CHAPTER 5

Supervised Learning: Regression (Including Time Series Models)

Supervised regression–based machine learning is a predictive form of modeling in which the goal is to model the relationship between a target and the predictor variable(s) in order to estimate a continuous set of possible outcomes. These are the most used machine learning models in finance.

One of the focus areas of analysts in financial institutions (and finance in general) is to predict investment opportunities, typically predictions of asset prices and asset returns. Supervised regression–based machine learning models are inherently suitable in this context. These models help investment and financial managers understand the properties of the predicted variable and its relationship with other variables, and help them identify significant factors that drive asset returns. This helps investors estimate return profiles, trading costs, technical and financial investment required in infrastructure, and thus ultimately the risk profile and profitability of a strategy or portfolio.

With the availability of large volumes of data and processing techniques, supervised regression–based machine learning isn't just limited to asset price prediction. These models are applied to a wide range of areas within finance, including portfolio management, insurance pricing, instrument pricing, hedging, and risk management.

In this chapter we cover three supervised regression–based case studies that span diverse areas, including asset price prediction, instrument pricing, and portfolio management. All of the case studies follow the standardized seven-step model development process presented in Chapter 2; those steps include defining the problem, loading the data, exploratory data analysis, data preparation, model evaluation, and

model tuning.[1] The case studies are designed not only to cover a diverse set of topics from the finance standpoint but also to cover multiple machine learning and modeling concepts, including models from basic linear regression to advanced deep learning that were presented in Chapter 4.

A substantial amount of asset modeling and prediction problems in the financial industry involve a time component and estimation of a continuous output. As such, it is also important to address *time series models*. In its broadest form, time series analysis is about inferring what has happened to a series of data points in the past and attempting to predict what will happen to it in the future. There have been a lot of comparisons and debates in academia and the industry regarding the differences between supervised regression and time series models. Most time series models are *parametric* (i.e., a known function is assumed to represent the data), while the majority of supervised regression models are *nonparametric*. Time series models primarily use historical data of the predicted variables for prediction, and supervised learning algorithms use *exogenous variables* as predictor variables.[2] However, supervised regression can embed the historical data of the predicted variable through a time-delay approach (covered later in this chapter), and a time series model (such as ARIMAX, also covered later in this chapter) can use exogenous variables for prediction. Hence, time series and supervised regression models are similar in the sense that both can use exogenous variables as well as historical data of the predicted variable for forecasting. In terms of the final output, both supervised regression and time series models estimate a continuous set of possible outcomes of a variable.

In Chapter 4, we covered the concepts of models that are common between supervised regression and supervised classification. Given that time series models are more closely aligned with supervised regression than supervised classification, we will go through the concepts of time series models separately in this chapter. We will also demonstrate how we can use time series models on financial data to predict future values. Comparison of time series models against the supervised regression models will be presented in the case studies. Additionally, some machine learning and deep learning models (such as LSTM) can be directly used for time series forecasting, and those will be discussed as well.

1 There may be reordering or renaming of the steps or substeps based on the appropriateness and intuitiveness of the steps/substeps.

2 An exogenous variable is one whose value is determined outside the model and imposed on the model.

In "Case Study 1: Stock Price Prediction" on page 95, we demonstrate one of the most popular prediction problems in finance, that of predicting stock returns. In addition to predicting future stock prices accurately, the purpose of this case study is to discuss the machine learning–based framework for general asset class price prediction in finance. In this case study we will discuss several machine learning and time series concepts, along with focusing on visualization and model tuning.

In "Case Study 2: Derivative Pricing" on page 114, we will delve into derivative pricing using supervised regression and show how to deploy machine learning techniques in the context of traditional quant problems. As compared to traditional derivative pricing models, machine learning techniques can lead to faster derivative pricing without relying on the several impractical assumptions. Efficient numerical computation using machine learning can be increasingly beneficial in areas such as financial risk management, where a trade-off between efficiency and accuracy is often inevitable.

In "Case Study 3: Investor Risk Tolerance and Robo-Advisors" on page 125, we illustrate supervised regression–based framework to estimate the risk tolerance of investors. In this case study, we build a robo-advisor dashboard in Python and implement this risk tolerance prediction model in the dashboard. We demonstrate how such models can lead to the automation of portfolio management processes, including the use of robo-advisors for investment management. The purpose is also to illustrate how machine learning can efficiently be used to overcome the problem of traditional risk tolerance profiling or risk tolerance questionnaires that suffer from several behavioral biases.

In "Case Study 4: Yield Curve Prediction" on page 141, we use a supervised regression–based framework to forecast different yield curve tenors simultaneously. We demonstrate how we can produce multiple tenors at the same time to model the yield curve using machine learning models.

In this chapter, we will learn about the following concepts related to supervised regression and time series techniques:

- Application and comparison of different time series and machine learning models.
- Interpretation of the models and results. Understanding the potential overfitting and underfitting and intuition behind linear versus nonlinear models.
- Performing data preparation and transformations to be used in machine learning models.
- Performing feature selection and engineering to improve model performance.
- Using data visualization and data exploration to understand outcomes.

- Algorithm tuning to improve model performance. Understanding, implementing, and tuning time series models such as ARIMA for prediction.
- Framing a problem statement related to portfolio management and behavioral finance in a regression-based machine learning framework.
- Understanding how deep learning–based models such as LSTM can be used for time series prediction.

The models used for supervised regression were presented in Chapters 3 and 4. Prior to the case studies, we will discuss time series models. We highly recommend readers turn to *Time Series Analysis and Its Applications*, 4th Edition, by Robert H. Shumway and David S. Stoffer (Springer) for a more in-depth understanding of time series concepts, and to *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*, 2nd Edition, by Aurélien Géron (O'Reilly) for more on concepts in supervised regression models.

This Chapter's Code Repository

A Python-based master template for supervised regression, a time series model template, and the Jupyter notebook for all case studies presented in this chapter are included in the folder *Chapter 5 - Sup. Learning - Regression and Time Series models* of the code repository for this book.

For any new supervised regression–based case study, use the common template from the code repository, modify the elements specific to the case study, and borrow the concepts and insights from the case studies presented in this chapter. The template also includes the implementation and tuning of the ARIMA and LSTM models.[3] The templates are designed to run on the cloud (i.e., Kaggle, Google Colab, and AWS). All the case studies have been designed on a uniform regression template.[4]

Time Series Models

A *time series* is a sequence of numbers that are ordered by a time index.

In this section we will cover the following aspects of time series models, which we further leverage in the case studies:

3 These models are discussed later in this chapter.

4 There may be reordering or renaming of the steps or substeps based on the appropriateness and intuitiveness of the steps/substeps.

- The components of a time series
- Autocorrelation and stationarity of time series
- Traditional time series models (e.g., ARIMA)
- Use of deep learning models for time series modeling
- Conversion of time series data for use in a supervised learning framework

Time Series Breakdown

A time series can be broken down into the following components:

Trend Component
: A trend is a consistent directional movement in a time series. These trends will be either *deterministic* or *stochastic*. The former allows us to provide an underlying rationale for the trend, while the latter is a random feature of a series that we will be unlikely to explain. Trends often appear in financial series, and many trading models use sophisticated trend identification algorithms.

Seasonal Component
: Many time series contain seasonal variation. This is particularly true in series representing business sales or climate levels. In quantitative finance we often see seasonal variation, particularly in series related to holiday seasons or annual temperature variation (such as natural gas).

We can write the components of a time series y_t as

$$y_t = S_t + T_t + R_t$$

where S_t is the seasonal component, T_t is the trend component, and R_t represents the remainder component of the time series not captured by seasonal or trend component.

The Python code for breaking down a time series (Y) into its component is as follows:

```
import statsmodels.api as sm
sm.tsa.seasonal_decompose(Y,freq=52).plot()
```

Figure 5-1 shows the time series broken down into trend, seasonality, and remainder components. Breaking down a time series into these components may help us better understand the time series and identify its behavior for better prediction.

The three time series components are shown separately in the bottom three panels. These components can be added together to reconstruct the actual time series shown in the top panel (shown as "observed"). Notice that the time series shows a trending component after 2017. Hence, the prediction model for this time series should incor-

porate the information regarding the trending behavior after 2017. In terms of seasonality there is some increase in the magnitude in the beginning of the calendar year. The residual component shown in the bottom panel is what is left over when the seasonal and trend components have been subtracted from the data. The residual component is mostly flat with some spikes and noise around 2018 and 2019. Also, each of the plots are on different scales, and the trend component has maximum range as shown by the scale on the plot.

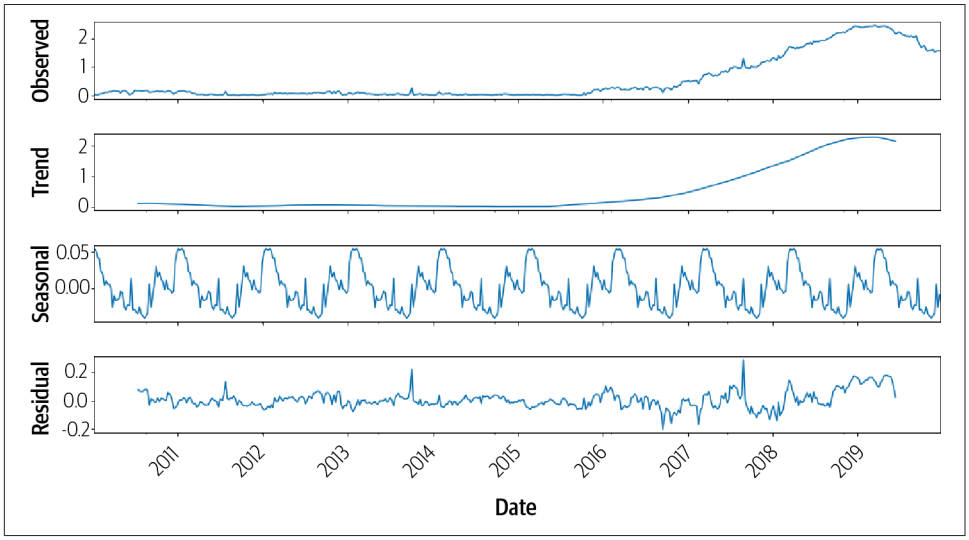


Figure 5-1. Time series components

Autocorrelation and Stationarity

When we are given one or more time series, it is relatively straightforward to decompose the time series into trend, seasonality, and residual components. However, there are other aspects that come into play when dealing with time series data, particularly in finance.

Autocorrelation

There are many situations in which consecutive elements of a time series exhibit correlation. That is, the behavior of sequential points in the series affect each other in a dependent manner. *Autocorrelation* is the similarity between observations as a function of the time lag between them. Such relationships can be modeled using an autoregression model. The term *autoregression* indicates that it is a regression of the variable against itself.

In an autoregression model, we forecast the variable of interest using a linear combination of past values of the variable.

Thus, an autoregressive model of order p can be written as

$$y_t = c + \phi_1 y_{t-1} + \phi_2 y_{t-2} + ...\phi_p y_{t-p} + \epsilon$$

where ϵ_t is white noise.[5] An autoregressive model is like a multiple regression but with lagged values of y_t as predictors. We refer to this as an AR(p) model, an autoregressive model of order p. Autoregressive models are remarkably flexible at handling a wide range of different time series patterns.

Stationarity

A time series is said to be stationary if its statistical properties do not change over time. Thus a time series with trend or with seasonality is not stationary, as the trend and seasonality will affect the value of the time series at different times. On the other hand, a white noise series is stationary, as it does not matter when you observe it; it should look similar at any point in time.

Figure 5-2 shows some examples of nonstationary series.

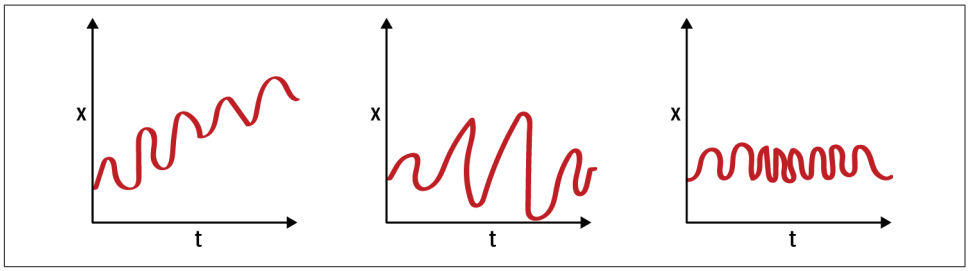


Figure 5-2. Nonstationary plots

In the first plot, we can clearly see that the mean varies (increases) with time, resulting in an upward trend. Thus this is a nonstationary series. For a series to be classified as stationary, it should not exhibit a trend. Moving on to the second plot, we certainly do not see a trend in the series, but the variance of the series is a function of time. A stationary series must have a constant variance; hence this series is a nonstationary series as well. In the third plot, the spread becomes closer as the time increases, which implies that the covariance is a function of time. The three examples shown in Figure 5-2 represent nonstationary time series. Now look at a fourth plot, as shown in Figure 5-3.

5 A white noise process is a random process of random variables that are uncorrelated and have a mean of zero and a finite variance.

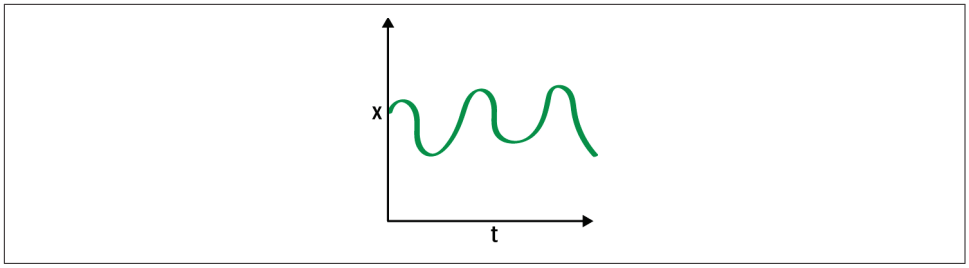


Figure 5-3. Stationary plot

In this case, the mean, variance, and covariance are constant with time. This is what a stationary time series looks like. Predicting future values using this fourth plot would be easier. Most statistical models require the series to be stationary to make effective and precise predictions.

