4

Dimensionality Reduction and Unsupervised Learning

*Present 1-2: The title slide and the Learning Objectives slide. An overview of what we will achieve in this course*

Lesson Objectives:

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

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

Introduction

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

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

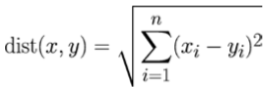
Hierarchical Cluster Analysis (HCA)

*Present x: Introduction to HCA and interpreting dendrograms*

Hierarchical cluster analysis (HCA) is best implemented when the user does not have an *a priori* number of clusters for which to build. 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 merges similar clusters that are closest in distance 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:



Equation 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:

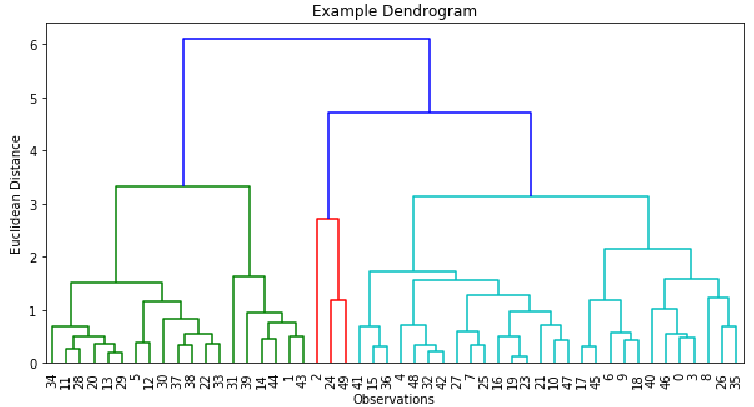


Figure 4.x. Example dendrogram

The y-axis indicates the Euclidean distance while the x-axis indicates the row index for each observation. Horizontal lines denote links between observations with links closer to the x-axis indicating closer distance and subsequent relationship. There appear to be 3 clusters in this example. 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 x: 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()).

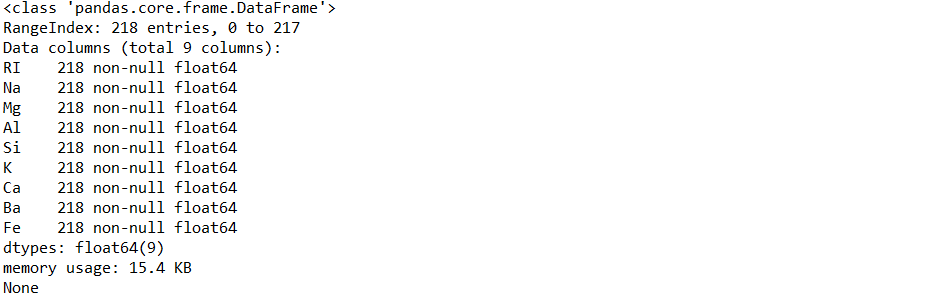


Figure 4.x: DataFrame information

***Discussion****: From Figure 4.x, how many rows do we have in our data? What about columns? Are there any missing observations? What is the data type for each column?*

***Answer****: There are 218 entries (I.e., rows) and 9 columns. There are no missing values (I.e., all are non-null) and that the data type for each column is float64.*

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')

Exercise 2: Plotting HCA Model

*Present x: Plotting HCA Model*

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()

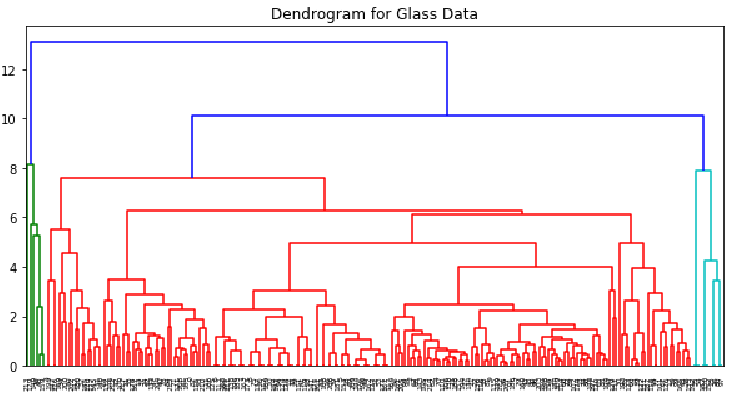


Figure 4.x: Dendogram for glass data

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.

