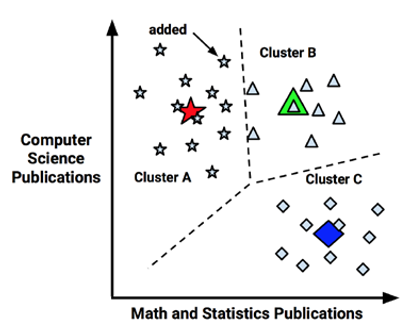


Figure 4.7: Shifting of the cluster centers to the cluster centroid.

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. This is shown in the following figure:

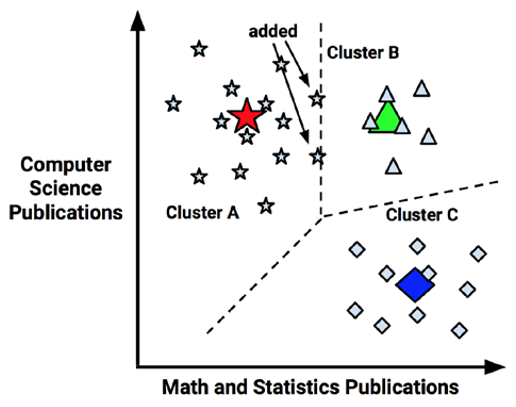


Figure 4.8: Updating of the centroids after observation reassignment.

This process of updating centroids continues until there are no further observation reassignments. The final centroid is as shown in the following figure:

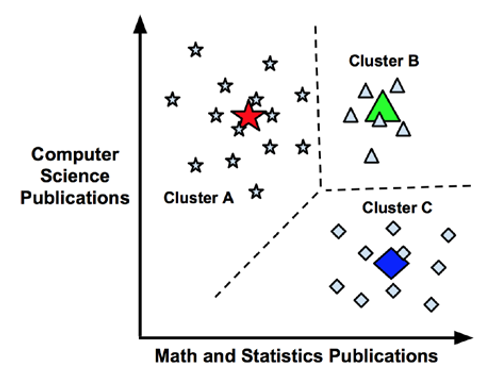


Figure 4.9: Final centroid position and cluster assignments.

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 mean *inertia*, or within-cluster sum-of-squares, by number of clusters and finding the point at which there are diminishing returns in inertia by adding more clusters.

Exercise 3: Fitting k-Means Model and Assigning Predictions

Present 12: Fitting k-Means Model and Assigning Predictions

Since our data has already been prepared (see *Exercise 1*) and we understand the concepts behind the k-Means algorithm, we will learn how easy it is to fit a k-Means model, generate predictions, and assign those predictions to the appropriate observation.

After the glass data set has been imported, shuffled, and standardized:

1. Instantiate a KMeans model with an arbitrary number of, in this case, 2 clusters., as follows:

from sklearn.cluster import KMeans

model = KMeans(n\_clusters=2)

1. Fit the model to scaled\_features using the following line of code:

model.fit(scaled\_features)

1. Save the cluster labels from our model into the array, labels, using labels = model.labels\_.
2. Generate a frequency table of the labels.

import pandas as pd

pd.value\_counts(labels)

To get a better idea, refer to the following screenshot:



Figure 4.10: Frequency table of two clusters

Using 2 clusters, 61 observations were placed into the first cluster and 157 observations were grouped into the second cluster.

1. Add the labels array as the column ‘Predicted Cluster’ into the df\_shuffled data frame and preview the first 5 rows using the following code:

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

print(df\_shuffled.head(5))

1. Check the output in the following figure:

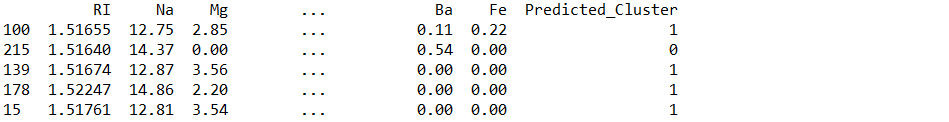


Figure 4.11: First 5 rows of df\_shuffled

Discuss 13: How confident are we in these predictions?