The two major reasons behind nonstationarity of a time series are trend and seasonality, as shown in Figure 5-2. In order to use time series forecasting models, we generally convert any nonstationary series to a stationary series, making it easier to model since statistical properties don't change over time.

Differencing

Differencing is one of the methods used to make a time series stationary. In this method, we compute the difference of consecutive terms in the series. Differencing is typically performed to get rid of the varying mean. Mathematically, differencing can be written as:

$$y_t' = y_t - y_{t-1}$$

where y_t is the value at a time t.

When the differenced series is white noise, the original series is referred to as a nonstationary series of degree one.

Traditional Time Series Models (Including the ARIMA Model)

There are many ways to model a time series in order to make predictions. Most of the time series models aim at incorporating the trend, seasonality, and remainder components while addressing the autocorrelation and stationarity embedded in the time series. For example, the autoregressive (AR) model discussed in the previous section addresses the autocorrelation in the time series.

One of the most widely used models in time series forecasting is the ARIMA model.

ARIMA

If we combine stationarity with autoregression and a moving average model (discussed further on in this section), we obtain an ARIMA model. *ARIMA* is an acronym for AutoRegressive Integrated Moving Average, and it has the following components:

AR(p)
: It represents autoregression, i.e., regression of the time series onto itself, as discussed in the previous section, with an assumption that current series values depend on its previous values with some lag (or several lags). The maximum lag in the model is referred to as p.

I(d)
: It represents order of integration. It is simply the number of differences needed to make the series stationary.

MA(q)
: It represents moving average. Without going into detail, it models the error of the time series; again, the assumption is that current error depends on the previous with some lag, which is referred to as q.

The moving average equation is written as:

$$y_t = c + \epsilon_t + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2}$$

where, ϵ_t is white noise. We refer to this as an *MA(q)* model of order q.

Combining all the components, the full ARIMA model can be written as:

$$y_t^{'} = c + \phi_1 y_{t-1}^{'} + \cdots + \phi_p y_{t-p}^{'} + \theta_1 \varepsilon_{t-1} + \cdots + \theta_q \varepsilon_{t-q} + \varepsilon_t$$

where $y_t^{'}$ is the differenced series (it may have been differenced more than once). The predictors on the right-hand side include both lagged values of $y_t^{'}$ and lagged errors. We call this an ARIMA(p,d,q) model, where p is the order of the autoregressive part, d is the degree of first differencing involved, and q is the order of the moving average part. The same stationarity and invertibility conditions that are used for autoregressive and moving average models also apply to an ARIMA model.

The Python code to fit the ARIMA model of the order (1,0,0) is shown in the following:

```
from statsmodels.tsa.arima_model import ARIMA
model=ARIMA(endog=Y_train,order=[1,0,0])
```

The family of ARIMA models has several variants, and some of them are as follows:

ARIMAX
: ARIMA models with exogenous variables included. We will be using this model in case study 1.

SARIMA
: "S" in this model stands for seasonal, and this model is targeted at modeling the seasonality component embedded in the time series, along with other components.

VARMA
: This is the extension of the model to multivariate case, when there are many variables to be predicted simultaneously. We predict many variables simultaneously in "Case Study 4: Yield Curve Prediction" on page 141.

Deep Learning Approach to Time Series Modeling

The traditional time series models such as ARIMA are well understood and effective on many problems. However, these traditional methods also suffer from several limitations. Traditional time series models are linear functions, or simple transformations of linear functions, and they require manually diagnosed parameters, such as time dependence, and don't perform well with corrupt or missing data.

If we look at the advancements in the field of deep learning for time series prediction, we see that *recurrent neural network* (RNN) has gained increasing attention in recent years. These methods can identify structure and patterns such as nonlinearity, can seamlessly model problems with multiple input variables, and are relatively robust to missing data. The RNN models can retain state from one iteration to the next by using their own output as input for the next step. These deep learning models can be referred to as time series models, as they can make future predictions using the data points in the past, similar to traditional time series models such as ARIMA. Therefore, there are a wide range of applications in finance where these deep learning models can be leveraged. Let us look at the deep learning models for time series forecasting.

RNNs

Recurrent neural networks (RNNs) are called "recurrent" because they perform the same task for every element of a sequence, with the output being dependent on the previous computations. RNN models have a memory, which captures information about what has been calculated so far. As shown in Figure 5-4, a recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor.

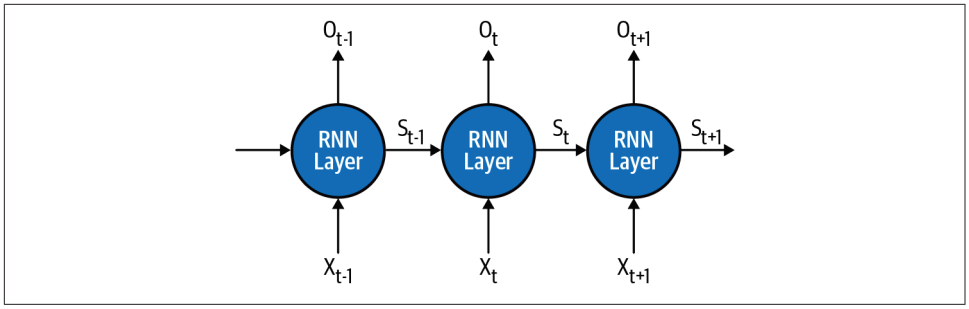


Figure 5-4. Recurrent Neural Network

In Figure 5-4:

- X_t is the input at time step t.
- O_t is the output at time step t.
- S_t is the hidden state at time step t. It's the memory of the network. It is calculated based on the previous hidden state and the input at the current step.

The main feature of an RNN is this hidden state, which captures some information about a sequence and uses it accordingly whenever needed.

Long short-term memory

Long short-term memory (LSTM) is a special kind of RNN explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically default behavior for an LSTM model.[6] These models are composed of a set of cells with features to memorize the sequence of data. These cells capture and store the data streams. Further, the cells interconnect one module of the past to another module of the present to convey information from several past time instants to the present one. Due to the use of gates in each cell, data in each cell can be disposed, filtered, or added for the next cells.

The *gates*, based on artificial neural network layers, enable the cells to optionally let data pass through or be disposed. Each layer yields numbers in the range of zero to one, depicting the amount of every segment of data that ought to be let through in each cell. More precisely, an estimation of zero value implies "let nothing pass through." An estimation of one indicates "let everything pass through." Three types of gates are involved in each LSTM, with the goal of controlling the state of each cell:

6 A detailed explanation of LSTM models can be found in this blog post by Christopher Olah .

Forget Gate
: Outputs a number between zero and one, where one shows "completely keep this" and zero implies "completely ignore this." This gate conditionally decides whether the past should be forgotten or preserved.

Input Gate
: Chooses which new data needs to be stored in the cell.

Output Gate
: Decides what will yield out of each cell. The yielded value will be based on the cell state along with the filtered and newly added data.

Keras wraps the efficient numerical computation libraries and functions and allows us to define and train LSTM neural network models in a few short lines of code. In the following code, LSTM module from `keras.layers` is used for implementing LSTM network. The network is trained with the variable `X_train_LSTM`. The network has a hidden layer with 50 LSTM blocks or neurons and an output layer that makes a single value prediction. Also refer to Chapter 3 for a more detailed description of all the terms (i.e., sequential, learning rate, momentum, epoch, and batch size).

A sample Python code for implementing an LSTM model in Keras is shown below:

```
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD
from keras.layers import LSTM

def create_LSTMmodel(learn_rate = 0.01, momentum=0):
       # create model
   model = Sequential()
   model.add(LSTM(50, input_shape=(X_train_LSTM.shape[1],\
      X_train_LSTM.shape[2])))
   #More number of cells can be added if needed
   model.add(Dense(1))
   optimizer = SGD(lr=learn_rate, momentum=momentum)
   model.compile(loss='mse', optimizer='adam')
   return model
LSTMModel = create_LSTMmodel(learn_rate = 0.01, momentum=0)
LSTMModel_fit = LSTMModel.fit(X_train_LSTM, Y_train_LSTM, validation_data=\
  (X_test_LSTM, Y_test_LSTM),epochs=330, batch_size=72, verbose=0, shuffle=False)
```

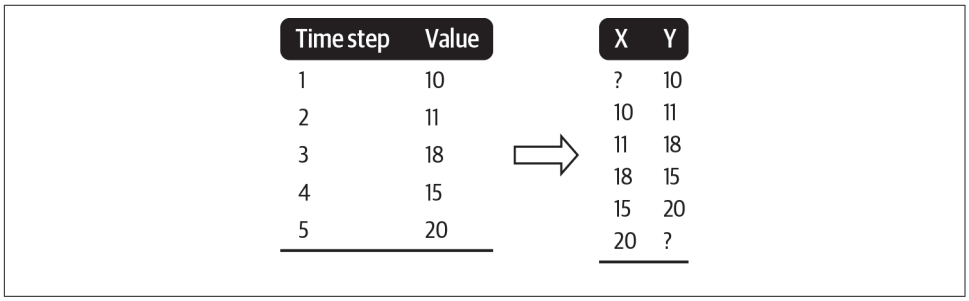
In terms of both learning and implementation, LSTM provides considerably more options for fine-tuning compared to ARIMA models. Although deep learning models have several advantages over traditional time series models, deep learning models are more complicated and difficult to train.[7]

7 An ARIMA model and a Keras-based LSTM model will be demonstrated in one of the case studies.

Modifying Time Series Data for Supervised Learning Models

A time series is a sequence of numbers that are ordered by a time index. Supervised learning is where we have input variables (*X*) and an output variable (*Y*). Given a sequence of numbers for a time series dataset, we can restructure the data into a set of predictor and predicted variables, just like in a supervised learning problem. We can do this by using previous time steps as input variables and using the next time step as the output variable. Let's make this concrete with an example.

We can restructure a time series shown in the left table in Figure 5-5 as a supervised learning problem by using the value at the previous time step to predict the value at the next time step. Once we've reorganized the time series dataset this way, the data would look like the table on the right.



Time step	Value
1	10
2	11
3	18
4	15
5	20

X	Y
?	10
10	11
11	18
18	15
15	20
20	?

Figure 5-5. Modifying time series for supervised learning models

We can see that the previous time step is the input (*X*) and the next time step is the output (*Y*) in our supervised learning problem. The order between the observations is preserved and must continue to be preserved when using this dataset to train a supervised model. We will delete the first and last row while training our supervised model as we don't have values for either *X* or *Y*.

In Python, the main function to help transform time series data into a supervised learning problem is the `shift()` function from the Pandas library. We will demonstrate this approach in the case studies. The use of prior time steps to predict the next time step is called the *sliding window*, *time delay*, or *lag* method.

Having discussed all the concepts of supervised learning and time series models, let us move to the case studies.

Case Study 1: Stock Price Prediction

One of the biggest challenges in finance is predicting stock prices. However, with the onset of recent advancements in machine learning applications, the field has been evolving to utilize nondeterministic solutions that learn what is going on in order to make more accurate predictions. Machine learning techniques naturally lend

themselves to stock price prediction based on historical data. Predictions can be made for a single time point ahead or for a set of future time points.

As a high-level overview, other than the historical price of the stock itself, the features that are generally useful for stock price prediction are as follows:

Correlated assets
: An organization depends on and interacts with many external factors, including its competitors, clients, the global economy, the geopolitical situation, fiscal and monetary policies, access to capital, and so on. Hence, its stock price may be correlated not only with the stock price of other companies but also with other assets such as commodities, FX, broad-based indices, or even fixed income securities.

Technical indicators
: A lot of investors follow technical indicators. Moving average, exponential moving average, and momentum are the most popular indicators.

Fundamental analysis
: Two primary data sources to glean features that can be used in fundamental analysis include:

  Performance reports
  : Annual and quarterly reports of companies can be used to extract or determine key metrics, such as ROE (Return on Equity) and P/E (Price-to-Earnings).

  News
  : News can indicate upcoming events that can potentially move the stock price in a certain direction.

In this case study, we will use various supervised learning–based models to predict the stock price of Microsoft using correlated assets and its own historical data. By the end of this case study, readers will be familiar with a general machine learning approach to stock prediction modeling, from gathering and cleaning data to building and tuning different models.

In this case study, we will focus on:

- Looking at various machine learning and time series models, ranging in complexity, that can be used to predict stock returns.
- Visualization of the data using different kinds of charts (i.e., density, correlation, scatterplot, etc.)
- Using deep learning (LSTM) models for time series forecasting.

- Implementation of the grid search for time series models (i.e., ARIMA model).
- Interpretation of the results and examining potential overfitting and underfitting of the data across the models.

Blueprint for Using Supervised Learning Models to Predict a Stock Price

1. Problem definition

In the supervised regression framework used for this case study, the weekly return of Microsoft stock is the predicted variable. We need to understand what affects Microsoft stock price and incorporate as much information into the model. Out of correlated assets, technical indicators, and fundamental analysis (discussed in the section before), we will focus on correlated assets as features in this case study.[8]

For this case study, other than the historical data of Microsoft, the independent variables used are the following potentially correlated assets:

Stocks
: IBM (IBM) and Alphabet (GOOGL)

Currency[9]
: USD/JPY and GBP/USD

Indices
: S&P 500, Dow Jones, and VIX

The dataset used for this case study is extracted from Yahoo Finance and the FRED website. In addition to predicting the stock price accurately, this case study will also demonstrate the infrastructure and framework for each step of time series and supervised regression–based modeling for stock price prediction. We will use the daily closing price of the last 10 years, from 2010 onward.