Exercise 3: Assigning Predicted Clusters to Observations

*Present x: Assigning Predicted Clusters to Observations*

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')

***Teaching Tip:***

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.

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

***Discussion****: Why can we not add labels as a column to any data frames generated before being shuffled?*

***Answer****:W*

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:

* HCA models are easy to build
* There is no need to specify the number of clusters in advance
* The 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 x: Brief explanation of 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.

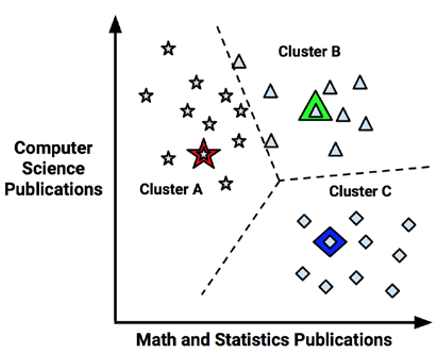


Figure 4.x: A scatterplot of observations and the cluster centers as denoted by the star, triangle, and diamond.

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.

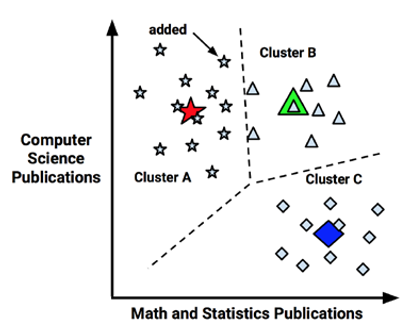


Figure 4.x: Shifting of the cluster centers to the cluster centroid.

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:

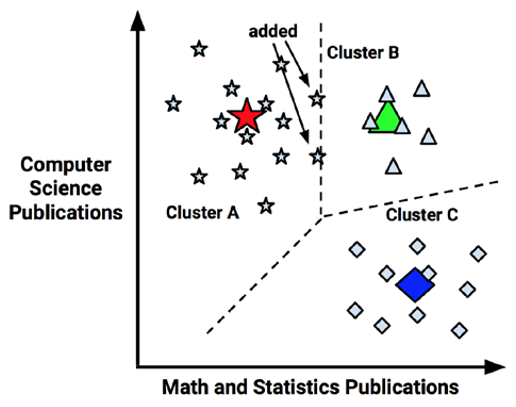


Figure 4.x: Updating of the centroids after observation reassignment.

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

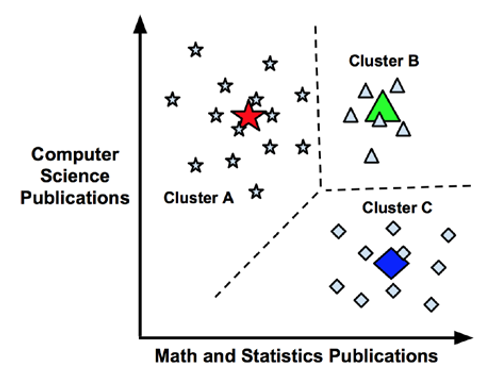


Figure 4.x: Final centroid position and cluster assignments.

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 4: Fitting k-Means Model

*Present x: Fitting k-Means Model*

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

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)

Exercise 5: Assigning Predicted Clusters to Observations

*Present x: Assigning Predicted Clusters to Observations*

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.x: Frequency table of two clusters

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))

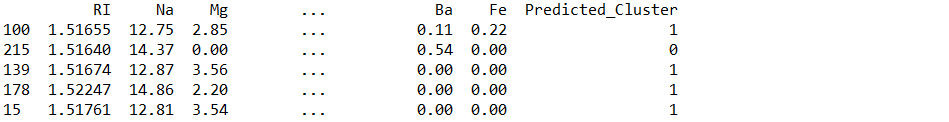


Figure 4.x: First 5 rows of Predicted\_Cluster

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

***Answer****: We are not very confident in these predictions because the predicted labels change each time the model is run. This is due to the initial group centers being chosen at random. Building an ensemble of models (I.e., numerous models), and calculating an average or mode of those models generates more stable predictions and subsequently increases the confidence in our predictions.*

Activity 1: Ensemble k-Means Clustering

*Present 14: Ensemble k-Means Clustering*

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

**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 the labels for each model 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. Iterate through 100 loops 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

Activity 2: Calculate Predicted Label

*Present x: Calculate Predicted Label*

Now, we will calculate the predicted label by assigning the most occurring value across each row as the predicted label.