Answer: We are not very confident in these predictions because the predicted labels change each time the model is run.

Activity 1: Ensemble k-Means Clustering and Calculating Predictions

Present 14: Ensemble k-Means Clustering and Assigning Predictions

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 identical models on the same data may result in different conclusions, limiting the confidence we have in our predictions. It is advised to run these 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.

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 as a new column at each iteration.
3. Calculate the mode for each row in labels\_df and save it as a new column in labels\_df.

Solution:

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

1. Instantiate an empty data frame for which to append each model’s and save it as the new data frame object labels\_df with the following code:

import pandas as pd

labels\_df = pd.DataFrame()

1. Import the KMeans function outside of the loop using from sklearn.cluster import KMeans
2. Complete 100 iterations as follows.

for i in range(0, 100):

1. Save a KMeans model object with 2 clusters (arbitrarily decided upon, a priori) using:

model = KMeans(n\_clusters=2)

1. Fit the model to scaled\_features using model.fit(scaled\_features)
2. Generate the labels array and save it as the labels object as follows:

labels = model.labels\_

1. Store labels as a column in labels\_df named after the iteration using the code:

labels\_df['Model\_{}\_Labels'.format(i+1)] = labels

1. After labels have been generated for each of the 100 models (see activity 1), calculate the mode for each row using row\_mode = labels\_df.mode(axis=1).
2. Assign row\_mode to a new column in labels\_df, as shown in the following code:

labels\_df['row\_mode'] = row\_mode

1. View the first 5 rows of labels\_df

print(labels\_df.head(5))

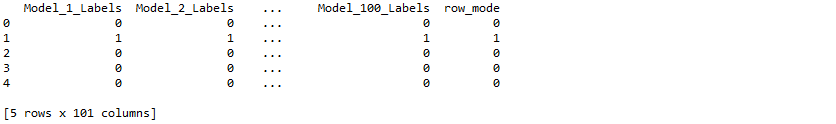


Figure 4.x: First 5 rows of labels\_df

Discuss 15: How confident are we in the predictions now that we have computed them from 100 models as opposed to 1?

Answer: We are much more confident in the predictions after running 100 models relative to a single model.

We have drastically increased the confidence in our predictions by iterating through numerous models, saving the predictions at each iteration, and assigning the final predictions as the mode of those predictions. However, these predictions were generated by models using a predetermined number of clusters. Unless we know the number of clusters *a priori*, we will want to discover the optimal number of cluster for which to segment our observations.

Exercise 4: Calculating Mean Inertia by n\_clusters

Present 16: Calculating Mean Inertia by n\_clusters

The k-Means algorithm groups observations into clusters by minimizing the within-cluster sum-of-squares, or inertia. Thus, to improve our confidence in the tuned number of clusters for our k-Means model, we will place the loop we created in Activity 1 (with a few minor adjustments) inside of another loop which will iterate through a range of n\_clusters. This creates a nested loop which iterates through 10 possible values for n\_clusters and build 100 models at each iteration. At each of the 100 inner iterations, model inertia will be calculated. For each of the 10 outer iterations, mean inertia over the 100 models will be computed. Resulting in the mean inertia value 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 as shown here:

from sklearn.cluster import KMeans

import numpy as np

1. 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 as shown here:

inertia\_list = []

1. In the for loop, we will iterate through 100 models using the following code.

for i in range(100):

1. Inside the loop, build KMeans model with n\_clusters=x as follows:

model = KMeans(n\_clusters=x)

Note

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

1. Fit the model to scaled\_features as shown here:

model.fit(scaled\_features)

1. Get the inertia value and save it to the object inertia as follows:

inertia = model.inertia\_

1. Append inertia to inertia\_list using inertia\_list.append(inertia)
2. Moving to the outside loop, instantiate another empty list to store the average inertia values, as shown here:

mean\_inertia\_list = []

1. Iterate through the values 1 through 10 for n\_clusters using the code below:

for x in range(1, 11):

1. After the inside for loop has run through 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 as the object, mean\_inertia as follows:

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

1. Append mean\_inertia to mean\_inertia\_list as shown here.

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

1. 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.
2. Print mean\_inertia\_list as shown in the following code. The values are shown in the following figure:

print(mean\_inertia\_list)



Figure 4.12: mean\_inertia\_list

Exercise 5: Plotting Mean Inertia by n\_clusters

Present 17: Plotting Mean Inertia by n\_clusters

Now that we have generated mean inertia over 100 models for each value of n\_clusters, we will plot mean inertia by n\_clusters. Then, we will discuss how to visually assess the best value to use for n\_clusters.