8 Refer to "Case Study 3: Bitcoin Trading Strategy" on page 179 presented in Chapter 6 and "Case Study 1: NLP and Sentiment Analysis–Based Trading Strategies" on page 362 presented in Chapter 10 to understand the usage of technical indicators and news-based fundamental analysis as features in the price prediction.

9 Equity markets have trading holidays, while currency markets do not. However, the alignment of the dates across all the time series is ensured before any modeling or analysis.

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 packages used for different purposes have been segregated in the Python code that follows. The details of most of these packages and functions were provided in Chapters 2 and 4. The use of these packages will be demonstrated in different steps of the model development process.

Function and modules for the supervised regression models

```
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Lasso
from sklearn.linear_model import ElasticNet
from sklearn.tree import DecisionTreeRegressor
from sklearn.neighbors import KNeighborsRegressor
from sklearn.svm import SVR
from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.ensemble import ExtraTreesRegressor
from sklearn.ensemble import AdaBoostRegressor
from sklearn.neural_network import MLPRegressor
```

Function and modules for data analysis and model evaluation

```
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_squared_error
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2, f_regression
```

Function and modules for deep learning models

```
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD
from keras.layers import LSTM
from keras.wrappers.scikit_learn import KerasRegressor
```

Function and modules for time series models

```
from statsmodels.tsa.arima_model import ARIMA
import statsmodels.api as sm
```

Function and modules for data preparation and visualization

```
# pandas, pandas_datareader, numpy and matplotlib
import numpy as np
import pandas as pd
import pandas_datareader.data as web
from matplotlib import pyplot
```

```
from pandas.plotting import scatter_matrix
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from pandas.plotting import scatter_matrix
from statsmodels.graphics.tsaplots import plot_acf
```

2.2. Loading the data. One of the most important steps in machine learning and predictive modeling is gathering good data. The following steps demonstrate the loading of data from the Yahoo Finance and FRED websites using the Pandas `DataReader` function:[10]

```
stk_tickers = ['MSFT', 'IBM', 'GOOGL']
ccy_tickers = ['DEXJPUS', 'DEXUSUK']
idx_tickers = ['SP500', 'DJIA', 'VIXCLS']

stk_data = web.DataReader(stk_tickers, 'yahoo')
ccy_data = web.DataReader(ccy_tickers, 'fred')
idx_data = web.DataReader(idx_tickers, 'fred')
```

Next, we define our dependent *(Y)* and independent *(X)* variables. The predicted variable is the weekly return of Microsoft (MSFT). The number of trading days in a week is assumed to be five, and we compute the return using five trading days. For independent variables we use the correlated assets and the historical return of MSFT at different frequencies.

The variables used as independent variables are lagged five-day return of stocks (IBM and GOOG), currencies (USD/JPY and GBP/USD), and indices (S&P 500, Dow Jones, and VIX), along with lagged 5-day, 15-day, 30-day and 60-day return of MSFT.

The lagged five-day variables embed the time series component by using a time-delay approach, where the lagged variable is included as one of the independent variables. This step is reframing the time series data into a supervised regression–based model framework.

```
return_period = 5
Y = np.log(stk_data.loc[:, ('Adj Close', 'MSFT')]).diff(return_period).\
shift(-return_period)
Y.name = Y.name[-1]+'_pred'

X1 = np.log(stk_data.loc[:, ('Adj Close', ('GOOGL', 'IBM'))]).diff(return_period)
X1.columns = X1.columns.droplevel()
X2 = np.log(ccy_data).diff(return_period)
X3 = np.log(idx_data).diff(return_period)

X4 = pd.concat([np.log(stk_data.loc[:, ('Adj Close', 'MSFT')]).diff(i) \
```

10 In different case studies across the book we will demonstrate loading the data through different sources (e.g., CSV, and external websites like quandl).

```
for i in [return_period, return_period*3,\
return_period*6, return_period*12]], axis=1).dropna()
X4.columns = ['MSFT_DT', 'MSFT_3DT', 'MSFT_6DT', 'MSFT_12DT']

X = pd.concat([X1, X2, X3, X4], axis=1)

dataset = pd.concat([Y, X], axis=1).dropna().iloc[::return_period, :]
Y = dataset.loc[:, Y.name]
X = dataset.loc[:, X.columns]
```

3. Exploratory data analysis

We will look at descriptive statistics, data visualization, and time series analysis in this section.

3.1. Descriptive statistics. Let's have a look at the dataset we have:

```
dataset.head()
```

```
Output
```

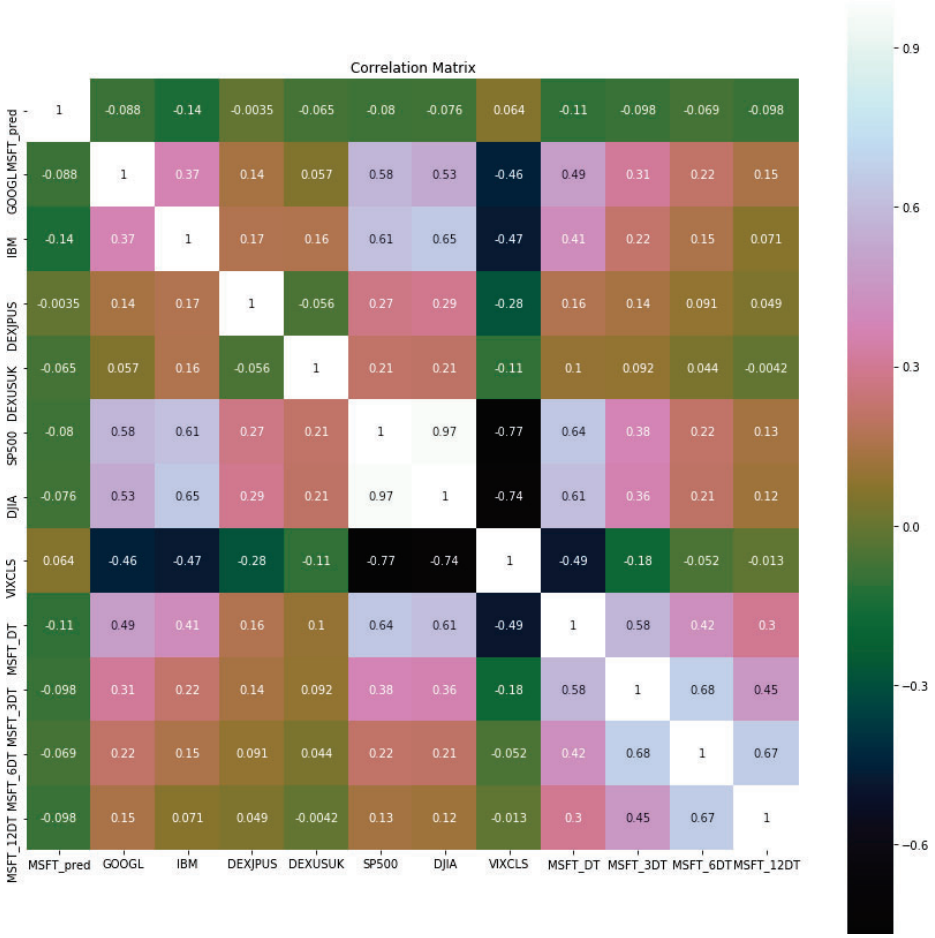
	MSFT_pred	GOOGL	IBM	DEXJPUS	DEXUSUK	SP500	DJIA	VIXCLS	MSFT_DT	MSFT_3DT	MSFT_6DT	MSFT_12DT
2010-03-31	0.021	1.741e-02	-0.002	1.630e-02	0.018	0.001	0.002	0.002	-0.012	0.011	0.024	-0.050
2010-04-08	0.031	6.522e-04	-0.005	-7.166e-03	-0.001	0.007	0.000	-0.058	0.021	0.010	0.044	-0.007
2010-04-16	0.009	-2.879e-02	0.014	-1.349e-02	0.002	-0.002	0.002	0.129	0.011	0.022	0.069	0.007
2010-04-23	-0.014	-9.424e-03	-0.005	2.309e-02	-0.002	0.021	0.017	-0.100	0.009	0.060	0.059	0.047
2010-04-30	-0.079	-3.604e-02	-0.008	6.369e-04	-0.004	-0.025	-0.018	0.283	-0.014	0.007	0.031	0.069

The variable MSFT_pred is the return of Microsoft stock and is the predicted variable. The dataset contains the lagged series of other correlated stocks, currencies, and indices. Additionally, it also consists of the lagged historical returns of MSFT.

3.2. Data visualization. The fastest way to learn more about the data is to visualize it. The visualization involves independently understanding each attribute of the dataset. We will look at the scatterplot and the correlation matrix. These plots give us a sense of the interdependence of the data. Correlation can be calculated and displayed for each pair of the variables by creating a correlation matrix. Hence, besides the relationship between independent and dependent variables, it also shows the correlation among the independent variables. This is useful to know because some machine learning algorithms like linear and logistic regression can have poor performance if there are highly correlated input variables in the data:

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

Output

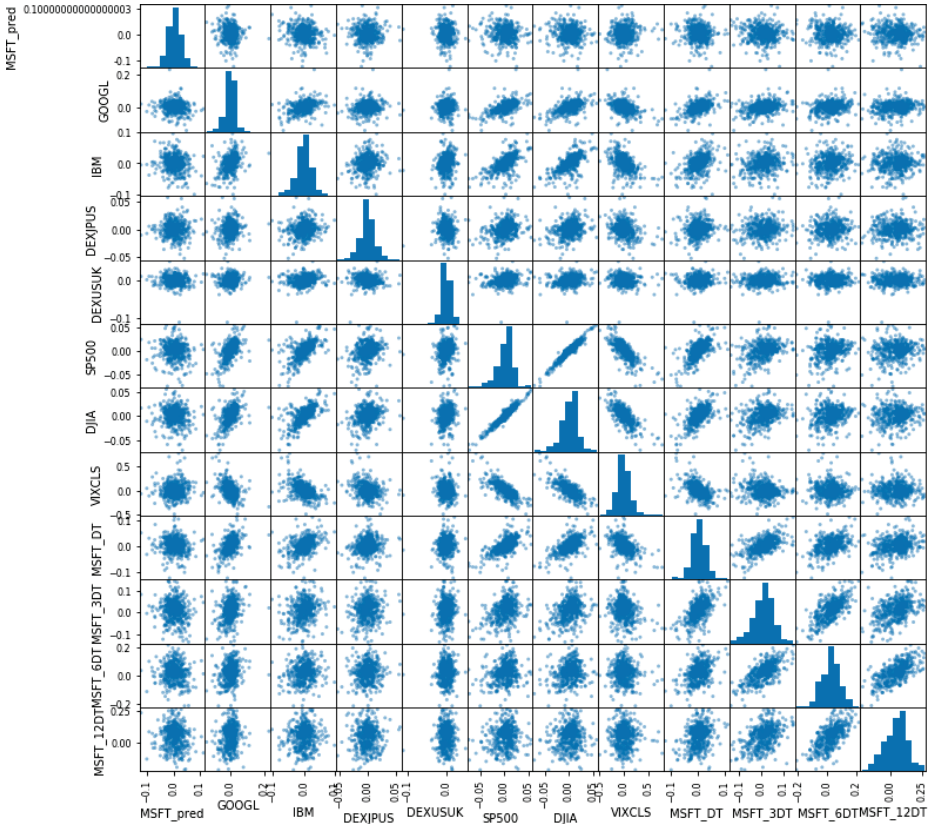


Looking at the correlation plot (full-size version available on GitHub), we see some correlation of the predicted variable with the lagged 5-day, 15-day, 30-day, and 60-day returns of MSFT. Also, we see a higher negative correlation of many asset returns versus VIX, which is intuitive.

Next, we can visualize the relationship between all the variables in the regression using the scatterplot matrix shown below:

```
pyplot.figure(figsize=(15,15))
scatter_matrix(dataset,figsize=(12,12))
pyplot.show()
```

Output

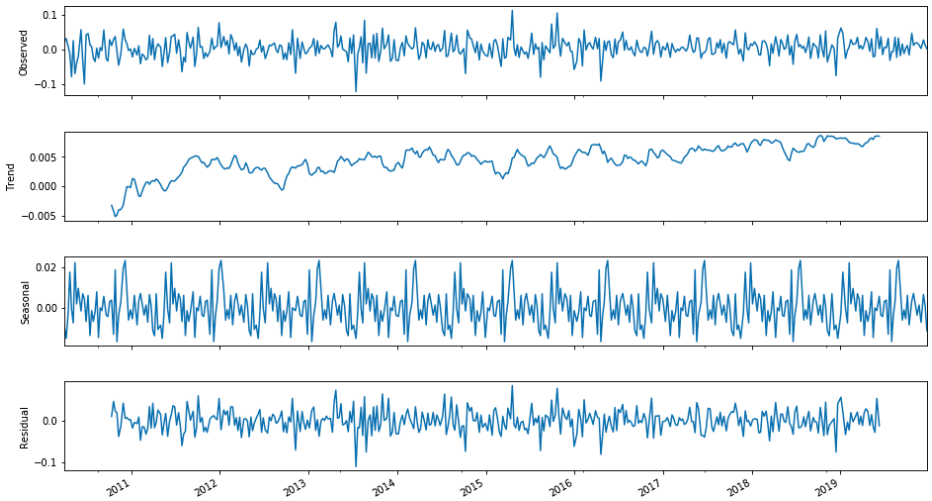


Looking at the scatterplot (full-size version available on GitHub), we see some linear relationship of the predicted variable with the lagged 15-day, 30-day, and 60-day returns of MSFT. Otherwise, we do not see any special relationship between our predicted variable and the features.

