CHAPTER 7

Unsupervised Learning: Dimensionality Reduction

In previous chapters, we used supervised learning techniques to build machine learning models using data where the answer was already known (i.e., the class labels were available in our input data). Now we will explore *unsupervised learning*, where we draw inferences from datasets consisting of input data when the answer is unknown. Unsupervised learning algorithms attempt to infer patterns from the data without any knowledge of the output the data is meant to yield. Without requiring labeled data, which can be time-consuming and impractical to create or acquire, this family of models allows for easy use of larger datasets for analysis and model development.

Dimensionality reduction is a key technique within unsupervised learning. It compresses the data by finding a smaller, different set of variables that capture what matters most in the original features, while minimizing the loss of information. Dimensionality reduction helps mitigate problems associated with high dimensionality and permits the visualization of salient aspects of higher-dimensional data that is otherwise difficult to explore.

In the context of finance, where datasets are often large and contain many dimensions, dimensionality reduction techniques prove to be quite practical and useful. Dimensionality reduction enables us to reduce noise and redundancy in the dataset and find an approximate version of the dataset using fewer features. With fewer variables to consider, exploration and visualization of a dataset becomes more straightforward. Dimensionality reduction techniques also enhance supervised learning–based models by reducing the number of features or by finding new ones. Practitioners use these dimensionality reduction techniques to allocate funds across asset classes and individual investments, identify trading strategies and signals, implement portfolio hedging and risk management, and develop instrument pricing models.

In this chapter, we will discuss fundamental dimensionality reduction techniques and walk through three case studies in the areas of portfolio management, interest rate modeling, and trading strategy development. The case studies are designed to not only cover diverse topics from a finance standpoint but also highlight multiple machine learning and data science concepts. The standardized template containing the detailed implementation of modeling steps in Python and machine learning and finance concepts can be used as a blueprint for any other dimensionality reduction–based problem in finance.

In "Case Study 1: Portfolio Management: Finding an Eigen Portfolio" on page 202, we use a dimensionality reduction algorithm to allocate capital into different asset classes to maximize risk-adjusted returns. We also introduce a backtesting framework to assess the performance of the portfolio we constructed.

In "Case Study 2: Yield Curve Construction and Interest Rate Modeling" on page 217, we use dimensionality reduction techniques to generate the typical movements of a yield curve. This will illustrate how dimensionality reduction techniques can be used for reducing the dimension of market variables across a number of asset classes to promote faster and effective portfolio management, trading, hedging, and risk management.

In "Case Study 3: Bitcoin Trading: Enhancing Speed and Accuracy" on page 227, we use dimensionality reduction techniques for algorithmic trading. This case study demonstrates data exploration in low dimension.

In addition to the points mentioned above, readers will understand the following points by the end of this chapter:

- Basic concepts of models and techniques used for dimensionality reduction and how to implement them in Python.
- Concepts of eigenvalues and eigenvectors of Principal Component Analysis (PCA), selecting the right number of principal components, and extracting the factor weights of principal components.
- Usage of dimensionality reduction techniques such as singular value decomposition (SVD) and t-SNE to summarize high-dimensional data for effective data exploration and visualization.
- How to reconstruct the original data using the reduced principal components.
- How to enhance the speed and accuracy of supervised learning algorithms using dimensionality reduction.

- A backtesting framework for the portfolio performance computing and analyzing portfolio performance metrics such as the Sharpe ratio and the annualized return of the portfolio.

This Chapter's Code Repository

A Python-based master template for dimensionality reduction, along with the Jupyter notebook for all the case studies in this chapter, is included in the folder *Chapter 7 - Unsup. Learning - Dimensionality Reduction* in the code repository for this book. To work through any dimensionality reduction–modeling machine learning problems in Python involving the dimensionality reduction models (such as PCA, SVD, Kernel PCA, or t-SNE) presented in this chapter, readers need to modify the template slightly to align with their problem statement. All the case studies presented in this chapter use the standard Python master template with the standardized model development steps presented in Chapter 3. For the dimensionality reduction case studies, steps 6 (i.e., model tuning) and 7 (i.e., finalizing the model) are relatively lighter compared to the supervised learning models, so these steps have been merged with step 5. For situations in which steps are irrelevant, they have been skipped or combined with others to make the flow of the case study more intuitive.

Dimensionality Reduction Techniques

Dimensionality reduction represents the information in a given dataset more efficiently by using fewer features. These techniques project data onto a lower dimensional space by either discarding variation in the data that is not informative or identifying a lower dimensional subspace on or near where the data resides.

There are many types of dimensionality reduction techniques. In this chapter, we will cover these most frequently used techniques for dimensionality reduction:

- Principal component analysis (PCA)
- Kernel principal component analysis (KPCA)
- t-distributed stochastic neighbor embedding (t-SNE)

After application of these dimensionality reduction techniques, the low-dimension feature subspace can be a linear or nonlinear function of the corresponding high-dimensional feature subspace. Hence, on a broad level these dimensionality reduction algorithms can be classified as linear and nonlinear. Linear algorithms, such as PCA, force the new variables to be linear combinations of the original features.

Nonlinear algorithms such KPCA and t-SNE can capture more complex structures in the data. However, given the infinite number of options, the algorithms still need to make assumptions to arrive at a solution.

Principal Component Analysis

The idea of principal component analysis (PCA) is to reduce the dimensionality of a dataset with a large number of variables, while retaining as much variance in the data as possible. PCA allows us to understand whether there is a different representation of the data that can explain a majority of the original data points.

PCA finds a set of new variables that, through a linear combination, yield the original variables. The new variables are called *principal components* (PCs). These principal components are orthogonal (or independent) and can represent the original data. The number of components is a hyperparameter of the PCA algorithm that sets the target dimensionality.

The PCA algorithm works by projecting the original data onto the principal component space. It then identifies a sequence of principal components, each of which aligns with the direction of maximum variance in the data (after accounting for variation captured by previously computed components). The sequential optimization also ensures that new components are not correlated with existing components. Thus the resulting set constitutes an orthogonal basis for a vector space.

The decline in the amount of variance of the original data explained by each principal component reflects the extent of correlation among the original features. The number of components that capture, for example, 95% of the original variation relative to the total number of features provides an insight into the linearly independent information of the original data. In order to understand how PCA works, let's consider the distribution of data shown in Figure 7-1.

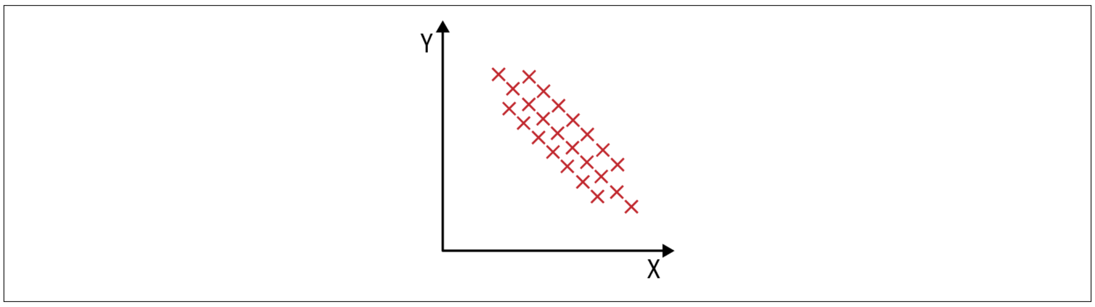


Figure 7-1. PCA-1

PCA finds a new quadrant system (*y'* and *x'* axes) that is obtained from the original through translation and rotation. It will move the center of the coordinate system from the original point *(0, 0)* to the center of the distribution of data points. It will then move the x-axis into the principal axis of variation, which is the one with the most variation relative to data points (i.e., the direction of maximum spread). Then it moves the other axis orthogonally to the principal one, into a less important direction of variation.

Figure 7-2 shows an example of PCA in which two dimensions explain nearly all the variance of the underlying data.

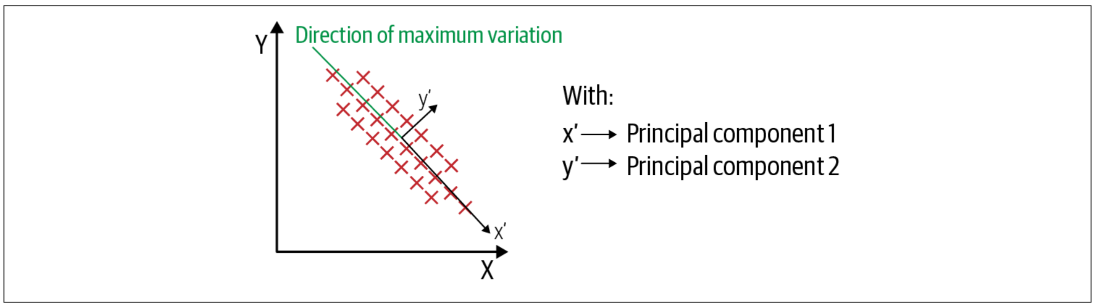

Direction of maximum variation
Y
X
y'
x'
With:
x' ⟶ Principal component 1
y' ⟶ Principal component 2

Figure 7-2. PCA-2

These new directions that contain the maximum variance are called principal components and are orthogonal to each other by design.

There are two approaches to finding the principal components: *Eigen decomposition* and *singular value decomposition* (SVD).

Eigen decomposition

The steps of Eigen decomposition are as follows:

1. First, a covariance matrix is created for the features.
2. Once the covariance matrix is computed, the *eigenvectors* of the covariance matrix are calculated.[1] These are the directions of maximum variance.
3. The *eigenvalues* are then created. They define the magnitude of the principal components.

So, for n dimensions, there will be an $n \times n$ variance-covariance matrix, and as a result, we will have an eigenvector of n values and n eigenvalues.

Python's sklearn library offers a powerful implementation of PCA. The `sklearn.decomposition.PCA` function computes the desired number of principal components and projects the data into the component space. The following code snippet illustrates how to create two principal components from a dataset.

1 Eigenvectors and eigenvalues are concepts of linear algebra.

`Implementation`

```python
# Import PCA Algorithm
from sklearn.decomposition import PCA
# Initialize the algorithm and set the number of PC's
pca = PCA(n_components=2)
# Fit the model to data
pca.fit(data)
# Get list of PC's
pca.components_
# Transform the model to data
pca.transform(data)
# Get the eigenvalues
pca.explained_variance_ratio
```

There are additional items, such as *factor loading*, that can be obtained using the functions in the sklearn library. Their use will be demonstrated in the case studies.

Singular value decomposition

Singular value decomposition (SVD) is factorization of a matrix into three matrices and is applicable to a more general case of $m \times n$ rectangular matrices.

If A is an $m \times n$ matrix, then SVD can express the matrix as:

$$A = U\Sigma V^T$$

where A is an $m \times n$ matrix, U is an $(m \times m)$ orthogonal matrix, Σ is an $(m \times n)$ nonnegative rectangular diagonal matrix, and V is an $(n \times n)$ orthogonal matrix. SVD of a given matrix tells us exactly how we can decompose the matrix. Σ is a diagonal matrix with m diagonal values called *singular values*. Their magnitude indicates how significant they are to preserving the information of the original data. V contains the principal components as column vectors.

As shown above, both Eigen decomposition and SVD tell us that using PCA is effectively looking at the initial data from a different angle. Both will always give the same answer; however, SVD can be much more efficient than Eigen decomposition, as it is able to handle sparse matrices (those which contain very few nonzero elements). In addition, SVD yields better numerical stability, especially when some of the features are strongly correlated.

Truncated SVD is a variant of SVD that computes only the largest singular values, where the number of computes is a user-specified parameter. This method is different from regular SVD in that it produces a factorization where the number of columns is equal to the specified truncation. For example, given an $n \times n$ matrix, SVD will produce matrices with n columns, whereas truncated SVD will produce matrices with a specified number of columns that may be less than n.

Implementation

```
from sklearn.decomposition import TruncatedSVD
svd = TruncatedSVD(ncomps=20).fit(X)
```

In terms of the weaknesses of the PCA technique, although it is very effective in reducing the number of dimensions, the resulting principal components may be less interpretable than the original features. Additionally, the results may be sensitive to the selected number of principal components. For example, too few principal components may miss some information compared to the original list of features. Also, PCA may not work well if the data is strongly nonlinear.

Kernel Principal Component Analysis

A main limitation of PCA is that it only applies linear transformations. Kernel principal component analysis (KPCA) extends PCA to handle nonlinearity. It first maps the original data to some nonlinear feature space (usually one of higher dimension). Then it applies PCA to extract the principal components in that space.