1. First, import matplotlib as follows:

import matplotlib.pyplot as plt

1. Create a list of numbers and save it as the object x so we can plot it on the x-axis as shown here.

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

1. Save mean\_inertia\_list as the object y as shown here.

y = mean\_inertia\_list

1. Plot mean inertia by number of clusters as follows:

plt.plot(x, y)

1. Set the plot title to read ‘Mean Inertia by n\_clusters’ using the following:

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

1. Label the x-axis ‘n\_clusters’ using plt.xlabel(‘n\_clusters’) and label the y-axis ‘Mean Inertia’ using plt.ylabel(‘Mean Inertia’).
2. Set the tick labels on the x-axis as the values in x using plt.xticks(x).
3. Display the plot used in plt.show(). To better understand, refer to the following code:

plt.plot(x, y)

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

plt.xlabel('n\_clusters')

plt.xticks(x)

plt.ylabel('Mean Inertia')

plt.show()

For the resultant output, refer to the following screenshot:

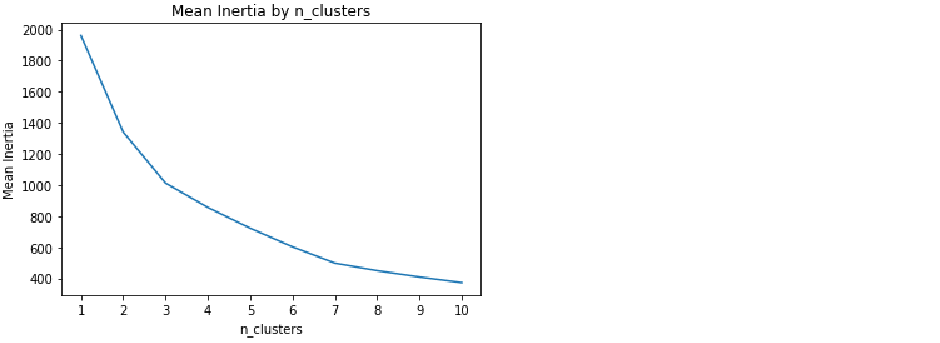


Figure 4.13: Mean inertia by n\_clusters

To determine the best number of n\_clusters we will use the “elbow method.” That is, the point in the plot where there are diminishing returns for the added complexity of more clusters. From Figure 4.13, we can see that there are rapid decreases in mean inertia from n\_clusters 1 to 3. After n\_clusters equals 3, the decreases in mean inertia seem to become less rapid and the decreases in inertia may not be worth the added complexity of adding additional clusters. Thus, the appropriate number of n\_clusters in this situation is 3.

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.

Discuss 18: Now that we know the best value for n\_clusters, how might we improve upon this model?

Answer: We can improve upon this model by using a dimensionality reduction technique prior to building the k-Means model.

Principal Component Analysis (PCA)

Present 19+: Slides 19-20 introduces 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:

* Fit PCA model with all 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 6: Fitting PCA Model

Present 21: Fitting PCA Model

Using the data we prepared in *Exercise 1* and the brief explanation of PCA, in this exercise we will learn to fit a generic PCA model.