3.3. Time series analysis. Next, we delve into the time series analysis and look at the decomposition of the time series of the predicted variable into trend and seasonality components:

```
res = sm.tsa.seasonal_decompose(Y,freq=52)
fig = res.plot()
fig.set_figheight(8)
fig.set_figwidth(15)
pyplot.show()
```

```
Output
```



We can see that for MSFT there has been a general upward trend in the return series. This may be due to the large run-up of MSFT in the recent years, causing more positive weekly return data points than negative.[11] The trend may show up in the constant/bias terms in our models. The residual (or white noise) term is relatively small over the entire time series.

4. Data preparation

This step typically involves data processing, data cleaning, looking at feature importance, and performing feature reduction. The data obtained for this case study is relatively clean and doesn't require further processing. Feature reduction might be useful here, but given the relatively small number of variables considered, we will keep all of them as is. We will demonstrate data preparation in some of the subsequent case studies in detail.

5. Evaluate models

5.1. Train-test split and evaluation metrics. As described in Chapter 2, it is a good idea to partition the original dataset into a *training set* and a *test set*. The test set is a sample of the data that we hold back from our analysis and modeling. We use it right at the end of our project to confirm the performance of our final model. It is the final test that gives us confidence on our estimates of accuracy on unseen data. We will use

11 The time series is not the stock price but stock return, so the trend is mild compared to the stock price series.

80% of the dataset for modeling and use 20% for testing. With time series data, the sequence of values is important. So we do not distribute the dataset into training and test sets in random fashion, but we select an arbitrary split point in the ordered list of observations and create two new datasets:

```
validation_size = 0.2
train_size = int(len(X) * (1-validation_size))
X_train, X_test = X[0:train_size], X[train_size:len(X)]
Y_train, Y_test = Y[0:train_size], Y[train_size:len(X)]
```

5.2. Test options and evaluation metrics. To optimize the various hyperparameters of the models, we use ten-fold cross validation (CV) and recalculate the results ten times to account for the inherent randomness in some of the models and the CV process. We will evaluate algorithms using the mean squared error metric. This metric gives an idea of the performance of the supervised regression models. All these concepts, including cross validation and evaluation metrics, have been described in Chapter 4:

```
num_folds = 10
scoring = 'neg_mean_squared_error'
```

5.3. Compare models and algorithms. Now that we have completed the data loading and designed the test harness, we need to choose a model.

5.3.1. Machine learning models from Scikit-learn. In this step, the supervised regression models are implemented using the sklearn package:

```
Regression and tree regression algorithms
```

```
models = []
models.append(('LR', LinearRegression()))
models.append(('LASSO', Lasso()))
models.append(('EN', ElasticNet()))
models.append(('KNN', KNeighborsRegressor()))
models.append(('CART', DecisionTreeRegressor()))
models.append(('SVR', SVR()))
```

```
Neural network algorithms
```

```
models.append(('MLP', MLPRegressor()))
```

```
Ensemble models
```

```
# Boosting methods
models.append(('ABR', AdaBoostRegressor()))
models.append(('GBR', GradientBoostingRegressor()))
# Bagging methods
models.append(('RFR', RandomForestRegressor()))
models.append(('ETR', ExtraTreesRegressor()))
```

Once we have selected all the models, we loop over each of them. First, we run the *k*-fold analysis. Next, we run the model on the entire training and testing dataset.

All the algorithms use default tuning parameters. We will calculate the mean and standard deviation of the evaluation metric for each algorithm and collect the results for model comparison later:

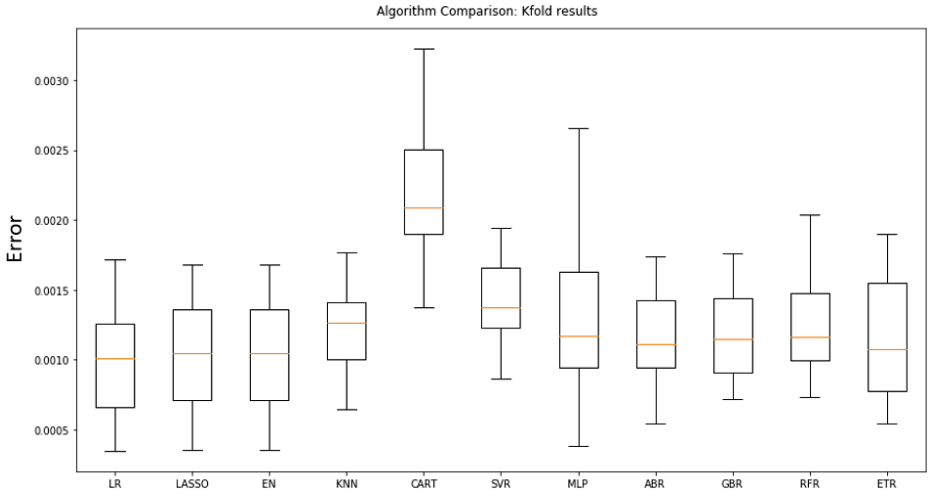
```
names = []
kfold_results = []
test_results = []
train_results = []
for name, model in models:
    names.append(name)
    ## k-fold analysis:
    kfold = KFold(n_splits=num_folds, random_state=seed)
    #converted mean squared error to positive. The lower the better
    cv_results = -1* cross_val_score(model, X_train, Y_train, cv=kfold, \
      scoring=scoring)
    kfold_results.append(cv_results)
    # Full Training period
    res = model.fit(X_train, Y_train)
    train_result = mean_squared_error(res.predict(X_train), Y_train)
    train_results.append(train_result)
    # Test results
    test_result = mean_squared_error(res.predict(X_test), Y_test)
    test_results.append(test_result)
```

Let's compare the algorithms by looking at the cross validation results:

```
Cross validation results
```

```
fig = pyplot.figure()
fig.suptitle('Algorithm Comparison: Kfold results')
ax = fig.add_subplot(111)
pyplot.boxplot(kfold_results)
ax.set_xticklabels(names)
fig.set_size_inches(15,8)
pyplot.show()
```

Output



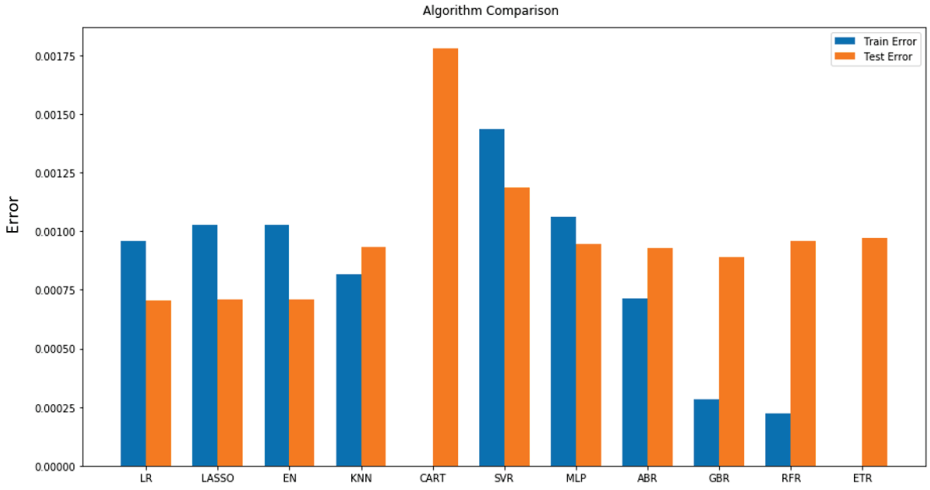
Although the results of a couple of the models look good, we see that the linear regression and the regularized regression including the lasso regression (LASSO) and elastic net (EN) seem to perform best. This indicates a strong linear relationship between the dependent and independent variables. Going back to the exploratory analysis, we saw a good correlation and linear relationship of the target variables with the different lagged MSFT variables.

Let us look at the errors of the test set as well:

Training and test error

```
    # compare algorithms
    fig = pyplot.figure()

    ind = np.arange(len(names))  # the x locations for the groups
    width = 0.35  # the width of the bars

    fig.suptitle('Algorithm Comparison')
    ax = fig.add_subplot(111)
    pyplot.bar(ind - width/2, train_results,  width=width, label='Train Error')
    pyplot.bar(ind + width/2, test_results, width=width, label='Test Error')
    fig.set_size_inches(15,8)
    pyplot.legend()
    ax.set_xticks(ind)
    ax.set_xticklabels(names)
    pyplot.show()
```

```
Output
```



Examining the training and test error, we still see a stronger performance from the linear models. Some of the algorithms, such as the decision tree regressor (CART), overfit on the training data and produced very high error on the test set. Ensemble models such as gradient boosting regression (GBR) and random forest regression (RFR) have low bias but high variance. We also see that the artificial neural network algorithm (shown as MLP in the chart) shows higher errors in both the training and test sets. This is perhaps due to the linear relationship of the variables not captured accurately by ANN, improper hyperparameters, or insufficient training of the model. Our original intuition from the cross validation results and the scatterplots also seem to demonstrate a better performance of linear models.

We now look at some of the time series and deep learning models that can be used. Once we are done creating these, we will compare their performance against that of the supervised regression–based models. Due to the nature of time series models, we are not able to run a *k*-fold analysis. We can still compare our results to the other models based on the full training and testing results.

5.3.2. Time series–based models: ARIMA and LSTM. The models used so far already embed the time series component by using a time-delay approach, where the lagged variable is included as one of the independent variables. However, for the time series–based models we do not need the lagged variables of MSFT as the independent variables. Hence, as a first step we remove MSFT's previous returns for these models. We use all other variables as the exogenous variables in these models.

Let us first prepare the dataset for ARIMA models by having only the correlated variables as exogenous variables:

```
X_train_ARIMA=X_train.loc[:, ['GOOGL', 'IBM', 'DEXJPUS', 'SP500', 'DJIA', \
'VIXCLS']]
X_test_ARIMA=X_test.loc[:, ['GOOGL', 'IBM', 'DEXJPUS', 'SP500', 'DJIA', \
'VIXCLS']]
tr_len = len(X_train_ARIMA)
te_len = len(X_test_ARIMA)
to_len = len (X)
```

We now configure the ARIMA model with the order *(1,0,0)* and use the independent variables as the exogenous variables in the model. The version of the ARIMA model where the exogenous variables are also used is known as the *ARIMAX* model, where "*X*" represents exogenous variables:

```
modelARIMA=ARIMA(endog=Y_train,exog=X_train_ARIMA,order=[1,0,0])
model_fit = modelARIMA.fit()
```

Now we fit the ARIMA model:

```
error_Training_ARIMA = mean_squared_error(Y_train, model_fit.fittedvalues)
predicted = model_fit.predict(start = tr_len -1 ,end = to_len -1, \
  exog = X_test_ARIMA)[1:]
error_Test_ARIMA = mean_squared_error(Y_test,predicted)
error_Test_ARIMA
```

`Output`

```
0.0005931919240399084
```

Error of this ARIMA model is reasonable.

Now let's prepare the dataset for the LSTM model. We need the data in the form of arrays of all the input variables and the output variables.

The logic behind the LSTM is that data is taken from the previous day (the data of all the other features for that day—correlated assets and the lagged variables of MSFT) and we try to predict the next day. Then we move the one-day window with one day and again predict the next day. We iterate like this over the whole dataset (of course in batches). The code below will create a dataset in which *X* is the set of independent variables at a given time (*t*) and *Y* is the target variable at the next time (*t* + *1*):

```
seq_len = 2 #Length of the seq for the LSTM

Y_train_LSTM, Y_test_LSTM = np.array(Y_train)[seq_len-1:], np.array(Y_test)
X_train_LSTM = np.zeros((X_train.shape[0]+1-seq_len, seq_len, X_train.shape[1]))
X_test_LSTM = np.zeros((X_test.shape[0], seq_len, X.shape[1]))
for i in range(seq_len):
    X_train_LSTM[:, i, :] = np.array(X_train)[i:X_train.shape[0]+i+1-seq_len, :]
    X_test_LSTM[:, i, :] = np.array(X)\
    [X_train.shape[0]+i-1:X.shape[0]+i+1-seq_len, :]
```

In the next step, we create the LSTM architecture. As we can see, the input of the LSTM is in `X_train_LSTM`, which goes into 50 hidden units in the LSTM layer and then is transformed to a single output—the stock return value. The hyperparameters

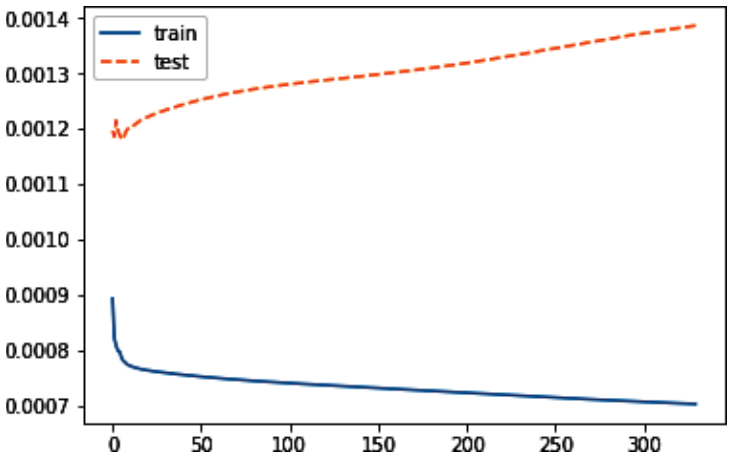
(i.e., learning rate, optimizer, activation function, etc.) were discussed in Chapter 3 of the book:

```
# LSTM Network
def create_LSTMmodel(learn_rate = 0.01, momentum=0):
        # create model
    model = Sequential()
    model.add(LSTM(50, input_shape=(X_train_LSTM.shape[1],\
      X_train_LSTM.shape[2])))
    #More cells can be added if needed
    model.add(Dense(1))
    optimizer = SGD(lr=learn_rate, momentum=momentum)
    model.compile(loss='mse', optimizer='adam')
    return model
LSTMModel = create_LSTMmodel(learn_rate = 0.01, momentum=0)
LSTMModel_fit = LSTMModel.fit(X_train_LSTM, Y_train_LSTM, \
  validation_data=(X_test_LSTM, Y_test_LSTM),\
  epochs=330, batch_size=72, verbose=0, shuffle=False)
```