A simple example of when KPCA is applicable is shown in Figure 7-3. Linear transformations are suitable for the blue and red data points on the left-hand plot. However, if all dots are arranged as per the graph on the right, the result is not linearly separable. We would then need to apply KPCA to separate the components.

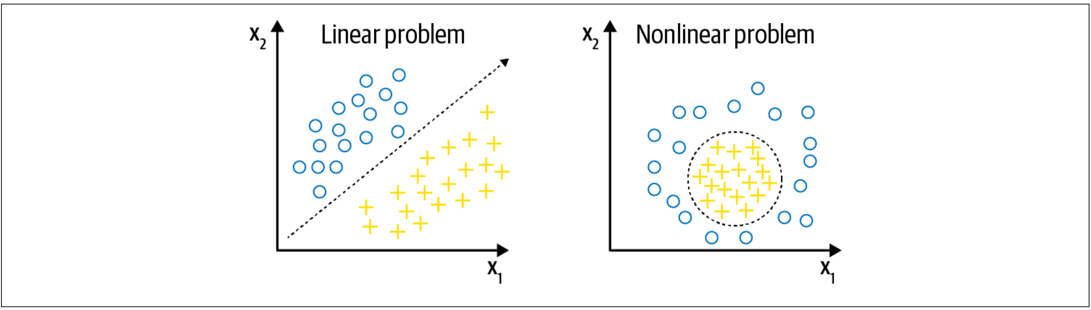


Figure 7-3. Kernel PCA

Implementation

```
from sklearn.decomposition import KernelPCA
kpca = KernelPCA(n_components=4, kernel='rbf').fit_transform(X)
```

In the Python code, we specify `kernel='rbf'`, which is the radial basis function kernel. This is commonly used as a kernel in machine learning techniques, such as in SVMs (see Chapter 4).

Using KPCA, component separation becomes easier in a higher dimensional space, as mapping into a higher dimensional space often provides greater classification power.

t-distributed Stochastic Neighbor Embedding

t-distributed stochastic neighbor embedding (t-SNE) is a dimensionality reduction algorithm that reduces the dimensions by modeling the probability distribution of neighbors around each point. Here, the term *neighbors* refers to the set of points closest to a given point. The algorithm emphasizes keeping similar points together in low dimensions as opposed to maintaining the distance between points that are apart in high dimensions.

The algorithm starts by calculating the probability of similarity of data points in corresponding high and low dimensional space. The similarity of points is calculated as the conditional probability that a point A would choose point B as its neighbor if neighbors were picked in proportion to their probability density under a normal distribution centered at A. The algorithm then tries to minimize the difference between these conditional probabilities (or similarities) in the high and low dimensional spaces for a perfect representation of data points in the low dimensional space.

`Implementation`

```
from sklearn.manifold import TSNE
X_tsne = TSNE().fit_transform(X)
```

An implementation of t-SNE is shown in the third case study presented in this chapter.

Case Study 1: Portfolio Management: Finding an Eigen Portfolio

A primary objective of portfolio management is to allocate capital into different asset classes to maximize risk-adjusted returns. Mean-variance portfolio optimization is the most commonly used technique for asset allocation. This method requires an estimated covariance matrix and expected returns of the assets considered. However, the erratic nature of financial returns leads to estimation errors in these inputs, especially when the sample size of returns is insufficient compared to the number of assets being allocated. These errors greatly jeopardize the optimality of the resulting portfolios, leading to poor and unstable outcomes.

Dimensionality reduction is a technique we can use to address this issue. Using PCA, we can take an $n \times n$ covariance matrix of our assets and create a set of n linearly uncorrelated principal portfolios (sometimes referred to in literature as an *eigen portfolio*) made up of our assets and their corresponding variances. The principal components of the covariance matrix capture most of the covariation among the assets and are mutually uncorrelated. Moreover, we can use standardized principal components as the portfolio weights, with the statistical guarantee that the returns from these principal portfolios are linearly uncorrelated.

By the end of this case study, readers will be familiar with a general approach to finding an eigen portfolio for asset allocation, from understanding concepts of PCA to backtesting different principal components.

This case study will focus on:

- Understanding eigenvalues and eigenvectors of PCA and deriving portfolio weights using the principal components.
- Developing a backtesting framework to evaluate portfolio performance.
- Understanding how to work through a dimensionality reduction modeling problem from end to end.

Blueprint for Using Dimensionality Reduction for Asset Allocation

1. Problem definition

Our goal in this case study is to maximize the risk-adjusted returns of an equity portfolio using PCA on a dataset of stock returns.

The dataset used for this case study is the Dow Jones Industrial Average (DJIA) index and its respective 30 stocks. The return data used will be from the year 2000 onwards and can be downloaded from Yahoo Finance.

We will also compare the performance of our hypothetical portfolios against a benchmark and backtest the model to evaluate the effectiveness of the approach.

2. Getting started—loading the data and Python packages

2.1. Loading the Python packages. The list of the libraries used for data loading, data analysis, data preparation, model evaluation, and model tuning are shown below. The details of most of these packages and functions can be found in Chapters 2 and 4.

`Packages for dimensionality reduction`

```
from sklearn.decomposition import PCA
from sklearn.decomposition import TruncatedSVD
from numpy.linalg import inv, eig, svd
from sklearn.manifold import TSNE
from sklearn.decomposition import KernelPCA
```

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

```
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
```

2.2. Loading the data. We import the dataframe containing the adjusted closing prices for all the companies in the DJIA index:

```
# load dataset
dataset = read_csv('Dow_adjcloses.csv', index_col=0)
```

3. Exploratory data analysis

Next, we inspect the dataset.

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

```
dataset.shape
```

```
Output
```

```
(4804, 30)
```

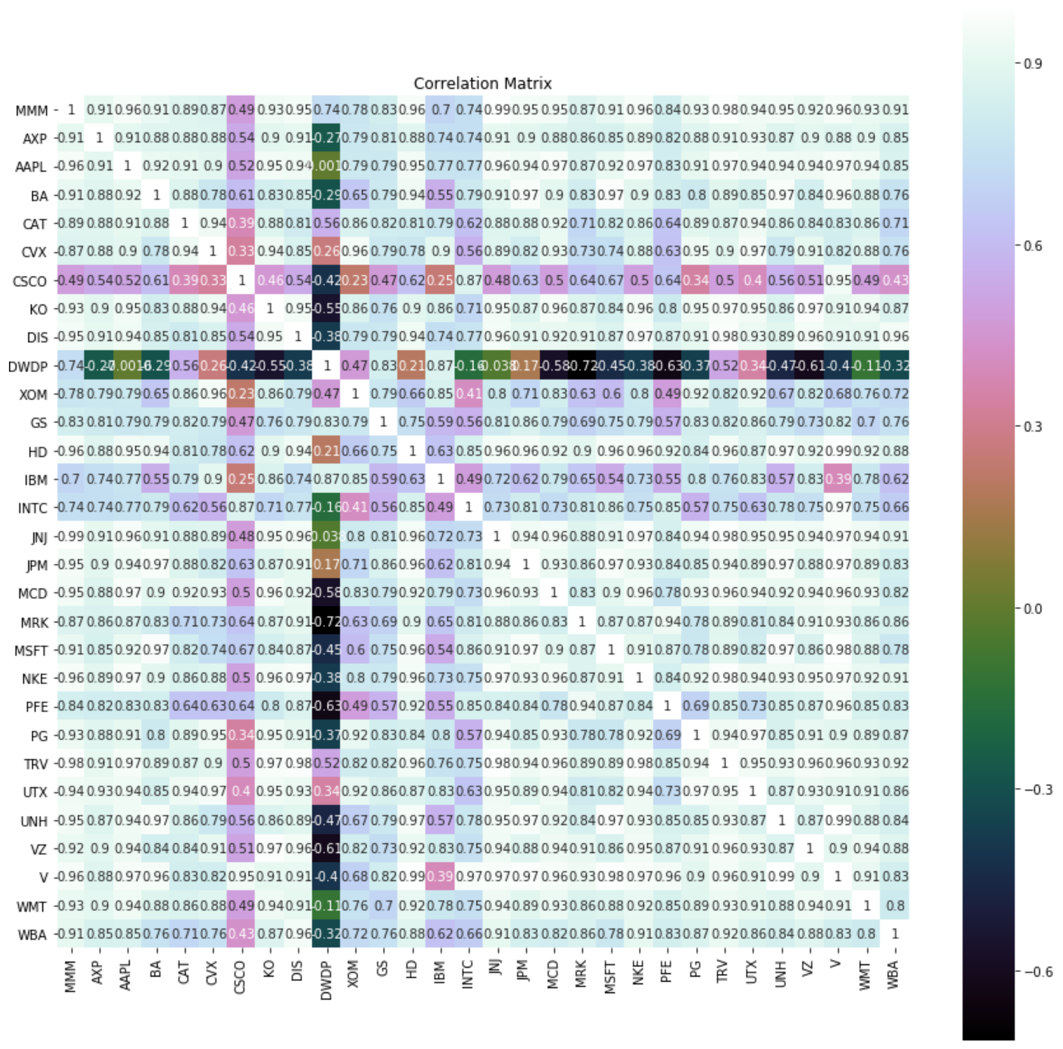
The data is comprised of 30 columns and 4,804 rows containing the daily closing prices of the 30 stocks in the index since 2000.

3.2. Data visualization. The first thing we must do is gather a basic sense of our data. Let us take a look at the return correlations:

```
correlation = dataset.corr()
plt.figure(figsize=(15, 15))
plt.title('Correlation Matrix')
sns.heatmap(correlation, vmax=1, square=True,annot=True, cmap='cubehelix')
```

There is a significant positive correlation between the daily returns. The plot (full-size version available on GitHub) also indicates that the information embedded in the data may be represented by fewer variables (i.e., something smaller than the 30 dimensions we have now). We will perform another detailed look at the data after implementing dimensionality reduction.

```
Output
```



4. Data preparation

We prepare the data for modeling in the following sections.

4.1. Data cleaning. First, 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
```

Some stocks were added to the index after our start date. To ensure proper analysis, we will drop those with more than 30% missing values. Two stocks fit this criteria—Dow Chemicals and Visa:

```
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`

```
(4804, 28)
```

We end up with return data for 28 companies and an additional one for the DJIA index. Now we fill the NAs with the mean of the columns:

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

4.2. Data transformation. In addition to handling the missing values, we also want to standardize the dataset features onto a unit scale (mean = 0 and variance = 1). All the variables should be on the same scale before applying PCA; otherwise, a feature with large values will dominate the result. We use `StandardScaler` in sklearn to standardize the dataset, as shown below:

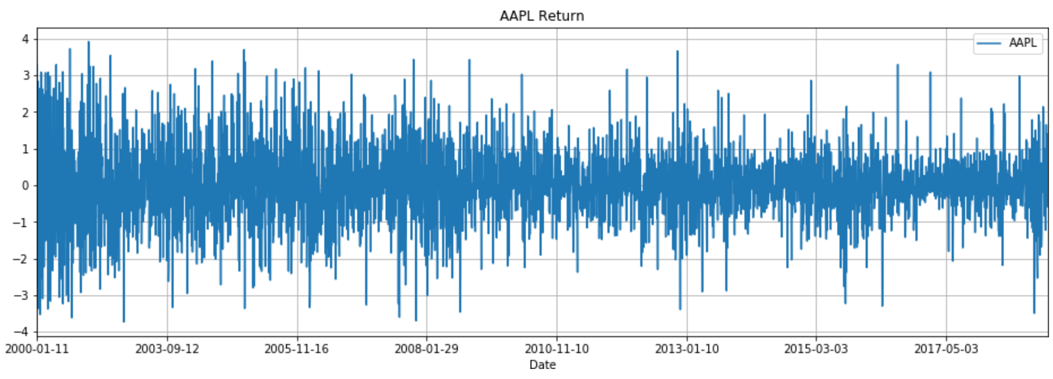
```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler().fit(datareturns)
rescaledDataset = pd.DataFrame(scaler.fit_transform(datareturns),columns =\
 datareturns.columns, index = datareturns.index)
# summarize transformed data
datareturns.dropna(how='any', inplace=True)
rescaledDataset.dropna(how='any', inplace=True)
```

Overall, cleaning and standardizing the data is important in order to create a meaningful and reliable dataset to be used in dimensionality reduction without error.