1. Instantiate a PCA model as shown here.

from sklearn.decomposition import PCA

model = PCA()

1. Fit the PCA model to scaled\_features as shown in the following code.

model.fit(scaled\_features)

1. Get the proportion of explained variance in the data for each component, save the array as the object explained\_var\_ratio, and print the values to the console as follows:

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

print(explained\_var\_ratio)

1. For the resultant output, refer to the following screenshot:



Figure 4.14: Explained variance in the data for each principal component

Each principal component explains a proportion of the variance in the data. In this exercise, the first principal component explained .35 of the variance in the data, the second explained .25, the third .13%, and so on and so forth. Altogether, these 9 components explain 100% of the variance in the data. The goal of dimensionality reduction is to decrease the number of dimensions in the data with the objectives of limiting overfitting and time to fit the subsequent model. Thus, we will not keep all 9 components. However, if we retain too few components the percent of explained variance in the data will be low and the subsequent model will under fit. Therefore, a challenge for data scientists exists in determining the number of n\_components that minimize over fitting and under fitting.

Exercise 7: Choosing n\_components using Threshold of Explained Variance

Present 22: Choosing n\_components using Threshold of Explained Variance

In Exercise 6, we learned to fit a PCA model with all available principal components. However, keeping all of the principal components does not reduce the number of dimensions in the data. In this exercise, we will reduce the number of dimensions in the data by retaining the components that explain a threshold of variance in the data.

1. Determine the number of principal components at which a minimum of 95% of the variance in the data is explained by calculating the cumulative sum of explained variance by principal component. Let’s have a look at the following code, to see how it’s done.

import numpy as np

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

print(cum\_sum\_explained\_var)

For the resultant output, refer to the following screenshot:



Figure 4.15: Cumulative sum of the explained variance for each principal component

1. Set the threshold for percent of variance to keep in the data as 95% as follows.

threshold = .95

1. 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)):.
2. Check to see if the item in cum\_sum\_explained\_var is greater than or equal to 0.95, as shown here.

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

1. 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. Look at the following code to get an idea.

best\_n\_components = i+1

break

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

else:

pass

1. Print a message detailing the best number of components using the following code:

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

View the output from the previous line of code:



Figure 4.15: Output message displaying number of components

The value for best\_n\_components is 6. 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-Means model 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

Present 24: Evaluating Mean Inertia by Cluster after PCA Transformation

Now that we know the number of components to retain at least 95% of the variance in the data, how to transform our features into principal components, and a way to tune the optimal number of clusters for k-Means clustering with a nested loop, we will put them all together in this activity.

Continuing from Exercise 7:

1. Instantiate a PCA model with the value for the n\_components argument equal to best\_n\_components (I.e., remember, best\_n\_components = 6).
2. Fit the model to scaled\_features and transform it into the first 6 principal components
3. Using a nested loop, calculate the mean inertia over 100 models at values 1 through 10 for n\_clusters (see Exercise 7).

Solution:

1. Instantiate a PCA model with the value for the n\_components argument equal to best\_n\_components (I.e., remember, best\_n\_components = 6) as follows:

from sklearn.decomposition import PCA

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

1. Fit the model to scaled\_features and transform them into the 6 components as shown here.

df\_pca = model.fit\_transform(scaled\_features)

1. Import numpy and the KMeans function outside the loop using the following code:

from sklearn.cluster import KMeans

import numpy as np

1. Instantiate an empty list, inertia\_list, for which we will be appending inertia values after each iteration using the following code.

inertia\_list = []

1. In the inside for loop, we will iterate through 100 models as follows:

for i in range(100):

1. Build our KMeans model with n\_clusters=x using:

model = KMeans(n\_clusters=x)

Note

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

1. Fit the model to df\_pca as follows:

model.fit(df\_pca)

1. Get the inertia value and save it to the object inertia using inertia = model.inertia\_
2. Append inertia to inertia\_list using the following code:

inertia\_list.append(inertia)

1. Moving to the outside loop, instantiate another empty list to store the average inertia values using the following code.

mean\_inertia\_list\_PCA = []

1. Since we want to check the average inertia over 100 models for n\_clusters 1 through 10 we will instantiate the outer loop as follows:

for x in range(1, 11):

1. 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 the following code:

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

1. Append mean\_inertia to mean\_inertia\_list\_PCA using the following code:

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

1. Print mean\_inertia\_list\_PCA to the console using print(mean\_inertia\_list\_PCA). Notice the output in the following screenshot:



Figure 4.x: mean\_inertia\_list\_PCA

Now, much like in Exercise 5, we have a mean inertia value for each value of n\_clusters (1 through 10). However, mean\_inertia\_list\_PCA contains the mean inertia value for each value of n\_clusters after PCA transformation. But, how do we know if the k-Means model performs better after PCA transformation? In the next exercise, we will visually compare the mean inertia values before and after PCA transformation at each value of n\_clusters.