Now we fit the LSTM model with the data and look at the change in the model performance metric over time simultaneously in the training set and the test set:

```
pyplot.plot(LSTMModel_fit.history['loss'], label='train', )
pyplot.plot(LSTMModel_fit.history['val_loss'], '--',label='test',)
pyplot.legend()
pyplot.show()
```

`Output`

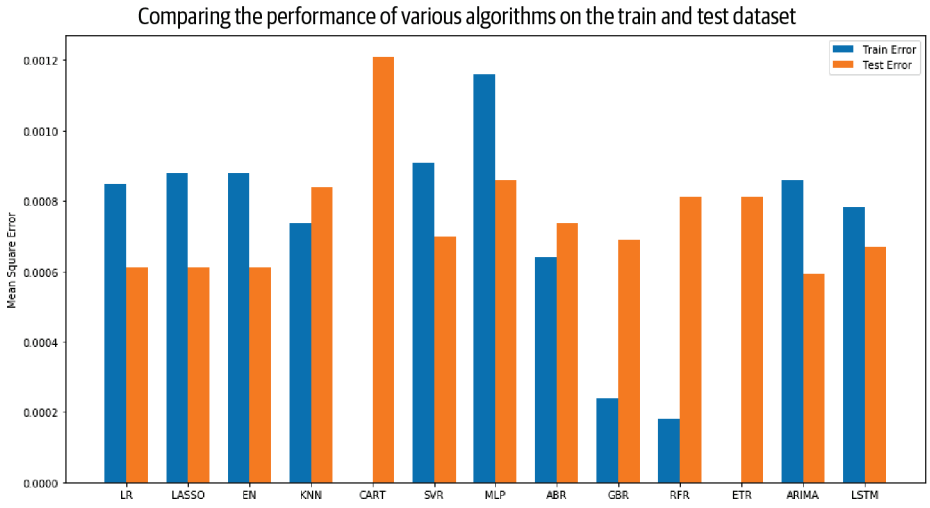



```
error_Training_LSTM = mean_squared_error(Y_train_LSTM,\
  LSTMModel.predict(X_train_LSTM))
predicted = LSTMModel.predict(X_test_LSTM)
error_Test_LSTM = mean_squared_error(Y_test,predicted)
```

Now, in order to compare the time series and the deep learning models, we append the result of these models to the results of the supervised regression–based models:

```
test_results.append(error_Test_ARIMA)
test_results.append(error_Test_LSTM)

train_results.append(error_Training_ARIMA)
train_results.append(error_Training_LSTM)

names.append("ARIMA")
names.append("LSTM")
```

`Output`



Looking at the chart, we find the time series–based ARIMA model comparable to the linear supervised regression models: linear regression (LR), lasso regression (LASSO), and elastic net (EN). This can primarily be due to the strong linear relationship as discussed before. The LSTM model performs decently; however, the ARIMA model outperforms the LSTM model in the test set. Hence, we select the ARIMA model for model tuning.

6. Model tuning and grid search

Let us perform the model tuning of the ARIMA model.

Model Tuning for the Supervised Learning or Time Series Models

The detailed implementation of grid search for all the supervised learning–based models, along with the ARIMA and LSTM models, is provided in the Regression-Master template under the GitHub repository for this book. For the grid search of the ARIMA and LSTM models, refer to the "ARIMA and LSTM Grid Search" section of the Regression-Master template.

The ARIMA model is generally represented as ARIMA*(p,d,q)* model, where *p* is the order of the autoregressive part, *d* is the degree of first differencing involved, and *q* is the order of the moving average part. The order of the ARIMA model was set to *(1,0,0)*. So we perform a grid search with different *p*, *d*, and *q* combinations in the ARIMA model's order and select the combination that minimizes the fitting error:

```
def evaluate_arima_model(arima_order):
    #predicted = list()
    modelARIMA=ARIMA(endog=Y_train,exog=X_train_ARIMA,order=arima_order)
    model_fit = modelARIMA.fit()
    error = mean_squared_error(Y_train, model_fit.fittedvalues)
    return error

# evaluate combinations of p, d and q values for an ARIMA model
def evaluate_models(p_values, d_values, q_values):
    best_score, best_cfg = float("inf"), None
    for p in p_values:
        for d in d_values:
            for q in q_values:
                order = (p,d,q)
                try:
                    mse = evaluate_arima_model(order)
                    if mse < best_score:
                        best_score, best_cfg = mse, order
                    print('ARIMA%s MSE=%.7f' % (order,mse))
                except:
                    continue
    print('Best ARIMA%s MSE=%.7f' % (best_cfg, best_score))

# evaluate parameters
p_values = [0, 1, 2]
d_values = range(0, 2)
q_values = range(0, 2)
warnings.filterwarnings("ignore")
evaluate_models(p_values, d_values, q_values)
```

Output

```
ARIMA(0, 0, 0) MSE=0.0009879
ARIMA(0, 0, 1) MSE=0.0009721
ARIMA(1, 0, 0) MSE=0.0009696
ARIMA(1, 0, 1) MSE=0.0009685
```

```
ARIMA(2, 0, 0) MSE=0.0009684
ARIMA(2, 0, 1) MSE=0.0009683
Best ARIMA(2, 0, 1) MSE=0.0009683
```

We see that the ARIMA model with the order *(2,0,1)* is the best performer out of all the combinations tested in the grid search, although there isn't a significant difference in the mean squared error (MSE) with other combinations. This means that the model with the autoregressive lag of two and moving average of one yields the best result. We should not forget the fact that there are other exogenous variables in the model that influence the order of the best ARIMA model as well.

7. Finalize the model

In the last step we will check the finalized model on the test set.

7.1. Results on the test dataset.

```
# prepare model
modelARIMA_tuned=ARIMA(endog=Y_train,exog=X_train_ARIMA,order=[2,0,1])
model_fit_tuned = modelARIMA_tuned.fit()

# estimate accuracy on validation set
predicted_tuned = model_fit.predict(start = tr_len -1 ,\
  end = to_len -1, exog = X_test_ARIMA)[1:]
print(mean_squared_error(Y_test,predicted_tuned))
```

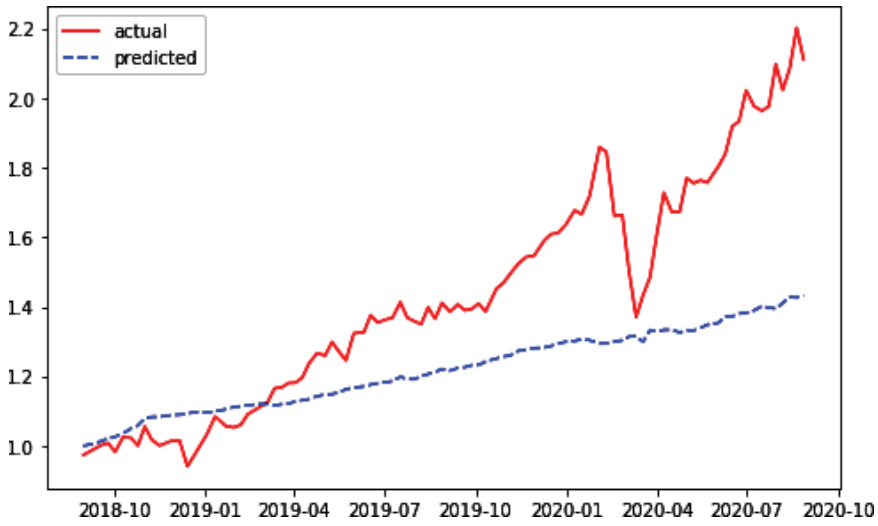
Output

```
0.0005970582461404503
```

The MSE of the model on the test set looks good and is actually less than that of the training set.

In the last step, we will visualize the output of the selected model and compare the modeled data against the actual data. In order to visualize the chart, we convert the return time series to a price time series. We also assume the price at the beginning of the test set as one for the sake of simplicity. Let us look at the plot of actual versus predicted data:

```
# plotting the actual data versus predicted data
predicted_tuned.index = Y_test.index
pyplot.plot(np.exp(Y_test).cumprod(), 'r', label='actual',)

# plotting t, a separately
pyplot.plot(np.exp(predicted_tuned).cumprod(), 'b--', label='predicted')
pyplot.legend()
pyplot.rcParams["figure.figsize"] = (8,5)
pyplot.show()
```



Looking at the chart, we clearly see the trend has been captured perfectly by the model. The predicted series is less volatile compared to the actual time series, and it aligns with the actual data for the first few months of the test set. A point to note is that the purpose of the model is to compute the next day's return given the data observed up to the present day, and not to predict the stock price several days in the future given the current data. Hence, a deviation from the actual data is expected as we move away from the beginning of the test set. The model seems to perform well for the first few months, with deviation from the actual data increasing six to seven months after the beginning of the test set.

Conclusion

We can conclude that simple models—linear regression, regularized regression (i.e., Lasso and elastic net)—along with the time series models, such as ARIMA, are promising modeling approaches for stock price prediction problems. This approach helps us deal with overfitting and underfitting, which are some of the key challenges in predicting problems in finance.

We should also note that we can use a wider set of indicators, such as P/E ratio, trading volume, technical indicators, or news data, which might lead to better results. We will demonstrate this in some of the future case studies in the book.

Overall, we created a supervised-regression and time series modeling framework that allows us to perform stock price prediction using historical data. This framework generates results to analyze risk and profitability before risking any capital.

Case Study 2: Derivative Pricing

In computational finance and risk management, several numerical methods (e.g., finite differences, fourier methods, and Monte Carlo simulation) are commonly used for the valuation of financial derivatives.

The *Black-Scholes formula* is probably one of the most widely cited and used models in derivative pricing. Numerous variations and extensions of this formula are used to price many kinds of financial derivatives. However, the model is based on several assumptions. It assumes a specific form of movement for the derivative price, namely a *Geometric Brownian Motion* (GBM). It also assumes a conditional payment at maturity of the option and economic constraints, such as no-arbitrage. Several other derivative pricing models have similarly impractical model assumptions. Finance practitioners are well aware that these assumptions are violated in practice, and prices from these models are further adjusted using practitioner judgment.

Another aspect of the many traditional derivative pricing models is model calibration, which is typically done not by historical asset prices but by means of derivative prices (i.e., by matching the market prices of heavily traded options to the derivative prices from the mathematical model). In the process of model calibration, thousands of derivative prices need to be determined in order to fit the parameters of the model, and the overall process is time consuming. Efficient numerical computation is increasingly important in financial risk management, especially when we deal with real-time risk management (e.g., high frequency trading). However, due to the requirement of a highly efficient computation, certain high-quality asset models and methodologies are discarded during model calibration of traditional derivative pricing models.

Machine learning can potentially be used to tackle these drawbacks related to impractical model assumptions and inefficient model calibration. Machine learning algorithms have the ability to tackle more nuances with very few theoretical assumptions and can be effectively used for derivative pricing, even in a world with frictions. With the advancements in hardware, we can train machine learning models on high performance CPUs, GPUs, and other specialized hardware to achieve a speed increase of several orders of magnitude as compared to the traditional derivative pricing models.

Additionally, market data is plentiful, so it is possible to train a machine learning algorithm to learn the function that is collectively generating derivative prices in the market. Machine learning models can capture subtle nonlinearities in the data that are not obtainable through other statistical approaches.

In this case study, we look at derivative pricing from a machine learning standpoint and use a supervised regression–based model to price an option from simulated data. The main idea here is to come up with a machine learning framework for derivative pricing. Achieving a machine learning model with high accuracy would mean that we

can leverage the efficient numerical calculation of machine learning for derivative pricing with fewer underlying model assumptions.

In this case study, we will focus on:

- Developing a machine learning–based framework for derivative pricing.
- Comparison of linear and nonlinear supervised regression models in the context of derivative pricing.

Blueprint for Developing a Machine Learning Model for Derivative Pricing

1. Problem definition

In the supervised regression framework we used for this case study, the predicted variable is the price of the option, and the predictor variables are the market data used as inputs to the Black-Scholes option pricing model.

The variables selected to estimate the market price of the option are stock price, strike price, time to expiration, volatility, interest rate, and dividend yield. The predicted variable for this case study was generated using random inputs and feeding them into the well-known Black-Scholes model.[12]

The price of a call option per the Black-Scholes option pricing model is defined in Equation 5-1.

Equation 5-1. Black-Scholes equation for call option

$$Se^{-q\tau}\Phi(d_1) - e^{-r\tau}K\Phi(d_2)$$

with

$$d_1 = \frac{\ln(S/K) + (r - q + \sigma^2/2)\tau}{\sigma\sqrt{\tau}}$$

and

12 The predicted variable, which is the option price, should ideally be directly obtained for the market. Given this case study is more for demonstration purposes, we use model-generated option price for the sake of convenience.

$$d_2 = \frac{\ln(S/K) + (r - q - \sigma^2/2)\tau}{\sigma\sqrt{\tau}} = d_1 - \sigma\sqrt{\tau}$$

where we have stock price S; strike price K; risk-free rate r; annual dividend yield q; time to maturity $\tau = T - t$ (represented as a unitless fraction of one year); and volatility σ.

