CHAPTER 8

Unsupervised Learning: Clustering

In the previous chapter, we explored dimensionality reduction, which is one type of unsupervised learning. In this chapter, we will explore *clustering*, a category of unsupervised learning techniques that allows us to discover hidden structures in data.

Both clustering and dimensionality reduction summarize the data. Dimensionality reduction compresses the data by representing it using new, fewer features while still capturing the most relevant information. Similarly, clustering is a way to reduce the volume of data and find patterns. However, it does so by categorizing the original data and not by creating new variables. Clustering algorithms assign observations to subgroups that consist of similar data points. The goal of clustering is to find a natural grouping in data so that items in a given cluster are more similar to each other than to those of different clusters. Clustering serves to better understand the data through the lens of several categories or groups created. It also permits the automatic categorization of new objects according to the learned criteria.

In the field of finance, clustering has been used by traders and investment managers to find homogeneous groups of assets, classes, sectors, and countries based on similar characteristics. Clustering analysis augments trading strategies by providing insights into categories of trading signals. The technique has been used to segment customers or investors into a number of groups to better understand their behavior and to perform additional analysis.

In this chapter, we will discuss fundamental clustering techniques and introduce three case studies in the areas of portfolio management and trading strategy development.

In "Case Study 1: Clustering for Pairs Trading" on page 243, we use clustering methods to select pairs of stocks for a trading strategy. A *pairs trading strategy* involves matching a long position with a short position in two financial instruments

that are closely related. Finding appropriate pairs can be a challenge when the number of instruments is high. In this case study, we demonstrate how clustering can be a useful technique in trading strategy development and other similar situations.

In "Case Study 2: Portfolio Management: Clustering Investors" on page 259, we identify clusters of investors with similar abilities and willingness to take risks. We show how clustering techniques can be used for effective asset allocation and portfolio rebalancing. This illustrates how part of the portfolio management process can be automated, which is immensely useful for investment managers and robo-advisors alike.

In "Case Study 3: Hierarchical Risk Parity" on page 267, we use a clustering-based algorithm to allocate capital into different asset classes and compare the results against other portfolio allocation techniques.

In this chapter, we will learn about the following concepts related to clustering techniques:

- Basic concepts of models and techniques used for clustering.
- How to implement different clustering techniques in Python.
- How to effectively perform visualizations of clustering outcomes.
- Understanding the intuitive meaning of clustering results.
- How to choose the right clustering techniques for a problem.
- Selecting the appropriate number of clusters in different clustering algorithms.
- Building hierarchical clustering trees using Python.

This Chapter's Code Repository

A Python-based master template for clustering, along with the Jupyter notebook for the case studies presented in this chapter are in Chapter 8 - Unsup. Learning - Clustering in the code repository for this book. To work through any machine learning problems in Python involving the models for clustering (such as *k*-means, hierarchical clustering, etc.) presented in this chapter, readers simply need to modify the template to align with their problem statement. Similar to the previous chapters, the case studies presented in this chapter use the standard Python master template with the standardized model development steps presented in Chapter 2. For the clustering case studies, steps 6 (Model Tuning and Grid Search) and 7 (Finalizing the Model) have merged with step 5 (Evaluate Algorithms and Models).

Clustering Techniques

There are many types of clustering techniques, and they differ with respect to their strategy of identifying groupings. Choosing which technique to apply depends on the nature and structure of the data. In this chapter, we will cover the following three clustering techniques:

- *k*-means clustering
- Hierarchical clustering
- Affinity propagation clustering

The following section summarizes these clustering techniques, including their strengths and weaknesses. Additional details for each of the clustering methods are provided in the case studies.

k-means Clustering

k-means is the most well-known clustering technique. The algorithm of *k*-means aims to find and group data points into classes that have high similarity between them. This similarity is understood as the opposite of the distance between data points. The closer the data points are, the more likely they are to belong to the same cluster.

The algorithm finds *k* centroids and assigns each data point to exactly one cluster with the goal of minimizing the within-cluster variance (called *inertia*). It typically uses the Euclidean distance (ordinary distance between two points), but other distance metrics can be used. The *k*-means algorithm delivers a local optimum for a given *k* and proceeds as follows:

1. This algorithm specifies the number of clusters.
2. Data points are randomly selected as cluster centers.
3. Each data point is assigned to the cluster center it is nearest to.
4. Cluster centers are updated to the mean of the assigned points.
5. Steps 3–4 are repeated until all cluster centers remain unchanged.

In simple terms, we randomly move around the specified number of centroids in each iteration, assigning each data point to the closest centroid. Once we have done that, we calculate the mean distance of all points in each centroid. Then, once we can no longer reduce the minimum distance from data points to their respective centroids, we have found our clusters.

k-means hyperparameters

The *k*-means hyperparameters include:

Number of clusters
: The number of clusters and centroids to generate.

Maximum iterations
: Maximum iterations of the algorithm for a single run.

Number initial
: The number of times the algorithm will be run with different centroid seeds. The final result will be the best output of the defined number of consecutive runs, in terms of inertia.

With *k*-means, different random starting points for the cluster centers often result in very different clustering solutions. Therefore, the *k*-means algorithm is run in sklearn with at least 10 different random initializations, and the solution occurring the greatest number of times is chosen.

The strengths of *k*-means include its simplicity, wide range of applicability, fast convergence, and linear scalability to large data while producing clusters of an even size. It is most useful when we know the exact number of clusters, *k*, beforehand. In fact, a main weakness of *k*-means is having to tune this hyperparameter. Additional drawbacks include the lack of a guarantee to find a global optimum and its sensitivity to outliers.

Implementation in Python

Python's sklearn library offers a powerful implementation of *k*-means. The following code snippet illustrates how to apply *k*-means clustering on a dataset:

```
from sklearn.cluster import KMeans
#Fit with k-means
k_means = KMeans(n_clusters=nclust)
k_means.fit(X)
```

The number of clusters is the key hyperparameter to be tuned. We will look at the *k*-means clustering technique in case studies 1 and 2 of this chapter, in which further details on choosing the right number of clusters and detailed visualizations are provided.

Hierarchical Clustering

Hierarchical clustering involves creating clusters that have a predominant ordering from top to bottom. The main advantage of hierarchical clustering is that we do not need to specify the number of clusters; the model determines that by itself. This

clustering technique is divided into two types: agglomerative hierarchical clustering and divisive hierarchical clustering.

Agglomerative hierarchical clustering is the most common type of hierarchical clustering and is used to group objects based on their similarity. It is a "bottom-up" approach where each observation starts in its own cluster, and pairs of clusters are merged as one moves up the hierarchy. The agglomerative hierarchical clustering algorithm delivers a *local optimum* and proceeds as follows:

1. Make each data point a single-point cluster and form *N* clusters.
2. Take the two closest data points and combine them, leaving *N-1* clusters.
3. Take the two closest clusters and combine them, forming *N-2* clusters.
4. Repeat step 3 until left with only one cluster.

Divisive hierarchical clustering works "top-down" and sequentially splits the remaining clusters to produce the most distinct subgroups.

Both produce *N-1* hierarchical levels and facilitate the clustering creation at the level that best partitions data into homogeneous groups. We will focus on the more common agglomerative clustering approach.

Hierarchical clustering enables the plotting of *dendrograms*, which are visualizations of a binary hierarchical clustering. A dendrogram is a type of tree diagram showing hierarchical relationships between different sets of data. They provide an interesting and informative visualization of hierarchical clustering results. A dendrogram contains the memory of the hierarchical clustering algorithm, so you can tell how the cluster is formed simply by inspecting the chart.

Figure 8-1 shows an example of dendrograms based on hierarchical clustering. The distance between data points represents dissimilarities, and the height of the blocks represents the distance between clusters.

Observations that fuse at the bottom are similar, while those at the top are quite different. With dendrograms, conclusions are made based on the location of the vertical axis rather than on the horizontal one.

The advantages of hierarchical clustering are that it is easy to implement it, does not require one to specify the number of clusters, and it produces dendrograms that are very useful in understanding the data. However, the time complexity for hierarchical clustering can result in long computation times relative to other algorithms, such as *k*-means. If we have a large dataset, it can be difficult to determine the correct number of clusters by looking at the dendrogram. Hierarchical clustering is very sensitive to outliers, and in their presence, model performance decreases significantly.

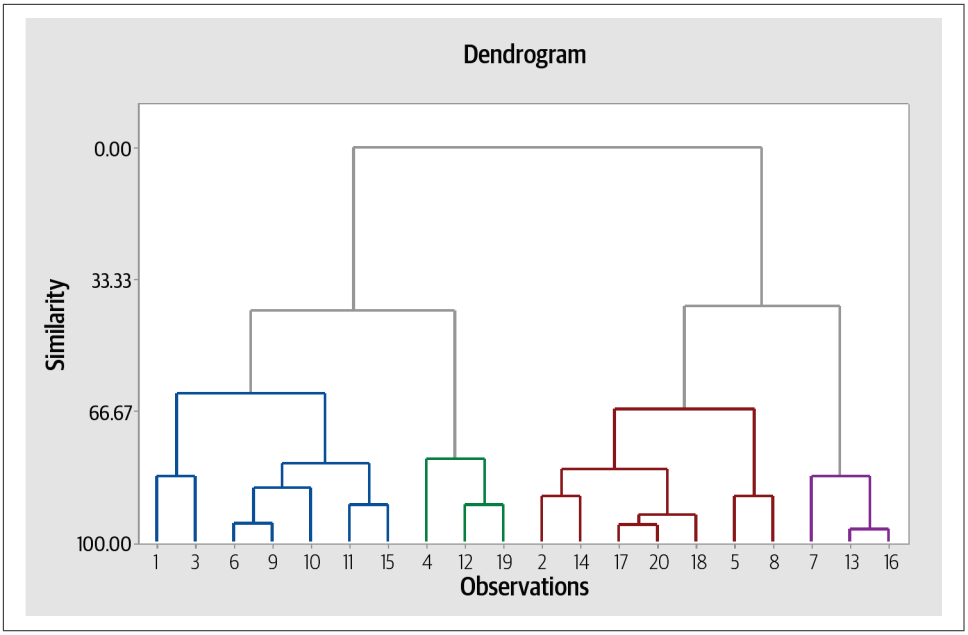


Figure 8-1. Hierarchical clustering

Implementation in Python

The following code snippet illustrates how to apply agglomerative hierarchical clustering with four clusters on a dataset:

```
from sklearn.cluster import AgglomerativeClustering
model = AgglomerativeClustering(n_clusters=4, affinity='euclidean',\
  linkage='ward')
clust_labels1 = model.fit_predict(X)
```

More details regarding the hyperparameters of agglomerative hierarchical clustering can be found on the sklearn website. We will look at the hierarchical clustering technique in case studies 1 and 3 in this chapter.