Exercise 8: Visual Comparison of Inertia by n\_clusters

Present 25: Visual Comparison of Inertia by n\_clusters

To visually compare mean inertia by n\_clusters before and after PCA transformation we will slightly modify the plot created in Exercise 5 by:

* Adding a second line to the plot showing mean inertia by n\_clusters after PCA transformation
* Creating a legend distinguishing the lines
* Changing title

Note:

For this visualization to work properly, mean\_inertia\_list from exercise 5 must still be in the environment.

Continuing from Activity 2:

1. Import matplotlib using import matplotlib.pyplot as plt.
2. Create a list of numbers and save it as the object x so we can plot it on the x-axis as follows:

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

1. Save mean\_inertia\_list\_PCA as the object y using the following code:

y = mean\_inertia\_list\_PCA

1. Save mean\_inertia\_list as the object y2 using y2 = mean\_inertia\_list
2. Plot mean inertia after PCA transformation by number of clusters using the following code:

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, label=’No PCA).

1. Set the plot title to read ‘Mean Inertia by n\_clusters for Original Features and PCA Transformed Features’ as follows:

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

1. Label the x-axis ‘n\_clusters’ using the following code:

plt.xlabel(‘n\_clusters’)

1. Label the y-axis ‘Mean Inertia’ using:

plt.ylabel(‘Mean Inertia’)

1. Set the tick labels on the x-axis as the values in x using plt.xticks(x).
2. Show a legend using and display the plot as follows:

plt.legend()

plt.show()

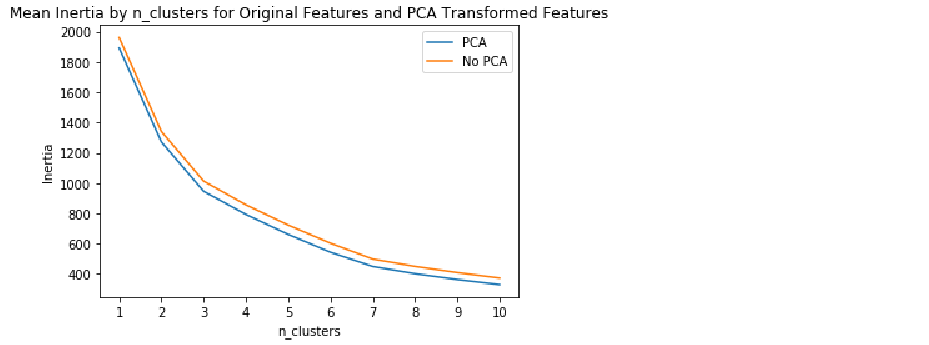


Figure 4.16: Mean inertia by n\_clusters for original features (orange) and PCA transformed features (blue).

From the plot we can see that inertia is lower at every number of clusters in the model using the PCA transformed features. 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 PCA transformation 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).

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 in an unsupervised fashion. Linear discriminant function analysis (LDA), on the other hand, is a supervised method for reducing the number of dimensions via data compression.

Supervised Data Compression using Linear Discriminant Function Analysis (LDA)

Present 25+: Slides 25-26 Linear Discriminant Function Analysis

As discussed previously, PCA transforms features into a set of variables to maximize the variance among the features. In PCA, the output labels are not considered when fitting the model. Meanwhile, LDA uses the dependent variable to help compress the data into features that best discriminate the classes of the outcome variable. In this section, we will walk through how to use LDA as a supervised data compression technique.

To demonstrate using LDA as supervised dimensionality compression technique we will:

* Fit an LDA model with all possible n\_components
* Transform our features to the n\_components
* Tune the number of n\_components

Exercise 9: Fitting LDA Model

Present 27: Fitting LDA Model

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 processed, and headlamps, respectively.