Let us look at the returns of one of the stocks from the cleaned and standardized dataset:

```
# Visualizing Log Returns for the DJIA
plt.figure(figsize=(16, 5))
plt.title("AAPL Return")
rescaledDataset.AAPL.plot()
plt.grid(True);
plt.legend()
plt.show()
```

`Output`



5. Evaluate algorithms and models

5.1. Train-test split. The portfolio is divided into training and test sets to perform the analysis regarding the best portfolio and to perform backtesting:

```
# Dividing the dataset into training and testing sets
percentage = int(len(rescaledDataset) * 0.8)
X_train = rescaledDataset[:percentage]
X_test = rescaledDataset[percentage:]

stock_tickers = rescaledDataset.columns.values
n_tickers = len(stock_tickers)
```

5.2. Model evaluation: applying principal component analysis. As the next step, we create a function to perform PCA using the sklearn library. This function generates the principal components from the data that will be used for further analysis:

```
pca = PCA()
PrincipalComponent=pca.fit(X_train)
```

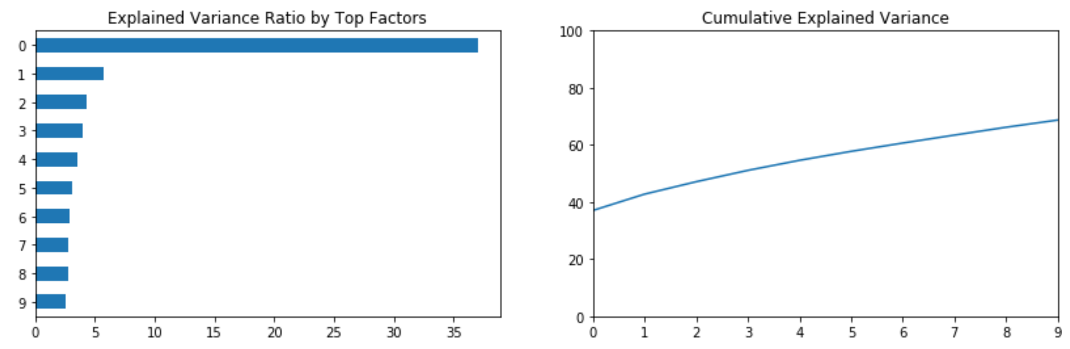
5.2.1. Explained variance using PCA. In this step, we look at the variance explained using PCA. The decline in the amount of variance of the original data explained by each principal component reflects the extent of correlation among the original features. The first principal component captures the most variance in the original data, the second component is a representation of the second highest variance, and so on. The eigenvectors with the lowest eigenvalues describe the least amount of variation within the dataset. Therefore, these values can be dropped.

The following charts show the number of principal components and the variance explained by each.

```
NumEigenvalues=20
fig, axes = plt.subplots(ncols=2, figsize=(14,4))
Series1 = pd.Series(pca.explained_variance_ratio_[:NumEigenvalues]).sort_values()
```

```
Series2 = pd.Series(pca.explained_variance_ratio_[:NumEigenvalues]).cumsum()
Series1.plot.barh(title='Explained Variance Ratio by Top Factors', ax=axes[0]);
Series1.plot(ylim=(0,1), ax=axes[1], title='Cumulative Explained Variance');
```

Output



We find that the most important factor explains around 40% of the daily return variation. This dominant principal component is usually interpreted as the "market" factor. We will discuss the interpretation of this and the other factors when looking at the portfolio weights.

The plot on the right shows the cumulative explained variance and indicates that around ten factors explain 73% of the variance in returns of the 28 stocks analyzed.

5.2.2. Looking at portfolio weights. In this step, we look more closely at the individual principal components. These may be less interpretable than the original features. However, we can look at the weights of the factors on each principal component to assess any intuitive themes relative to the 28 stocks. We construct five portfolios, defining the weights of each stock as each of the first five principal components. We then create a scatterplot that visualizes an organized descending plot with the respective weight of every company at the current chosen principal component:

```
def PCWeights():
    #Principal Components (PC) weights for each 28 PCs

    weights = pd.DataFrame()
    for i in range(len(pca.components_)):
        weights["weights_{}".format(i)] = \
        pca.components_[i] / sum(pca.components_[i])
    weights = weights.values.T
    return weights
weights=PCWeights()
```

```
sum(pca.components_[0])
```

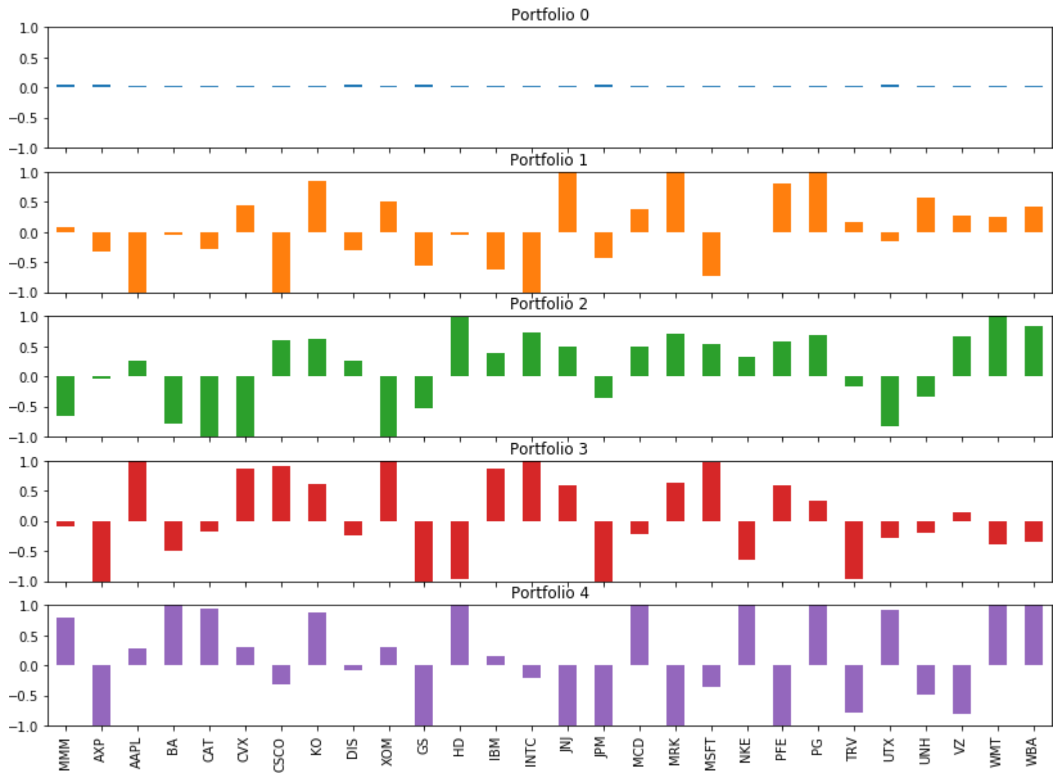
Output

```
-5.247808242068631
```

```
NumComponents=5
topPortfolios = pd.DataFrame(pca.components_[:NumComponents],\
   columns=dataset.columns)
eigen_portfolios = topPortfolios.div(topPortfolios.sum(1), axis=0)
eigen_portfolios.index = [f'Portfolio {i}' for i in range( NumComponents)]
np.sqrt(pca.explained_variance_)
eigen_portfolios.T.plot.bar(subplots=True, layout=(int(NumComponents),1),  \
figsize=(14,10), legend=False, sharey=True, ylim= (-1,1))
```

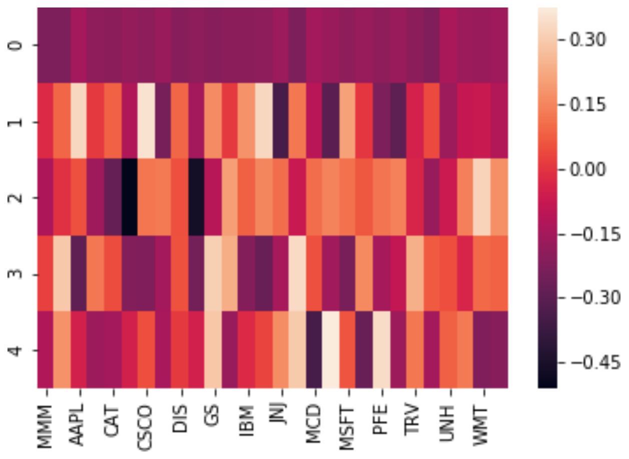
Given that scale for the plots are the same, we can also look at the heatmap as follows:

`Output`



```
# plotting heatmap
sns.heatmap(topPortfolios)
```

```
Output
```



The heatmap and barplots show the contribution of different stocks in each eigenvector.

Traditionally, the intuition behind each principal portfolio is that it represents some sort of independent risk factor. The manifestation of those risk factors depends on the assets in the portfolio. In our case study, the assets are all U.S. domestic equities. The principal portfolio with the largest variance is typically a systematic risk factor (i.e., “market” factor). Looking at the first principal component (*Portfolio 0*), we see that the weights are distributed homogeneously across the stocks. This nearly equal weighted portfolio explains 40% of the variance in the index and is a fair representation of a systematic risk factor.

The rest of the eigen portfolios typically correspond to sector or industry factors. For example, *Portfolio 1* assigns a high weight to JNJ and MRK, which are stocks from the health care sector. Similarly, *Portfolio 3* has high weights on technology and electronics companies, such AAPL, MSFT, and IBM.

When the asset universe for our portfolio is expanded to include broad, global investments, we may identify factors for international equity risk, interest rate risk, commodity exposure, geographic risk, and many others.

In the next step, we find the best eigen portfolio.

5.2.3. Finding the best eigen portfolio. To determine the best eigen portfolio, we use the *Sharpe ratio*. This is an assessment of risk-adjusted performance that explains the annualized returns against the annualized volatility of a portfolio. A high Sharpe ratio explains higher returns and/or lower volatility for the specified portfolio. The annualized Sharpe ratio is computed by dividing the annualized returns against the annualized volatility. For annualized return we apply the geometric average of all the returns in respect to the periods per year (days of operations in the exchange in a year). Annualized volatility is computed by taking the standard deviation of the returns and multiplying it by the square root of the periods per year.

The following code computes the Sharpe ratio of a portfolio:

```
# Sharpe Ratio Calculation
# Calculation based on conventional number of trading days per year (i.e., 252).
def sharpe_ratio(ts_returns, periods_per_year=252):
    n_years = ts_returns.shape[0]/ periods_per_year
    annualized_return = np.power(np.prod(1+ts_returns), (1/n_years))-1
    annualized_vol = ts_returns.std() * np.sqrt(periods_per_year)
    annualized_sharpe = annualized_return / annualized_vol

    return annualized_return, annualized_vol, annualized_sharpe
```

