PREV
Chapter 3

NEXT
Chapter 5

Chapter 4

Dimensionality Reduction and Unsupervised Learning

Learning Objectives

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

- Compare hierarchical cluster analysis (HCA) and k-means clustering
- Conduct an HCA and interpret the output
- Tune a number of clusters for k-means clustering
- Select an optimal number of principal components for dimension reduction
- Perform supervised dimension compression using linear discriminant function analysis (LDA)

This chapter will cover 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 placed 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 selected customers for marketing advertisements and promotions 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 a model to improve it, and which features to exclude because they diminish a 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 a model. Additionally, numerous features expand the complexity of a model and increase the time for model tuning and fitting.

This becomes troublesome with larger datasets. Fortunately, another use case for unsupervised learning is to reduce the number of features in a dataset by creating combinations of the original features. Reducing the number of features in 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. It is a problem in linear regression models because it does not allow the isolation of the relationship between each independent variable and the outcome measure. Thus, coefficients and p-values become unstable and less precise.

In this chapter, we will be covering two widely used unsupervised clustering algorithms: *Hierarchical Cluster Analysis (HCA)* and *k-means clustering*. Additionally, we will explore dimension reduction using *principal component analysis (PCA)* and observe how reducing dimensionality can improve model performance. Lastly, we will implement linear discriminant function analysis *(LDA)* for supervised dimensionality reduction.

Hierarchical Cluster Analysis (HCA)

Hierarchical cluster analysis (HCA) is best implemented when the user does not have a priori number of clusters 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 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*, the Euclidean distance is calculated using the formula:

$$\operatorname{dist}(x,y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

Figure 4.1: The 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 as 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 neurons.

This creates a hierarchical structure in which items that are closely related are "chunked" together into clusters. An example dendrogram is displayed here:

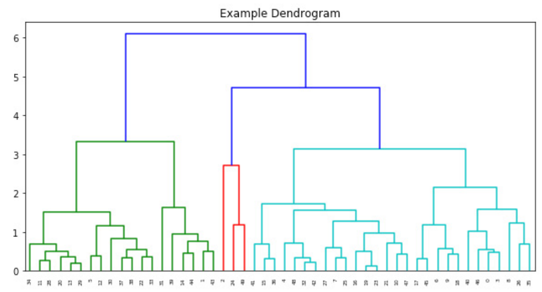


Figure 4.2: An example dendrogram

The y-axis indicates the Euclidean distance, while the x-axis indicates the row index for each observation. Horizontal lines denote links between observations; links closer to the x-axis indicate shorter distance and a subsequent closer relationship. In this example, there appear to be three clusters. The first cluster includes observations colored in green, the second cluster includes observations colored in red, and the third cluster includes observations colored in turquoise.

EXERCISE 34: BUILDING AN HCA MODEL

To demonstrate HCA, we will be use an adapted version of the glass dataset from the University of California – Irvine (https://github.com/TrainingByPackt/Data-Science-with-Python/tree/master/Chapter04). 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

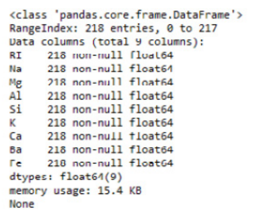
In this exercise, we will use the refractive index (RI) and weight percent in each oxide to segment the glass type.

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')
```

2. Look for some basic data frame information by printing **`df.info()`** to the console using the following code:

```
print(df.info()):
```



```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 218 entries, 0 to 217
Data columns (total 9 columns):
RI    218 non-null float64
Na    218 non-null float64
Mg    218 non-null float64
Al    218 non-null float64
Si    218 non-null float64
K     218 non-null float64
Ca    218 non-null float64
Ba    218 non-null float64
Fe    218 non-null float64
dtypes: float64(9)
memory usage: 15.4 KB
None
```

Figure 4.3: DataFrame information

3. 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)
```