Affinity Propagation Clustering

Affinity propagation creates clusters by sending messages between data points until convergence. Unlike clustering algorithms such as *k*-means, affinity propagation does not require the number of clusters to be determined or estimated before running the algorithm. Two important parameters are used in affinity propagation to determine the number of clusters: the *preference*, which controls how many *exemplars* (or prototypes) are used; and the *damping factor*, which dampens the responsibility and availability of messages to avoid numerical oscillations when updating these messages.

A dataset is described using a small number of exemplars. These are members of the input set that are representative of clusters. The affinity propagation algorithm takes in a set of pairwise similarities between data points and finds clusters by maximizing the total similarity between data points and their exemplars. The messages sent between pairs represent the suitability of one sample to be the exemplar of the other, which is updated in response to the values from other pairs. This updating happens iteratively until convergence, at which point the final exemplars are chosen, and we obtain the final clustering.

In terms of strengths, affinity propagation does not require the number of clusters to be determined before running the algorithm. The algorithm is fast and can be applied to large similarity matrices. However, the algorithm often converges to suboptimal solutions, and at times it can fail to converge.

Implementation in Python

The following code snippet illustrates how to implement the affinity propagation algorithm for a dataset:

```
from sklearn.cluster import AffinityPropagation
# Initialize the algorithm and set the number of PC's
ap = AffinityPropagation()
ap.fit(X)
```

More details regarding the hyperparameters of affinity propagation clustering can be found on the sklearn website. We will look at the affinity propagation technique in case studies 1 and 2 in this chapter.

Case Study 1: Clustering for Pairs Trading

A pairs trading strategy constructs a portfolio of correlated assets with similar market risk factor exposure. Temporary price discrepancies in these assets can create opportunities to profit through a long position in one instrument and a short position in another. A pairs trading strategy is designed to eliminate market risk and exploit these temporary discrepancies in the relative returns of stocks.

The fundamental premise in pairs trading is that *mean reversion* is an expected dynamic of the assets. This mean reversion should lead to a long-run equilibrium relationship, which we try to approximate through statistical methods. When moments of (presumably temporary) divergence from this long-term trend arise, one can possibly profit. The key to successful pairs trading is the ability to select the right pairs of assets to be used.

Traditionally, trial and error was used for pairs selection. Stocks or instruments that were merely in the same sector or industry were grouped together. The idea was that if these stocks were for companies in similar industries, their stocks should move

similarly as well. However, this was and is not necessarily the case. Additionally, with a large universe of stocks, finding a suitable pair is a difficult task, given that there are a total of $n(n–1)/2$ possible pairs, where n is the number of instruments. Clustering can be a useful technique here.

In this case study, we will use clustering algorithms to select pairs of stocks for a pairs trading strategy.

This case study will focus on:

- Evaluating three main clustering methods: *k*-means, hierarchical clustering, and affinity propagation clustering.
- Understanding approaches to finding the right number of clusters in *k*-means and hierarchical clustering.
- Visualizing data in the clusters, including viewing dendrograms.
- Selecting the right clustering algorithm.

Blueprint for Using Clustering to Select Pairs

1. Problem definition

Our goal in this case study is to perform clustering analysis on the stocks in the S&P 500 to come up with pairs for a pairs trading strategy. S&P 500 stock data was obtained using `pandas_datareader` from Yahoo Finance. It includes price data from 2018 onwards.

2. Getting started—loading the data and Python packages

The list of the libraries used for data loading, data analysis, data preparation, and model evaluation are shown below.

2.1. Loading the Python packages. The details of most of these packages and functions have been provided in Chapters 2 and 4. The use of these packages will be demonstrated in different steps of the model development process.

```
Packages for clustering

    from sklearn.cluster import KMeans, AgglomerativeClustering, AffinityPropagation
    from scipy.cluster.hierarchy import fcluster
```

```
from scipy.cluster.hierarchy import dendrogram, linkage, cophenet
from scipy.spatial.distance import pdist
from sklearn.metrics import adjusted_mutual_info_score
from sklearn import cluster, covariance, manifold
```

```
Packages for data processing and visualization
```

```
# Load libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from pandas import read_csv, set_option
from pandas.plotting import scatter_matrix
import seaborn as sns
from sklearn.preprocessing import StandardScaler
import datetime
import pandas_datareader as dr
import matplotlib.ticker as ticker
from itertools import cycle
```

2.2. Loading the data. The stock data is loaded below.[1]

```
dataset = read_csv('SP500Data.csv', index_col=0)
```

3. Exploratory data analysis

We take a quick look at the data in this section.

3.1. Descriptive statistics. Let us look at the shape of the data:

```
# shape
dataset.shape
```

```
Output
```

```
(448, 502)
```

The data contains 502 columns and 448 observations.

3.2. Data visualization. We will take a detailed look into the visualization postclustering.

4. Data preparation

We prepare the data for modeling in the following sections.

1 Refer to the Jupyter notebook to understand fetching price data using `pandas_datareader`.

4.1. Data cleaning. In this step, we check for NAs in the rows and either drop them or fill them with the mean of the column:

```
#Checking for any null values and removing the null values'''
print('Null Values =',dataset.isnull().values.any())
```

Output

```
Null Values = True
```

Let us get rid of the columns with more than 30% missing values:

```
missing_fractions = dataset.isnull().mean().sort_values(ascending=False)
missing_fractions.head(10)
drop_list = sorted(list(missing_fractions[missing_fractions > 0.3].index))
dataset.drop(labels=drop_list, axis=1, inplace=True)
dataset.shape
```

Output

```
(448, 498)
```

Given that there are null values, we drop some rows:

```
# Fill the missing values with the last value available in the dataset.
dataset=dataset.fillna(method='ffill')
```

The data cleaning steps identified those with missing values and populated them. This step is important for creating a meaningful, reliable, and clean dataset that can be used without any errors in the clustering.

4.2. Data transformation. For the purpose of clustering, we will be using *annual returns* and *variance* as the variables, as they are primary indicators of stock performance and volatility. The following code prepares these variables:

```
#Calculate average annual percentage return and volatilities
returns = pd.DataFrame(dataset.pct_change().mean() * 252)
returns.columns = ['Returns']
returns['Volatility'] = dataset.pct_change().std() * np.sqrt(252)
data = returns
```

All the variables should be on the same scale before applying clustering; otherwise, a feature with large values will dominate the result. We use `StandardScaler` in sklearn to standardize the dataset features onto unit scale (mean = 0 and variance = 1):

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler().fit(data)
rescaledDataset = pd.DataFrame(scaler.fit_transform(data),\
  columns = data.columns, index = data.index)
# summarize transformed data
rescaledDataset.head(2)
```

```
Output
```

	Returns		Volatility
ABT	0.794067	–0.702741	ABBV

With the data prepared, we can now explore the clustering algorithms.

5. Evaluate algorithms and models

We will look at the following models:

- *k*-means
- Hierarchical clustering (agglomerative clustering)
- Affinity propagation

5.1. k-means clustering. Here, we model using *k*-means and evaluate two ways to find the optimal number of clusters.

5.1.1. Finding the optimal number of clusters. We know that *k*-means initially assigns data points to clusters randomly and then calculates centroids or mean values. Further, it calculates the distances within each cluster, squares these, and sums them to get the sum of squared errors.

The basic idea is to define *k* clusters so that the total within-cluster variation (or error) is minimized. The following two methods are useful in finding the number of clusters in *k*-means:

Elbow method
: Based on the sum of squared errors (SSE) within clusters

Silhouette method
: Based on the silhouette score

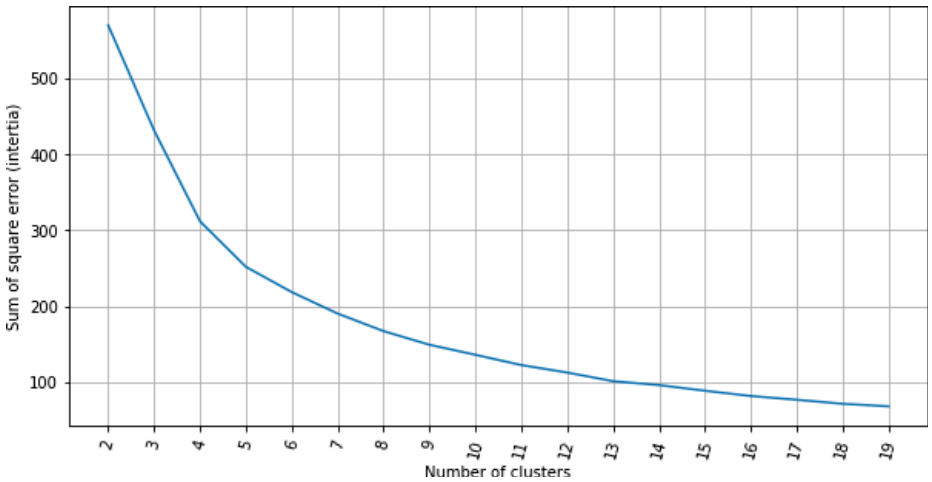
First, let's examine the elbow method. The SSE for each point is the square of the distance of the point from its representation (i.e., its predicted cluster center). The sum of squared errors is plotted for a range of values for the number of clusters. The first cluster will add much information (explain a lot of variance), but eventually the marginal gain will drop, giving an angle in the graph. The number of clusters is chosen at this point; hence it is referred to as the "elbow criterion."

Let us implement this in Python using the sklearn library and plot the SSE for a range of values for *k*:

```
distortions = []
max_loop=20
for k in range(2, max_loop):
```

```
        kmeans = KMeans(n_clusters=k)
        kmeans.fit(X)
        distortions.append(kmeans.inertia_)
    fig = plt.figure(figsize=(15, 5))
    plt.plot(range(2, max_loop), distortions)
    plt.xticks([i for i in range(2, max_loop)], rotation=75)
    plt.grid(True)
```

Output



Inspecting the sum of squared errors chart, it appears the elbow kink occurs around five or six clusters for this data. Certainly we can see that as the number of clusters increases past six, the SSE within clusters begins to plateau.

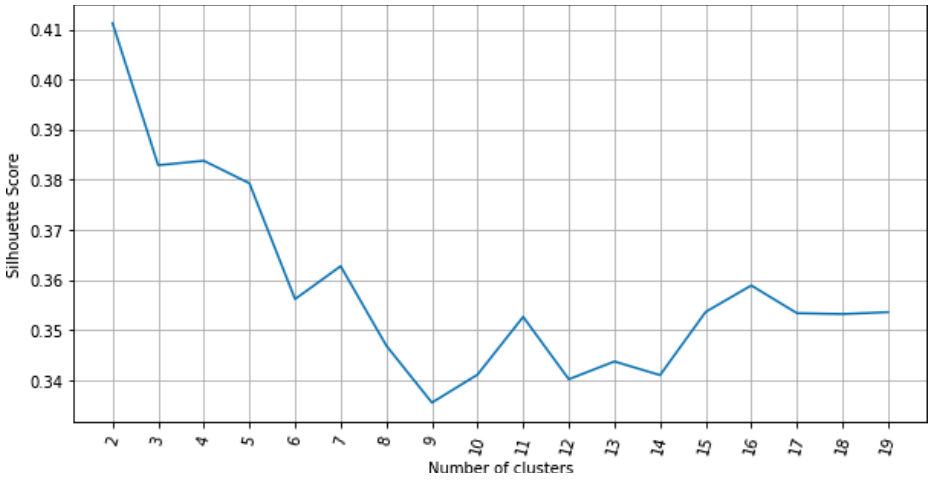
Now let's look at the silhouette method. The silhouette score measures how similar a point is to its own cluster (*cohesion*) compared to other clusters (*separation*). The range of the silhouette value is between 1 and –1. A high value is desirable and indicates that the point is placed in the correct cluster. If many points have a negative silhouette value, that may indicate that we have created too many or too few clusters.

Let us implement this in Python using the sklearn library and plot the silhouette score for a range of values for *k*:

```
from sklearn import metrics

silhouette_score = []
for k in range(2, max_loop):
        kmeans = KMeans(n_clusters=k,  random_state=10, n_init=10, n_jobs=-1)
        kmeans.fit(X)
        silhouette_score.append(metrics.silhouette_score(X, kmeans.labels_, \
          random_state=10))
fig = plt.figure(figsize=(15, 5))
```

```
plt.plot(range(2, max_loop), silhouette_score)
plt.xticks([i for i in range(2, max_loop)], rotation=75)
plt.grid(True)
```

Output



Looking at the silhouette score chart, we can see that there are various parts of the graph at which a kink can be seen. Since there is not much of a difference in the SSE after six clusters, it implies that six clusters is a preferred choice in this *k*-means model.

Combining information from both methods, we infer the optimum number of clusters to be six.

5.1.2. Clustering and visualization. Let us build the *k*-means model with six clusters and visualize the results:

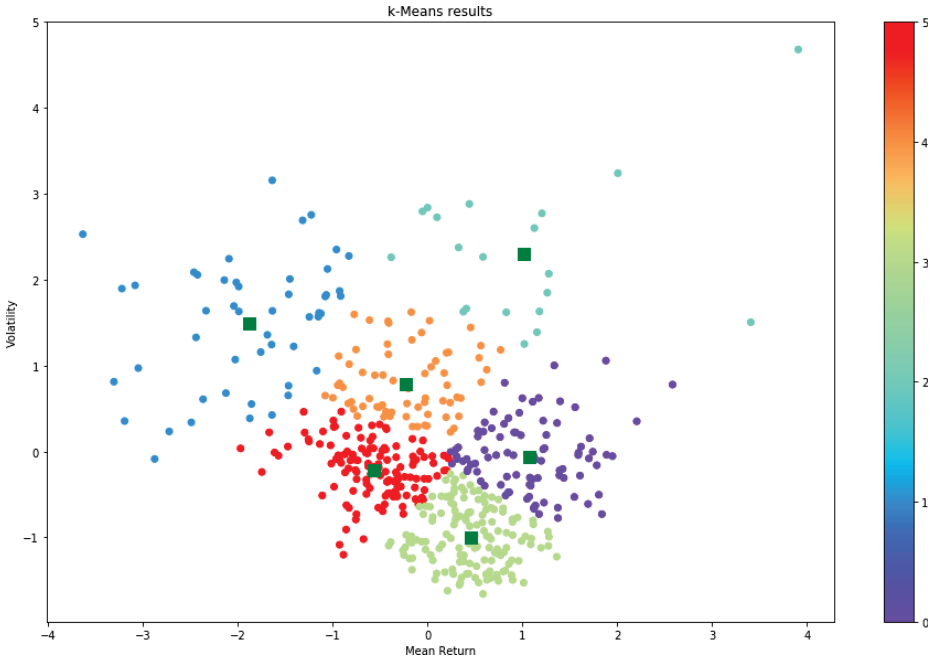
```
nclust=6
#Fit with k-means
k_means = cluster.KMeans(n_clusters=nclust)
k_means.fit(X)
#Extracting labels
target_labels = k_means.predict(X)
```

Visualizing how clusters are formed is no easy task when the number of variables in the dataset is very large. A basic scatterplot is one method for visualizing a cluster in a two-dimensional space. We create one below to identify the relationships inherent in our data:

```
centroids = k_means.cluster_centers_
fig = plt.figure(figsize=(16,10))
ax = fig.add_subplot(111)
```

```
scatter = ax.scatter(X.iloc[:,0],X.iloc[:,1], c=k_means.labels_, \
  cmap="rainbow", label = X.index)
ax.set_title('k-means results')
ax.set_xlabel('Mean Return')
ax.set_ylabel('Volatility')
plt.colorbar(scatter)

plt.plot(centroids[:,0],centroids[:,1],'sg',markersize=11)
```

Output



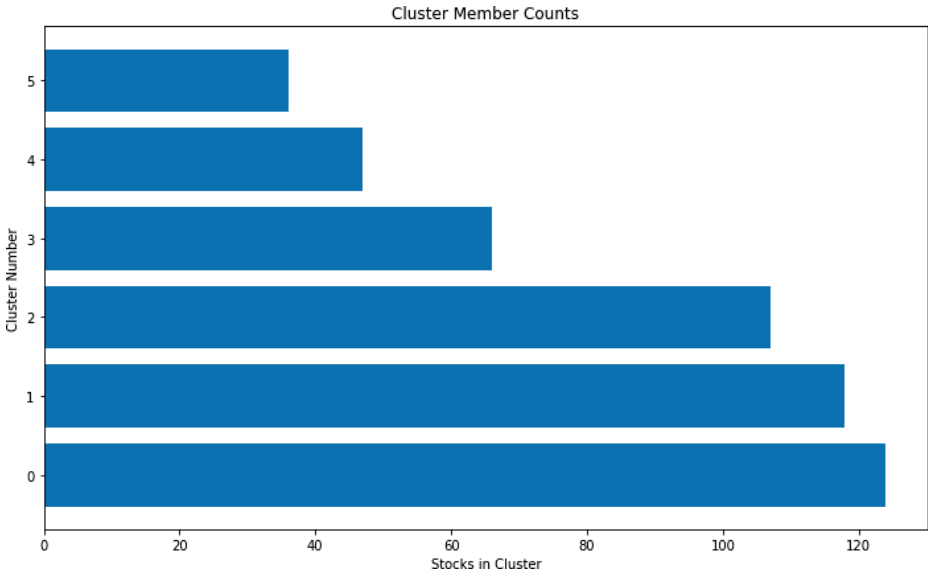
In the preceding plot, we can somewhat see that there are distinct clusters separated by different colors (full-color version available on GitHub). The grouping of data in the plot seems to be separated quite well. There is also a degree of separation in the centroids of the clusters, represented by square dots.

Let us look at the number of stocks in each of the clusters:

```
# show number of stocks in each cluster
clustered_series = pd.Series(index=X.index, data=k_means.labels_.flatten())
# clustered stock with its cluster label
clustered_series_all = pd.Series(index=X.index, data=k_means.labels_.flatten())
clustered_series = clustered_series[clustered_series != -1]

plt.figure(figsize=(12,7))
plt.barh(
```

```
        range(len(clustered_series.value_counts())), # cluster labels, y axis
        clustered_series.value_counts()
    )
    plt.title('Cluster Member Counts')
    plt.xlabel('Stocks in Cluster')
    plt.ylabel('Cluster Number')
    plt.show()
```

Output



The number of stocks per cluster ranges from around 40 to 120. Although the distribution is not equal, we have a significant number of stocks in each cluster.

Let's look at the hierarchical clustering.

5.2. Hierarchical clustering (agglomerative clustering). In the first step, we look at the hierarchy graph and check for the number of clusters.

5.2.1. Building hierarchy graph/dendrogram. The hierarchy class has a dendrogram method that takes the value returned by the *linkage method* of the same class. The linkage method takes the dataset and the method to minimize distances as parameters. We use *ward* as the method since it minimizes the variance of distances between the clusters:

```
from scipy.cluster.hierarchy import dendrogram, linkage, ward

#Calculate linkage
Z= linkage(X, method='ward')
Z[0]
```

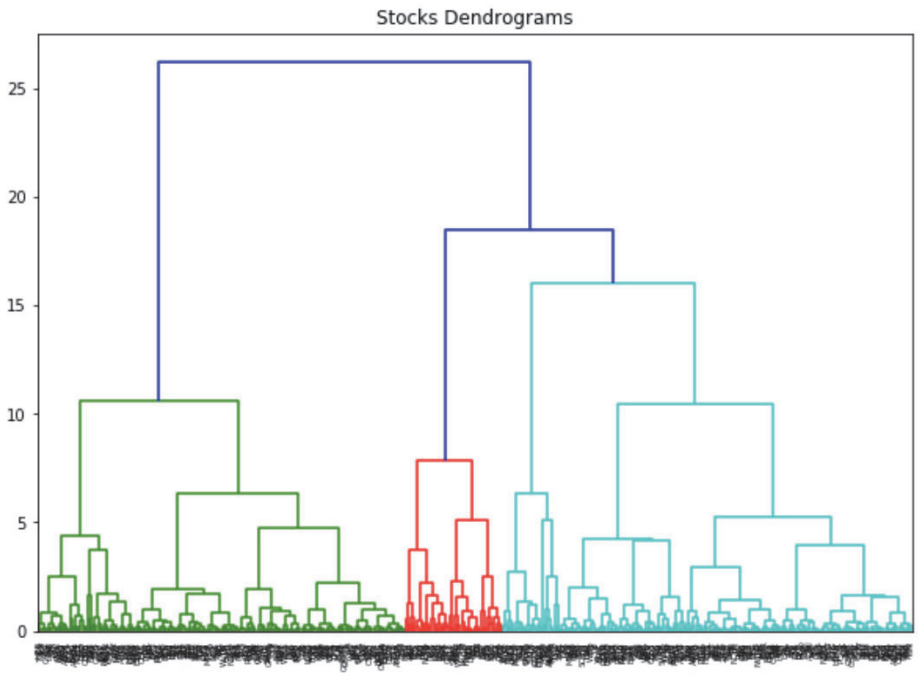
Output

```
array([3.30000000e+01, 3.14000000e+02, 3.62580431e-03, 2.00000000e+00])
```

The best way to visualize an agglomerative clustering algorithm is through a dendrogram, which displays a cluster tree, the leaves being the individual stocks and the root being the final single cluster. The distance between each cluster is shown on the y-axis. The longer the branches are, the less correlated the two clusters are:

```
#Plot Dendrogram
plt.figure(figsize=(10, 7))
plt.title("Stocks Dendrograms")
dendrogram(Z,labels = X.index)
plt.show()
```

Output



This chart can be used to visually inspect the number of clusters that would be created for a selected distance threshold (although the names of the stocks on the horizontal axis are not very clear, we can see that they are grouped into several clusters). The number of vertical lines a hypothetical straight, horizontal line will pass through is the number of clusters created for that distance threshold value. For example, at a value of 20, the horizontal line would pass through two vertical branches of the dendrogram, implying two clusters at that distance threshold. All data points (leaves) from that branch would be labeled as that cluster that the horizontal line passed through.

Choosing a threshold cut at 13 yields four clusters, as confirmed in the following Python code:

```
distance_threshold = 13
clusters = fcluster(Z, distance_threshold, criterion='distance')
chosen_clusters = pd.DataFrame(data=clusters, columns=['cluster'])
chosen_clusters['cluster'].unique()
```

`Output`

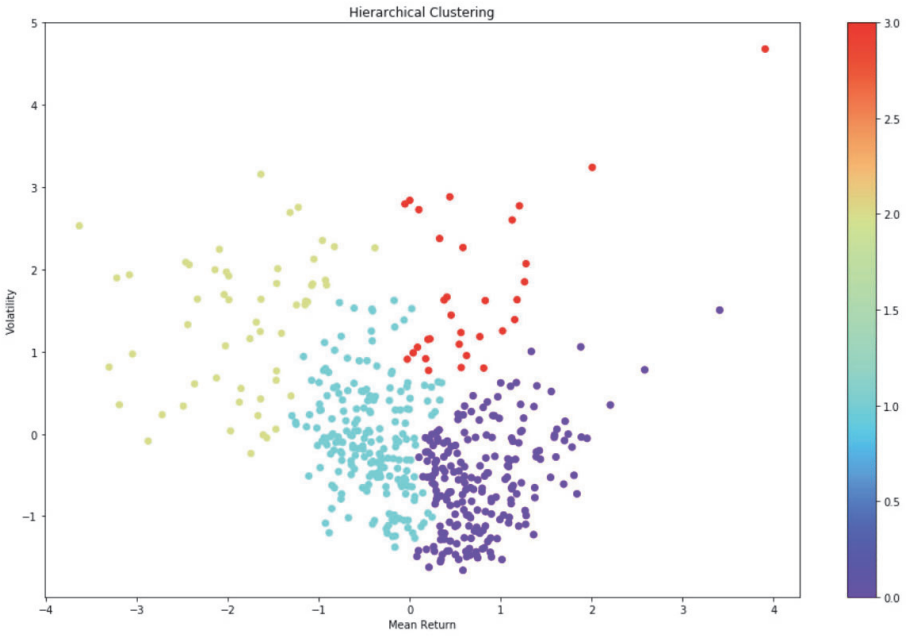
```
array([1, 4, 3, 2], dtype=int64)
```

5.2.2. Clustering and visualization. Let us build the hierarchical clustering model with four clusters and visualize the results:

```
nclust = 4
hc = AgglomerativeClustering(n_clusters=nclust, affinity='euclidean', \
linkage='ward')
clust_labels1 = hc.fit_predict(X)

fig = plt.figure(figsize=(16,10))
ax = fig.add_subplot(111)
scatter = ax.scatter(X.iloc[:,0],X.iloc[:,1], c=clust_labels1, cmap="rainbow")
ax.set_title('Hierarchical Clustering')
ax.set_xlabel('Mean Return')
ax.set_ylabel('Volatility')
plt.colorbar(scatter)
```

Similar to the plot of *k*-means clustering, we see that there are some distinct clusters separated by different colors (full-size version available on GitHub).

Output

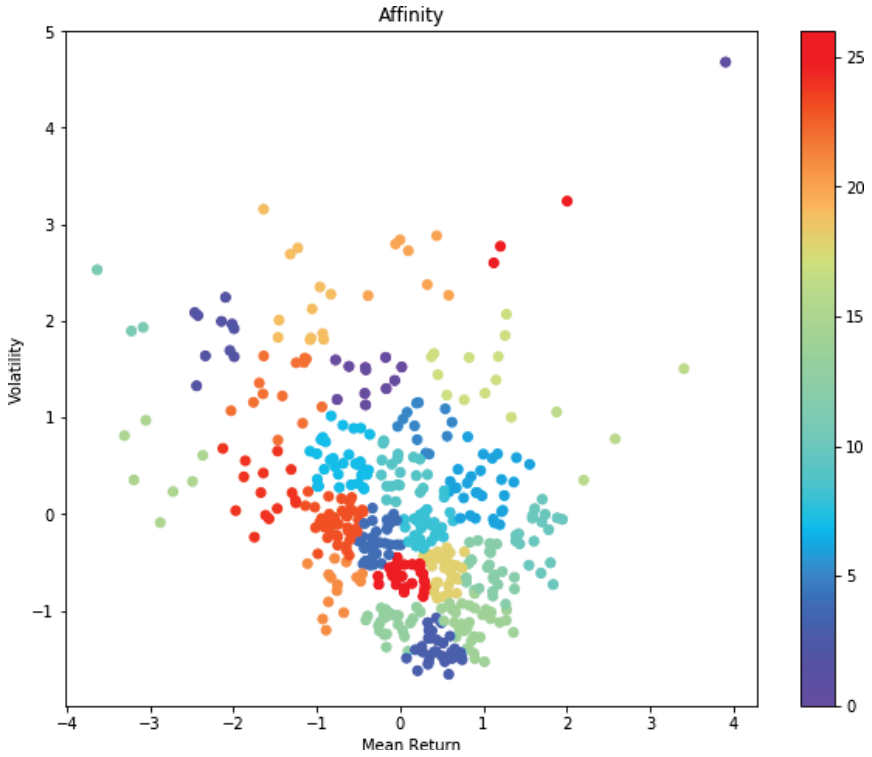


Now let us look at affinity propagation clustering.

5.3. Affinity propagation. Let us build the affinity propagation model and visualize the results:

```
ap = AffinityPropagation()
ap.fit(X)
clust_labels2 = ap.predict(X)

fig = plt.figure(figsize=(10,8))
ax = fig.add_subplot(111)
scatter = ax.scatter(X.iloc[:,0],X.iloc[:,1], c=clust_labels2, cmap="rainbow")
ax.set_title('Affinity')
ax.set_xlabel('Mean Return')
ax.set_ylabel('Volatility')
plt.colorbar(scatter)
```

Output



The affinity propagation model with the chosen hyperparameters produced many more clusters than *k*-means and hierarchical clustering. There is some clear grouping, but also more overlap due to the larger number of clusters (full-size version available on GitHub). In the next step, we will evaluate the clustering techniques.

5.4. Cluster evaluation. If the ground truth labels are not known, evaluation must be performed using the model itself. The silhouette coefficient (`sklearn.metrics.silhouette_score`) is one example we can use. A higher silhouette coefficient score implies a model with better defined clusters. The silhouette coefficient is computed for each of the clustering methods defined above:

```
from sklearn import metrics
print("km", metrics.silhouette_score(X, k_means.labels_, metric='euclidean'))
print("hc", metrics.silhouette_score(X, hc.fit_predict(X), metric='euclidean'))
print("ap", metrics.silhouette_score(X, ap.labels_, metric='euclidean'))
```

Output

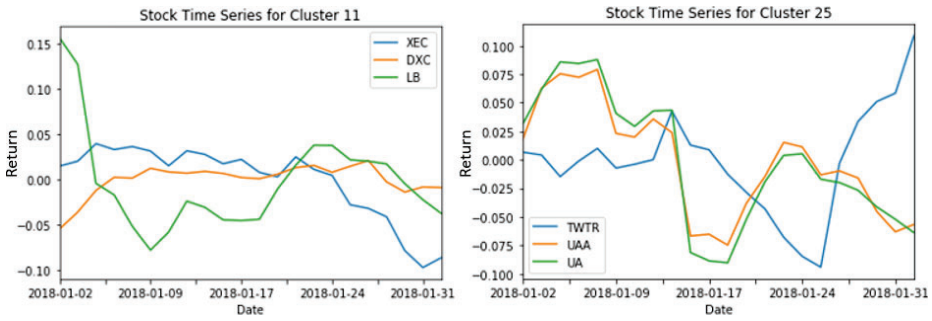
```
km 0.3350720873411941
hc 0.3432149515640865
ap 0.3450647315156527
```

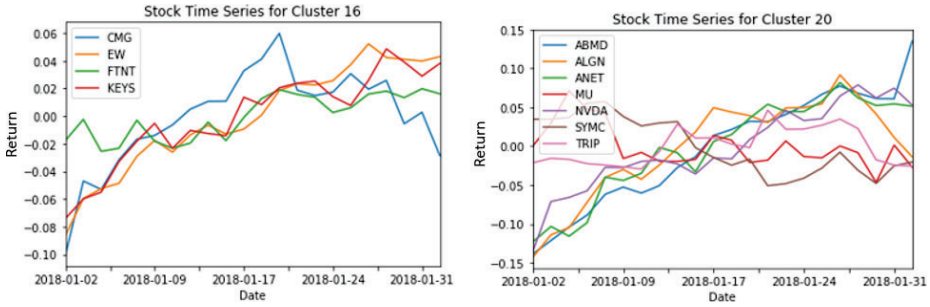
Given that affinity propagation performs the best, we proceed with affinity propagation and use 27 clusters as specified by this clustering method.

Visualizing the return within a cluster. We have the clustering technique and the number of clusters finalized, but we need to check whether the clustering leads to a sensible output. To do this, we visualize the historical behavior of the stocks in a few clusters:

```
# all stock with its cluster label (including -1)
clustered_series = pd.Series(index=X.index, data=ap.fit_predict(X).flatten())
# clustered stock with its cluster label
clustered_series_all = pd.Series(index=X.index, data=ap.fit_predict(X).flatten())
clustered_series = clustered_series[clustered_series != -1]
# get the number of stocks in each cluster
counts = clustered_series_ap.value_counts()
# let's visualize some clusters
cluster_vis_list = list(counts[(counts<25) & (counts>1)].index)[::-1]
cluster_vis_list
# plot a handful of the smallest clusters
plt.figure(figsize=(12, 7))
cluster_vis_list[0:min(len(cluster_vis_list), 4)]

for clust in cluster_vis_list[0:min(len(cluster_vis_list), 4)]:
    tickers = list(clustered_series[clustered_series==clust].index)
    # calculate the return (lognormal) of the stocks
    means = np.log(dataset.loc[:"2018-02-01", tickers].mean())
    data = np.log(dataset.loc[:"2018-02-01", tickers]).sub(means)
    data.plot(title='Stock Time Series for Cluster %d' % clust)
plt.show()
```

Output





Looking at the charts above, across all the clusters with small number of stocks, we see similar movement of the stocks under different clusters, which corroborates the effectiveness of the clustering technique.

6. Pairs selection

Once the clusters are created, several cointegration-based statistical techniques can be applied on the stocks within a cluster to create the pairs. Two or more time series are considered to be cointegrated if they are nonstationary and tend to move together.[2] The presence of cointegration between time series can be validated through several statistical techniques, including the Augmented Dickey-Fuller test and the Johansen test.

In this step, we scan through a list of securities within a cluster and test for cointegration between the pairs. First, we write a function that returns a cointegration test score matrix, a p-value matrix, and any pairs for which the p-value was less than 0.05.

Cointegration and pair selection function.