To make the logic simpler, we define moneyness as $M = K / S$ and look at the prices in terms of per unit of current stock price. We also set q as 0.

This simplifies the formula to the following:

$$e^{-q\tau}\Phi\left(\frac{-\ln(M) + (r + \sigma^2/2)\tau}{\sigma\sqrt{\tau}}\right) - e^{-r\tau}M\Phi\left(\frac{-\ln(M) + (r - \sigma^2/2)\tau}{\sigma\sqrt{\tau}}\right)$$

Looking at the equation above, the parameters that feed into the Black-Scholes option pricing model are moneyness, risk-free rate, volatility, and time to maturity.

The parameter that plays the central role in derivative market is volatility, as it is directly related to the movement of the stock prices. With the increase in the volatility, the range of share price movements becomes much wider than that of a low volatility stock.

In the options market, there isn't a single volatility used to price all the options. This volatility depends on the option moneyness and time to maturity. In general, the volatility increases with higher time to maturity and with moneyness. This behavior is referred to as volatility smile/skew. We often derive the volatility from the price of the options existing in the market, and this volatility is referred to as "implied" volatility. In this exercise, we assume the structure of the volatility surface and use function in Equation 5-2, where volatility depends on the option moneyness and time to maturity to generate the option volatility surface.

Equation 5-2. Equation for vloatility

$$\sigma(M, \tau) = \sigma_0 + \alpha\tau + \beta(M - 1)^2$$

2. Getting started—loading the data and Python packages

2.1. Loading the Python packages. The loading of Python packages is similar to case study 1 in this chapter. Please refer to the Jupyter notebook of this case study for more details.

2.2. Defining functions and parameters. To generate the dataset, we need to simulate the input parameters and then create the predicted variable.

As a first step we define the constant parameters. The constant parameters required for the volatility surface are defined below. These parameters are not expected to have a significant impact on the option price; therefore, these parameters are set to some meaningful values:

```
true_alpha = 0.1
true_beta = 0.1
true_sigma0 = 0.2
```

The risk-free rate, which is an input to the Black-Scholes option pricing model, is defined as follows:

```
risk_free_rate = 0.05
```

Volatility and option pricing functions. In this step we define the function to compute the volatility and price of a call option as per Equations 5-1 and 5-2:

```
def option_vol_from_surface(moneyness, time_to_maturity):
    return true_sigma0 + true_alpha * time_to_maturity +\
     true_beta * np.square(moneyness - 1)

def call_option_price(moneyness, time_to_maturity, option_vol):
    d1=(np.log(1/moneyness)+(risk_free_rate+np.square(option_vol))*\
    time_to_maturity)/ (option_vol*np.sqrt(time_to_maturity))
    d2=(np.log(1/moneyness)+(risk_free_rate-np.square(option_vol))*\
    time_to_maturity)/(option_vol*np.sqrt(time_to_maturity))
    N_d1 = norm.cdf(d1)
    N_d2 = norm.cdf(d2)

    return N_d1 - moneyness * np.exp(-risk_free_rate*time_to_maturity) * N_d2
```

2.3. Data generation. We generate the input and output variables in the following steps:

- Time to maturity (*Ts*) is generated using the `np.random.random` function, which generates a uniform random variable between zero and one.
- Moneyness (*Ks*) is generated using the `np.random.randn` function, which generates a normally distributed random variable. The random number multiplied by 0.25 generates the deviation of strike from spot price,[13] and the overall equation ensures that the moneyness is greater than zero.
- Volatility (*sigma*) is generated as a function of time to maturity and moneyness using Equation 5-2.
- The option price is generated using Equation 5-1 for the Black-Scholes option price.

13 When the spot price is equal to the strike price, at-the-money option.

In total we generate 10,000 data points (*N*):

```
N = 10000

Ks = 1+0.25*np.random.randn(N)
Ts = np.random.random(N)
Sigmas = np.array([option_vol_from_surface(k,t) for k,t in zip(Ks,Ts)])
Ps = np.array([call_option_price(k,t,sig) for k,t,sig in zip(Ks,Ts,Sigmas)])
```

Now we create the variables for predicted and predictor variables:

```
Y = Ps
X = np.concatenate([Ks.reshape(-1,1), Ts.reshape(-1,1), Sigmas.reshape(-1,1)], \
axis=1)

dataset = pd.DataFrame(np.concatenate([Y.reshape(-1,1), X], axis=1),
                       columns=['Price', 'Moneyness', 'Time', 'Vol'])
```

3. Exploratory data analysis

Let's have a look at the dataset we have.

3.1. Descriptive statistics.

```
dataset.head()
```

`Output`

	Price	Moneyness	Time	Vol
0	1.390e-01	0.898	0.221	0.223
1	3.814e-06	1.223	0.052	0.210
2	1.409e-01	0.969	0.391	0.239
3	1.984e-01	0.950	0.628	0.263
4	2.495e-01	0.914	0.810	0.282

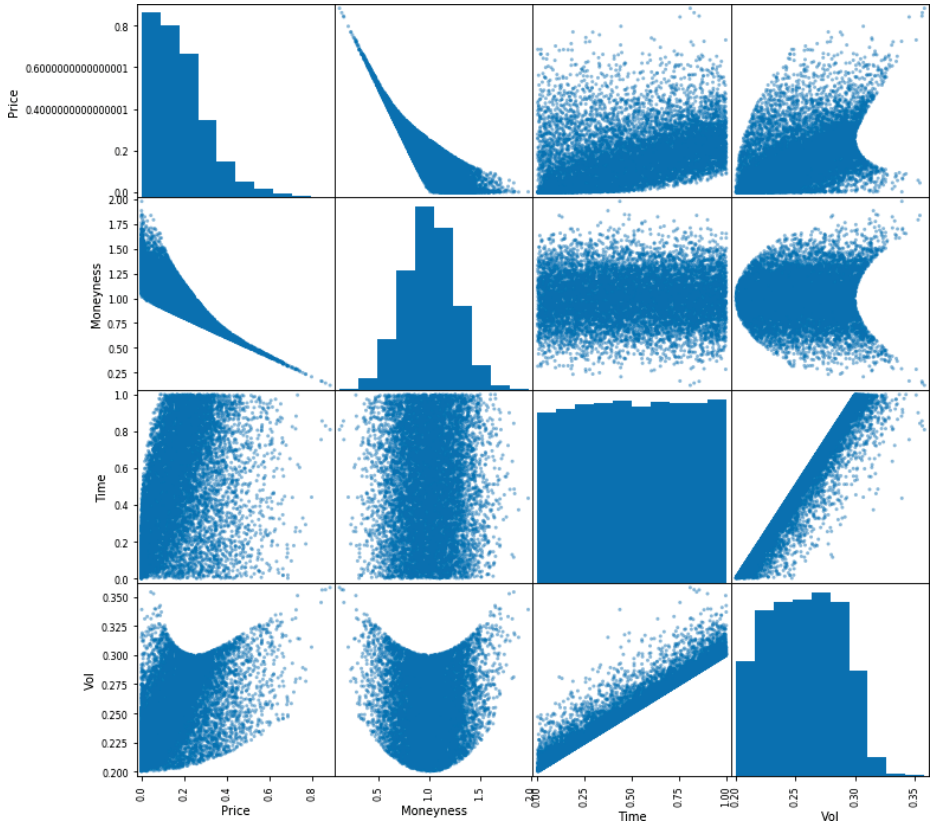
The dataset contains *price*—which is the price of the option and is the predicted variable—along with *moneyness* (the ratio of strike and spot price), *time to maturity*, and *volatility*, which are the features in the model.

3.2. Data visualization. In this step we look at scatterplot to understand the interaction between different variables:[14]

```
pyplot.figure(figsize=(15,15))
scatter_matrix(dataset,figsize=(12,12))
pyplot.show()
```

14 Refer to the Jupyter notebook of this case study to go through other charts such as histogram plot and correlation plot.

Output



The scatterplot reveals very interesting dependencies and relationships between the variables. Let us look at the first row of the chart to see the relationship of price to different variables. We observe that as moneyness decreases (i.e., strike price decreases as compared to the stock price), there is an increase in the price, which is in line with the rationale described in the previous section. Looking at the price versus time to maturity, we see an increase in the option price. The price versus volatility chart also shows an increase in the price with the volatility. However, option price seems to exhibit a nonlinear relationship with most of the variables. This means that we expect our nonlinear models to do a better job than our linear models.

Another interesting relationship is between volatility and strike. The more we deviate from the moneyness of one, the higher the volatility we observe. This behavior is shown due to the volatility function we defined before and illustrates the volatility smile/skew.

4. Data preparation and analysis

We performed most of the data preparation steps (i.e., getting the dependent and independent variables) in the preceding sections. In this step we look at the feature importance.

4.1. Univariate feature selection. We start by looking at each feature individually and, using the single variable regression fit as the criteria, look at the most important variables:

```
bestfeatures = SelectKBest(k='all', score_func=f_regression)
fit = bestfeatures.fit(X,Y)
dfscores = pd.DataFrame(fit.scores_)
dfcolumns = pd.DataFrame(['Moneyness', 'Time', 'Vol'])
#concat two dataframes for better visualization
featureScores = pd.concat([dfcolumns,dfscores],axis=1)
featureScores.columns = ['Specs','Score']  #naming the dataframe columns
featureScores.nlargest(10,'Score').set_index('Specs')
```

```
Output
```

```
Moneyness : 30282.309
Vol : 2407.757
Time : 1597.452
```

We observe that the moneyness is the most important variable for the option price, followed by volatility and time to maturity. Given there are only three predictor variables, we retain all the variables for modeling.

5. Evaluate models

5.1. Train-test split and evaluation metrics. First, we separate the training set and test set:

```
validation_size = 0.2

train_size = int(len(X) * (1-validation_size))
X_train, X_test = X[0:train_size], X[train_size:len(X)]
Y_train, Y_test = Y[0:train_size], Y[train_size:len(X)]
```

We use the prebuilt sklearn models to run a *k*-fold analysis on our training data. We then train the model on the full training data and use it for prediction of the test data. We will evaluate algorithms using the mean squared error metric. The parameters for the *k*-fold analysis and evaluation metrics are defined as follows:

```
num_folds = 10
seed = 7
scoring = 'neg_mean_squared_error'
```

5.2. Compare models and algorithms. Now that we have completed the data loading and have designed the test harness, we need to choose a model out of the suite of the supervised regression models.

`Linear models and regression trees`

```
models = []
models.append(('LR', LinearRegression()))
models.append(('KNN', KNeighborsRegressor()))
models.append(('CART', DecisionTreeRegressor()))
models.append(('SVR', SVR()))
```

`Artificial neural network`

```
models.append(('MLP', MLPRegressor()))
```

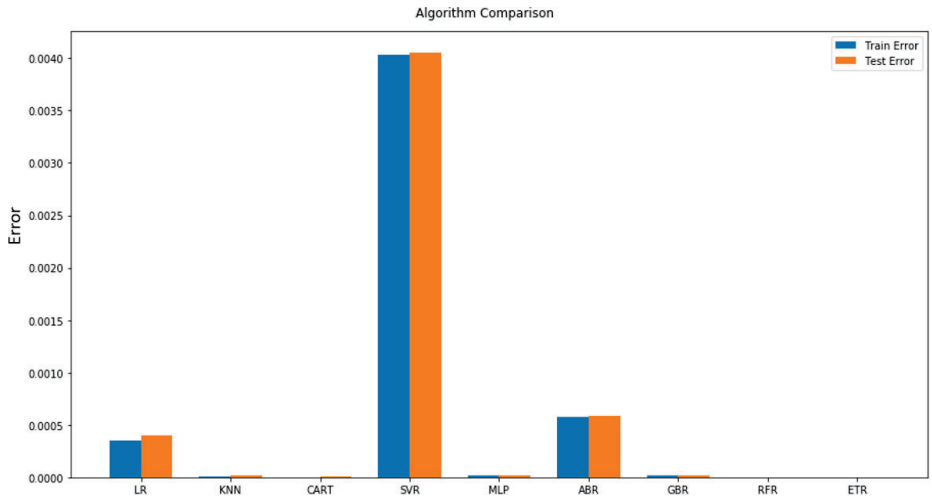
`Boosting and bagging methods`

```
# Boosting methods
models.append(('ABR', AdaBoostRegressor()))
models.append(('GBR', GradientBoostingRegressor()))
# Bagging methods
models.append(('RFR', RandomForestRegressor()))
models.append(('ETR', ExtraTreesRegressor()))
```

Once we have selected all the models, we loop over each of them. First, we run the *k*-fold analysis. Next, we run the model on the entire training and testing dataset.

The algorithms use default tuning parameters. We will calculate the mean and standard deviation of error metric and save the results for use later.

`Output`



The Python code for the *k*-fold analysis step is similar to that used in case study 1. Readers can also refer to the Jupyter notebook of this case study in the code repository for more details. Let us look at the performance of the models in the training set.

We see clearly that the nonlinear models, including classification and regression tree (CART), ensemble models, and artificial neural network (represented by MLP in the chart above), perform a lot better that the linear algorithms. This is intuitive given the nonlinear relationships we observed in the scatterplot.

Artificial neural networks (ANN) have the natural ability to model any function with fast experimentation and deployment times (definition, training, testing, inference). ANN can effectively be used in complex derivative pricing situations. Hence, out of all the models with good performance, we choose ANN for further analysis.