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

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

scaled_features = scaler.fit_transform(df_shuffled)
```

5. Perform hierarchical clustering using the linkage function on **`scaled_features`**. The following code will show you how:

```
from scipy.cluster.hierarchy import linkage

model = linkage(scaled_features, method='complete')
```

Congratulations! You've successfully built an HCA model.

EXERCISE 35: PLOTTING AN HCA MODEL AND ASSIGNING PREDICTIONS

Now that the HCA model has been built, we will continue with the analysis by visualizing 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()
```

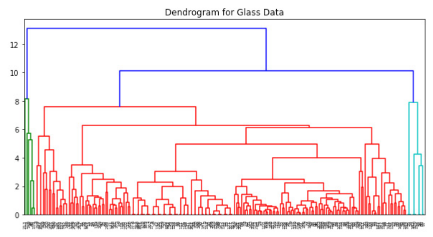


Figure 4.4: Dendogram for glass data

Note

The index for each observation or row in a dataset is on the x-axis. The Euclidean distance is on the y-axis. 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 rows in **`df_shuffled`**.

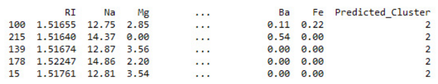
2. Generate predicted labels of the cluster which an observation belongs to using the following code:

```
from scipy.cluster.hierarchy import fcluster

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

3. Add the labels array as a column in the shuffled data and preview the first five rows using the following code:

```
df_shuffled['Predicted_Cluster'] = labels

print(df_shuffled.head(5))
```

4. Check the output in the following figure:



```
           RI     Na    Mg         ...          Ba    Fe  Predicted_Cluster
100  1.51655  12.75  2.85         ...        0.11  0.22                  2
215  1.51640  14.37  0.00         ...        0.54  0.00                  2
139  1.51674  12.87  3.56         ...        0.00  0.00                  2
178  1.52247  14.86  2.20         ...        0.00  0.00                  2
15   1.51761  12.81  3.54         ...        0.00  0.00                  2
```

Figure 4.5: The first five rows of df_shuffled after predictions have been matched to observations.

We have successfully learned the difference 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 pros of HCA models include:

- They are easy to build
- There is 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 (that is, 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 datasets with many features

Next, we will introduce you to another clustering algorithm, k-means clustering. This algorithm addresses some of the HCA shortcomings by having the ability to adjust when the clusters have been initially generated. It is more computationally frugal than HCA.

K-means Clustering

Like HCA, K-means also uses distance to assign observations into clusters not labeled in data. However, rather than linking observations to each other as in HCA, k-means assigns observations to *k* (user-defined number) 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 cluster centers have been randomly generated there are two phases:

- Assignment phase
- Updating phase

Note

The randomly generated cluster centers are important to remember, and we will be visiting it later in this 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 to the appropriate cluster. We can turn it into an advantage 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:

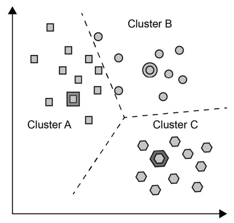


Figure 4.6: A scatterplot of observations and the cluster centers as denoted by the star, triangle, and diamond.

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:

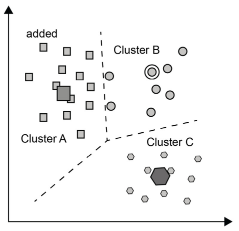


Figure 4.7: Shifting of the cluster centers to the cluster centroid.

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:

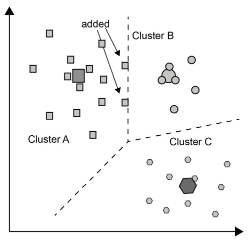


Figure 4.8: Updating of the centroids after observation reassignment.

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

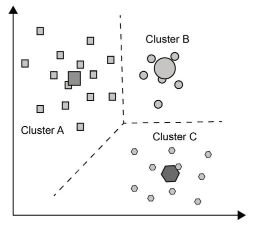


Figure 4.9: The final centroid position and cluster assignments.

Using the same glass dataset from *Exercise 34, Building an HCA Model*, we will fit a k-means model with 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 36: FITTING K-MEANS MODEL AND ASSIGNING PREDICTIONS

Since our data has already been prepared (see *Exercise 34, Building an HCA Model*), and we understand concepts behind the k-Means algorithm, we will learn how easy it is to fit a k-means model, generate predictions, and assign these predictions to the appropriate observation.

After the glass dataset has been imported, shuffled, and standardized:

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

   ```
   from sklearn.cluster import KMeans

