

Table of Contents

Chapter 1: 7. Clustering data with Unsupervised Machine Learning	1
The K-Means Algorithm	1
Assignment of centroids	2
When does the algorithm stop iterating?	4
Implementing the K-Means algorithm in scikit-learn	4
Creating the base K-Means model	5
Optimal number of clusters	6
Feature Engineering for optimization	8
Scaling	9
Principal Component Analysis	11
Cluster Visualization	14
t-SNE	14
Hierarchical Clustering	17
Step 1: Individual features as individual clusters	17
Step 2: The merge	18
Step 3: Iteration	19
Implementing the hierarchical clustering	20
Going from unsupervised to supervised learning	21
Creating a labelled dataset	22
Building the decision tree	23
Summary	24
Index	26

1

7. Clustering data with Unsupervised Machine Learning

Most of the data that you will encounter out in the wild do not come with labels. Applying supervised machine learning techniques is impossible when your data does not come with labels. Unsupervised machine learning addresses this issue by grouping data into clusters after which we can assign labels based on these clusters.

Once the data is clustered into a specific number of groups, we can then proceed to giving these groups labels. Therefore, unsupervised machine learning is the first step that you, as the data scientist will have to implement before applying supervised machine learning techniques such as classification in order to make meaningful predictions.

A common application of the unsupervised machine learning algorithm is customer data that you will find across a wide range of industries. As a data scientist your job is to find groups of customers that you can segment and deliver targeted products/adverts to.

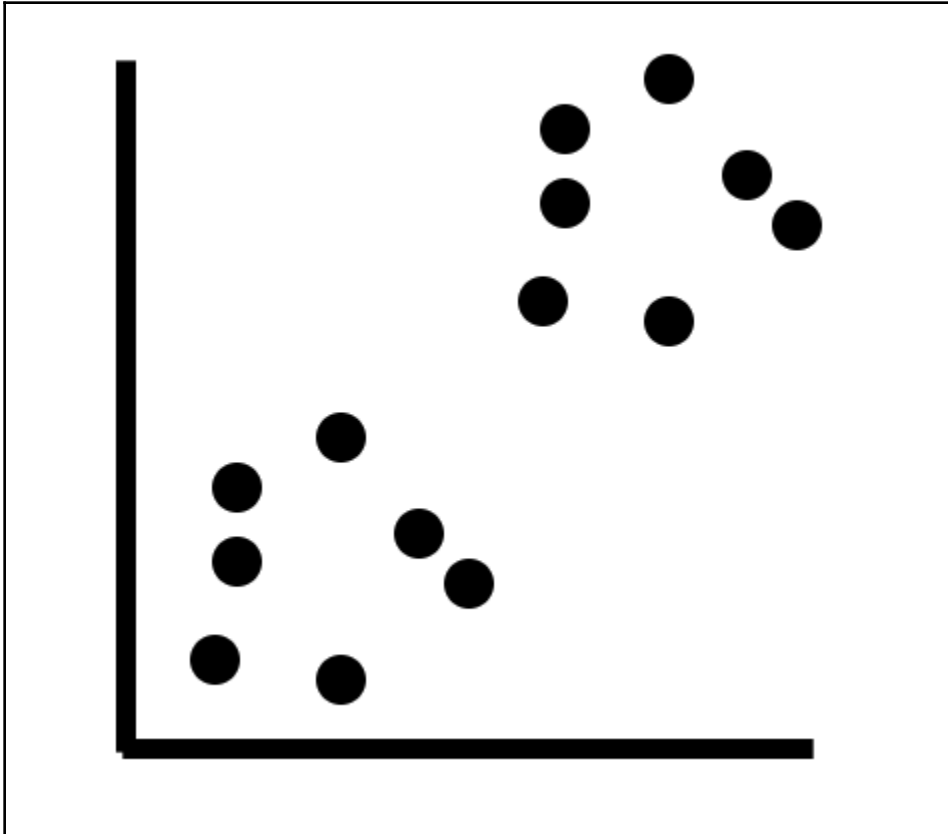
In this chapter you will learn about:

- The K-means algorithm and how it works internally in order to cluster unlabelled data.
- Implementing the K-means algorithm in scikit-learn.
- Feature Engineering for optimizing the unsupervised machine learning
- Cluster visualization
- Going from unsupervised to supervised machine learning

The K-Means Algorithm

In this section you will learn about how the K-Means algorithm works 'under-the-hood' in order to cluster data into groups that make logical sense.

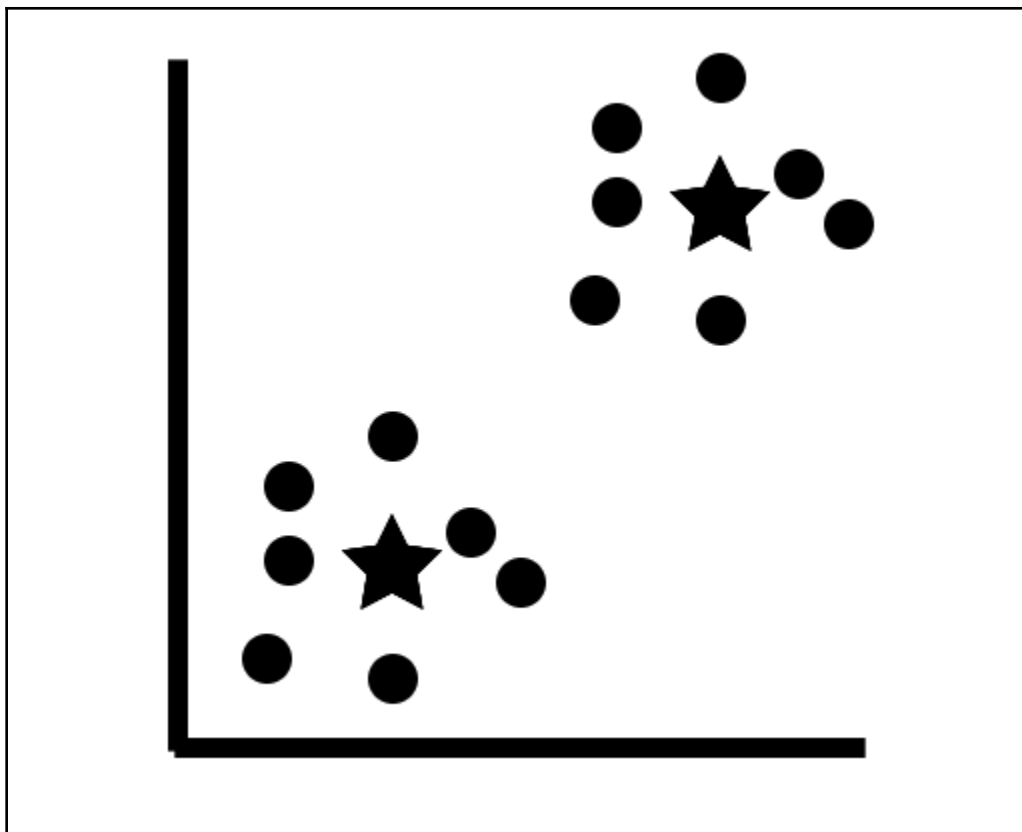
Let's consider a set of points as illustrated in the image below:



A random set of points

Assignment of centroids

The first step that the algorithm takes is to assign a set of random centroids. Assuming that we want to find two distinct clusters or groups, the algorithm assigns two centroids as illustrated in the image below:

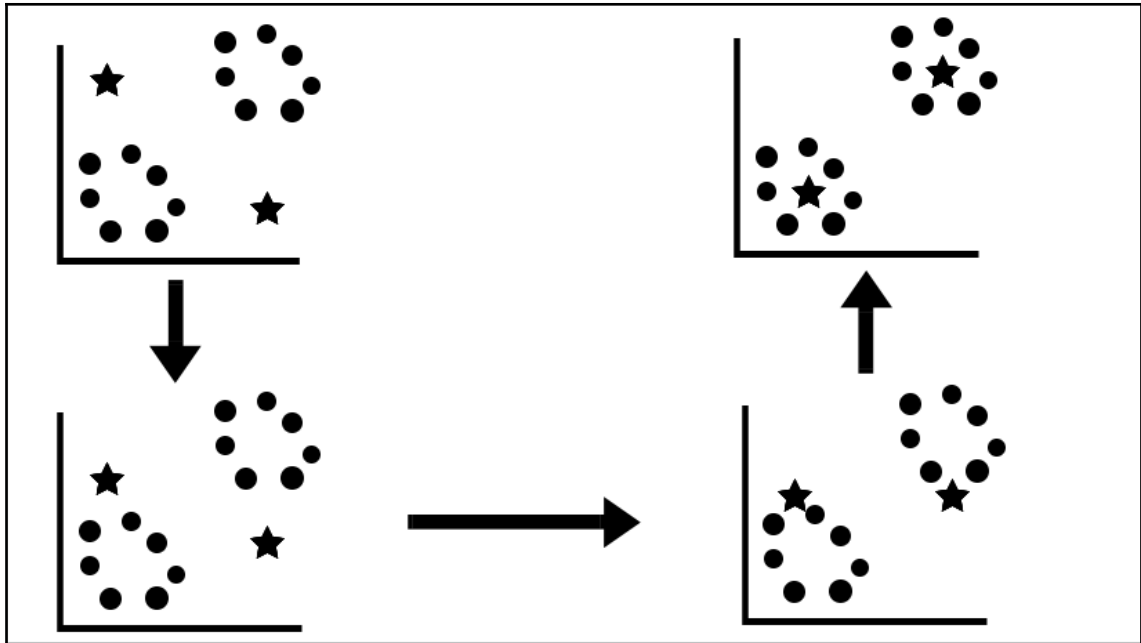


Centroids represented by stars

In the image above, the stars represent the centroids of the algorithm. Notice how in the case above, the cluster centers perfectly fit the two distinct groups. This however, is the most ideal case. In reality, the means or centroids are assigned randomly and with every iteration the cluster centroids move closer to the center of the two groups.

The algorithm is known as the K-means algorithm as we try to find the mean of a group of points as the centroid. Since the mean can only be computed for a set of numeric points, such clustering algorithms only work with numeric data.

In reality the process of grouping these points into two distinct clusters is not this straightforward. A visual representation of this process is illustrated below:



The process assigning centroids in the K-Means algorithm

In the figure above, the process of assigning the random centroids begins on the top left corner. As we go down and towards the top right you will notice how the centroids move closer to the center of the two distinct groups. In reality, the algorithm does not have an optimal end point at which it stops the iteration.

When does the algorithm stop iterating?

Typically, the algorithm looks for two metrics in order to stop the iteration process:

- The distance between the distinct groups or clusters that are formed.
- The distance between each point and the centroid of a cluster.

The most optimal case of cluster formation is when the distance between the distance groups or clusters are as large as possible while the distance between each point and the centroid of a cluster is as small as possible.

Implementing the K-Means algorithm in scikit-learn

Now that you understand how the K-Means algorithm works internally, we can now proceed to implementing the same in scikit-learn. We are going to work with the same fraud detection dataset that we have used in all previous chapters. The key difference is that we are going to drop the target feature which contains the labels and identify the two clusters that are used to detect fraud.

Creating the base K-Means model

In order to load the dataset into our workspace & drop the target feature with the labels we use the code shown below:

```
import pandas as pd
#Reading in the dataset
df = pd.read_csv('fraud_prediction.csv')
#Dropping the target feature & the index
df = df.drop(['Unnamed: 0', 'isFraud'], axis = 1)
```

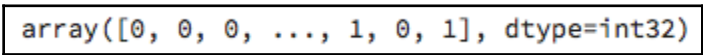
Next, we can implement the K-Means algorithm with 2 cluster means. This choice of 2 cluster means is arbitrary in nature since we know that there should be two distinct clusters as a result of two labels - 'Fraud' & 'Not Fraud' transactions. We can do this by using the code shown below:

```
from sklearn.cluster import KMeans
#Initializing K-means with 2 clusters
k_means = KMeans(n_clusters = 2)
#Fitting the model on the data
k_means.fit(df)
```

In the code above we first import the *KMeans* package from scikit-learn and initialize a model with 2 clusters. We then fit this model to the data using the *.fit()* function. This results in a set of labels as the output. We can extract the labels by using the code shown below:

```
#Extracting labels
target_labels = k_means.predict(df)
#Printing the labels
target_labels
```

The output produced by the code above is an array of labels for each mobile transaction as illustrate in the image below:



```
array([0, 0, 0, ..., 1, 0, 1], dtype=int32)
```

Array of labels

Now that we have a set of labels, we know which cluster each transaction falls into. Mobile transactions that have a label - 0 fall into one group while transactions that have a label - 1 fall into the second group.