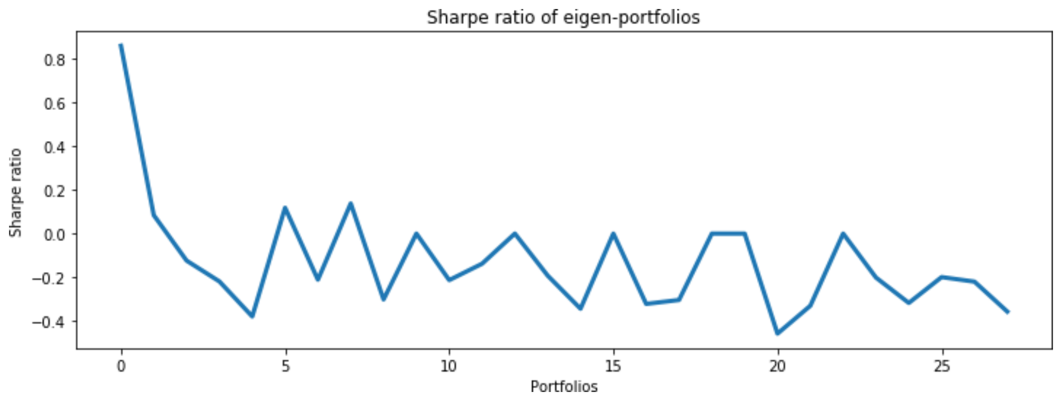
We construct a loop to compute the principal component weights for each eigen portfolio. Then it uses the Sharpe ratio function to look for the portfolio with the highest Sharpe ratio. Once we know which portfolio has the highest Sharpe ratio, we can visualize its performance against the index for comparison:

```
def optimizedPortfolio():
    n_portfolios = len(pca.components_)
    annualized_ret = np.array([0.] * n_portfolios)
    sharpe_metric = np.array([0.] * n_portfolios)
    annualized_vol = np.array([0.] * n_portfolios)
    highest_sharpe = 0
    stock_tickers = rescaledDataset.columns.values
    n_tickers = len(stock_tickers)
    pcs = pca.components_

    for i in range(n_portfolios):

        pc_w = pcs[i] / sum(pcs[i])
        eigen_prtfi = pd.DataFrame(data ={'weights': pc_w.squeeze()*100}, \
        index = stock_tickers)
        eigen_prtfi.sort_values(by=['weights'], ascending=False, inplace=True)
        eigen_prti_returns = np.dot(X_train_raw.loc[:, eigen_prtfi.index], pc_w)
        eigen_prti_returns = pd.Series(eigen_prti_returns.squeeze(),\
         index=X_train_raw.index)
        er, vol, sharpe = sharpe_ratio(eigen_prti_returns)
        annualized_ret[i] = er
        annualized_vol[i] = vol
        sharpe_metric[i] = sharpe

        sharpe_metric= np.nan_to_num(sharpe_metric)

    # find portfolio with the highest Sharpe ratio
    highest_sharpe = np.argmax(sharpe_metric)

    print('Eigen portfolio #%d with the highest Sharpe. Return %.2f%%,\
     vol = %.2f%%, Sharpe = %.2f' %
```

```
            (highest_sharpe,
             annualized_ret[highest_sharpe]*100,
             annualized_vol[highest_sharpe]*100,
             sharpe_metric[highest_sharpe]))


    fig, ax = plt.subplots()
    fig.set_size_inches(12, 4)
    ax.plot(sharpe_metric, linewidth=3)
    ax.set_title('Sharpe ratio of eigen-portfolios')
    ax.set_ylabel('Sharpe ratio')
    ax.set_xlabel('Portfolios')

    results = pd.DataFrame(data={'Return': annualized_ret,\
    'Vol': annualized_vol,
    'Sharpe': sharpe_metric})
    results.dropna(inplace=True)
    results.sort_values(by=['Sharpe'], ascending=False, inplace=True)
    print(results.head(5))

    plt.show()

optimizedPortfolio()
```

`Output`

```
Eigen portfolio #0 with the highest Sharpe. Return 11.47%, vol = 13.31%, \
Sharpe = 0.86
    Return    Vol   Sharpe
0    0.115  0.133   0.862
7    0.096  0.693   0.138
5    0.100  0.845   0.118
1    0.057  0.670   0.084
```

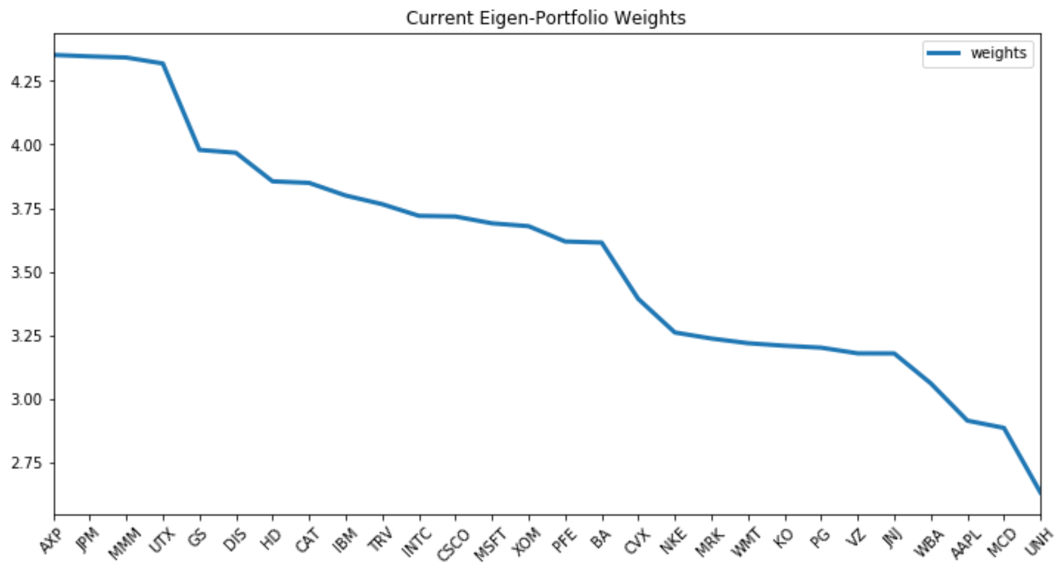


As shown by the results above, *Portfolio 0* is the best performing one, with the highest return *and* the lowest volatility. Let us look at the composition of this portfolio:

```
weights = PCWeights()
portfolio = portfolio = pd.DataFrame()
```

```
def plotEigen(weights, plot=False, portfolio=portfolio):
    portfolio = pd.DataFrame(data ={'weights': weights.squeeze() * 100}, \
    index = stock_tickers)
    portfolio.sort_values(by=['weights'], ascending=False, inplace=True)
    if plot:
        portfolio.plot(title='Current Eigen-Portfolio Weights',
            figsize=(12, 6),
            xticks=range(0, len(stock_tickers), 1),
            rot=45,
            linewidth=3
            )
        plt.show()


    return portfolio

# Weights are stored in arrays, where 0 is the first PC's weights.
plotEigen(weights=weights[0], plot=True)
```

`Output`



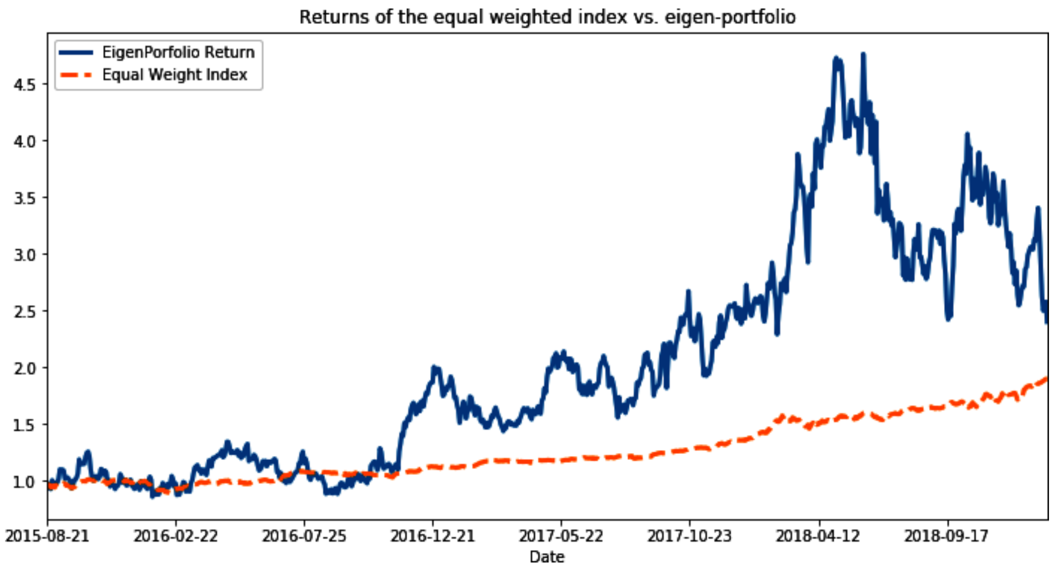
Recall that this is the portfolio that explains 40% of the variance and represents the systematic risk factor. Looking at the portfolio weights (in percentages in the y-axis), they do not vary much and are in the range of 2.7% to 4.5% across all stocks. However, the weights seem to be higher in the financial sector, and stocks such as AXP, JPM, and GS have higher-than-average weights.

5.2.4. Backtesting the eigen portfolios. We will now try to backtest this algorithm on the test set. We will look at a few of the top performers and the worst performer. For the top performers we look at the 3rd- and 4th-ranked eigen portfolios (*Portfolios 5* and *1*), while the worst performer reviewed was ranked 19th (*Portfolio 14*):

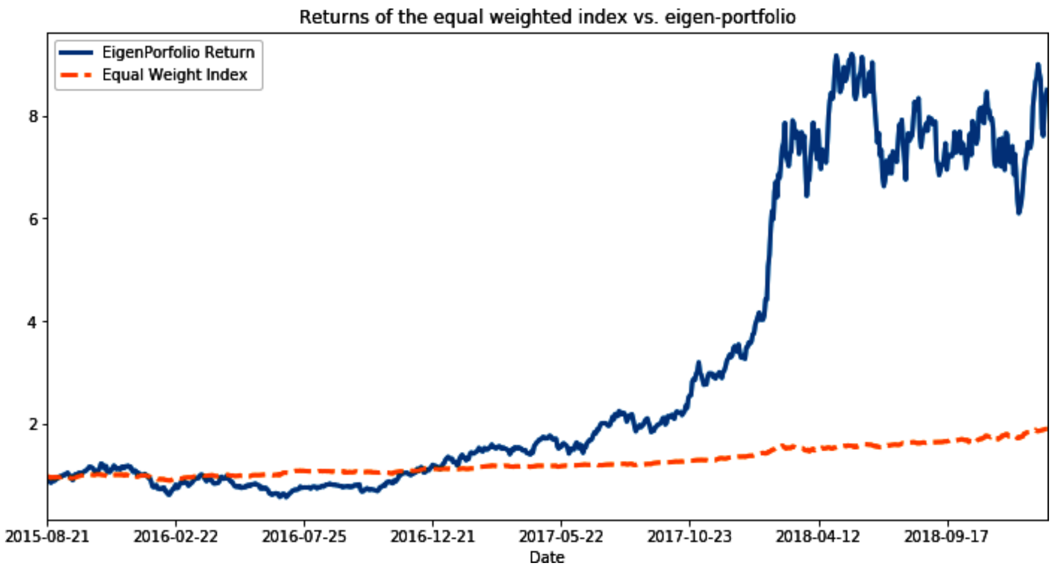
```
def Backtest(eigen):

    '''
    Plots principal components returns against real returns.
    '''

    eigen_prtfi = pd.DataFrame(data ={'weights': eigen.squeeze()}, \
    index=stock_tickers)
    eigen_prtfi.sort_values(by=['weights'], ascending=False, inplace=True)

    eigen_prti_returns = np.dot(X_test_raw.loc[:, eigen_prtfi.index], eigen)
    eigen_portfolio_returns = pd.Series(eigen_prti_returns.squeeze(),\
     index=X_test_raw.index)
    returns, vol, sharpe = sharpe_ratio(eigen_portfolio_returns)
    print('Current Eigen-Portfolio:\nReturn = %.2f%%\nVolatility = %.2f%%\n\
    Sharpe = %.2f' % (returns * 100, vol * 100, sharpe))
    equal_weight_return=(X_test_raw * (1/len(pca.components_))).sum(axis=1)
    df_plot = pd.DataFrame({'EigenPorfolio Return': eigen_portfolio_returns, \
    'Equal Weight Index': equal_weight_return}, index=X_test.index)
    np.cumprod(df_plot + 1).plot(title='Returns of the equal weighted\
     index vs. First eigen-portfolio',
                          figsize=(12, 6), linewidth=3)
    plt.show()

Backtest(eigen=weights[5])
Backtest(eigen=weights[1])
Backtest(eigen=weights[14])
```

`Output`

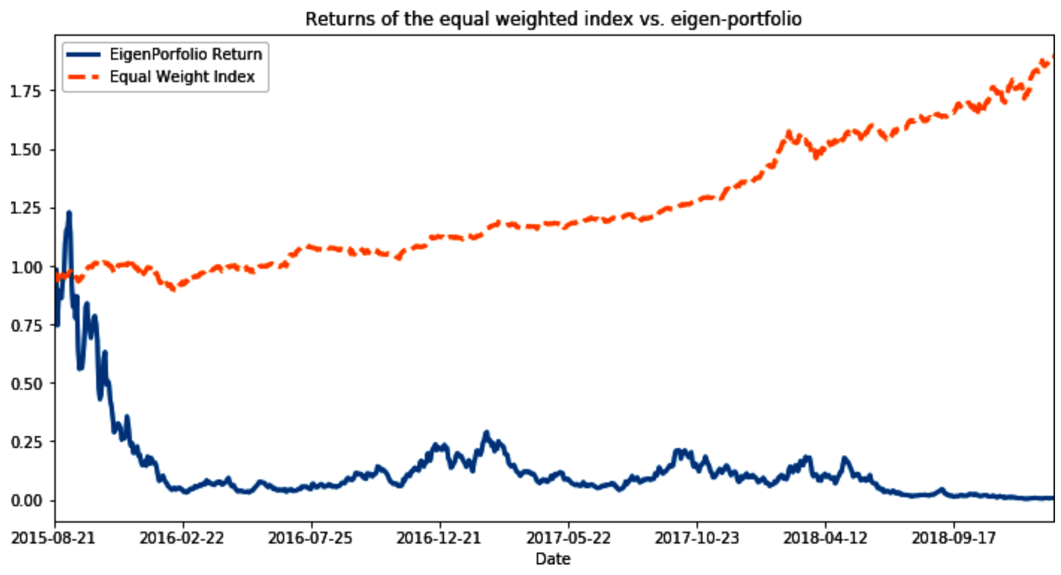
```
Current Eigen-Portfolio:
Return = 32.76%
Volatility = 68.64%
Sharpe = 0.48
```



```
Current Eigen-Portfolio:
Return = 99.80%
Volatility = 58.34%
Sharpe = 1.71
```



```
Current Eigen-Portfolio:
Return = -79.42%
Volatility = 185.30%
Sharpe = -0.43
```



As shown in the preceding charts, the eigen portfolio return of the top portfolios outperforms the equally weighted index. The eigen portfolio ranked 19th underperformed the market significantly in the test set. The outperformance and underperformance are attributed to the weights of the stocks or sectors in the eigen portfolio. We can drill down further to understand the individual drivers of each portfolio. For example, *Portfolio 1* assigns high weight to several stocks in the health care sector, as discussed previously. This sector saw a significant increase in 2017 onwards, which is reflected in the chart for *Eigen Portfolio 1*.