   model = KMeans(n_clusters=2)
   ```

2. Fit the model to **`scaled_features`** using the following line of code:

   ```
   model.fit(scaled_features)
   ```

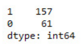
3. Save the cluster labels from our model into the array, labels, using the following:

   ```
   labels = model.labels_
   ```

4. 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:



```
1    157
0     61
dtype: int64
```

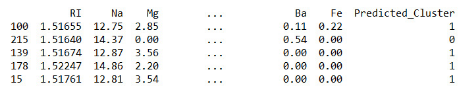
Figure 4.10: Frequency table of two clusters

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

5. Add the labels array as the '**`Predicted Cluster`**' column into the **`df_shuffled`** data frame and preview the first five rows using the following code:

   ```
   df_shuffled['Predicted_Cluster'] = labels

   print(df_shuffled.head(5))
   ```

6. Check the output in the following figure:



```
           RI     Na    Mg        ...          Ba    Fe  Predicted_Cluster
100  1.51655  12.75  2.85        ...        0.11  0.22                  1
215  1.51640  14.37  0.00        ...        0.54  0.00                  0
139  1.51674  12.87  3.56        ...        0.00  0.00                  1
178  1.52247  14.86  2.20        ...        0.00  0.00                  1
15   1.51761  12.81  3.54        ...        0.00  0.00                  1
```

Figure 4.11: First five rows of df_shuffled

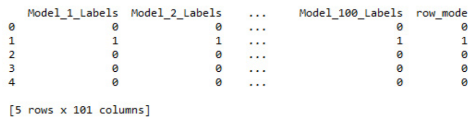
ACTIVITY 12: ENSEMBLE K-MEANS CLUSTERING AND CALCULATING PREDICTIONS

When algorithms use randomness as part of their method for finding the optimal solution (that is, 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 (that is, mean, median, and mode). In this activity, we will build an ensemble of 100 k-means clustering models.

After the glass dataset has been imported, shuffled, and standardized (see *Exercise 34, Building an HCA Model*):

1. Instantiate an empty data frame 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. Calculate the mode for each row in **`labels_df`** and save it as a new column in **`labels_df`**. The output should be as follows:



```
   Model_1_Labels  Model_2_Labels    ...     Model_100_Labels  row_mode
0               0               0    ...                    0         0
1               1               1    ...                    1         1
2               0               0    ...                    0         0
3               0               0    ...                    0         0
4               0               0    ...                    0         0

[5 rows x 101 columns]
```

Figure 4.12: First five rows of labels_df

Note

The solution for this activity can be found on page 356.

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 these 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 clusters to segment our observations.

EXERCISE 37: 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 A*ctivity 12, Ensemble k-means Clustering and Calculating Predictions* (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 builds 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 **`n_clusters`** value.

After the glass dataset has been imported, shuffled, and standardized (see *Exercise 34, Building an HCA Model*):

1. Import the packages we need outside of the loop as shown here:

   ```
   from sklearn.cluster import KMeans

   import numpy as np
   ```

2. 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 append inertia values after each iteration of the inside loop as shown here:

   ```
   inertia_list = []
   ```

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

   ```
   for i in range(100):
   ```

4. Inside the loop, build a **`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.

5. Fit the model to **`scaled_features`** as shown here:

   ```
   model.fit(scaled_features)
   ```

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

   ```
   inertia = model.inertia_
   ```

7. Append inertia to **`inertia_list`** using the following code:

   ```
   inertia_list.append(inertia)
   ```

8. Move to the outside loop, instantiate another empty list to store the average inertia values, as shown here:

   ```
   mean_inertia_list = []
   ```

9. Iterate through the values 1 through 10 for **`n_clusters`** using the following code:

   ```
   for x in range(1, 11):
   ```

10. 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)
```