Optimal number of clusters

While explaining how the K-Means algorithm works, we mentioned how the algorithm terminates once it finds the optimal number of clusters. When picking clusters arbitrarily using scikit-learn this is not always the case. We need to find the optimal number of clusters in this case.

One way we can do this is by a measure known as - **Inertia**. Inertia measures how close the data points in a cluster are to its centroid. Obviously, a lower inertia signifies that the groups or cluster are tightly packed, which is good.

In order to compute the inertia for the model we use the code shown below:

```
# Inertia of present model
k_means.inertia_
```

The code above produced an inertia value of 4.99×10^{17} which is extremely large and is not a good value of inertia that the model should be producing. This suggests that the individual data points are spread out and are not tightly packed together.

In most cases, we do not really know what the optimal number of clusters are and so we need to plot the inertia scores for different number of clusters. We can do this by using the code shown below:

```
import matplotlib.pyplot as plt
import seaborn as sns

#Initialize a list of clusters from 1 to 10 clusters

clusters = [1,2,3,4,5,6,7,8,9,10]

#Create an empty list in order to store the inertia values

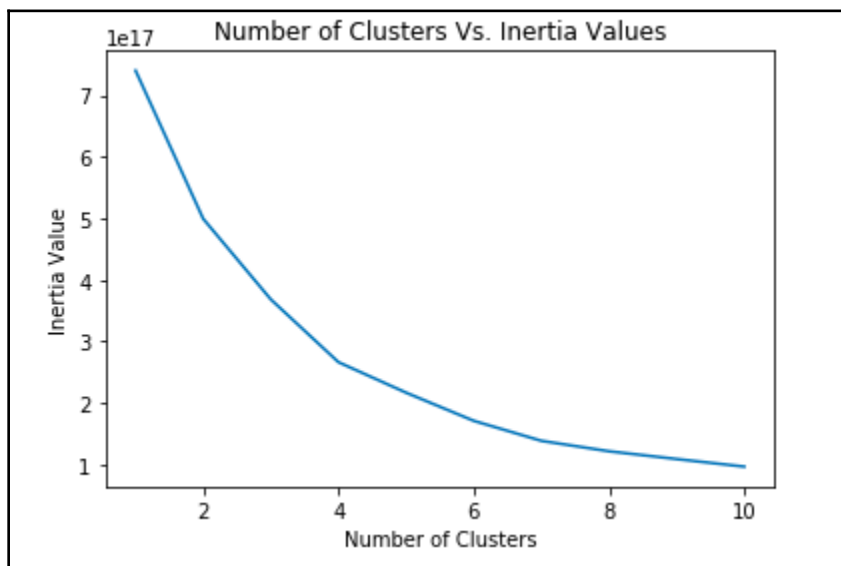
inertia_values = []

for cluster in clusters:
    #Build a k-means model for each cluster value
    k_means = KMeans(n_clusters = cluster)
    #Fit the model to the data
    k_means.fit(df)
    # Store inertia value of each model into the empty list
```

```
inertia_values.append(k_means.inertia_)
# Plot the result

sis.lineplot(x = clusters, y = inertia_values)
plt.xlabel('Number of Clusters')
plt.ylabel('Inertia Value')
plt.title('Number of Clusters Vs. Inertia Values')
plt.show()
```

This results in a plot as illustrated in the image below:



Inertia as a function of the number of clusters

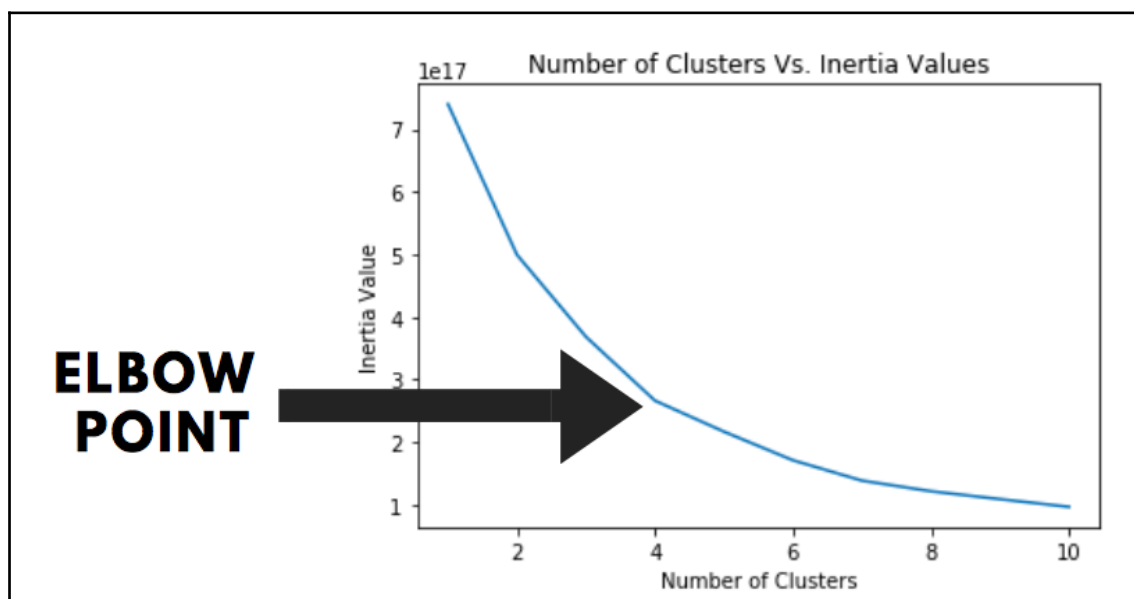
In the code above, we first create a list of clusters that has values from 1 to 10. Each value denotes the number of clusters that will be used in the machine learning model. Next, we create an empty list that will store all the inertia values that each model will produce.

Now we loop over the list of clusters and build and evaluate a K-Means model for each cluster value in the list. Each model now produces an inertia which is stored in the list we initialized at the start of the code block. A simple line plot is then constructed by using the list of clusters along the x-axis and the corresponding inertia values along the y-axis using matplotlib.

The plot tells us that the inertia values are the lowest when the number of clusters is equal to 10. However, having a large number of clusters is also something that we must aim at avoiding as too many groups does not help us generalize well and the characteristics about

each group becomes very specific.