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

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

***Answer****: We are much more confident in the predictions after running 100 models relative to a single model. To further increase our confidence that we have appropriately segmented this sample we will iterate through 100 models over a range of n\_clusters.*

Exercise 6: Calculating Mean Inertia by n\_clusters

*Present x: 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

***Discussion****: Why do we import our packages outside of the loop rather than inside the loop?*

***Answer****:*

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.

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\_

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

*Discussion: After all 100 iterations of n\_clusters=x have concluded, how many values will inertia\_list contain?*

*Answer: inertia\_list will contain 100 values, one for each model.*

1. 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):

*Discussion: Why are we iterating through the range 1 to 11 rather than the range 0 to 10?*

*Answer: The number of clusters cannot equal 0, thus the reason for iterating through the range 1 through 11 as opposed to 0 through 10.*

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)

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.

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

print(mean\_inertia\_list)



Figure 4.x: mean\_inertia\_list

Exercise 7: Plotting Mean Inertia by n\_clusters

*Present x: 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 to visually assess the best value 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 plt.title(‘Mean Inertia by n\_clusters’).
2. Label the x-axis ‘n\_clusters’ using plt.xlabel(‘n\_clusters’) and label the y-axis ‘Mean Inertia’ using plt.ylabel(‘Mean Inertia’).
3. Set the tick labels on the x-axis as the values in x using plt.xticks(x).
4. Display the plot use 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:

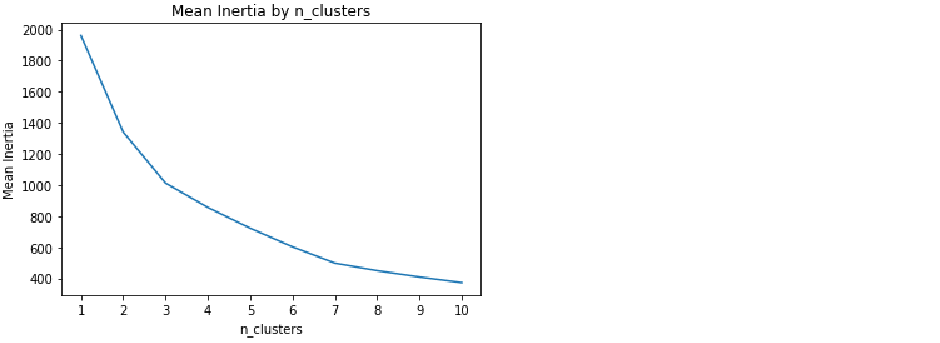


Figure 4.x: Mean inertia by n\_clusters

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.x, 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.

*Discuss x: 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.*

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

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

Principal Component Analysis (PCA)

*Present x: Brief introduction of Principal Component Analysis*

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

To demonstrate PCA, we will:

* 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 8: Fitting PCA Model

*Fitting PCA Model*

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

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)

For the resultant output, refer to the following screenshot:



Figure 4.x: Explained variance in the data for each principal component

Each principal component explains a proportion of the variance in the data. For example, the first principal component explains .35 of the variance in the data, the second explains .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 reduce 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 9: Choosing n\_components using Threshold of Explained Variance

*Present x: Choosing n\_components using Threshold of Explained Variance*

The objective in this exercise is to programmatically determine which ones will combine to capture no less than 95% of the explained variance in the data, a user-defined threshold.

Continuing from Exercise 9:

1. Determine the principal component at which at least 95% of the variance in the data is explained by calculating the cumulative sum of explained variance by principal component. Let’s take 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.x: Cumulative sum of the explained variance for each principal component

*Discuss: Why might we want to set up logic so a threshold is checked programmatically as opposed to checking manually?*

*Answer: Checking a threshold manually may be difficult with a large number of features, human error is introduced with manual inspection, and manual inspection makes the analysis less automated.*

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

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-Meansmodel to lower the inertia values. Additionally, we can compare the inertia values across n\_clusters values for the models built using PCA transformed data to those using data that was not PCA transformed.