6. Model tuning and finalizing the model

Determining the proper number of nodes for the middle layer of an ANN is more of an art than a science, as discussed in Chapter 3. Too many nodes in the middle layer, and thus too many connections, produce a neural network that memorizes the input data and lacks the ability to generalize. Therefore, increasing the number of nodes in the middle layer will improve performance on the training set, while decreasing the number of nodes in the middle layer will improve performance on a new dataset.

As discussed in Chapter 3, the ANN model has several other hyperparameters such as learning rate, momentum, activation function, number of epochs, and batch size. All these hyperparameters can be tuned during the grid search process. However, in this step, we stick to performing grid search on the number of hidden layers for the purpose of simplicity. The approach to perform grid search on other hyperparameters is the same as described in the following code snippet:

```
'''
hidden_layer_sizes : tuple, length = n_layers - 2, default (100,)
    The ith element represents the number of neurons in the ith
    hidden layer.
'''
param_grid={'hidden_layer_sizes': [(20,), (50,), (20,20), (20, 30, 20)]}
model = MLPRegressor()
kfold = KFold(n_splits=num_folds, random_state=seed)
grid = GridSearchCV(estimator=model, param_grid=param_grid, scoring=scoring, \
  cv=kfold)
grid_result = grid.fit(X_train, Y_train)
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

Output

```
Best: -0.000024 using {'hidden_layer_sizes': (20, 30, 20)}
-0.000580 (0.000601) with: {'hidden_layer_sizes': (20,)}
-0.000078 (0.000041) with: {'hidden_layer_sizes': (50,)}
-0.000090 (0.000140) with: {'hidden_layer_sizes': (20, 20)}
-0.000024 (0.000011) with: {'hidden_layer_sizes': (20, 30, 20)}
```

The best model has three layers, with 20, 30, and 20 nodes in each hidden layer, respectively. Hence, we prepare a model with this configuration and check its performance on the test set. This is a crucial step, because a greater number of layers may lead to overfitting and have poor performance in the test set.

```
# prepare model
model_tuned = MLPRegressor(hidden_layer_sizes=(20, 30, 20))
model_tuned.fit(X_train, Y_train)

# estimate accuracy on validation set
# transform the validation dataset
predictions = model_tuned.predict(X_test)
print(mean_squared_error(Y_test, predictions))
```

Output

```
3.08127276609567e-05
```

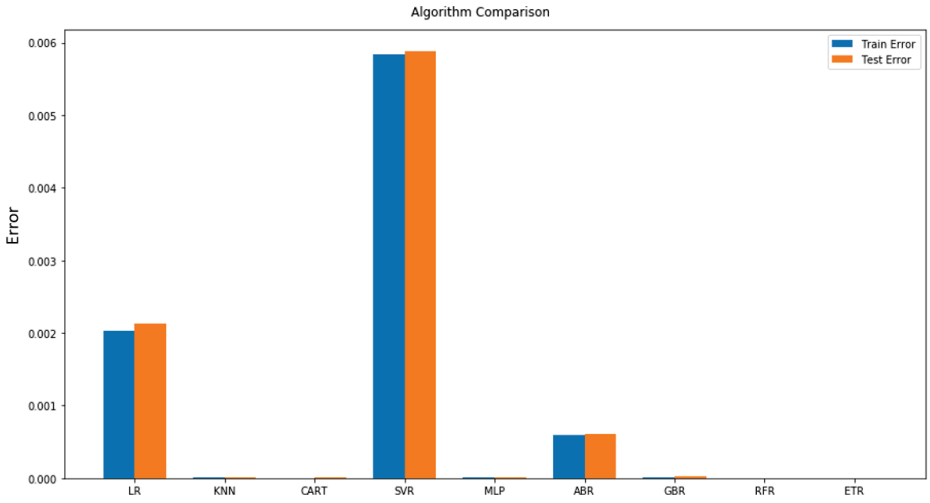
We see that the root mean squared error (RMSE) is 3.08e–5, which is less than one cent. Hence, the ANN model does an excellent job of fitting the Black-Scholes option pricing model. A greater number of layers and tuning of other hyperparameters may enable the ANN model to capture the complex relationship and nonlinearity in the data even better. Overall, the results suggest that ANN may be used to train an option pricing model that matches market prices.

7. Additional analysis: removing the volatility data

As an additional analysis, we make the process harder by trying to predict the price without the volatility data. If the model performance is good, we will eliminate the need to have a volatility function as described before. In this step, we further compare the performance of the linear and nonlinear models. In the following code snippet, we remove the volatility variable from the dataset of the predictor variable and define the training set and test set again:

```
X = X[:, :2]
validation_size = 0.2
train_size = int(len(X) * (1-validation_size))
X_train, X_test = X[0:train_size], X[train_size:len(X)]
Y_train, Y_test = Y[0:train_size], Y[train_size:len(X)]
```

Next, we run the suite of the models (except the regularized regression model) with the new dataset, with the same parameters and similar Python code as before. The performance of all the models after removing the volatility data is as follows:



Looking at the result, we have a similar conclusion as before and see a poor performance of the linear regression and good performance of the ensemble and ANN models. The linear regression now does even a worse job than before. However, the performance of ANN and other ensemble models does not deviate much from their previous performance. This implies the information of the volatility is likely captured in other variables, such as moneyness and time to maturity. Overall, it is good news as it means that fewer variables might be needed to achieve the same performance.

Conclusion

We know that derivative pricing is a nonlinear problem. As expected, our linear regression model did not do as well as our nonlinear models, and the non-linear models have a very good overall performance. We also observed that removing the volatility increases the difficulty of the prediction problem for the linear regression. However, the nonlinear models such as ensemble models and ANN are still able to do well at the prediction process. This does indicate that one might be able to sidestep the development of an option volatility surface and achieve a good prediction with a smaller number of variables.

We saw that an artificial neural network (ANN) can reproduce the Black-Scholes option pricing formula for a call option to a high degree of accuracy, meaning we can leverage efficient numerical calculation of machine learning in derivative pricing without relying on the impractical assumptions made in the traditional derivative pricing models. The ANN and the related machine learning architecture can easily be extended to pricing derivatives in the real world, with no knowledge of the theory of derivative pricing. The use of machine learning techniques can lead to much faster derivative pricing compared to traditional derivative pricing models. The price we

might have to pay for this extra speed is some loss of accuracy. However, this reduced accuracy is often well within reasonable limits and acceptable from a practical point of view. New technology has commoditized the use of ANN, so it might be worthwhile for banks, hedge funds, and financial institutions to explore these models for derivative pricing.

Case Study 3: Investor Risk Tolerance and Robo-Advisors

The risk tolerance of an investor is one of the most important inputs to the portfolio allocation and rebalancing steps of the portfolio management process. There is a wide variety of risk profiling tools that take varied approaches to understanding the risk tolerance of an investor. Most of these approaches include qualitative judgment and involve significant manual effort. In most of the cases, the risk tolerance of an investor is decided based on a risk tolerance questionnaire.

Several studies have shown that these risk tolerance questionnaires are prone to error, as investors suffer from behavioral biases and are poor judges of their own risk perception, especially during stressed markets. Also, given that these questionnaires must be manually completed by investors, they eliminate the possibility of automating the entire investment management process.

So can machine learning provide a better understanding of an investor's risk profile than a risk tolerance questionnaire can? Can machine learning contribute to automating the entire portfolio management process by cutting the client out of the loop? Could an algorithm be written to develop a personality profile for the client that would be a better representation of how they would deal with different market scenarios?

The goal of this case study is to answer these questions. We first build a supervised regression–based model to predict the risk tolerance of an investor. We then build a robo-advisor dashboard in Python and implement the risk tolerance prediction model in the dashboard. The overall purpose is to demonstrate the automation of the manual steps in the portfolio management process with the help of machine learning. This can prove to be immensely useful, specifically for robo-advisors.

A *dashboard* is one of the key features of a robo-advisor as it provides access to important information and allows users to interact with their accounts free of any human dependency, making the portfolio management process highly efficient.

Figure 5-6 provides a quick glance at the robo-advisor dashboard built for this case study. The dashboard performs end-to-end asset allocation for an investor, embedding the machine learning–based risk tolerance model constructed in this case study.

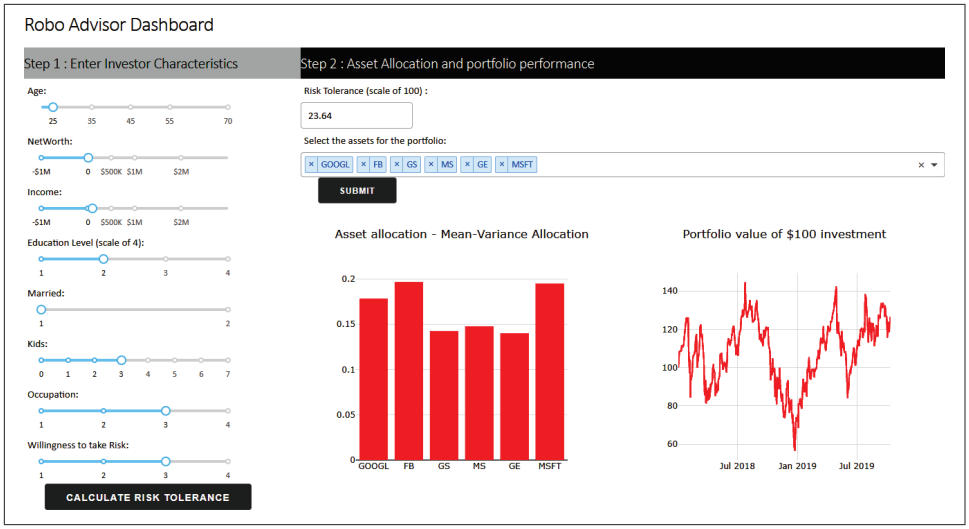


Figure 5-6. Robo-advisor dashboard

This dashboard has been built in Python and is described in detail in an additional step in this case study. Although it has been built in the context of robo-advisors, it can be extended to other areas in finance and can embed the machine learning models discussed in other case studies, providing finance decision makers with a graphical interface for analyzing and interpreting model results.

In this case study, we will focus on:

- Feature elimination and feature importance/intuition.
- Using machine learning to automate manual processes involved in portfolio management process.
- Using machine learning to quantify and model the behavioral bias of investors/individuals.
- Embedding machine learning models into user interfaces or dashboards using Python.

Blueprint for Modeling Investor Risk Tolerance and Enabling a Machine Learning–Based Robo-Advisor

1. Problem definition

In the supervised regression framework used for this case study, the predicted variable is the "true" risk tolerance of an individual,[15] and the predictor variables are demographic, financial, and behavioral attributes of an individual.

The data used for this case study is from the Survey of Consumer Finances (SCF), which is conducted by the Federal Reserve Board. The survey includes responses about household demographics, net worth, financial, and nonfinancial assets for the same set of individuals in 2007 (precrisis) and 2009 (postcrisis). This enables us to see how each household's allocation changed after the 2008 global financial crisis. Refer to the data dictionary for more information on this survey.

2. Getting started—loading the data and Python packages

2.1. Loading the Python packages. The details on loading the standard Python packages were presented in the previous case studies. Refer to the Jupyter notebook for this case study for more details.

2.2. Loading the data. In this step we load the data from the Survey of Consumer Finances and look at the data shape:

```
# load dataset
dataset = pd.read_excel('SCFP2009panel.xlsx')
```

Let us look at the size of the data:

```
dataset.shape
```

```
Output
```

```
(19285, 515)
```

As we can see, the dataset has a total of 19,285 observations with 515 columns. The number of columns represents the number of features.

15 Given that the primary purpose of the model is to be used in the portfolio management context, the individual is also referred to as investor in the case study.

3. Data preparation and feature selection

In this step we prepare the predicted and predictor variables to be used for modeling.

3.1. Preparing the predicted variable. In the first step, we prepare the predicted variable, which is the true risk tolerance.

The steps to compute the true risk tolerance are as follows:

1. Compute the risky assets and the risk-free assets for all the individuals in the survey data. Risky and risk-free assets are defined as follows:

   Risky assets
   : Investments in mutual funds, stocks, and bonds.

   Risk-free assets
   : Checking and savings balances, certificates of deposit, and other cash balances and equivalents.

2. Take the ratio of risky assets to total assets (where total assets is the sum of risky and risk-free assets) of an individual and consider that as a measure of the individual's risk tolerance.[16] From the SCF, we have the data of risky and risk-free assets for the individuals in 2007 and 2009. We use this data and normalize the risky assets with the price of a stock index (S&P500) in 2007 versus 2009 to get risk tolerance.
3. Identify the "intelligent" investors. Some literature describes an intelligent investor as one who does not change their risk tolerance during changes in the market. So we consider the investors who changed their risk tolerance by less than 10% between 2007 and 2009 as the intelligent investors. Of course, this is a qualitative judgment, and there can be several other ways of defining an intelligent investor. However, as mentioned before, beyond coming up with a precise definition of true risk tolerance, the purpose of this case study is to demonstrate the usage of machine learning and provide a machine learning–based framework in portfolio management that can be further leveraged for more detailed analysis.

Let us compute the predicted variable. First, we get the risky and risk-free assets and compute the risk tolerance for 2007 and 2009 in the following code snippet:

```
# Compute the risky assets and risk-free assets for 2007
dataset['RiskFree07']= dataset['LIQ07'] + dataset['CDS07'] + dataset['SAVBND07']\
 + dataset['CASHLI07']
dataset['Risky07'] = dataset['NMMF07'] + dataset['STOCKS07'] + dataset['BOND07']
```

16 There potentially can be several ways of computing the risk tolerance. In this case study, we use the intuitive ways to measure the risk tolerance of an individual.