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

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 as follows:

from sklearn.utils import shuffle

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

1. Save 'Type’ as DV (I.e., dependent variable) as follows:

DV = 'Type'

1. Split the shuffled data into features (i.e., X) and outcome (i.e., y) using X = df\_shuffled.drop(DV, axis=1) and y = df\_shuffled[DV], respectively.
2. X

X

1. Scale X\_train and X\_test separately using the following code:

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train)

X\_test\_scaled = scaler.fit\_transform(X\_test)

1. Instantiate the LDA model and save it as model. The following will show you how.

from sklearn.discriminant\_analysis import LinearDiscriminantAnalysis

model = LinearDiscriminantAnalysis()

Note

By instantiating an LDA model with no argument for n\_components we will return all possible components.

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

See the resultant output below:



**Figure 4.x: Output from fitting linear discriminant function analysis**

1. Much like in PCA, we can return the percentage of variance explained by each component. The output is shown in the following figure.

model.explained\_variance\_ratio\_



Figure 4.17: Explained variance by component.

Note

The first component explains 95.86% of the variance in the data and the second component explains 4.14% of the variance in the data for a total of 100%.

We have successfully fit a LDA model to compress our data from 9 features to 2 features. Decreasing the features to 2 cuts the time to tune and fit machine learning models. However, prior to using these features in a classifier model we must transform the training and testing features into their 2 components. In the next exercise, we will show how this is done.

Discuss 28: What is the advantage of saving our outcome variable as DV?

Answer: By saving target as DV we do not need to write Type in the two places where it refers to DV in step 4. This allows the code to be flexible to different data sets. By saving Type as DV ahead of time, the dependent variable needs to be changed in one place rather than several.

Exercise 10: Using LDA Transformed Components in Classification Model

Present 29: Using LDA Transformed Components in Classification Model

Using supervised data compression, we will transform our training and testing features (i.e., X\_train\_scaled and X\_test\_scaled, respectively) into their components and fit a RandomForestClassifier model on them.

Continuing from Exercise 9:

1. Compress X\_train\_scaled into its components as follows:

X\_train\_LDA = model.transform(X\_train\_scaled)

1. Compress X\_test into its components using:

X\_test\_LDA = model.transform(X\_test\_scaled)

Instructor Note:

Make sure to split the data into testing and training and scaling each prior to transforming them into components. If we fit the LDA model to X (I.e., prior to splitting into testing and training) then we would allow the LDA model to peek at the test data. Thus, to avoid peeking at the test data, we will fit the LDA model on the scaled training data (I.e., X\_train\_scaled) and then transform both the training (I.e., X\_train\_scaled) and testing (X\_test\_scaled) data using the model fit on the training data.

1. Instantiate a RandomForestClassifier model as follows:

from sklearn.ensemble import RandomForestClassifier

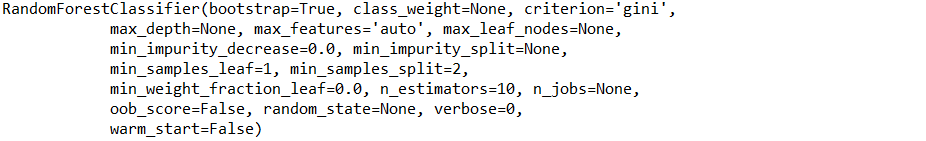
model = RandomForestClassifier()

Note

We will be using the default hyperparameters of the RandomForestClassifier model because tuning hyperparameters is beyond the scope of this lesson.

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

See the resultant output below:



**Figure 4.x: Output after fitting random forest classifier model**

1. Generate predictions on X\_test\_LDA and save them as the array, predictions using the following code:

predictions = model.predict(X\_test\_LDA)

1. Evaluate model performance by comparing predictions to y\_test using a confusion matrix. To generate and print a confusion matrix see the code below:

from sklearn.metrics import confusion\_matrix

import pandas as pd

import numpy as np

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

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)

The output is shown in the following figure:

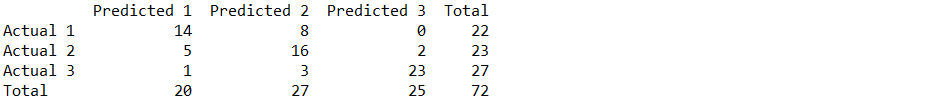


Figure 4.19: 3x3 confusion matrix for evaluating RandomForestClassifier model performance using the LDA compressed data

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, 14 were predicted to be in class 1 and 8 were predicted to be in class 2. Of the 23 observations that were in class 2, 16 were predicted to be in class 2, 5 in class 1, and 2 in class 3. Of the 27 observations that were in class 3, 23 were predicted to be in class 3, 1 were predicted to be in class 1, and 3 were predicted to be in class 2. The Random Forest classifier algorithm predicted with 73.61% accuracy using the components generated from the LDA algorithm.

Summary

Present 32: Summary

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. We were introduced to an algorithm, LDA, which reduces dimensionality in a supervised manner. Lastly, we transformed our scaled training and testing features into the LDA components and used them to fit a Random Forest classifier model. We should now feel much more comfortable with dimensionality reduction and unsupervised learning techniques.

We were briefly introduced to creating plots in this lesson, however, in the next lesson, we will focus on using the Matplotlib data visualization library to build and customize a variety of plots and subplots.

Present 33: Assessment questions

Assessment Questions

1. In unsupervised learning, *descriptive models* are used for exploratory analysis to uncover patterns in unlabeled data.
   1. True
   2. False
2. Examples of unsupervised learning tasks include:
   1. clustering
   2. regression
   3. dimension reduction
   4. a and c
3. In clustering, observations are assigned to groups in which there is:
   1. High within-group homogeneity and high between-group heterogeneity
   2. High within-group homogeneity and low between-group heterogeneity
   3. Low within-group homogeneity and high between-group heterogeneity
   4. Low within-group homogeneity and low between-group heterogeneity
4. Fill in the blanks: Too many features can be troublesome because the greater the number of variables in a model the higher the probability of \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ and subsequent *\_\_\_\_\_\_\_\_\_\_\_* of the model.
   1. overfitting, multicollinearity
   2. accuracy, success
   3. multicollinearity, overfitting
   4. None of the above
5. True/False: Model fitting time is not affected by the number of features in the data.
   1. True
   2. False
6. Which 2 methods did we cover for reducing the number of features in our data?
   1. Stepwise model selection and factor analysis
   2. Principal Component Analysis (PCA) and
   3. None of the above
   4. All of the above
7. Fill in the blank: Reducing the number of features in the data helps eliminate \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ and converges on a combination of features to best produce a model that performs well on \_\_\_\_\_\_, \_\_\_\_, data.
   1. multicollinearity, unseen, test
   2. variance, unseen, training
   3. multicollinearity, unseen, training
   4. None of the above
8. Fill in the blank: Hierarchical Cluster Analysis (HCA) and k-Means clustering BOTH use \_\_\_\_\_\_\_\_ to determine similarity.
   1. sum
   2. distance
   3. z-score
   4. inertia
9. True/False: When algorithms use an element of randomness in determining predictions, we feel confident in our predictions after building a single model?
   1. True
   2. False
10. To increase the confidence we have in our predictions, we can:
    1. Iterate through numerous models
    2. Run the model 1 more time and see if the predictions changed
    3. Set a random state
    4. None of the above
11. To determine the best number of n\_clusters in a k-Means model we evaluate the inertia by n\_clusters plot and pick the number of clusters where:
    1. There is the smallest inertia
    2. There is the smallest number of clusters
    3. There are diminishing returns of decreases in inertia with added clusters
    4. None of the above
12. True/False: In Exercise 7, when we plotted Mean Inertia by n\_clusters using the original features and the PCA transformed features, the PCA transformed features had lower mean inertia at each value of n\_clusters?
    1. True
    2. False
13. True/False: Linear Discriminant Function Analysis (LFA) is used for supervised data compression
    1. True
    2. False
14. What happens if we fit the LDA model to X rather than X\_train prior to transforming X\_train and X\_test into its LDA components?
    1. Python will throw an error
    2. The algorithm is allowed to peek at the test data
    3. a and b
    4. None of the above