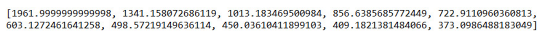
11. Append **mean_inertia** to **mean_inertia_list** as shown here:

```
mean_inertia_list.append(mean_inertia)
```

12. 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**.

13. Print **mean_inertia_list** as shown in the following code. The values are shown in the following figure:

```
print(mean_inertia_list)
```



```
[1961.9999999999998, 1341.158072686119, 1013.183469500984, 856.6385685772449, 722.9110960360813,
603.1272461641258, 498.57219149636114, 450.03610411899103, 409.1821381484066, 373.0986488183049]
```

Figure 4.13: mean_inertia_list

EXERCISE 38: PLOTTING MEAN INERTIA BY N_CLUSTERS

Continuing from Exercise 38:

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
```

2. 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))
```

3. Save **mean_inertia_list**, as the object y as shown here:

```
y = mean_inertia_list
```

4. Plot the mean inertia by number of clusters, as follows:

```
plt.plot(x, y)
```

5. Set the plot title to read '**Mean Inertia by n_clusters**' using the following:

```
plt.title('Mean Inertia by n_clusters')
```

6. Label the x-axis '**n_clusters**' using **plt.xlabel('n_clusters')**, and label the y-axis '**Mean Inertia**' using the following code:

```
plt.ylabel ('Mean Inertia')
```

7. Set the tick labels on the x-axis as the values in x using the following:

```
plt.xticks(x)
```

8. 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:

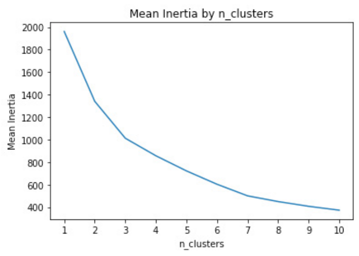


Figure 4.14: Mean inertia by n_clusters

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 4.14, 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 decrease 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.

Principal Component Analysis (PCA)

At a high level, PCA is a technique for creating uncorrelated linear combinations from the original features termed **components**. Of the principal components, the first component explains the greatest proportion of variance in 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 data
- Fit those components to a k-means cluster analysis and compare k-means performance before and after the PCA transformation

EXERCISE 39: FITTING A PCA MODEL

In this exercise, you will learn to fit a generic PCA model using data we prepared in *Exercise 34, Building an HCA Model* and the brief explanation of PCA.

1. Instantiate a PCA model as shown here:

   ```
   from sklearn.decomposition import PCA

   model = PCA()
   ```

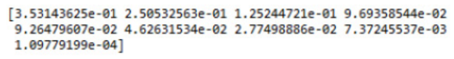
2. Fit the PCA model to **`scaled_features`**, as shown in the following code:

   ```
   model.fit(scaled_features)
   ```

3. 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)
   ```

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



```
[3.53143625e-01 2.50532563e-01 1.25244721e-01 9.69358544e-02
 9.26479607e-02 4.62631534e-02 2.77498886e-02 7.37245537e-03
 1.09779199e-04]
```

Figure 4.15: Explained variance in the data for each principal component

Each principal component explains a proportion of the variance in data. In this exercise, the first principal component explained .35 of the variance in data, the second explained. 25, the third .13%, and so on. Altogether, these nine components explain 100% of the variance in data. The goal of dimensionality reduction is to decrease the number of dimensions in data with the objectives of limiting overfitting and time to fit the subsequent model. Thus, we will not keep all nine 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 40: CHOOSING N_COMPONENTS USING THRESHOLD OF EXPLAINED VARIANCE

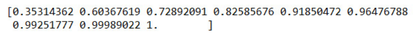
In *Exercise 39*, *Fitting PCA Model*, you learned to fit a PCA model with all available principal components. However, keeping all of the principal components does not reduce the number of dimensions in data. In this exercise, we will reduce the number of dimensions in data by retaining the components that explain a threshold of variance in it.

1. Determine the number of principal components in which a minimum of 95% of the variance in the data is explained by calculating the cumulative sum of explained variance by the

principal component. Let's 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:



```
[0.35314362 0.60367619 0.72892091 0.82585676 0.91850472 0.96476788
 0.99251777 0.99989022 1.        ]
```

Figure 4.16: The cumulative sum of the explained variance for each principal component

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

```
threshold = .95
```

3. 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 data. Since we will be looping through the indices of **`cum_sum_explained_var`**, we will instantiate our loop using the following:

```
for i in range(len(cum_sum_explained_var)):
```

4. 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:
```

5. 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
```

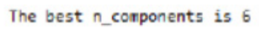
The last two lines in the if statement instruct the loop not to do anything if the logic is not met:

```
else:

pass
```

6. 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:



```
The best n_components is 6
```

Figure 4.17: The 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 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 13: EVALUATING MEAN INERTIA BY CLUSTER AFTER PCA TRANSFORMATION

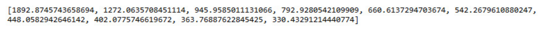
Now that we know the number of components to retain at least 95% of the variance in the data, how to transform our features into principal components, and a way to tune the optimal number of clusters for k-means clustering with a nested loop, we will put them all together in this activity.

Continuing from *Exercise 40*:

1. Instantiate a PCA model with the value for the **`n_components`** argument equal to **`best_n_components`** (that is, remember, **`best_n_components = 6`**).

2. Fit the model to **`scaled_features`** and transform it into the first six principal components

3. Using a nested loop, calculate the mean inertia over 100 models at values 1 through 10 for **`n_clusters`** (see *Exercise 40*, *Choosing n_components using Threshold of Explained Variance*).



```
[1892.8745743658694, 1272.0635708451114, 945.9585011131066, 792.9280542109909, 660.6137294703674, 542.2679610880247,
448.0582942646142, 402.0775746619672, 363.76887622845425, 330.43291214440774]
```

Figure 4.18: mean_inertia_list_PCA

Now, much like in *Exercise 38*, *Plotting Mean Inertia by n_clusters*, 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`**.

Note

The solution for this activity can be found on page 357.

EXERCISE 41: 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 38, Plotting Mean Inertia by n_clusters,* by:

- Adding a second line to the plot showing mean inertia by **`n_clusters`** after PCA transformation
- Creating a legend distinguishing the lines
- Changing the title

Note

For this visualization to work properly, **`mean_inertia_list`** *from Exercise 38, Plotting Mean Inertia by n_clusters, must still be in the environment.*

Continuing from *Activity 13*:

1. Import Matplotlib using the following code:

```
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))
```

3. Save **`mean_inertia_list_PCA`** as the object y using the following code:

```
y = mean_inertia_list_PCA
```

4. Save **`mean_inertia_list`** as the object y2 using the following:

```
y2 = mean_inertia_list
```

5. Plot mean inertia after a PCA transformation by number of clusters using the following code:

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

   Add our second line of mean inertia before a PCA transformation by number of clusters using the following:

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

6. 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')
```

7. Label the x-axis '**`n_clusters`**' using the following code:

```
plt.xlabel('n_clusters')
```

8. Label the y-axis '**`Mean Inertia`**' using:

```
plt.ylabel('Mean Inertia')
```

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

10. Show a legend using and display the plot as follows:

```
plt.legend()

plt.show()
```

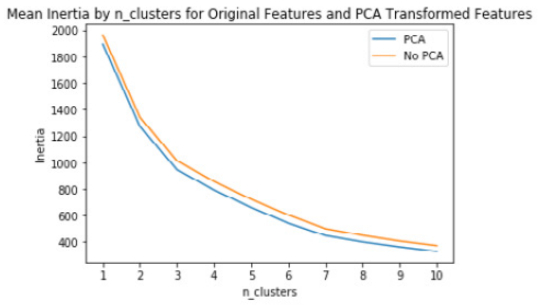


Figure 4.19: Mean inertia by n_clusters for original features (orange) and PCA transformed features (blue)