Activity 3: Evaluating Mean Inertia by Cluster after PCA Transformation

*Present x: 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 how to tune the optimal number of clusters for k-Means clustering with a nested loop, we will put them all together in this activity.

We will also visually compare the inertia by n\_clusters prior to the PCA transformation and after the PCA transformation to see the effect of PCA transformation on k-Means model performance.

Continuing from Exercise 10:

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)

Now, much like in Exercise 7, 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 10: Visual Comparison of Inertia by n\_clusters

*Present x: 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 7 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

Continuing from Activity 3:

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’)

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

plt.xlabel(‘n\_clusters’)

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()

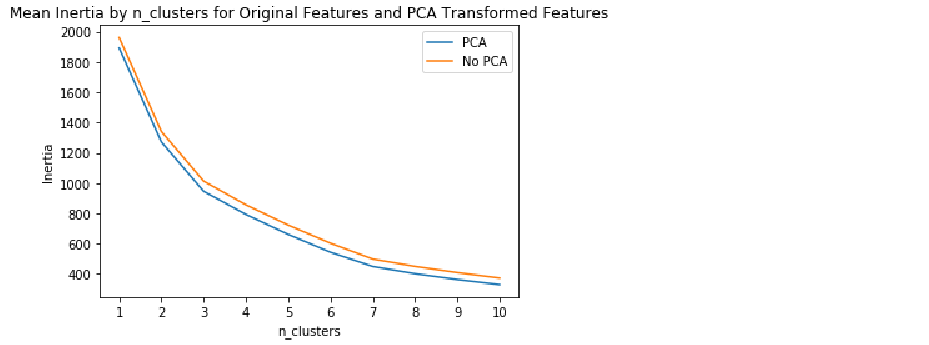


Figure 4.x: Mean inertia by n\_clusters for original features (orange) and PCA transformed features (blue).

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

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 x: Brief introduction of 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 how to use 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 11: Fitting LDA Model

*Present x: 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 process, 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*.*

***Discussion****: 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 just needs to be changed in one place rather than several.*

1. Scale df\_shuffled by, first, instantiating a StandardScaler object named scaler using scaler = StandardScaler(). Fit scaler to X and transform X into the scaled features using scaled\_features = scaler.fit\_transform(X).
2. Split scaled\_features and y into testing and training as follows:

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

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

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, y\_train)



Figure 4.x: Explained variance by component.

5.864.14

We have successfully fit a LDA model to compress our data from 9 features to 2 features. Prior to using these features in a classifier model we must transform the data into its 2 components.

Exercise 12: Using LDA Transformed Components in Classification Model

*Present x: Using LDA Transformed Components in Classification Model*

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

Continuing from Exercise 11:

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

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

Compress X\_test into its components using:

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

***Teaching tip:***

Make sure to split the data into testing and training 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 training data (I.e., X\_train) and then transform both the training (I.e., X\_train) and testing (X\_test) 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 \_LDA.
2. 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 a confusion matrix, begin by importing the function confusion\_matrix using from sklearn.metrics import confusion\_matrix. Then, save the confusion matrix into a 3x3 array object using conf\_matrix = confusion\_matrix(y\_test, predictions). Print the confusion matrix using print(conf\_matrix). The output is shown in the following figure.



Figure 4.x: 3x3 Confusion Matrix for evaluating RandomForestClassifier model performance using the LDA compressed data.

***Note***

This 3x3 array can be difficult to comprehend without guidance. The first, second, and third columns represent predicted 1, predicted 2, and predicted 3, respectively. The first, second, and third rows represent actual 1, actual 2, and actual 3, respectively. Additionally, like k-Means, RandomForestClassifier models deliver different predictions each time a model is run. This is due to Random Forests combining numerous randomized decision trees. Thus, output will vary model to model.

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

import pandas as pd

cm = pd.DataFrame(conf\_matrix)

1. Create a new column named ‘Total’ which contains the row totals with the following code:

import numpy as np

cm[‘Total’] = np.sum(cm, axis=1)

Compute a row below the matrix containing the column totals as follows:

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

Assign names to the columns in cm

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

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

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

Print the new confusion matrix to the console using print(cm). The output is shown in the following figure.