```
def find_cointegrated_pairs(data, significance=0.05):
    # This function is from https://www.quantopian.com
    n = data.shape[1]
    score_matrix = np.zeros((n, n))
    pvalue_matrix = np.ones((n, n))
    keys = data.keys()
    pairs = []
    for i in range(1):
        for j in range(i+1, n):
            S1 = data[keys[i]]
            S2 = data[keys[j]]
            result = coint(S1, S2)
            score = result[0]
            pvalue = result[1]
            score_matrix[i, j] = score
```

2 Refer to Chapter 5 for more details.

```
                pvalue_matrix[i, j] = pvalue
                if pvalue < significance:
                    pairs.append((keys[i], keys[j]))
    return score_matrix, pvalue_matrix, pairs
```

Next, we check the cointegration of different pairs within several clusters using the function created above and return the pairs found:

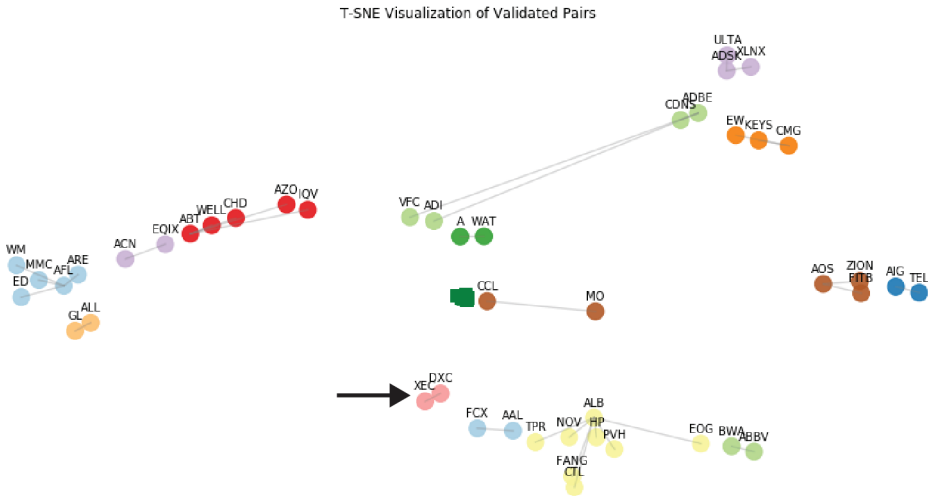
```
from statsmodels.tsa.stattools import coint
cluster_dict = {}
for i, which_clust in enumerate(ticker_count_reduced.index):
    tickers = clustered_series[clustered_series == which_clust].index
    score_matrix, pvalue_matrix, pairs = find_cointegrated_pairs(
        dataset[tickers]
    )
    cluster_dict[which_clust] = {}
    cluster_dict[which_clust]['score_matrix'] = score_matrix
    cluster_dict[which_clust]['pvalue_matrix'] = pvalue_matrix
    cluster_dict[which_clust]['pairs'] = pairs

pairs = []
for clust in cluster_dict.keys():
    pairs.extend(cluster_dict[clust]['pairs'])

print ("Number of pairs found : %d" % len(pairs))
print ("In those pairs, there are %d unique tickers." % len(np.unique(pairs)))
```

`Output`

```
Number of pairs found : 32
In those pairs, there are 47 unique tickers.
```

Let us visualize the results of the pair selection process now. Refer to the Jupyter notebook of this case study for the details of the steps related to the pair visualization using the t-SNE technique.

The following chart shows the strength of *k*-means for finding nontraditional pairs (pointed out with an arrow in the visualization). DXC is the ticker symbol for DXC Technology, and XEC is the ticker symbol for Cimarex Energy. These two stocks are from different sectors and appear to have nothing in common on the surface, but they are identified as pairs using *k*-means clustering and cointegration testing. This implies that a long-run stable relationship exists between their stock price movements.



Once the pairs are created, they can be used in a pairs trading strategy. When the share prices of the pair deviate from the identified long-run relationship, an investor would seek to take a long position in the underperforming security and sell short the outperforming security. If the securities return to their historical relationship, a profit is made from the convergence of the prices.

Conclusion

In this case study, we demonstrated the efficiency of clustering techniques by finding small pools of stocks in which to identify pairs to be used in a pairs trading strategy. A next step beyond this case study would be to explore and backtest various long/short trading strategies with pairs of stocks from the groupings of stocks.

Clustering can be used for dividing stocks and other types of assets into groups with similar characteristics for several other kinds of trading strategies. It can also be effective in portfolio construction, helping to ensure we choose a pool of assets with sufficient diversification between them.

Case Study 2: Portfolio Management: Clustering Investors

Asset management and investment allocation is a tedious and time-consuming process in which investment managers often must design customized approaches for each client or investor.

What if we were able to organize these clients into particular investor profiles, or clusters, wherein each group is indicative of investors with similar characteristics?

Clustering investors based on similar characteristics can lead to simplicity and standardization in the investment management process. These algorithms can group investors based on different factors, such as age, income, and risk tolerance. It can help investment managers identify distinct groups within their investors base. Additionally, by using these techniques, managers can avoid introducing any biases that otherwise could adversely impact decision making. The factors analyzed through clustering can have a big impact on asset allocation and rebalancing, making it an invaluable tool for faster and effective investment management.

In this case study, we will use clustering methods to identify different types of investors.

The data used for this case study is from the Survey of Consumer Finances, which is conducted by the Federal Reserve Board. The same dataset was used in "Case Study 3: Investor Risk Tolerance and Robo-Advisors" on page 125 in Chapter 5.

In this case study, we focus on:

- Understanding the intuitive meaning of the groupings coming out of clustering.
- Choosing the right clustering techniques.
- Visualization of the clustering outcome and selecting the correct number of clusters in *k*-means.

Blueprint for Using Clustering for Grouping Investors

1. Problem definition

The goal of this case study is to build a clustering model to group individuals or investors based on parameters related to the ability and willingness to take risk. We will focus on using common demographic and financial characteristics to accomplish this.

The survey data we're using includes responses from 10,000+ individuals in 2007 (precrisis) and 2009 (postcrisis). There are over 500 features. Since the data has many variables, we will first reduce the number of variables and select the most intuitive features directly linked to an investor's ability and willingness to take risk.

2. Getting started—loading the data and Python packages

2.1. Loading the Python packages. The packages loaded for this case study are similar to those loaded in the case study presented in Chapter 5. However, some additional packages related to the clustering techniques are shown in the following code snippet:

```
#Import packages for clustering techniques
from sklearn.cluster import KMeans, AgglomerativeClustering,AffinityPropagation
from sklearn.metrics import adjusted_mutual_info_score
from sklearn import cluster, covariance, manifold
```

2.2. Loading the data. The data (again, previously used in Chapter 5) is further processed to give the following attributes that represent an individual's ability and willingness to take risk. This preprocessed data is for the 2007 survey and is loaded below:

```
# load dataset
dataset = pd.read_excel('ProcessedData.xlsx')
```

3. Exploratory data analysis

Next, we take a closer look at the different columns and features found in the data.

3.1. Descriptive statistics. First, looking at the shape of the data:

```
dataset.shape
```

```
Output
```

```
(3866, 13)
```

The data has information for 3,886 individuals across 13 columns:

```
# peek at data
set_option('display.width', 100)
dataset.head(5)
```

	ID	AGE	EDUC	MARRIED	KIDS	LIFECL	OCCAT	RISK	HHOUSES	WSAVED	SPENDMOR	NWCAT	INCCL
0	1	3	2	1	0	2	1	3	1	1	5	3	4
1	2	4	4	1	2	5	2	3	0	2	5	5	5
2	3	3	1	1	2	3	2	2	1	2	4	4	4
3	4	3	1	1	2	3	2	2	1	2	4	3	4
4	5	4	3	1	1	5	1	2	1	3	3	5	5

As we can see in the table above, there are 12 attributes for each of the individuals. These attributes can be categorized as demographic, financial, and behavioral attributes. They are summarized in Figure 8-2.

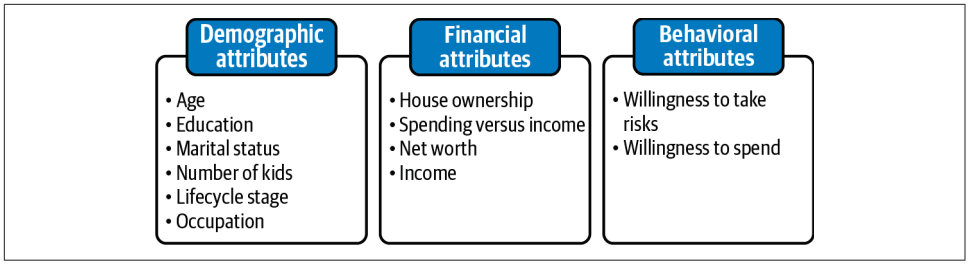


Figure 8-2. Attributes for clustering individuals

Many of these were previously used and defined in the Chapter 5 case study. A few additional attributes (LIFECYCL, HHOUSES, and SPENDMOR) are used in this case study and are defined below:

LIFECYCL
: This is a lifecycle variable, used to approximate a person's ability to take on risk. There are six categories in increasing level of ability to take risk. A value of 1 represents "age under 55, not married, and no kids," and a value of 6 represents "age over 55 and not working."

HHOUSES
: This is a flag indicating whether the individual is a homeowner. A value of 1 (0) implies the individual does (does not) own a home.

SPENDMOR
: This represents higher spending preference if assets appreciated on a scale of 1 to 5.

3.2. Data visualization. We will take a detailed look into the visualization postclustering.

4. Data preparation

Here, we perform any necessary changes to the data in preparation for modeling.

4.1. Data cleaning. In this step, we check for NAs in the rows and either drop them or fill them with the mean of the column:

```
print('Null Values =', dataset.isnull().values.any())
```

```
Output

    Null Values = False
```

Given that there is not any missing data, and the data is already in categorical format, no further data cleaning was performed. The *ID* column is unnecessary and is dropped:

```
X=X.drop(['ID'], axis=1)
```

4.2. Data transformation. As we saw in Section 3.1, all the columns represent categorical data with similar numeric scale, with no outliers. Hence, no data transformation will be required for clustering.

5. Evaluate algorithms and models

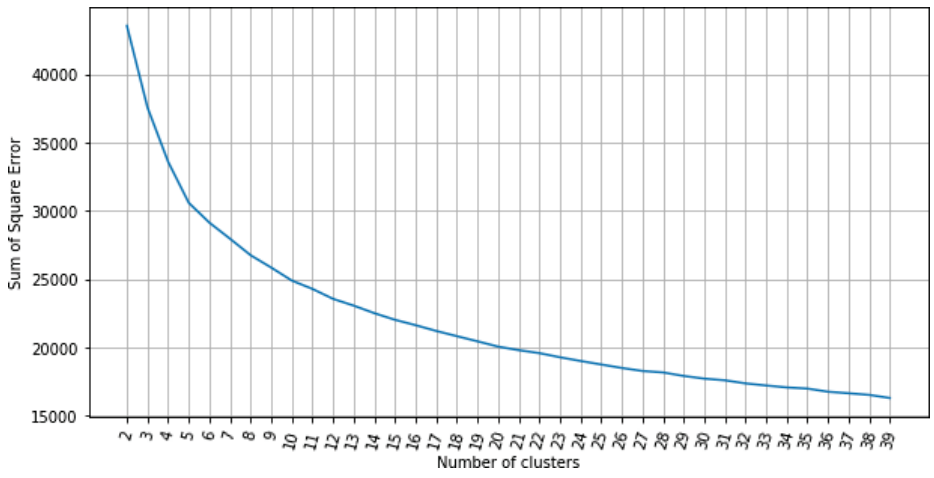
We will analyze the performance of *k*-means and affinity propagation.

5.1. k-means clustering. We look at the details of the *k*-means clustering in this step. First, we find the optimal number of clusters, followed by the creation of a model.

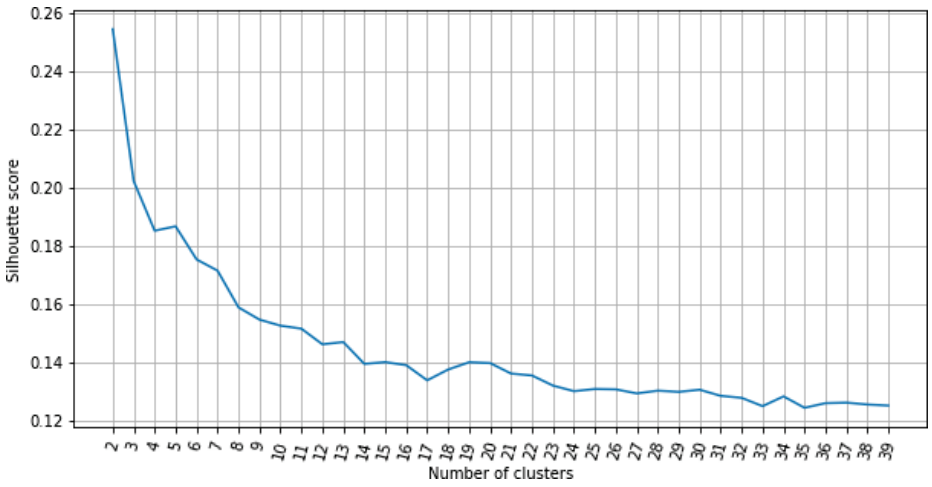
5.1.1. Finding the optimal number of clusters. We look at the following two metrics to evaluate the number of clusters in the *k*-means model. The Python code to get these two metrics is the same as in case study 1:

1. Sum of squared errors (SSE)
2. Silhouette score

```
Sum of squared errors (SSE) within clusters
```



```
Silhouette score
```



Looking at both of the preceding charts, the optimum number of clusters seems to be around 7. We can see that as the number of clusters increases past 6, the SSE within clusters begins to plateau. From the second graph, we can see that there are various parts of the graph where a kink can be seen. Since there is not much of a difference in the SSE after 7 clusters, we proceed with using 7 clusters in the *k*-means model below.

5.1.2. Clustering and visualization. Let us create a *k*-means model with 7 clusters:

```
nclust=7

#Fit with k-means
k_means = cluster.KMeans(n_clusters=nclust)
k_means.fit(X)
```

Let us assign a target cluster to each individual in the dataset. This assignment is used further for exploratory data analysis to understand the behavior of each cluster:

```
#Extracting labels
target_labels = k_means.predict(X)
```

5.2. Affinity propagation. Here, we build an affinity propagation model and look at the number of clusters:

```
ap = AffinityPropagation()
ap.fit(X)
clust_labels2 = ap.predict(X)

cluster_centers_indices = ap.cluster_centers_indices_
labels = ap.labels_
```

```
n_clusters_ = len(cluster_centers_indices)
print('Estimated number of clusters: %d' % n_clusters_)
```

Output

```
Estimated number of clusters: 161
```

The affinity propagation resulted in over 150 clusters. Such a large number will likely make it difficult to ascertain proper differentiation between them.

5.3. Cluster evaluation. In this step, we check the performance of the clusters using silhouette coefficient (*sklearn.metrics.silhouette_score*). Recall that a higher silhouette coefficient score relates to a model with better defined clusters:

```
from sklearn import metrics
print("km", metrics.silhouette_score(X, k_means.labels_))
print("ap", metrics.silhouette_score(X, ap.labels_))
```

Output

```
km 0.170585217843582
ap 0.09736878398868973
```

The *k*-means model has a much higher silhouette coefficient compared to the affinity propagation. Additionally, the large number of clusters resulting from the affinity propagation is untenable. In the context of the problem at hand, having fewer clusters, or categorizations of investors, helps build simplicity and standardization in the investment management process. It gives the users of this information (e.g., financial advisors) some manageable intuition around the representation of the clusters. Comprehending and being able to speak to six to eight investor types is much more practical than maintaining a meaningful understanding of over 100 different profiles. With this in mind, we proceed with *k*-means as the preferred clustering technique.

6. Cluster intuition

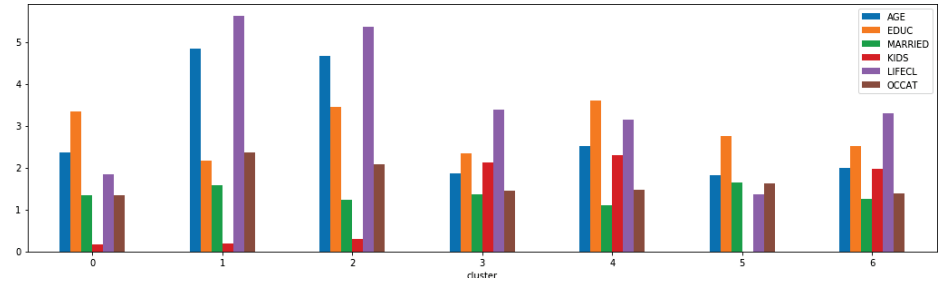
In the next step, we will analyze the clusters and attempt to draw conclusions from them. We do that by plotting the average of each variable of the cluster and summarizing the findings:

```
cluster_output= pd.concat([pd.DataFrame(X),  pd.DataFrame(k_means.labels_, \
  columns = ['cluster'])],axis=1)
output=cluster_output.groupby('cluster').mean()
```

Demographics Features: Plot for each of the clusters

```
output[['AGE','EDUC','MARRIED','KIDS','LIFECL','OCCAT']].\
plot.bar(rot=0, figsize=(18,5));
```

Output

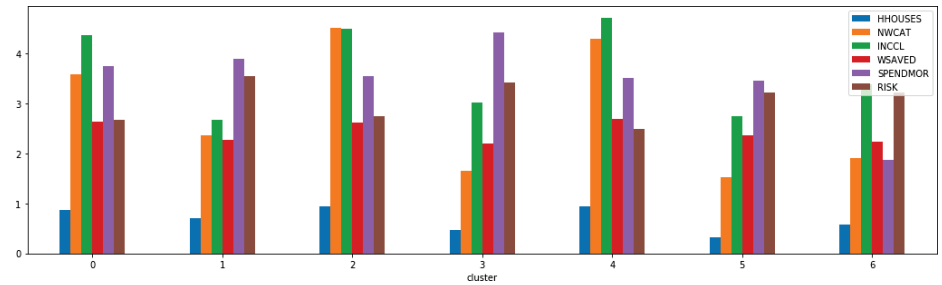


The plot here shows the average values of the attributes for each of the clusters (full size version available on GitHub). For example, in comparing clusters 0 and 1, cluster 0 has *lower* average age, yet *higher* average education. However, these two clusters are more similar in marital status and number of children. So, based on the demographic attributes, the individuals in cluster 0 will, on average, have higher risk tolerance compared to those in cluster 1.

Financial and Behavioral Attributes: Plot for each of the clusters