Therefore, the ideal way to choose the best number of clusters for a give problem given that we do not have prior information about the number of groups that we want beforehand is to identify the "elbow point" of the plot. The elbow point is the point at which the rate of decrease in inertia values slows down. This elbow point is illustrated in the image below:



Elbow point of the graph

From the plot above it is clear that the elbow point corresponds to 4 clusters. This could mean that there are 4 distinct types of fraudulent transactions apart from the standard categorization of 'fraud' and 'not fraud'. However, since we know beforehand that the dataset has a binary target feature with 2 categories we will not ponder deeply into why 4 is the ideal number of groups/clusters for this dataset.

Feature Engineering for optimization

Engineering the features in your dataset is a concept that is fundamentally used to improve the performance of your model. Fine-tuning the features to the algorithms design is beneficial because it may lead to a potential increase in accuracy while reducing the generalization errors at the same time. The different kinds of feature engineering techniques that you will learn for optimizing your dataset in this section are as follows:

1. Scaling
2. Principal Component Analysis

Scaling

Scaling is the process of standardizing your data so that the values under every feature fall within a certain range such as -1 to +1. In order to scale the data we subtract each value of a particular feature with the mean of that feature and divide it by the variance of that feature. In order to scale the features in our fraud detection dataset we use the code shown below:

```
from sklearn.preprocessing import StandardScaler

#Setting up the standard scaler

scale_data = StandardScaler()

#Scaling the data

scale_data.fit(df)

df_scaled = scale_data.transform(df)

#Applying the K-Means algorithm on the scaled data

#Initializing K-means with 2 clusters

k_means = KMeans(n_clusters = 2)

#Fitting the model on the data

k_means.fit(df_scaled)

# Inertia of present model

k_means.inertia_
```

In the code above we use the *StandardScalar()* function in order to scale our dataframe and then build a K-Means model with 2 clusters on this scaled data. After evaluating the inertia of the model, the value output is 295,000 which is substantially better than the value of 4.99×10^{17} produced by the model without scaling.

We can then create a new plot of the number of clusters versus the inertia values using the same code as we did earlier with the only difference of replacing the original dataframe with the scaled dataframe as shown below:

```
#Initialize a list of clusters from 1 to 10 clusters

clusters = [1,2,3,4,5,6,7,8,9,10]

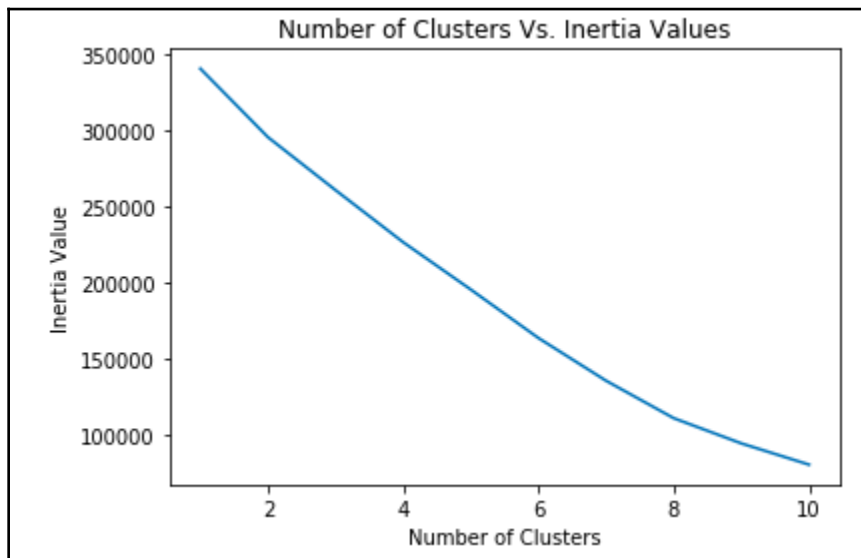
#Create an empty list in order to store the inertia values

inertia_values = []

for cluster in clusters:
    #Build a k-means model for each cluster value
    k_means = KMeans(n_clusters = cluster)
    #Fit the model to the data
    k_means.fit(df_scaled)
    # Store inertia value of each model into the empty list
    inertia_values.append(k_means.inertia_)
# Plot the result

sns.lineplot(x = clusters, y = inertia_values)
plt.xlabel('Number of Clusters')
plt.ylabel('Inertia Value')
plt.title('Number of Clusters Vs. Inertia Values')
plt.show()
```

This produces an output as shown below:



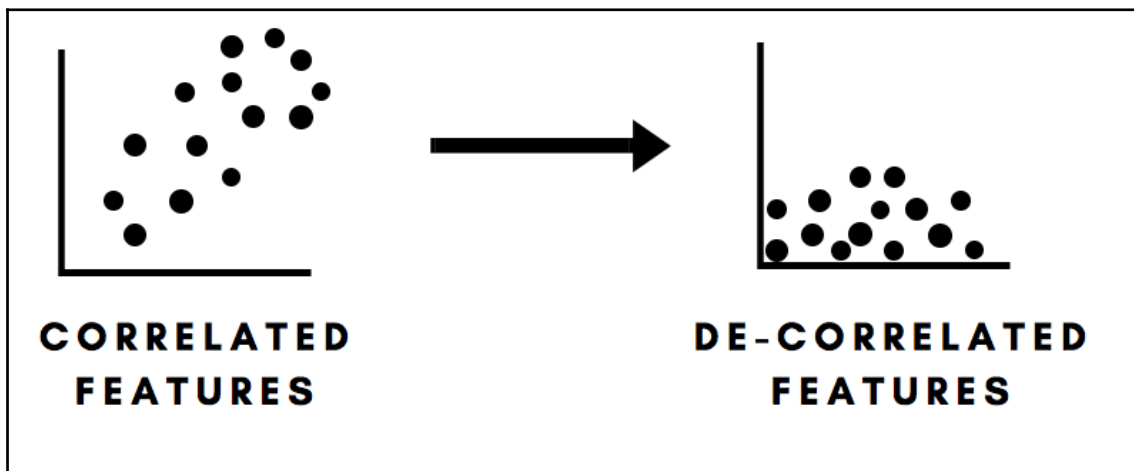
Optimal number of clusters post scaling

We notice that the plot above does not have a very clear "elbow" point where the rate of decrease in the inertia values is lower. However, if we look closely we can find this point at 8 clusters. A good implementation practice is to pick the "elbow" point prior to scaling your data if it's not clear post scaling.