Given that these eigen portfolios are independent, they also provide diversification opportunities. As such, we can invest across these uncorrelated eigen portfolios, providing other potential portfolio management benefits.

Conclusion

In this case study, we applied dimensionality reduction techniques in the context of portfolio management, using eigenvalues and eigenvectors from PCA to perform asset allocation.

We demonstrated that, while some interpretability is lost, the initution behind the resulting portfolios can be matched to risk factors. In this example, the first eigen portfolio represented a systematic risk factor, while others exhibited sector or industry concentration.

Through backtesting, we found that the portfolio with the best result on the training set also achieved the strongest performance on the test set. Several of the portfolios outperformed the index based on the Sharpe ratio, the risk-adjusted performance metric used in this exercise.

Overall, we found that using PCA and analyzing eigen portfolios can yield a robust methodology for asset allocation and portfolio management.

Case Study 2: Yield Curve Construction and Interest Rate Modeling

A number of problems in portfolio management, trading, and risk management require a deep understanding and modeling of yield curves.

A yield curve represents interest rates, or yields, across a range of maturities, usually depicted in a line graph, as discussed in "Case Study 4: Yield Curve Prediction" on page 141 in Chapter 5. Yield curve illustrates the "price of funds" at a given point in time and, due to the time value of money, often shows interest rates rising as a function of maturity.

Researchers in finance have studied the yield curve and found that shifts or changes in the shape of the yield curve are attributable to a few unobservable factors. Specifically, empirical studies reveal that more than 99% of the movement of various U.S. Treasury bond yields are captured by three factors, which are often referred to as level, slope, and curvature. The names describe how each influences the yield curve shape in response to a shock. A level shock changes the interest rates of all maturities by almost identical amounts, inducing a *parallel shift* that changes the level of the entire curve up or down. A shock to the slope factor changes the difference in short-term and long-term rates. For instance, when long-term rates increase by a larger amount than do short-term rates, it results in a curve that becomes steeper (i.e., visually, the curve becomes more upward sloping). Changes in the short- and long-term rates can also produce a flatter yield curve. The main effects of the shock to the curvature factor focuses on medium-term interest rates, leading to hump, twist, or U-shaped characteristics.

Dimensionality reduction breaks down the movement of the yield curve into these three factors. Reducing the yield curve into fewer components means we can focus on a few intuitive dimensions in the yield curve. Traders and risk managers use this technique to condense the curve in risk factors for hedging the interest rate risk. Similarly, portfolio managers then have fewer dimensions to analyze when allocating funds. Interest rate structurers use this technique to model the yield curve and analyze its shape. Overall, it promotes faster and more effective portfolio management, trading, hedging, and risk management.

In this case study, we use PCA to generate typical movements of a yield curve and show that the first three principal components correspond to a yield curve's level, slope, and curvature, respectively.

This case study will focus on:

- Understanding the intuition behind eigenvectors.
- Using dimensions resulting from dimensionality reduction to reproduce the original data.

Blueprint for Using Dimensionality Reduction to Generate a Yield Curve

1. Problem definition

Our goal in this case study is to use dimensionality reduction techniques to generate the typical movements of a yield curve.

The data used for this case study is obtained from Quandl, a premier source for financial, economic, and alternative datasets. We use the data of 11 tenors (or maturities), from 1-month to 30-years, of Treasury curves. These are of daily frequency and are available from 1960 onwards.

2. Getting started—loading the data and Python packages

2.1. Loading the Python packages. The loading of Python packages is similar to the previous dimensionality reduction case study. Please refer to the Jupyter notebook of this case study for more details.

2.2. Loading the data. In the first step, we load the data of different tenors of the Treasury curves from Quandl:

```
# In order to use quandl, ApiConfig.api_key will need to be
# set to identify you to the quandl API. Please see API
# Documentation of quandl for more details
quandl.ApiConfig.api_key = 'API Key'

treasury = ['FRED/DGS1MO','FRED/DGS3MO','FRED/DGS6MO','FRED/DGS1',\
'FRED/DGS2','FRED/DGS3','FRED/DGS5','FRED/DGS7','FRED/DGS10',\
'FRED/DGS20','FRED/DGS30']

treasury_df = quandl.get(treasury)
treasury_df.columns = ['TRESY1mo','TRESY3mo','TRESY6mo','TRESY1y',\
'TRESY2y','TRESY3y','TRESY5y','TRESY7y','TRESY10y',\'TRESY20y','TRESY30y']
dataset = treasury_df
```

3. Exploratory data analysis

Here, we will take our first look at the data.

3.1. Descriptive statistics. In the next step we look at the shape of the dataset:

```
# shape
dataset.shape
```

Output

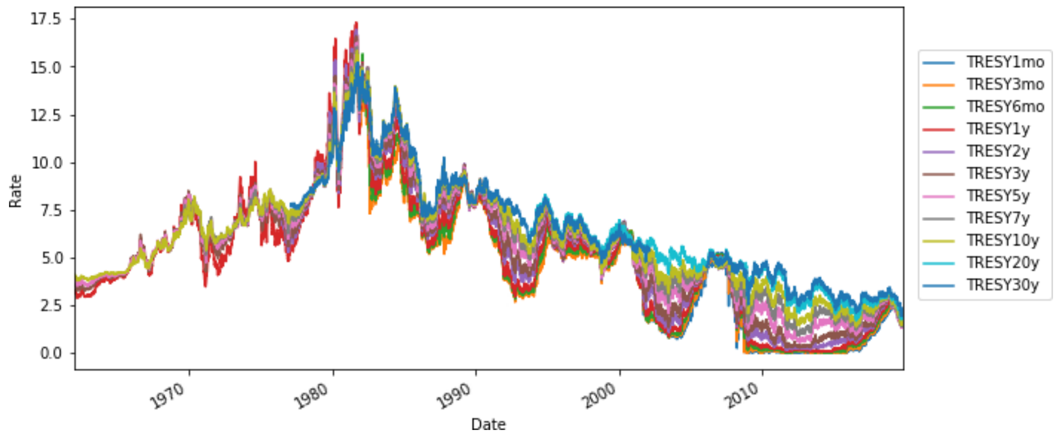
```
(14420, 11)
```

The dataset has 14,420 rows and has the data of 11 tenors of the Treasury curve for more than 50 years.

3.2. Data visualization. Let us look at the movement of the rates from the downloaded data:

```
dataset.plot(figsize=(10,5))
plt.ylabel("Rate")
plt.legend(bbox_to_anchor=(1.01, 0.9), loc=2)
plt.show()
```

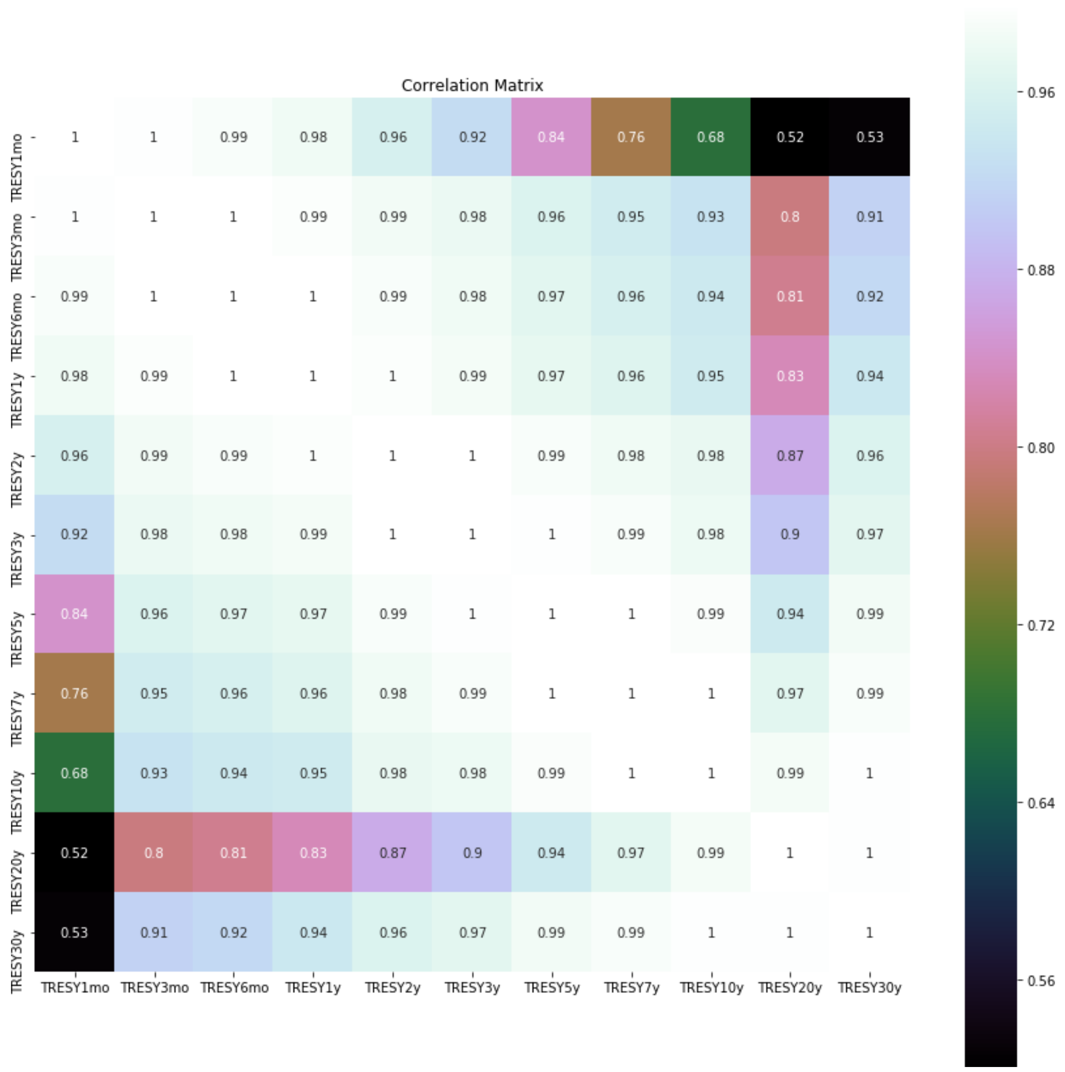
Output



In the next step we look at the correlations across tenors:

```
# correlation
correlation = dataset.corr()
plt.figure(figsize=(15, 15))
plt.title('Correlation Matrix')
sns.heatmap(correlation, vmax=1, square=True, annot=True, cmap='cubehelix')
```

```
Output
```



There is a significant positive correlation between the tenors, as you can see in the output (full-size version available on GitHub). This is an indication that reducing the number dimensions may be useful when modeling with the data. Additional visualizations of the data will be performed after implementing the dimensionality reduction models.

4. Data preparation

Data cleaning and transformation are a necessary modeling prerequisite in this case study.