```
output[['HHOUSES','NWCAT','INCCL','WSAVED','SPENDMOR','RISK']].\
plot.bar(rot=0, figsize=(18,5));
```

Output



The plot here shows the average values of the financial and behavior attributes for each of the clusters (full size version available on GitHub). Again, comparing clusters 0 and 1, the former has higher average house ownership, higher average net worth and income, and a lower willingness to take risk compared to the latter. In terms of saving versus income comparison and willingness to save, the two clusters are comparable. Therefore, we can posit that the individuals in cluster 0 will, on average, have a higher ability and yet a lower willingness to take risks compared to the individuals in cluster 1.

Combining the information from the demographics, financial, and behavioral attributes for these two clusters, the overall ability to take risks for an individual in cluster 0 is higher than someone in cluster 1. Performing similar analyses across all other clusters, we summarize the results in the table below. The risk tolerance column represents the subjective assessment of the risk tolerance of each cluster.

Cluster	Features	Risk capacity
Cluster 0	Low age, high net worth and income, less risky life category, willingness to spend more	High
Cluster 1	High age, low net worth and income, highly risky life category, willingness to take risk, low education	Low
Cluster 2	High age, high net worth and income, highly risky life category, willingness to take risk, owns home	Medium
Cluster 3	Low age, very low income and net worth, high willingness to take risk, many kids	Low
Cluster 4	Medium age, very high income and net worth, high willingness to take risk, many kids, owns home	High
Cluster 5	Low age, very low income and net worth, high willingness to take risk, no kids	Medium
Cluster 6	Low age, medium income and net worth, high willingness to take risk, many kids, owns home	Low

Conclusion

One of the key takeaways from this case study is the approach to understanding the cluster intuition. We used visualization techniques to understand the expected behavior of a cluster member by qualitatively interpreting mean values of the variables in each cluster. We demonstrated the efficiency of clustering in discovering the natural groups of different investors based on their risk tolerance.

Given that clustering algorithms can successfully group investors based on different factors (such as age, income, and risk tolerance), they can be further used by portfolio managers to standardize portfolio allocation and rebalance strategies across the clusters, making the investment management process faster and more effective.

Case Study 3: Hierarchical Risk Parity

Markowitz's *mean-variance portfolio optimization* is the most commonly used technique for portfolio construction and asset allocation. In this technique, we need to estimate the covariance matrix and expected returns of assets to be used as inputs. As discussed in "Case Study 1: Portfolio Management: Finding an Eigen Portfolio" on page 202 in Chapter 7, the erratic nature of financial returns causes estimation errors in the expected returns and the covariance matrix, especially when the number of assets is large compared to the sample size. These errors greatly jeopardize the optimality of the resulting portfolios, which leads to erroneous and unstable results. Additionally, small changes in the assumed asset returns, volatilities, or covariances

can lead to large effects on the output of the optimization procedure. In this sense, the Markowitz mean-variance optimization is an ill-posed (or ill-conditioned) inverse problem.

In "Building Diversified Portfolios That Outperform Out-of-Sample" by Marcos López de Prado (2016), the author proposes a portfolio allocation method based on clustering called *hierarchical risk parity*. The main idea of hierarchical risk parity is to run hierarchical clustering on the covariance matrix of stock returns and then find a diversified weighting by distributing capital equally to each cluster hierarchy (so that many correlated strategies will receive the same total allocation as a single uncorrelated one). This alleviates some of the issues (highlighted above) found in Markowitz's mean-variance optimization and improves numerical stability.

In this case study, we will implement hierarchical risk parity based on clustering methods and compare it against Markowitz's mean-variance optimization method.

The dataset used for this case study is price data for stocks in the S&P 500 from 2018 onwards. The dataset can be downloaded from Yahoo Finance. It is the same dataset as was used in case study 1.

In this case study, we will focus on:

- Application of clustering-based techniques for portfolio allocation.
- Developing a framework for comparing portfolio allocation methods.

Blueprint for Using Clustering to Implement Hierarchical Risk Parity

1. Problem definition

Our goal in this case study is to use a clustering-based algorithm on a dataset of stocks to allocate capital into different asset classes. In order to backtest and compare the portfolio allocation against the traditional Markowitz mean-variance optimization, we will perform visualization and use performance metrics, such as the Sharpe ratio.

2. Getting started—loading the data and Python packages

2.1. Loading the Python packages. The packages loaded for this case study are similar to those loaded in the previous case study. However, some additional packages related to the clustering techniques are shown in the following code snippet:

```
#Import Model Packages
import scipy.cluster.hierarchy as sch
from sklearn.cluster import AgglomerativeClustering
from scipy.cluster.hierarchy import fcluster
from scipy.cluster.hierarchy import dendrogram, linkage, cophenet
from sklearn.metrics import adjusted_mutual_info_score
from sklearn import cluster, covariance, manifold
import ffn

#Package for optimization of mean variance optimization
import cvxopt as opt
from cvxopt import blas, solvers
```

Since this case study uses the same data as case study 1, some of the next steps (i.e., loading the data) have been skipped to avoid repetition. As a reminder, the data contains around 500 stocks and 448 observations.

3. Exploratory data analysis

We will take a detailed look into the visualization postclustering later in this case study.

4. Data preparation

4.1. Data cleaning. Refer to case study 1 for data cleaning steps.

4.2. Data transformation. We will be using annual returns for clustering. Additionally, we will train the data and then test the data. Here, we prepare the dataset for training and testing by separating 20% of the dataset for testing, and we generate the return series:

```
X= dataset.copy('deep')
row= len(X)
train_len = int(row*.8)

X_train = X.head(train_len)
X_test = X.tail(row-train_len)

#Calculate percentage return
returns = X_train.to_returns().dropna()
returns_test=X_test.to_returns().dropna()
```

5. Evaluate algorithms and models

In this step, we will look at hierarchical clustering and perform further analysis and visualization.

5.1. Building a hierarchy graph/dendrogram. The first step is to look for clusters of correlations using the agglomerative hierarchical clustering technique. The hierarchy class has a dendrogram method that takes the value returned by the linkage method of the same class. The linkage method takes the dataset and the method to minimize distances as parameters. There are different options for measurement of the distance. The option we will choose is ward, since it minimizes the variance of distances between the clusters. Other possible measures of distance include single and centroid.

Linkage does the actual clustering in one line of code and returns a list of the clusters joined in the format:

```
Z= [stock_1, stock_2, distance, sample_count]
```

As a precursor, we define a function to convert correlation into distances:

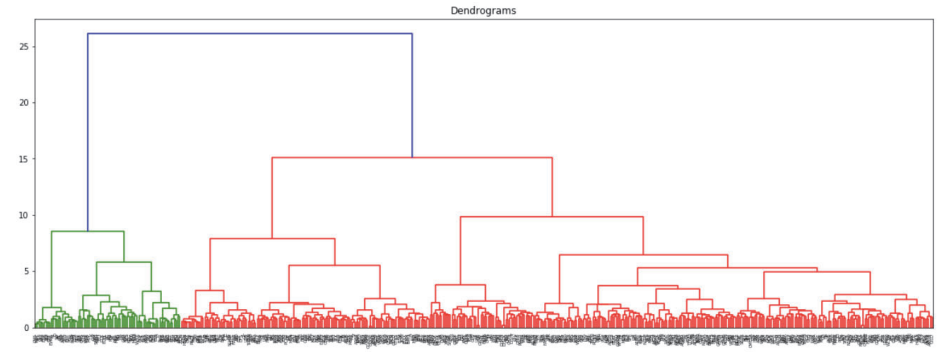
```
def correlDist(corr):
    # A distance matrix based on correlation, where 0<=d[i,j]<=1
    # This is a proper distance metric
    dist = ((1 - corr) / 2.) ** .5  # distance matrix
    return dist
```

Now we convert the correlation of the returns of the stocks into distances, followed by the computation of linkages in the step below. Computation of linkages is followed by the visualization of the clusters through a dendrogram. Again, the leaves are the individual stocks, and the root is the final single cluster. The distance between each cluster is shown on the y-axis; the longer the branches are, the less correlated two clusters are.

```
#Calculate linkage
dist = correlDist(returns.corr())
link = linkage(dist, 'ward')

#Plot Dendrogram
plt.figure(figsize=(20, 7))
plt.title("Dendrograms")
dendrogram(link,labels = X.columns)
plt.show()
```

In the following chart, the horizontal axis represents the clusters. Although the names of the stocks on the horizontal axis are not very clear (not surprising, given that there are 500 stocks), we can see that they are grouped into several clusters. The appropriate number of clusters appears to be 2, 3, or 6, depending on the desired distance threshold level. Next, we will leverage the linkages computed from this step to compute the asset allocation based on hierarchical risk parity.

Output



5.2. Steps for hierarchical risk parity. The hierarchical risk parity (HRP) algorithm works in three stages, as outlined in Prado's paper:

Tree clustering
: Grouping similar investments into clusters based on their correlation matrix. Having a hierarchical structure helps us improve stability issues of quadratic optimizers when inverting the covariance matrix.

Quasi-diagonalization
: Reorganizing the covariance matrix so similar investments will be placed together. This matrix diagonalization allows us to distribute weights optimally following an inverse-variance allocation.

Recursive bisection
: Distributing the allocation through recursive bisection based on cluster covariance.

Having performed the first stage in the previous section, where we identified clusters based on the distance metrics, we proceed to quasi-diagonalization.

5.2.1. Quasi-diagonalization. Quasi-diagonalization is a process known as *matrix seriation*, which reorganizes the rows and columns of a covariance matrix so that the largest values lie along the diagonal. As shown in the following code, the process reorganizes the covariance matrix so similar investments are placed together. This matrix diagonalization allows us to distribute weights optimally following an inverse-variance allocation:

```
def getQuasiDiag(link):
    # Sort clustered items by distance
    link = link.astype(int)
    sortIx = pd.Series([link[-1, 0], link[-1, 1]])
    numItems = link[-1, 3]  # number of original items
```

```
    while sortIx.max() >= numItems:
        sortIx.index = range(0, sortIx.shape[0] * 2, 2)  # make space
        df0 = sortIx[sortIx >= numItems]  # find clusters
        i = df0.index
        j = df0.values - numItems
        sortIx[i] = link[j, 0]  # item 1
        df0 = pd.Series(link[j, 1], index=i + 1)
        sortIx = sortIx.append(df0)  # item 2
        sortIx = sortIx.sort_index()  # re-sort
        sortIx.index = range(sortIx.shape[0])  # re-index
    return sortIx.tolist()
```

5.2.2. Recursive bisection. In the next step, we perform recursive bisection, which is a top-down approach to splitting portfolio weights between subsets based on the inverse proportion to their aggregated variances. The function `getClusterVar` computes the cluster variance, and in this process, it requires the inverse-variance portfolio from the function `getIVP`. The output of the function `getClusterVar` is used by the function `getRecBipart` to compute the final allocation through recursive bisection based on cluster covariance:

```
def getIVP(cov, **kargs):
# Compute the inverse-variance portfolio
ivp = 1. / np.diag(cov)
ivp /= ivp.sum()
return ivp

def getClusterVar(cov,cItems):
    # Compute variance per cluster
    cov_=cov.loc[cItems,cItems] # matrix slice
    w_=getIVP(cov_).reshape(-1, 1)
    cVar=np.dot(np.dot(w_.T,cov_),w_)[0, 0]
    return cVar

def getRecBipart(cov, sortIx):
    # Compute HRP alloc
    w = pd.Series(1, index=sortIx)
    cItems = [sortIx]  # initialize all items in one cluster
    while len(cItems) > 0:
        cItems = [i[j:k] for i in cItems for j, k in ((0,\
           len(i) // 2), (len(i) // 2, len(i))) if len(i) > 1]  # bi-section
        for i in range(0, len(cItems), 2):  # parse in pairs
            cItems0 = cItems[i]  # cluster 1
            cItems1 = cItems[i + 1]  # cluster 2
            cVar0 = getClusterVar(cov, cItems0)
            cVar1 = getClusterVar(cov, cItems1)
            alpha = 1 - cVar0 / (cVar0 + cVar1)
            w[cItems0] *= alpha  # weight 1
            w[cItems1] *= 1 - alpha  # weight 2
    return w
```

The following function `getHRP` combines the three stages—clustering, quasi-diagonalization, and recursive bisection—to produce the final weights:

```
def getHRP(cov, corr):
    # Construct a hierarchical portfolio
    dist = correlDist(corr)
    link = sch.linkage(dist, 'single')
    #plt.figure(figsize=(20, 10))
    #dn = sch.dendrogram(link, labels=cov.index.values)
    #plt.show()
    sortIx = getQuasiDiag(link)
    sortIx = corr.index[sortIx].tolist()
    hrp = getRecBipart(cov, sortIx)
    return hrp.sort_index()
```

5.3. Comparison against other asset allocation methods. A main focus of this case study is to develop an alternative to Markowitz's mean-variance portfolio optimization using clustering. In this step, we define a function to compute the allocation of a portfolio based on Markowitz's mean-variance technique. This function (`getMVP`) takes the covariance matrix of the assets as an input, performs the mean-variance optimization, and produces the portfolio allocations:

```
def getMVP(cov):
    cov = cov.T.values
    n = len(cov)
    N = 100
    mus = [10 ** (5.0 * t / N - 1.0) for t in range(N)]

    # Convert to cvxopt matrices
    S = opt.matrix(cov)
    #pbar = opt.matrix(np.mean(returns, axis=1))
    pbar = opt.matrix(np.ones(cov.shape[0]))

    # Create constraint matrices
    G = -opt.matrix(np.eye(n))  # negative n x n identity matrix
    h = opt.matrix(0.0, (n, 1))
    A = opt.matrix(1.0, (1, n))
    b = opt.matrix(1.0)

    # Calculate efficient frontier weights using quadratic programming
    solvers.options['show_progress'] = False
    portfolios = [solvers.qp(mu * S, -pbar, G, h, A, b)['x']
                  for mu in mus]
    ## Calculate risk and return of the frontier
    returns = [blas.dot(pbar, x) for x in portfolios]
    risks = [np.sqrt(blas.dot(x, S * x)) for x in portfolios]
    ## Calculate the 2nd degree polynomial of the frontier curve.
    m1 = np.polyfit(returns, risks, 2)
    x1 = np.sqrt(m1[2] / m1[0])
    # CALCULATE THE OPTIMAL PORTFOLIO
    wt = solvers.qp(opt.matrix(x1 * S), -pbar, G, h, A, b)['x']
```

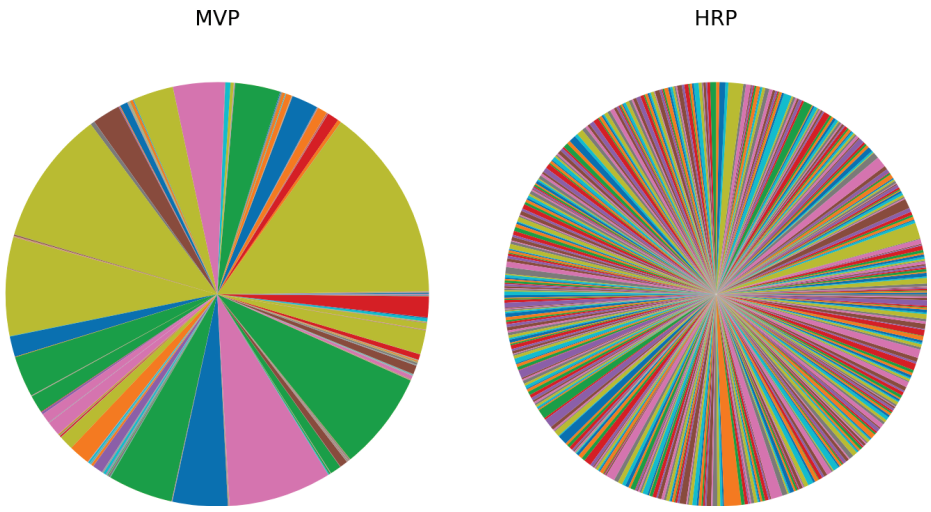
```
    return list(wt)
```

5.4. Getting the portfolio weights for all types of asset allocation. In this step, we use the functions above to compute the asset allocation using the two asset allocation methods. We then visualize the asset allocation results:

```
def get_all_portfolios(returns):

    cov, corr = returns.cov(), returns.corr()
    hrp = getHRP(cov, corr)
    mvp = getMVP(cov)
    mvp = pd.Series(mvp, index=cov.index)
    portfolios = pd.DataFrame([mvp, hrp], index=['MVP', 'HRP']).T
    return portfolios

#Now getting the portfolios and plotting the pie chart
portfolios = get_all_portfolios(returns)

portfolios.plot.pie(subplots=True, figsize=(20, 10),legend = False);
fig, (ax1, ax2) = plt.subplots(1, 2,figsize=(30,20))
ax1.pie(portfolios.iloc[:, 0], );
ax1.set_title('MVP',fontsize=30)
ax2.pie(portfolios.iloc[:, 1]);
ax2.set_title('HRP',fontsize=30)
```

The following pie charts show the asset allocation of MVP versus HRP. We clearly see more diversification in HRP. Now let us look at the backtesting results.

`Output`

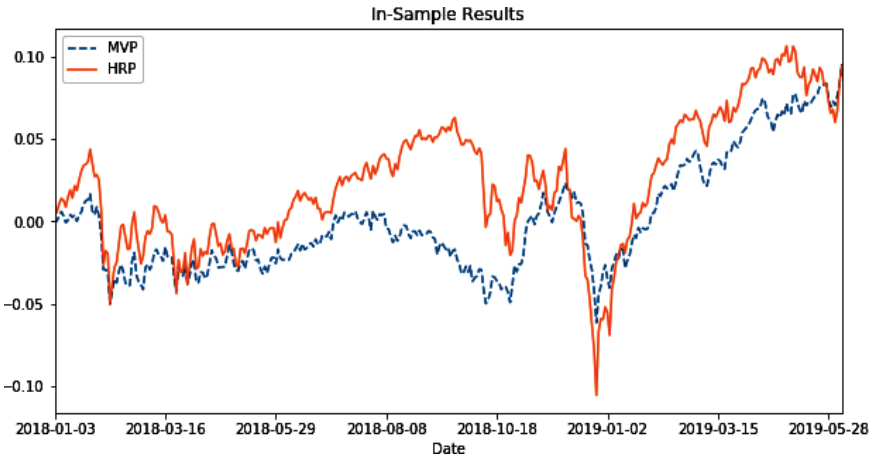


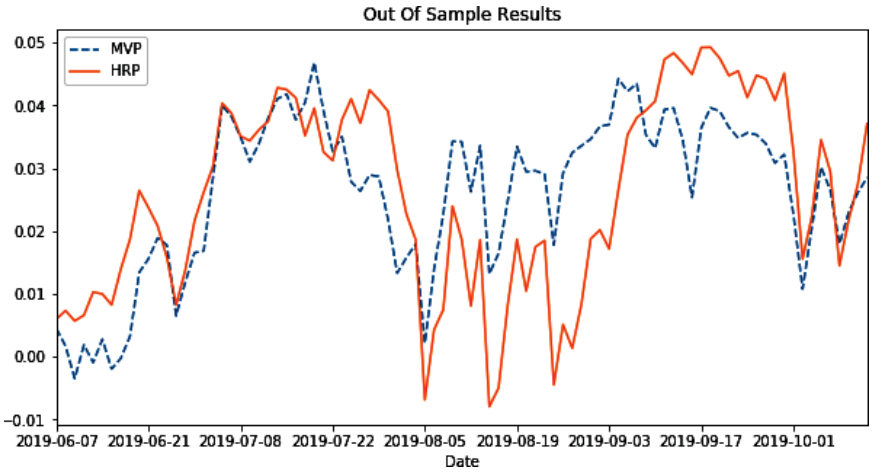
6. Backtesting

We will now backtest the performance of portfolios produced by the algorithms, looking at both in-sample and out-of-sample results:

```
Insample_Result=pd.DataFrame(np.dot(returns,np.array(portfolios)), \
'MVP','HRP'], index = returns.index)
OutOfSample_Result=pd.DataFrame(np.dot(returns_test,np.array(portfolios)), \
columns=['MVP', 'HRP'], index = returns_test.index)

Insample_Result.cumsum().plot(figsize=(10, 5), title ="In-Sample Results",\
                              style=['--','-'])
OutOfSample_Result.cumsum().plot(figsize=(10, 5), title ="Out Of Sample Results",\
                                 style=['--','-'])
```

`Output`





Looking at the charts, MVP underperforms for a significant amount of time in the in-sample test. In the out-of-sample test, MVP performed better than HRP for a brief period of time from August 2019 to mid-September 2019. In the next step, we examine the Sharpe ratio for the two allocation methods:

In-sample results.

```
#In_sample Results
stddev = Insample_Result.std() * np.sqrt(252)
sharp_ratio = (Insample_Result.mean()*np.sqrt(252))/(Insample_Result).std()
Results = pd.DataFrame(dict(stdev=stddev, sharp_ratio = sharp_ratio))
Results
```

`Output`

	stdev	sharp_ratio
MVP	0.086	0.785
HRP	0.127	0.524

Out-of-sample results.

```
#OutOf_sample Results
stddev_oos = OutOfSample_Result.std() * np.sqrt(252)
sharp_ratio_oos = (OutOfSample_Result.mean()*np.sqrt(252))/(OutOfSample_Result).\
std()
Results_oos = pd.DataFrame(dict(stdev_oos=stddev_oos, sharp_ratio_oos = \
  sharp_ratio_oos))
Results_oos
```

`Output`

	stdev_oos	sharp_ratio_oos
MVP	0.103	0.787
HRP	0.126	0.836

Although the in-sample results of MVP look promising, the out-of-sample Sharpe ratio and overall return of the portfolio constructed using the hierarchical clustering approach are better. The diversification that HRP achieves across uncorrelated assets makes the methodology more robust against shocks.

Conclusion

In this case study, we saw that portfolio allocation based on hierarchical clustering offers better separation of assets into clusters with similar characteristics without relying on classical correlation analysis used in Markowitz's mean-variance portfolio optimization.

Using Markowitz's technique yields a less diverse portfolio, concentrated in a few stocks. The HRP approach, leveraging hierarchical clustering–based allocation, results in a more diverse and distributed portfolio. This approach presented the best out-of-sample performance and offers better tail risk management due to the diversification.

Indeed, the corresponding hierarchical risk parity strategies address the shortcomings of minimum-variance-based portfolio allocation. It is visual and flexible, and it seems to offer a robust methodology for portfolio allocation and portfolio management.

Chapter Summary

In this chapter, we learned about different clustering techniques and used them to capture the natural structure of data to enhance decision making across several areas of finance. Through the case studies, we demonstrated that clustering techniques can be useful in enhancing trading strategies and portfolio management.

In addition to offering an approach to different finance problems, the case studies focused on understanding the concepts of clustering models, developing intuition, and visualizing clusters. Overall, the concepts in Python, machine learning, and finance presented in this chapter through the case studies can used as a blueprint for any other clustering-based problem in finance.

Having covered supervised and unsupervised learning, we will explore another type of machine learning, reinforcement learning, in the next chapter.

Exercises

- Use hierarchical clustering to form clusters of investments in a different asset class, such as forex or commodities.
- Apply clustering analysis for pairs trading in the interest rate market on the universe of bonds.