Principal Component Analysis

The Principal Component Analysis or the "PCA" for short is a subset of Dimensionality Reduction. Dimensionality Reduction is a process of reducing the number of features that provide no predictive value to a predictive model. We also optimize and improve the computational efficiency of processing the algorithms. This is because a dataset with a smaller number of features will make it easier for the algorithm to detect patterns faster.

The first step in PCA is called "Decorrelation". Features that are highly correlated with each other provide no value to the predictive model. Therefore, in the decorrelation step, the PCA first takes two highly correlated features and spreads it's data points such that it's aligned across the axis and is not correlated anymore. This process is illustrated in the image shown below:



The process of decorrelation

Once the features are decorrelated, the principal components or features are extracted from the data. These features, are the ones that have high variance and in-turn provide the most value to a predictive model. The features with low variance are discarded and thus the number of dimensions in the dataset is reduced.

In order to perform dimensionality reduction using PCA, we use the code shown below:

```
from sklearn.decomposition import PCA

#Initialize a PCA model with 5 features

pca_model = PCA(n_components = 5)

#Fit the model to the scaled dataframe

pca_model.fit(df_scaled)

#Transform the features so that it is de-correlated

pca_transform = pca_model.transform(df_scaled)

#Check to see if there are only 5 features

pca_transform.shape
```

In the code above, we first import the *PCA* method from scikit-learn. Next, we initialize a *PCA* model with 5 components. Here, we are specifying that we want the PCA to reduce the dataset to the 5 most important features only.

We then fit the PCA model to the dataframe and transform it in order to obtain the decorrelated features. Checking the shape of the final array of features we see that it has only 5 features. Finally, we create a new K-Means model with the principal component features only as shown in the code below:

```
#Applying the K-Means algorithm on the scaled data

#Initializing K-means with 2 clusters

k_means = KMeans(n_clusters = 2)

#Fitting the model on the data

k_means.fit(pca_transform)

# Inertia of present model

k_means.inertia_
```

Evaluating the inertia of the new model did indeed improve its performance. We obtained a lower value of inertia than in the case of the scaled model. Let's now evaluate the inertia scores for different number of principal components or features. In order to this, we use the code shown below:

```
#Initialize a list of principal components

components = [1,2,3,4,5,6,7,8,9,10]

#Create an empty list in order to store the inertia values

inertia_values = []

for comp in components:
    #Initialize a PCA model

    pca_model = PCA(n_components = comp)
    #Fit the model to the dataframe

    pca_model.fit(df_scaled)
    #Transform the features so that it is de-correlated

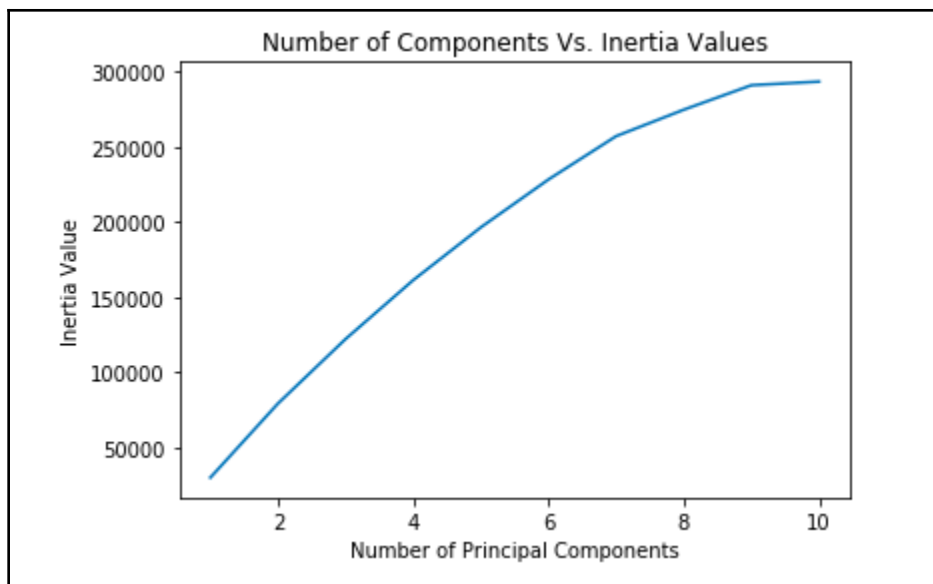
    pca_transform = pca_model.transform(df_scaled)
    #Build a k-means model
    k_means = KMeans(n_clusters = 2)
    #Fit the model to the data
    k_means.fit(pca_transform)
    # Store inertia value of each model into the empty list
    inertia_values.append(k_means.inertia_)
# Plot the result

sns.lineplot(x = components, y = inertia_values)
plt.xlabel('Number of Principal Components')
plt.ylabel('Inertia Value')
plt.title('Number of Components Vs. Inertia Values')
plt.show()
```

In the code above:

1. We first initialize an list in order to store the different principal component values that we want build our models with. These values are from 1 to 10.
2. Next, we initialize an empty list in order to store the inertia values from each and every model.
3. Using each principal component value, we build a new K-Means model and append the inertia value for that model into the empty list.
4. Finally, a plot is constructed between the inertia values and the different values of components.

This plot is illustrated in the image below:



Inertia Values Vs. Number of Principal Components

From the plot above it is clear that the inertia value is the lowest for 1 component.

Cluster Visualization

Visualizing how your clusters are formed is no easy task when the number of variables / dimensions in your dataset are very large. There are two main methods that you can use in order to visualize how the clusters are distributed. They are:

- **t-SNE** : "t-SNE" creates a map of the dataset in two dimensional space.
- **Hierarchical Clustering** : "Hierarchical Clustering" uses a tree based visualization known as a dendrogram in order to create hierarchies

In this section you will learn how you, as the data scientist can implement these visualization techniques in order to create compelling cluster visuals.

t-SNE

The "t-SNE" is an abbreviation that stands for t-distributed Stochastic Neighbor Embedding. The fundamental concept behind the t-SNE is to map a higher dimension to a

two dimensional space. In simple terms, if your dataset has more than 2 features, the t-SNE does a great job at showing you how your entire dataset is visualized on your computer screen!

The first step is to implement the K-Means algorithm and create a set of prediction labels that we can then merge into the un-labelled dataset.