4.1. Data cleaning. Here, we check for NAs in the data and either drop them or fill them with the mean of the column.

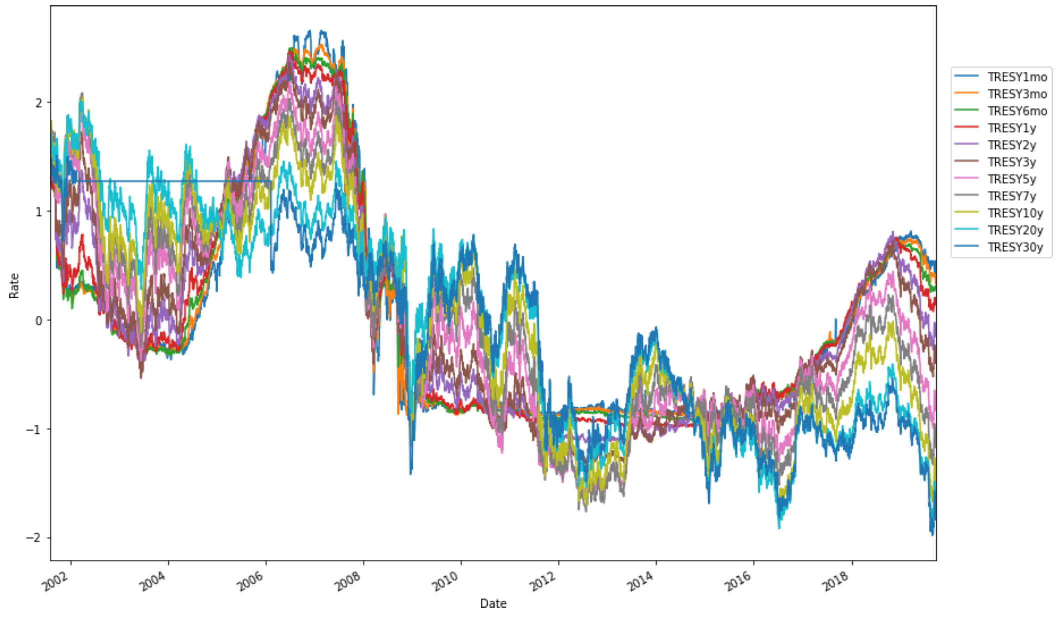
4.2. Data transformation. We standardize the variables on the same scale before applying PCA in order to prevent a feature with large values from dominating the result. We use the `StandardScaler` function in sklearn to standardize the dataset's features onto a unit scale (mean = 0 and variance = 1):

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler().fit(dataset)
rescaledDataset = pd.DataFrame(scaler.fit_transform(dataset),\
columns = dataset.columns,
index = dataset.index)
# summarize transformed data
dataset.dropna(how='any', inplace=True)
rescaledDataset.dropna(how='any', inplace=True)
```

`Visualizing the standardized dataset`

```
rescaledDataset.plot(figsize=(14, 10))
plt.ylabel("Rate")
plt.legend(bbox_to_anchor=(1.01, 0.9), loc=2)
plt.show()
```

`Output`



5. Evaluate algorithms and models

5.2. Model evaluation—applying principal component analysis. As a next step, we create a function to perform PCA using the sklearn library. This function generates the principal components from the data that will be used for further analysis:

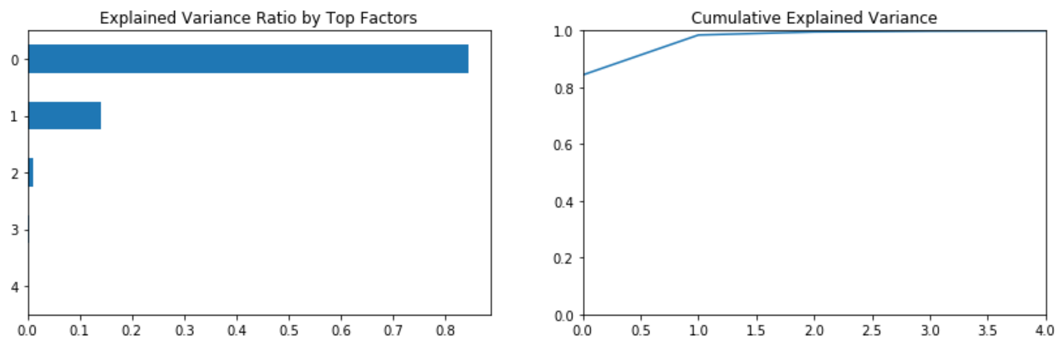
```
pca = PCA()
PrincipalComponent=pca.fit(rescaledDataset)
```

5.2.1. Explained variance using PCA.

```
NumEigenvalues=5
fig, axes = plt.subplots(ncols=2, figsize=(14, 4))
pd.Series(pca.explained_variance_ratio_[:NumEigenvalues]).sort_values().\
plot.barh(title='Explained Variance Ratio by Top Factors',ax=axes[0]);
pd.Series(pca.explained_variance_ratio_[:NumEigenvalues]).cumsum()\
.plot(ylim=(0,1),ax=axes[1], title='Cumulative Explained Variance');
# explained_variance
pd.Series(np.cumsum(pca.explained_variance_ratio_)).to_frame\
('Explained Variance_Top 5').head(NumEigenvalues).style.format('{:,.2%}'.format)
```

```
Output
```

	Explained Variance_Top 5
0	84.36%
1	98.44%
2	99.53%
3	99.83%
4	99.94%

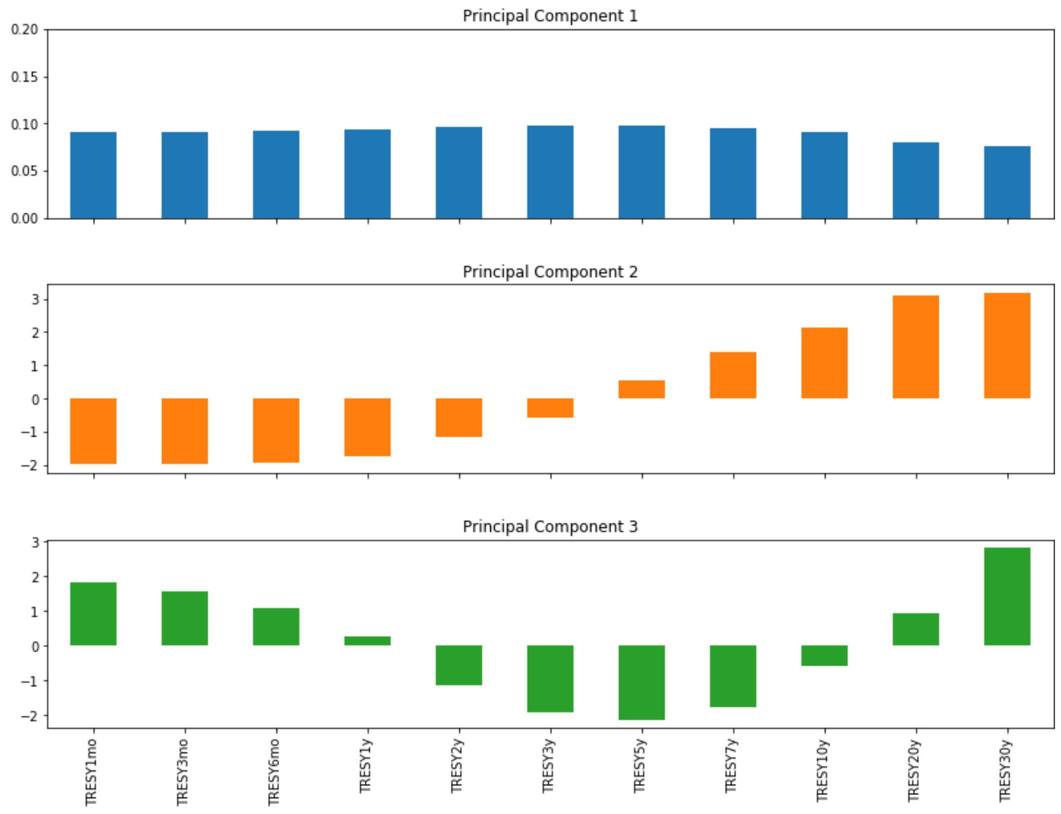


The first three principal components account for 84.4%, 14.08%, and 1.09% of variance, respectively. Cumulatively, they describe over 99.5% of all movement in the data. This is an incredibly efficient reduction in dimensions. Recall that in the first case study, we saw the first 10 components account for only 73% of variance.

5.2.2. Intuition behind the principal components. Ideally, we can have some intuition and interpretation of these principal components. To explore this, we first have a function to determine the weights of each principal component, and then perform the visualization of the principal components:

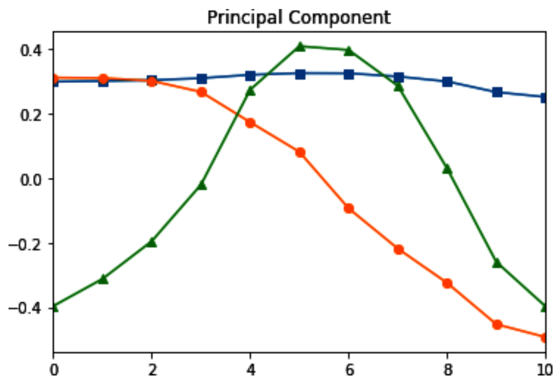
```
def PCWeights():
    '''
    Principal Components (PC) weights for each 28 PCs
    '''
    weights = pd.DataFrame()

    for i in range(len(pca.components_)):
        weights["weights_{}".format(i)] = \
        pca.components_[i] / sum(pca.components_[i])

    weights = weights.values.T
    return weights

weights=PCWeights()

weights = PCWeights()
NumComponents=3

topPortfolios = pd.DataFrame(weights[:NumComponents], columns=dataset.columns)
topPortfolios.index = [f'Principal Component {i}' \
for i in range(1, NumComponents+1)]

axes = topPortfolios.T.plot.bar(subplots=True, legend=False, figsize=(14, 10))
plt.subplots_adjust(hspace=0.35)
axes[0].set_ylim(0, .2);
```

```
Output
```



```
    pd.DataFrame(pca.components_[0:3].T).plot(style= ['s-','o-','^-'], \
                                legend=False, title="Principal Component")
```

```
Output
```



By plotting the components of the eigenvectors we can make the following interpretation:

Principal Component 1
: This eigenvector has all positive values, with all tenors weighted in the same direction. This means that the first principal component reflects movements that cause all maturities to move in the same direction, corresponding to *directional movements* in the yield curve. These are movements that shift the entire yield curve up or down.

Principal Component 2
: The second eigenvector has the first half of the components negative and the second half positive. Treasury rates on the short end (long end) of the curve are weighted positively (negatively). This means that the second principal component reflects movements that cause the short end to go in one direction and the long end in the other. Consequently, it represents *slope movements* in the yield curve.

Principal Component 3
: The third eigenvector has the first third of the components negative, the second third positive, and the last third negative. This means that the third principal component reflects movements that cause the short and long end to go in one direction, and the middle to go in the other, resulting in *curvature movements* of the yield curve.

5.2.3. Reconstructing the curve using principal components. One of the key features of PCA is the ability to reconstruct the initial dataset using the outputs of PCA. Using simple matrix reconstruction, we can generate a near exact replica of the initial data:

```
pca.transform(rescaledDataset)[:, :2]
```

`Output`

```
array([[ 4.97514826, -0.48514999],
       [ 5.03634891, -0.52005102],
       [ 5.14497849, -0.58385444],
       ...,
       [-1.82544584,  2.82360062],
       [-1.69938513,  2.6936174 ],
       [-1.73186029,  2.73073137]])
```

Mechanically, PCA is just a matrix multiplication:

$$Y = XW$$

where Y is the principal components, X is input data, and W is a matrix of coefficients, which we can use to recover the original matrix as per the equation below:

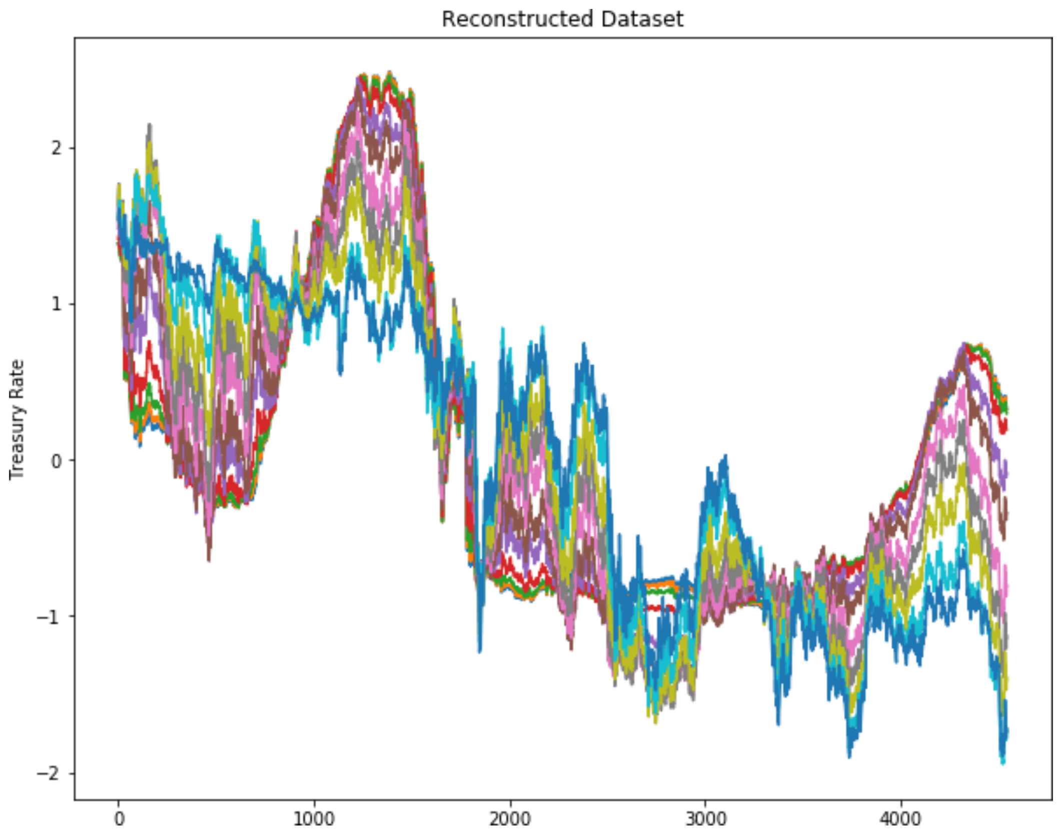
$$X = YW'$$

where $W'$is the inverse of the matrix of coefficients W.

```
nComp=3
reconst= pd.DataFrame(np.dot(pca.transform(rescaledDataset)[:, :nComp],\
pca.components_[:nComp,:]),columns=dataset.columns)
plt.figure(figsize=(10,8))
plt.plot(reconst)
plt.ylabel("Treasury Rate")
plt.title("Reconstructed Dataset")
plt.show()
```

This figure shows the replicated Treasury rate chart and demonstrates that, using just the first three principal components, we are able to replicate the original chart. Despite reducing the data from 11 dimensions to three, we still retain more than 99% of the information and can reproduce the original data easily. Additionally, we also have intuition around these three drivers of yield curve moments. Reducing the yield curve into fewer components means practictioners can focus on fewer factors that influence interest rates. For example, in order to hedge a portfolio, it may be sufficient to protect the portfolio against moves in the first three principal components only.

`Output`



Conclusion

In this case study, we introduced dimensionality reduction to break down the Treasury rate curve into fewer components. We saw that the principal components are quite intuitive for this case study. The first three principal components explain more than 99.5% of the variation and represent directional movements, slope movements, and curvature movements, respectively.

By using principal component analysis, analyzing the eigenvectors, and understanding the intuition behind them, we demonstrated how using dimensionality reduction led to fewer intuitive dimensions in the yield curve. Such dimensionality reduction of the yield curve can potentially lead to faster and more effective portfolio management, trading, hedging, and risk management.

Case Study 3: Bitcoin Trading: Enhancing Speed and Accuracy

As trading becomes more automated, traders will continue to seek to use as many features and technical indicators as they can to make their strategies more accurate and efficient. One of the many challenges in this is that adding more variables leads to ever more complexity, making it increasingly difficult to arrive at solid conclusions. Using dimensionality reduction techniques, we can compress many features and technical indicators into a few logical collections, while still maintaining a significant amount of the variance of the original data. This helps speed up model training and tuning. Additionally, it helps prevent overfitting by getting rid of correlated variables, which can ultimately cause more harm than good. Dimensionality reduction also enhances exploration and visualization of a dataset to understand grouping or relationships, an important task when building and continuously monitoring trading strategies.

In this case study, we will use dimensionality reduction to enhance "Case Study 3: Bitcoin Trading Strategy" on page 179 presented in Chapter 6. In this case study, we design a trading strategy for bitcoin that considers the relationship between the short-term and long-term prices to predict a buy or sell signal. We create several new intuitive, technical indicator features, including trend, volume, volatility, and momentum. We apply dimensionality reduction techniques on these features in order to achieve better results.

In this case study, we will focus on:

- Reducing the dimensions of a dataset to yield better and faster results for supervised learning.
- Using SVD and t-SNE to visualize data in lower dimensions.

Blueprint for Using Dimensionality Reduction to Enhance a Trading Strategy

1. Problem definition

Our goal in this case study is to use dimensionality reduction techniques to enhance an algorithmic trading strategy. The data and the variables used in this case study are the same as in "Case Study 3: Bitcoin Trading Strategy" on page 179. For reference, we are using intraday bitcoin price data, volume, and weighted bitcoin price from January 2012 to October 2017. Steps 3 and 4 presented in this case study use the same steps as the case study in Chapter 6. As such, these steps are condensed in this case study to avoid repetition.

2. Getting started—loading the data and Python packages

2.1. Loading the Python packages. The Python packages used for this case study are the same as those presented in the previous two case studies in this chapter.

3. Exploratory data analysis

Refer to "3. Exploratory data analysis" on page 181 for more details of this step.

4. Data preparation

We prepare the data for modeling in the following sections.

4.1. Data cleaning. We clean the data by filling the NAs with the last available values:

```
dataset[dataset.columns] = dataset[dataset.columns].ffill()
```

4.2. Preparing the data for classification. We attach the following label to each movement: 1 if the short-term price increases compared to the long-term price; 0 if the short-term price decreases compared to the long-term price. This label is assigned to a variable we will call *signal*, which is the predicted variable for this case study. Let us look at the data for prediction:

```
dataset.tail(5)
```

```
Output
```

	Open	High	Low	Close	Volume_(BTC)	Volume_(Currency)	Weighted_Price	short_mavg	long_mavg	signal
2841372	2190.49	2190.49	2181.37	2181.37	1.700	3723.785	2190.247	2179.259	2189.616	0.0
2841373	2190.50	2197.52	2186.17	2195.63	6.561	14402.812	2195.206	2181.622	2189.877	0.0
2841374	2195.62	2197.52	2191.52	2191.83	15.663	34361.024	2193.792	2183.605	2189.943	0.0
2841375	2195.82	2216.00	2195.82	2203.51	27.090	59913.493	2211.621	2187.018	2190.204	0.0
2841376	2201.70	2209.81	2196.98	2208.33	9.962	21972.309	2205.649	2190.712	2190.510	1.0

The dataset contains the signal column along with all other columns.

4.3. Feature engineering. In this step, we construct a dataset that contains the predictors that will be used to make the signal prediction. Using the bitcoin intraday price data, including daily open, high, low, close, and volume, we compute the following technical indicators:

- Moving Average
- Stochastic Oscillator %K and %D
- Relative Strength Index (RSI)
- Rate Of Change (ROC)
- Momentum (MOM)

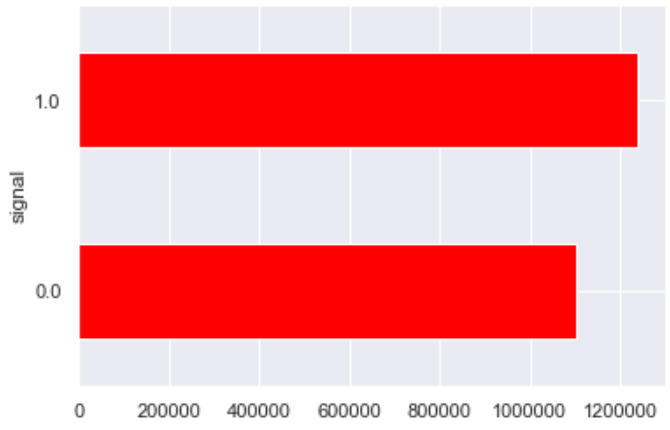
The code for the construction of all of the indicators, along with their descriptions, is presented in Chapter 6. The final dataset and the columns used are as follows:

	Close	Volume_(BTC)	Weighted_Price	signal	EMA10	EMA30	EMA200	ROC10	ROC30	MOM10	...	RSI200	%K10	%D10	%K30
2841372	2181.37	1.700	2190.247	0.0	2181.181	2182.376	2211.244	0.431	-0.649	8.42	...	46.613	56.447	73.774	47.883
2841373	2195.63	6.561	2195.206	0.0	2183.808	2183.231	2211.088	1.088	-0.062	23.63	...	47.638	93.687	71.712	93.805
2841374	2191.83	15.663	2193.792	0.0	2185.266	2183.786	2210.897	1.035	-0.235	19.83	...	47.395	80.995	77.043	81.350
2841375	2203.51	27.090	2211.621	0.0	2188.583	2185.058	2210.823	1.479	0.297	34.13	...	48.213	74.205	82.963	74.505
2841376	2208.33	9.962	2205.649	1.0	2192.174	2186.560	2210.798	1.626	0.516	36.94	...	48.545	82.810	79.337	84.344

4.4. Data visualization. Let us look at the distribution of the predicted variable:

```
fig = plt.figure()
plot = dataset.groupby(['signal']).size().plot(kind='barh', color='red')
plt.show()
```

Output



The predicted signal is "buy" 52.9% of the time.

5. Evaluate algorithms and models

Next, we perform dimensionality reduction and evaluate the models.

5.1. Train-test split. In this step, we split the dataset into training and test sets:

```
Y= subset_dataset["signal"]
X = subset_dataset.loc[:, dataset.columns != 'signal'] validation_size = 0.2
X_train, X_validation, Y_train, Y_validation = train_test_split\
(X, Y, test_size=validation_size, random_state=1)
```

We standardize the variables on the same scale before applying dimensionality reduction. Data standardization is performed using the following Python code:

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler().fit(X_train)
rescaledDataset = pd.DataFrame(scaler.fit_transform(X_train),\
columns = X_train.columns, index = X_train.index)
# summarize transformed data
X_train.dropna(how='any', inplace=True)
rescaledDataset.dropna(how='any', inplace=True)
rescaledDataset.head(2)
```

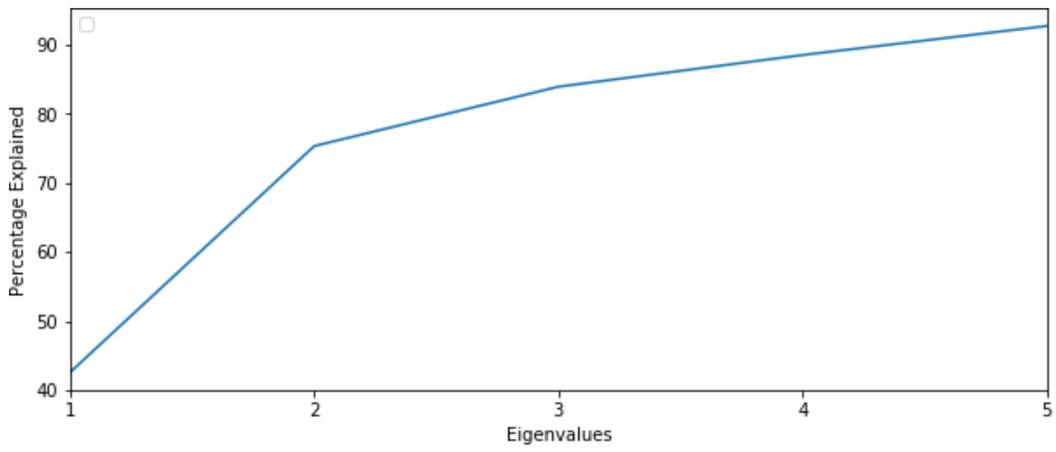
Output

	Close	Volume_(BTC)	Weighted_Price	EMA10	EMA30	EMA200	ROC10	ROC30	MOM10	MOM30	...	RSI200	%K10	%D10	%K30	%D30
2834071	1.072	-0.367	1.040	1.064	1.077	1.014	0.005	-0.159	0.009	-0.183	...	-0.325	1.322	0.427	-0.205	-0.412
2836517	-1.738	1.126	-1.714	-1.687	-1.653	-1.733	-0.533	-0.597	-0.066	-0.416	...	-0.465	-1.620	-0.511	-1.283	-0.970

5.2. Singular value decomposition (feature reduction). Here we will use SVD to perform PCA. Specifically, we are using the `TruncatedSVD` method in the sklearn package to transform the full dataset into a representation using the top five components:

```
ncomps = 5
svd = TruncatedSVD(n_components=ncomps)
svd_fit = svd.fit(rescaledDataset)
Y_pred = svd.fit_transform(rescaledDataset)
ax = pd.Series(svd_fit.explained_variance_ratio_.cumsum()).plot(kind='line', \
figsize=(10, 3))
ax.set_xlabel("Eigenvalues")
ax.set_ylabel("Percentage Explained")
print('Variance preserved by first 5 components == {:.2%}'.\
format(svd_fit.explained_variance_ratio_.cumsum()[-1]))
```

`Output`



Following the computation, we preserve 92.75% of the variance by using just five components rather than the full 25+ original features. This is a tremendously useful compression for the analysis and iterations of the model.