Figure 4.x: 3x3 confusion matrix styled for improved interpretability

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, 15 were predicted to be in class 1 and 7 were predicted to be in class 2. Of the 23 observations that were in class 2, 16 were predicted to be in class 2, 6 in class 1, and 1 in class 3. Of the 27 observations that were in class 3, 27 were predicted to be in class 3. The Random Forest classifier algorithm predicted with 80.56% accuracy using all the components generated from the LDA algorithm.

However, in Exercise 10 we saw the k-Means model perform better when 95% of the data was retained relative to 100%. Thus, to optimize model performance, we will find the number of components returned by the LDA model that results in the best accuracy score as calculated using RandomForestClassifier.

Exercise 13: Tuning LDA n\_components

*Present x: Tuning LDA n\_components*

To be sure we are using the optimal number of LDA n\_components for which to use in our model, we will iterate through all possible n\_components, transforming the features, fitting a RandomForestClassifier model, and evaluating accuracy at each iteration. This way, we can be sure we are using the optimal number of n\_components for which to use in our model.

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

1. Construct a loop to iterate through the possible values for n\_components, transform the features, split the transformed features into testing and training data, fit a RandomForestClassifier model, and append the accuracy scores to a list.

First, import the dependencies outside the loop so fewer computational resources are spent on importing libraries using the following code:

from sklearn.discriminant\_analysis import LinearDiscriminantAnalysis

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

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy\_score

1. Instantiate an empty list for which to append accuracy scores outside of the loop as follows:

accuracy\_list = []

1. Instantiate the loop and instruct the computer to loop through the values 0 to 2 (I.e., the maximal number of components). The below code will show you how:

for i in range(2):

1. Instantiate an LDA model with the number of components equaling i+1 using:

model = LinearDiscriminantAnalysis(n\_components=i+1)

1. Fit the LDA model to the training data as follows:

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

1. Transform X\_train into its components using X\_train\_LDA = model.transform(X\_train). Transform X\_test into its components using X\_test\_LDA = model.transform(X\_test).
2. Instantiate a RandomForestClassifier model and fit it to the training data using the following code:

model = RandomForestClassifier()  
model.fit(X\_train\_LDA, y\_train)

1. Generate predictions on the test data using:

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

1. Calculate model accuracy as follows:

accuracy = accuracy\_score(y\_test, predictions)

1. Append accuracy to accuracy\_list.

accuracy\_list.append(accuracy)

1. Find the maximum of accuracy list and print it to the console using the following code. The output is shown in the following figure.

max\_accuracy = max(accuracy\_list)

print(max\_accuracy)



Figure 4.x: Maximum value of accuracy\_list

1. Find the index of the maximum value in the list.

index\_max\_accuracy = accuracy\_list.index(max\_accuracy)

1. Print off a message that tells the user which number of components maximize accuracy and what the accuracy is. The output is shown in the following figure.

print('{0} component(s) are used to achieve {1:0.2f}% accuracy'.format(index\_max\_accuracy+1, max\_accuracy\*100))



Figure 4.x: Number of components and accuracy of the RandomForestClassifier model

***Discussion****: Why do we add 1 to index\_max\_accuracy in the formatted string? Why do we multiply max\_accuracy by 100 in the formatted string?*

***Answer****: We add 1 to index\_max\_accuracy because indexing starts at 0. Additionally, we multiply max\_accuracy by 100 to convert the proportion to a percentage.*

Fitting our RandomForestClassifier model with 2 LDA components resulted in better accuracy than our RandomForestClassifier model with 1 LDA component. As mentioned previously, Random Forest models work by combining numerous randomized decision trees, so predictions vary each time a model is run. Thus, we may consider building our RandomForestClassifier models in an ensemble much like we did in k-Means to increase the confidence in our predictions.

Summary

*Present x: 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 tuned the number of components in LDA by iterating through all possible values for components and programmatically returning which value resulted in the best accuracy score from a Random Forest classifier model. We should now feel 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.

Assessment Questions:

*Present x: 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
   2. Principal Component Analysis (PCA)
   3. Linear Discriminant Function Analysis (LDA)
   4. a and b
   5. b and c
   6. a and c
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