We can do this by using the code shown below:

```
#Reading in the dataset

df = pd.read_csv('fraud_prediction.csv')

#Dropping the target feature & the index

df = df.drop(['Unnamed: 0', 'isFraud'], axis = 1)

#Initializing K-means with 2 clusters

k_means = KMeans(n_clusters = 2)

#Fitting the model on the data

k_means.fit(df)

#Extracting labels

target_labels = k_means.predict(df)

#Converting the labels to a series

target_labels = pd.Series(target_labels)

#Merging the labels to the dataset

df = pd.merge(df, pd.DataFrame(target_labels), left_index=True,
               right_index=True)

#Renaming the target

df['fraud'] = df[0]
df = df.drop([0], axis = 1)
```

Don't worry about how the above segment of code works for the moment as this will be explained in detail in a later section within this chapter when we deal with converting an unsupervised machine learning problem into that of a supervised learning one.

Next, we will create an t-SNE object and fit that into our array of data points that consists of the features only. We then transform the features in the same time so that we can view all the features on a two-dimensional space. This is done in the code segment shown below:

```
from sklearn.manifold import TSNE

#Initialize a TSNE object

tsne_object = TSNE()

#Fit and transform the features using the TSNE object

transformed = tsne_object.fit_transform(features)
```

In the code above:

1. We first initialize the t-SNE object by using the *TSNE()* function.
2. Using the t-SNE object, we fit and transform the data in our features using the *fit_transform()* method.

Next, we create the t-SNE visualization by using the code shown below:

```
#Creating a t-SNE visualization

x_axis = transformed[:,0]

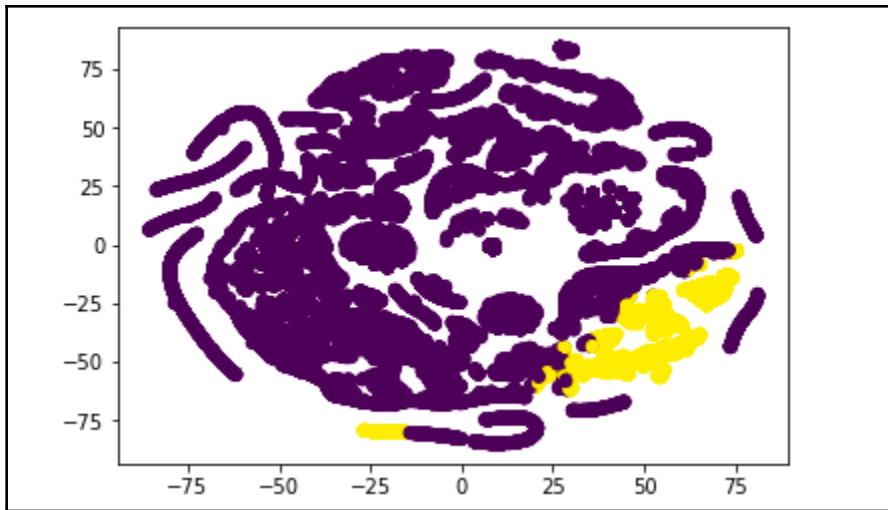
y_axis = transformed[:,1]

plt.scatter(x_axis, y_axis, c = target)

plt.show()
```

In the code above:

1. We extract the first and second features from the set of transformed features for the x and y axis respectively.
2. We then plot a scatter plot and color it by the target labels which was generated earlier using the K-Means algorithm. This generates a plot as illustrated below:



t-SNE visualization

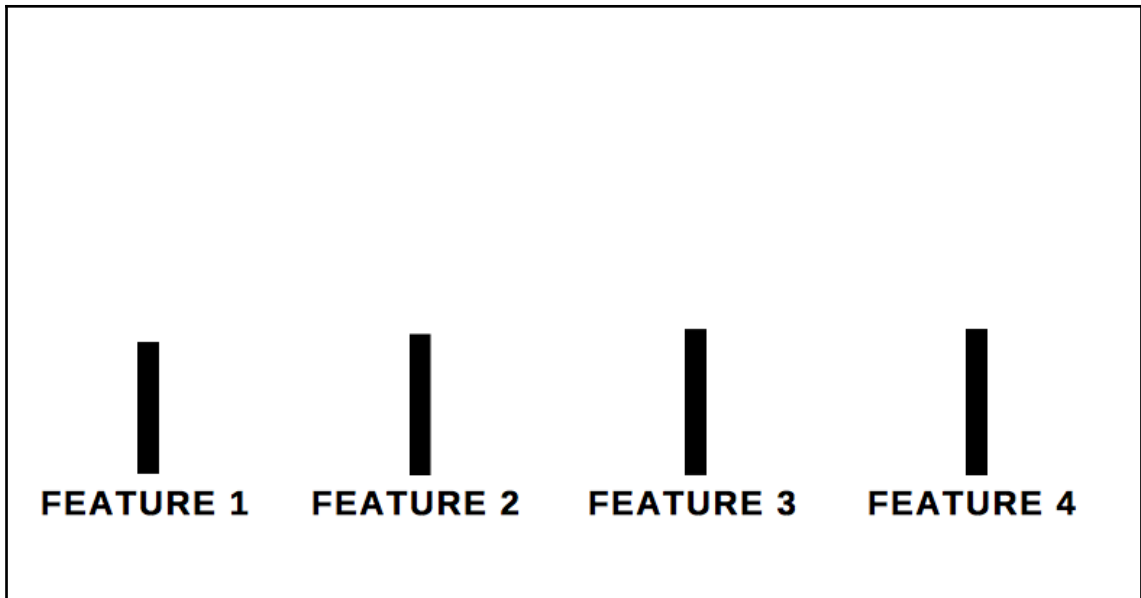
In the plot above, the yellow color represents the transactions which have been assigned the 'fraud' label while the purple labels represent the transactions that have been assigned the 'non-fraudulent' labels.

Hierarchical Clustering

As discussed initially, the hierarchical clustering technique uses the dendrogram in order to visualize clusters or groups. In order to understand how the dendrogram works, we will consider a dataset with 4 features.