For convenience, we will create a Python dataframe specifically for these top five components:

```
dfsvd = pd.DataFrame(Y_pred, columns=['c{}'.format(c) for \
c in range(ncomps)], index=rescaledDataset.index)
print(dfsvd.shape)
dfsvd.head()
```

`Output`

```
(8000, 5)
```

	c0	c1	c2	c3	c4
2834071	–2.252	1.920	0.538	–0.019	–0.967
2836517	5.303	–1.689	–0.678	0.473	0.643
2833945	–2.315	–0.042	1.697	–1.704	1.672
2835048	–0.977	0.782	3.706	–0.697	0.057
2838804	2.115	–1.915	0.475	–0.174	–0.299

5.2.1. Basic visualization of reduced features.

Let us visualize the compressed dataset:

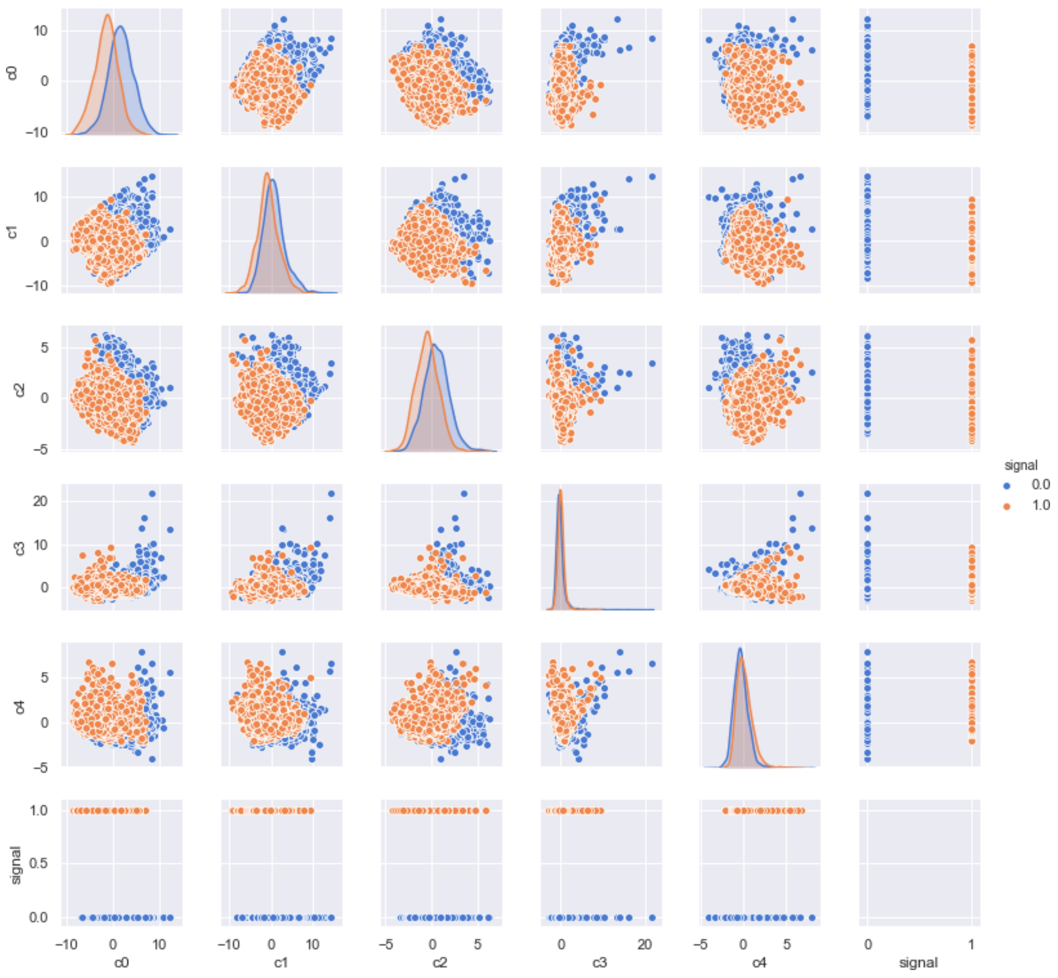
```
svdcols = [c for c in dfsvd.columns if c[0] == 'c']
```

Pairs-plots

Pairs-plots are a simple representation of a set of 2D scatterplots, with each component plotted against every other component. The data points are colored according to their signal classification:

```
plotdims = 5
ploteorows = 1
dfsvdplot = dfsvd[svdcols].iloc[:, :plotdims]
dfsvdplot['signal']=Y_train
ax = sns.pairplot(dfsvdplot.iloc[::ploteorows, :], hue='signal', size=1.8)
```

```
Output
```



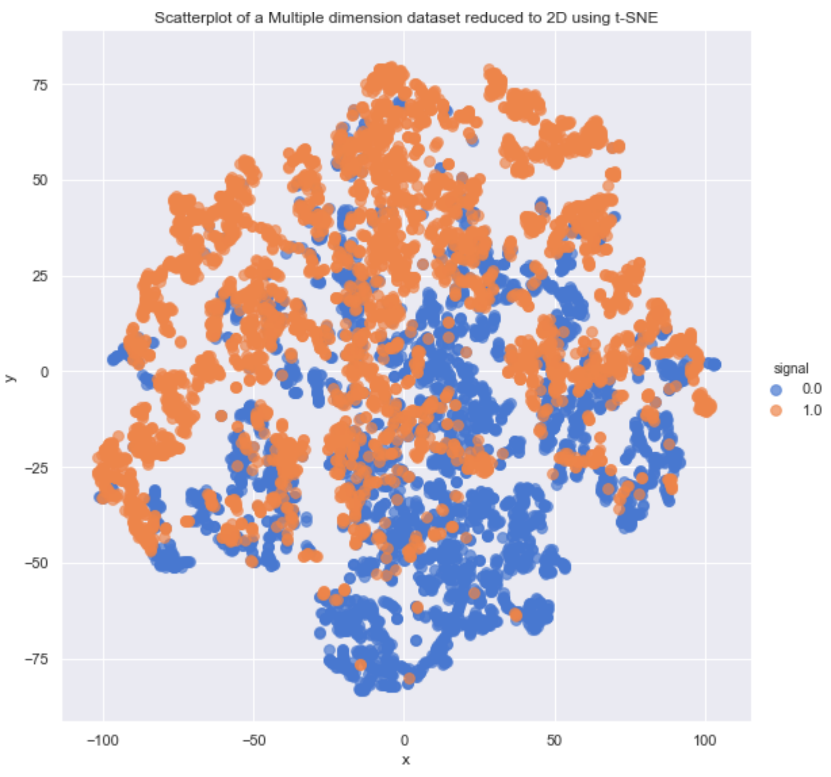
We can see that there is clear separation of the colored dots (full color version available on GitHub), meaning that data points from the same signal tend to cluster together. The separation is more distinct for the first components, with the characteristics of signal distributions growing more similar as you progress from the first to the fifth component. That said, the plot provides support for using all five components in our model.

5.3. t-SNE visualization. In this step, we implement t-SNE and look at the related visualization. We will use the basic implementation available in Scikit-learn:

```
tsne = TSNE(n_components=2, random_state=0)

Z = tsne.fit_transform(dfsvd[svdcols])
dftsne = pd.DataFrame(Z, columns=['x','y'], index=dfsvd.index)
```

```
dftsne['signal'] = Y_train

g = sns.lmplot('x', 'y', dftsne, hue='signal', fit_reg=False, size=8
                , scatter_kws={'alpha':0.7,'s':60})
```

`Output`



The plot shows us that there is a good degree of clustering for the trading signal. There is some overlap of the long and short signals, but they can be distinguished quite well using the reduced number of features.

5.4. Compare models with and without dimensionality reduction. In this step, we analyze the impact of the dimensionality reduction on the classification and the impact on the overall accuracy and computation time:

```
# test options for classification
scoring = 'accuracy'
```

5.4.1. Models. We first look at the time taken by the model without dimensionality reduction, where we have all the technical indicators:

```
import time
start_time = time.time()

# spot-check the algorithms
```

```
models =  RandomForestClassifier(n_jobs=-1)
cv_results_XTrain= cross_val_score(models, X_train, Y_train, cv=kfold, \
  scoring=scoring)
print("Time Without Dimensionality Reduction--- %s seconds ---" % \
(time.time() - start_time))
```

`Output`

```
Time Without Dimensionality Reduction
7.781347990036011 seconds
```

The total time taken without dimensionality reduction is around eight seconds. Let us look at the time it takes with dimensionality reduction, when only the five principal components from the truncated SVD are used:

```
start_time = time.time()
X_SVD= dfsvd[svdcols].iloc[:, :5]
cv_results_SVD = cross_val_score(models, X_SVD, Y_train, cv=kfold, \
  scoring=scoring)
print("Time with Dimensionality Reduction--- %s seconds ---" % \
(time.time() - start_time))
```

`Output`

```
Time with Dimensionality Reduction
2.281977653503418 seconds
```

The total time taken with dimensionality reduction is around two seconds—four times a reduction in time, which is a significant improvement. Let us investigate whether there is any decline in the accuracy when using the condensed dataset:

```
print("Result without dimensionality Reduction: %f (%f)" %\
 (cv_results_XTrain.mean(), cv_results_XTrain.std()))
print("Result with dimensionality Reduction: %f (%f)" %\
 (cv_results_SVD.mean(), cv_results_SVD.std()))
```

`Output`

```
Result without dimensionality Reduction: 0.936375 (0.010774)
Result with dimensionality Reduction: 0.887500 (0.012698)
```

Accuracy declines roughly 5%, from 93.6% to 88.7%. The improvement in speed has to be balanced against this loss in accuracy. Whether the loss in accuracy is acceptable likely depends on the problem. If this is a model that needs to be recalibrated very frequently, then a lower computation time will be essential, especially when handling large, high-velocity datasets. The improvement in the computation time does have other benefits, especially in the early stages of trading strategy development. It enables us to test a greater number of features (or technical indicators) in less time.

Conclusion

In this case study, we demonstrated the efficiency of dimensionality reduction and principal components analysis in reducing the number of dimensions in the context of a trading strategy. Through dimensionality reduction, we achieved a commensurate accuracy rate with a fourfold improvement in the modeling speed. In trading strategy development involving expansive datasets, such speed enhancements can lead to improvements for the entire process.

We demonstrated that both SVD and t-SNE yield reduced datasets that can easily be visualized for evaluating trading signal data. This allowed us to distinguish the long and short signals of this trading strategy in ways not possible with the original number of features.

Chapter Summary

The case studies presented in this chapter focused on understanding the concepts of the different dimensionality reduction methods, developing intuition around the principal components, and visualizing the condensed datasets.

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 dimensionality reduction–based problem in finance.

In the next chapter, we explore concepts and case studies for another type of unsupervised learning—clustering.

Exercises

1. Using dimensionality reduction, extract the different factors from the stocks within a different index and use them to build a trading strategy.
2. Pick any of the regression-based case studies in Chapter 5 and use dimensionality reduction to see whether there is any improvement in computation time. Explain the components using the factor loading and develop some high-level intuition of them.
3. For case study 3 presented in this chapter, perform factor loading of the principal components and understand the intuition of the different components.
4. Get the principal components of different currency pairs or different commodity prices. Identify the drivers of the primary principal components and link them to some intuitive macroeconomic variables.