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 observations in each cluster after the PCA transformation relative to before the transformation. Thus, 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 (that is, 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 Analysis (LDA)

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 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 **`n_components`**
- Tune the number of **`n_components`**

EXERCISE 42: 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`**. (https://github.com/TrainingByPackt/Data-Science-with-Python/tree/master/Chapter04) This dataset contains the same nine 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')
```

2. 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)
```

3. Save '**`Type`**' as **`DV`** (I.e., dependent variable) as follows:

```
DV = 'Type'
```

4. 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.

5. Split X 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(X,
y, test_size=0.33, random_state=42)
```

6. 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)
```

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

```
from sklearn.discriminant_analysis import
LinearDiscriminantAnalysis

-model = LinearDiscriminantAnalysis()
```

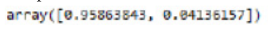
Note

By instantiating an LDA model with no argument **`for n_components`** *we will return all possible components.*

8. Fit the model to the training data using the following:

```
model.fit(X_train_scaled, y_train)
```

9. See the resultant output below:



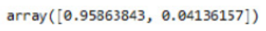
```
array([0.95863843, 0.04136157])
```

Figure 4.20: Output from fitting linear discriminant function analysis

10. Much like in PCA, we can return the percentage of variance explained by each component.

```
model.explained_variance_ratio_
```

The output is shown in the following figure.



```
array([0.95863843, 0.04136157])
```

Figure 4.21: 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 an LDA model to compress our data from nine features to two features. Decreasing the features to two 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 two components. In the next exercise, we will show how this is done.

EXERCISE 43: 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 42*:

1. Compress **`X_train_scaled`** into its components as follows:

   ```
   X_train_LDA = model.transform(X_train_scaled)
   ```

2. Compress **`X_test`** into its components using:

   ```
   X_test_LDA = model.transform(X_test_scaled)
   ```

3. Instantiate a **`RandomForestClassifier`** model as follows:

   ```
   from sklearn.ensemble import RandomForestClassifier

   model = RandomForestClassifier()
   ```

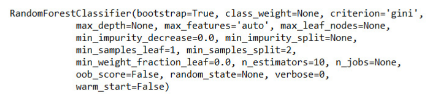
   ### *Note*

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

4. Fit the model to the compressed training data using the following code:

   ```
   model.fit(X_train_LDA, y_train)
   ```

   See the resultant output below:



```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
            max_depth=None, max_features='auto', max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=None,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False)
```

Figure 4.22: Output after fitting random forest classifier model

5. Generate predictions on **`X_test_LDA`** and save them as the array, predictions using the following code:

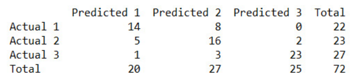
   ```
   predictions = model.predict(X_test_LDA)
   ```

6. 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:



```
          Predicted 1  Predicted 2  Predicted 3  Total
Actual 1           14            8            0     22
Actual 2            5           16            2     23
Actual 3            1            3           23     27
Total              20           27           25     72
```

Figure 4.23: 3x3 confusion matrix for evaluating RandomForestClassifier model performance using the LDA compressed data

Summary

This chapter introduced you 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 the value resulting in the best accuracy score from a Random Forest classifier model. You should now feel comfortable with dimensionality reduction and unsupervised learning techniques.

We were briefly introduced to creating plots in this chapter; however, in the next chapter, we will learn about structured data and how to work with XGboost and Keras libraries

Support / Sign Out

PREV
Chapter 3

Chap