Step 1: Individual features as individual clusters

In the first step each feature in the dataset is considered to be its own cluster. This is illustrated in the image shown below:

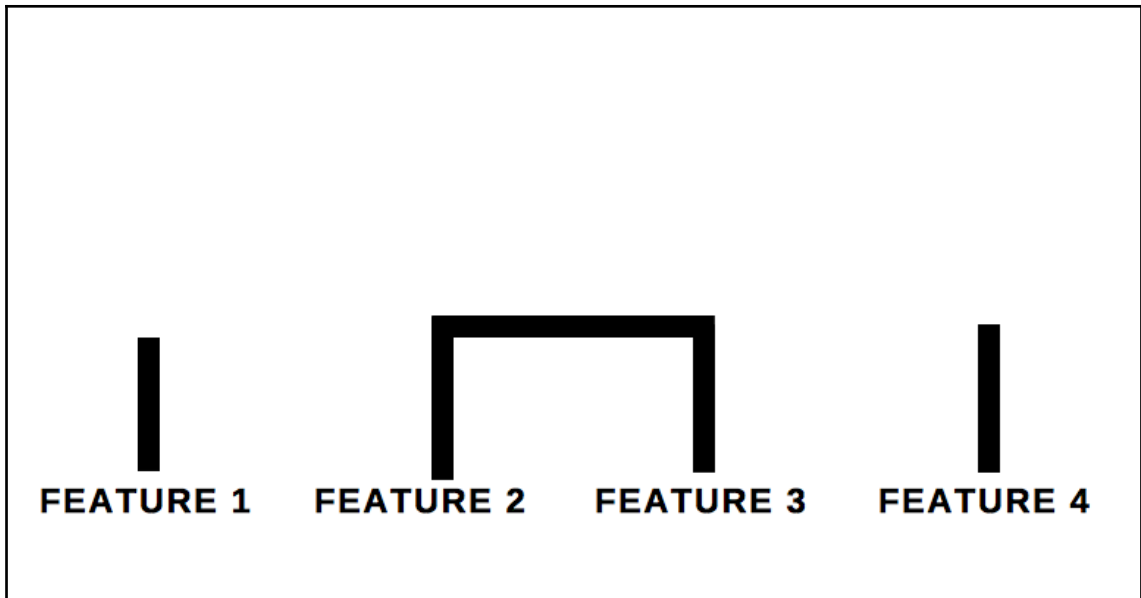


Each feature as a single cluster in the dendrogram

Each feature in the image above is one single cluster at this point of time. The algorithm now searches to find the two features that are closest to each other and merges them into a single cluster.

Step 2: The merge

In this step, the algorithm merges the data points in the two closest features together as one single cluster. This is illustrated in the image below:

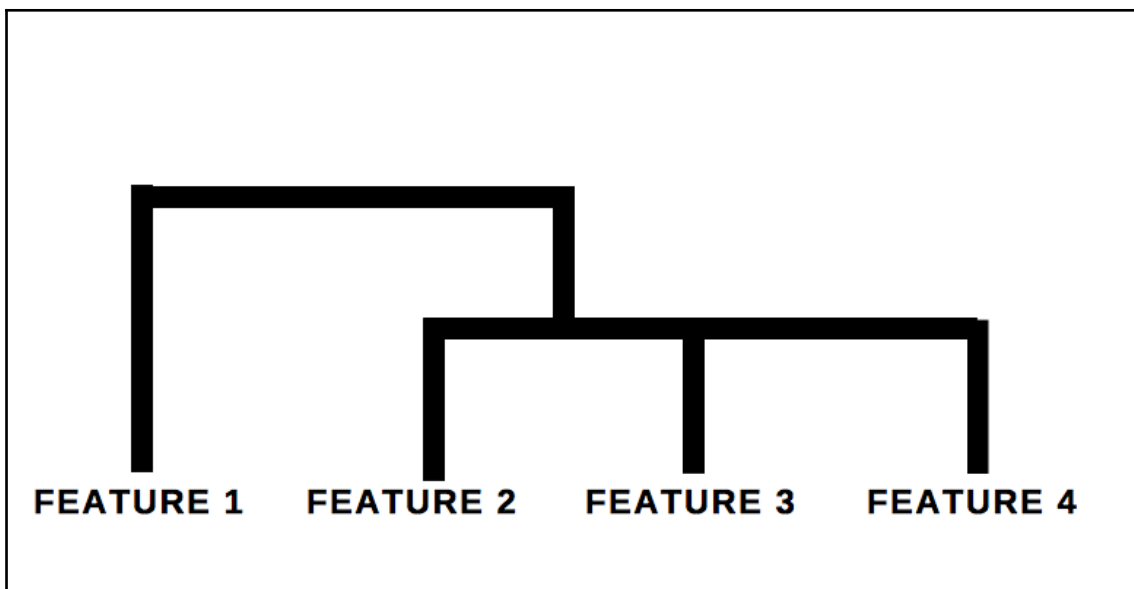


The process in which features merge into a single cluster

From the image above it is clear that the algorithm has now chosen feature 2 and feature 2 and has decided that the data under these two features are the closest to each other.

Step 3: Iteration

The algorithm now continues this process of merging features together iteratively until no more clusters can be formed. The final dendrogram that is formed is illustrated in the image below:



In the image above, feature 2 and feature 3 were grouped into a single cluster. The algorithm then decided that feature 1 and the cluster of feature 2 & 3 are closest to each other. Therefore, these 3 features were clustered into one group. Finally, feature 4 was grouped together with feature 3 alone.

Implementing the hierarchical clustering

Now that you have learnt how the hierarchical clustering works we can now implement this concept. In order to create a hierarchical cluster we use the code shown below:

```
from scipy.cluster.hierarchy import linkage
from scipy.cluster.hierarchy import dendrogram
import numpy as np
import matplotlib.pyplot as plt

#Creating an array of 4 features

array = np.array([[1,2,3,4], [5,6,7,8], [2,3,4,5], [5,6,4,3]])

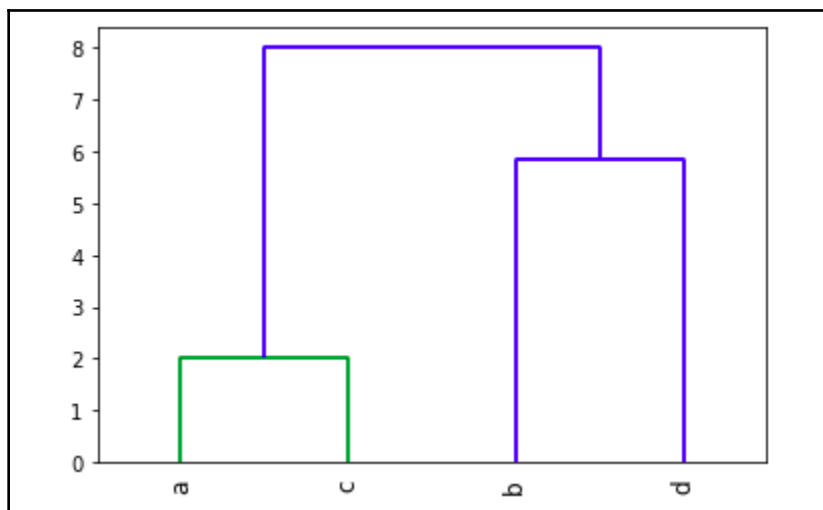
feature_names = ['a', 'b', 'c', 'd']

#Creating clusters

clusters = linkage(array, method = 'complete')
```

```
#Creating a dendrogram  
  
dendrogram(clusters, labels = feature_names, leaf_rotation = 90)  
  
plt.show()
```

The code above results in a dendrogram as illustrated in the image below:



Dendrogram

In the code above:

1. We first create an array with 4 columns.
2. We then use the *linkage* function in order to create the clusters. Within the function, we specify the method argument as complete in order to indicate that we want the entire dendrogram.
3. Finally, we use the *dendrogram* function in order to create the dendrogram with the clusters. We set the label names to the list of feature names created earlier in the code.

Going from unsupervised to supervised learning

The eventual goal of unsupervised learning to take a dataset with no labels and assign these

labels to each row of the dataset so that we can run a supervised learning algorithm through it. This allows us to create predictions that make use of the labels.

In this section you will learn how to convert the labels generated by the unsupervised machine learning algorithm into a decision tree that makes use of these labels.

Creating a labelled dataset

The first step is to convert the labels generated by an unsupervised machine learning algorithm such as the K-Means and append it to the dataset. We can do this by using the code shown below:

```
#Reading in the dataset

df = pd.read_csv('fraud_prediction.csv')

#Dropping the target feature & the index

df = df.drop(['Unnamed: 0', 'isFraud'], axis = 1)
```

In the code above, we have read in the fraud detection dataset and dropped the target and index columns.

```
#Initializing K-means with 2 clusters

k_means = KMeans(n_clusters = 2)

#Fitting the model on the data

k_means.fit(df)
```

Next, we initialize and fit a K-Means model with 2 clusters.

```
#Extracting labels

target_labels = k_means.predict(df)

#Converting the labels to a series

target_labels = pd.Series(target_labels)

#Merging the labels to the dataset

df = pd.merge(df, pd.DataFrame(target_labels), left_index=True,
right_index=True)
```

Finally, we create the target labels using the *predict()* method and convert it into a pandas series. We then merge this series to the data frame in order to create our labelled dataset.

Building the decision tree

Now that we have the labelled dataset we can now create a decision tree in order to convert the unsupervised machine learning problem into that of a supervised machine learning one.

In order to do this we first start with all the necessary package imports as shown in the code below:

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.externals.six import StringIO
from IPython.display import Image
from sklearn.tree import export_graphviz
import pydotplus
from sklearn import tree
```

Next, we rename the target column into a name that is appropriate. This is because, when we merged the target labels created by the K-Means algorithm, it produced '0' as the default name. We can do this by using the code shown below:

```
#Renaming the target

df['fraud'] = df[0]
df = df.drop([0], axis = 1)
```

Next, we build the decision tree classification algorithm by using the code shown below:

```
#Creating the features

features = df.drop('fraud', axis = 1).values

target = df['fraud'].values

#Initializing an empty DT classifier with a random state value of 42

dt_classifier = DecisionTreeClassifier(criterion = 'gini', random_state =
42)

#Fitting the classifier on the training data

dt_classifier.fit(features, target)
```

In the code above we first create the features and target variables and initialize a decision

tree classifier. We then fit the classifier on the features and target.

Finally, we want to visualize the decision tree. We can do this by using the code shown below:

```
#Creating a data frame with the features only

features = df.drop('fraud', axis = 1)

dot_data = tree.export_graphviz(dt_classifier, out_file=None,
                                feature_names= features.columns)

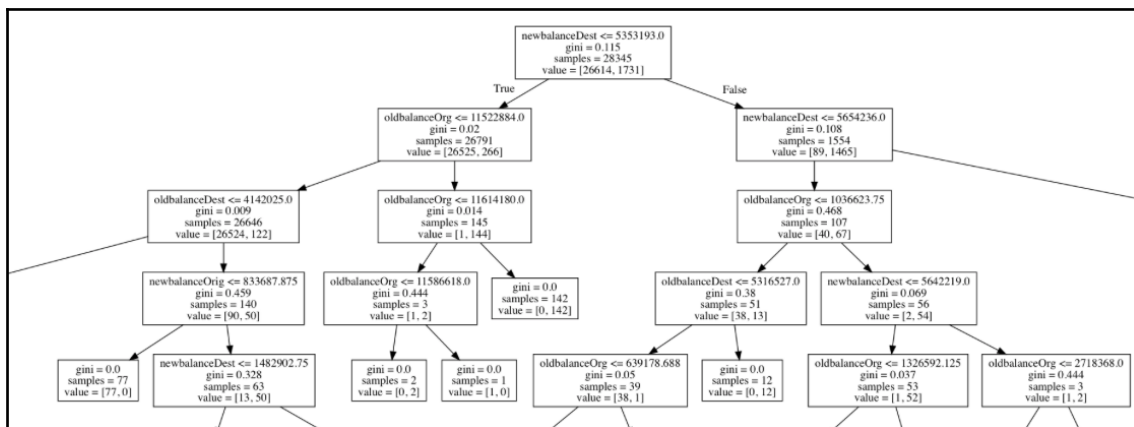
# Draw graph

graph = pydotplus.graph_from_dot_data(dot_data)

#Show graph

Image(graph.create_png())
```

This results in a decision tree as illustrated in the image below:



A part of the decision tree created

Summary

In this chapter you learn the underlying mechanism behind how the K-Means algorithm works in order to cluster un-labelled data points into clusters or groups. You then learnt how you can implement the same using scikit-learn and further expanded on the feature

engineering aspect of the implementation.

Having learnt how to visualize clusters using hierarchical clustering and t-SNE, you now learnt how to map a multi-dimensional dataset into a two dimensional space. Finally, you learnt how to convert the unsupervised machine learning problem into that of a supervised learning one using decision trees.

In the next and final chapter, you will learn how to formally evaluate the performance of all the machine learning algorithms that you have built so far!

Index