```
# Compute the risky assets and risk-free assets for 2009
dataset['RiskFree09']= dataset['LIQ09'] + dataset['CDS09'] + dataset['SAVBND09']\
+ dataset['CASHLI09']
dataset['Risky09'] = dataset['NMMF09'] + dataset['STOCKS09'] + dataset['BOND09']

# Compute the risk tolerance for 2007
dataset['RT07'] = dataset['Risky07']/(dataset['Risky07']+dataset['RiskFree07'])

#Average stock index for normalizing the risky assets in 2009
Average_SP500_2007=1478
Average_SP500_2009=948

# Compute the risk tolerance for 2009
dataset['RT09'] = dataset['Risky09']/(dataset['Risky09']+dataset['RiskFree09'])*\
                (Average_SP500_2009/Average_SP500_2007)
```

Let us look at the details of the data:

```
dataset.head()
```

`Output`

	YY1	Y1	WGT09	AGE07	AGECL07	EDUC07	EDCL07	MARRIED07	KIDS07	LIFECL07	...
0	1	11	11668.134198	47	3	12	2	1	0	2	...
1	1	12	11823.456494	47	3	12	2	1	0	2	...
2	1	13	11913.228354	47	3	12	2	1	0	2	...
3	1	14	11929.394266	47	3	12	2	1	0	2	...
4	1	15	11917.722907	47	3	12	2	1	0	2	...

5 rows × 521 columns

The data above displays some of the columns out of the 521 columns of the dataset.

Let us compute the percentage change in risk tolerance between 2007 and 2009:

```
dataset['PercentageChange'] = np.abs(dataset['RT09']/dataset['RT07']-1)
```

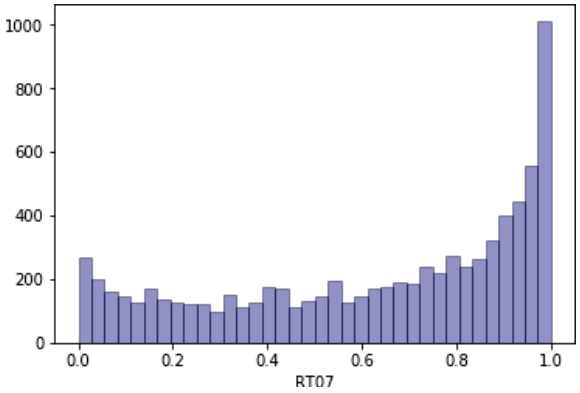
Next, we drop the rows containing "NA" or "NaN":

```
# Drop the rows containing NA
dataset=dataset.dropna(axis=0)

dataset=dataset[~dataset.isin([np.nan, np.inf, -np.inf]).any(1)]
```

Let us investigate the risk tolerance behavior of individuals in 2007 versus 2009. First we look at the risk tolerance in 2007:

```
sns.distplot(dataset['RT07'], hist=True, kde=False,
             bins=int(180/5), color = 'blue',
             hist_kws={'edgecolor':'black'})
```

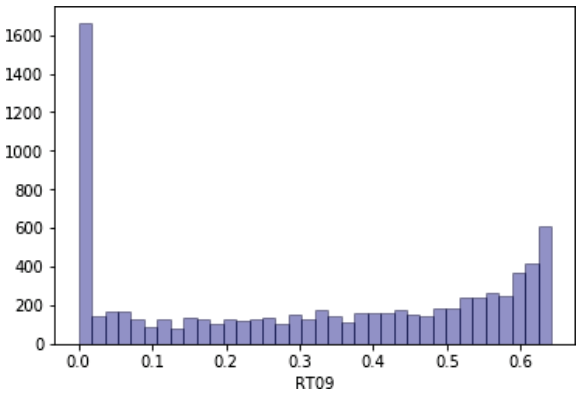
Output



Looking at the risk tolerance in 2007, we see that a significant number of individuals had a risk tolerance close to one, meaning investments were skewed more toward the risky assets. Now let us look at the risk tolerance in 2009:

```
sns.distplot(dataset['RT09'], hist=True, kde=False,
             bins=int(180/5), color = 'blue',
             hist_kws={'edgecolor':'black'})
```

Output



Clearly, the behavior of the individuals reversed after the crisis. Overall risk tolerance decreased, which is shown by the outsized proportion of households having risk tolerance close to zero in 2009. Most of the investments of these individuals were in risk-free assets.

In the next step, we pick the intelligent investors whose change in risk tolerance between 2007 and 2009 was less than 10%, as described in "3.1. Preparing the predicted variable" on page 128:

```
dataset3 = dataset[dataset['PercentageChange']<=.1]
```

We assign the true risk tolerance as the average risk tolerance of these intelligent investors between 2007 and 2009:

```
dataset3['TrueRiskTolerance'] = (dataset3['RT07'] + dataset3['RT09'])/2
```

This is the predicted variable for this case study.

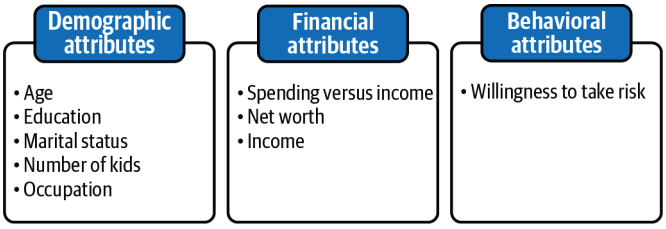
Let us drop other labels that might not be needed for the prediction:

```
dataset3.drop(labels=['RT07', 'RT09'], axis=1, inplace=True)
dataset3.drop(labels=['PercentageChange'], axis=1, inplace=True)
```

3.2. Feature selection—limit the feature space. In this section, we will explore ways to condense the feature space.

3.2.1. Feature elimination. To filter the features further, we check the description in the data dictionary and keep only the features that are relevant.

Looking at the entire data, we have more than *500* features in the dataset. However, academic literature and industry practice indicate risk tolerance is heavily influenced by investor demographic, financial, and behavioral attributes, such as age, current income, net worth, and willingness to take risk. All these attributes were available in the dataset and are summarized in the following section. These attributes are used as features to predict investors' risk tolerance.



In the dataset, each of the columns contains a numeric value corresponding to the value of the attribute. The details are as follows:

AGE
: There are six age categories, where 1 represents age less than 35 and 6 represents age more than 75.

EDUC
: There are four education categories, where 1 represents no high school and 4 represents college degree.

MARRIED
: There are two categories to represent marital status, where 1 represents married and 2 represents unmarried.

OCCU
: This represents occupation category. A value of 1 represents managerial status and 4 represents unemployed.

KIDS
: Number of children.

WSAVED
: This represents the individual's spending versus income, split into three categories. For example, 1 represents spending exceeded income.

NWCAT
: This represents net worth category. There are five categories, where 1 represents net worth less than the 25th percentile and 5 represents net worth more than the 90th percentile.

INCCL
: This represents income category. There are five categories, where 1 represents income less than $10,000 and 5 represents income more than $100,000.

RISK
: This represents the willingness to take risk on a scale of 1 to 4, where 1 represents the highest level of willingness to take risk.

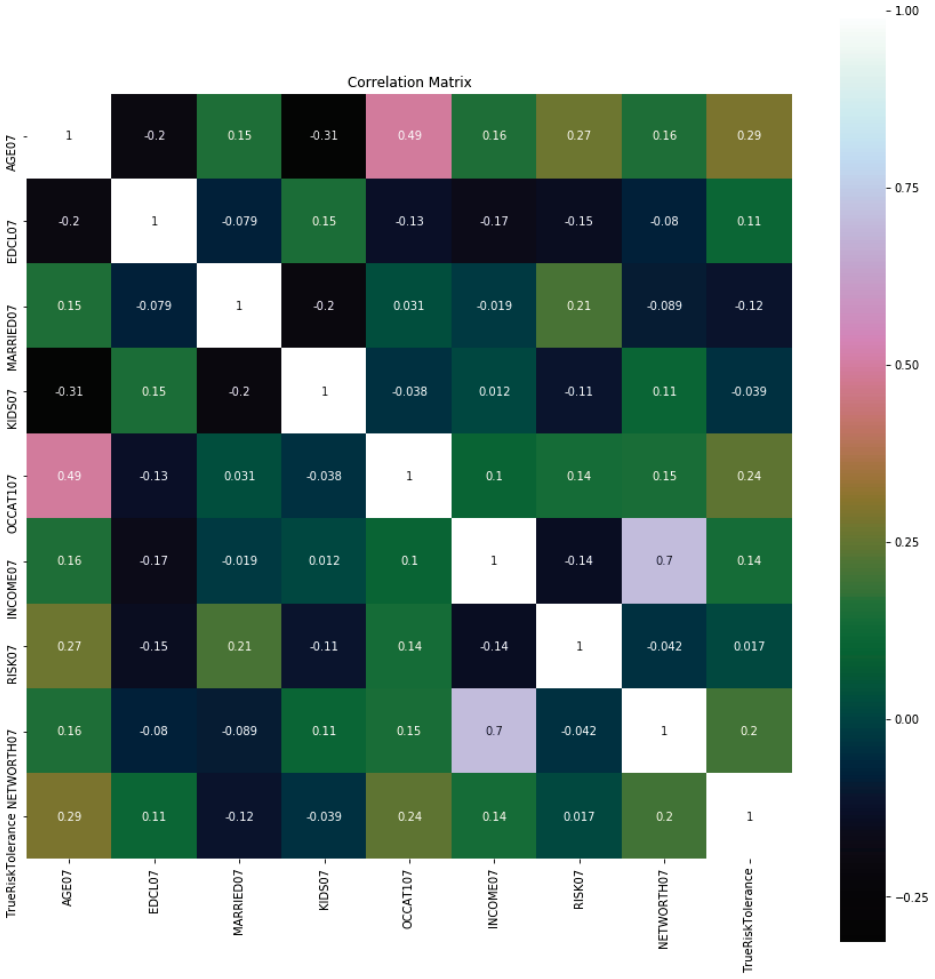
We keep only the intuitive features as of 2007 and remove all the intermediate features and features related to 2009, as the variables of 2007 are the only ones required for predicting the risk tolerance:

```
keep_list2 = ['AGE07','EDCL07','MARRIED07','KIDS07','OCCAT107','INCOME07',\
'RISK07','NETWORTH07','TrueRiskTolerance']

drop_list2 = [col for col in dataset3.columns if col not in keep_list2]

dataset3.drop(labels=drop_list2, axis=1, inplace=True)
```

Now let us look at the correlation among the features:

```
# correlation
correlation = dataset3.corr()
plt.figure(figsize=(15,15))
plt.title('Correlation Matrix')
sns.heatmap(correlation, vmax=1, square=True,annot=True,cmap='cubehelix')
```

Output



Looking at the correlation chart (full-size version available on GitHub), net worth and income are positively correlated with risk tolerance. With a greater number of kids and marriage, risk tolerance decreases. As the willingness to take risks decreases, the risk tolerance decreases. With age there is a positive relationship of the risk

tolerance. As per Hui Wang and Sherman Hanna's paper "Does Risk Tolerance Decrease with Age?," risk tolerance increases as people age (i.e., the proportion of net wealth invested in risky assets increases as people age) when other variables are held constant.

So in summary, the relationship of these variables with risk tolerance seems intuitive.

4. Evaluate models

4.1. Train-test split. Let us split the data into training and test set:

```
Y= dataset3["TrueRiskTolerance"]
X = dataset3.loc[:, dataset3.columns != 'TrueRiskTolerance']
validation_size = 0.2
seed = 3
X_train, X_validation, Y_train, Y_validation = \
train_test_split(X, Y, test_size=validation_size, random_state=seed)
```

4.2. Test options and evaluation metrics. We use R^2 as the evaluation metric and select 10 as the number of folds for cross validation.[17]

```
num_folds = 10
scoring = 'r2'
```

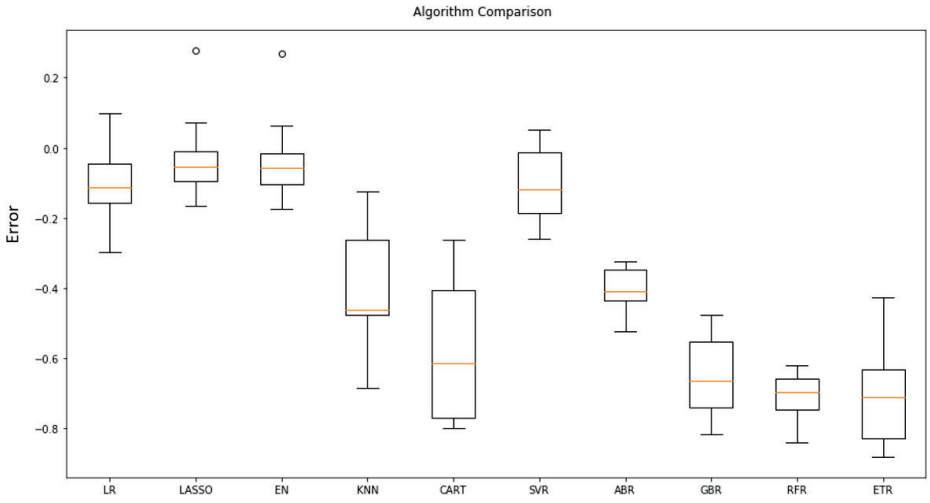
4.3. Compare models and algorithms. Next, we select the suite of the regression model and perform the *k*-folds cross validation.

`Regression Models`

```
# spot-check the algorithms
models = []
models.append(('LR', LinearRegression()))
models.append(('LASSO', Lasso()))
models.append(('EN', ElasticNet()))
models.append(('KNN', KNeighborsRegressor()))
models.append(('CART', DecisionTreeRegressor()))
models.append(('SVR', SVR()))
#Ensemble Models
# Boosting methods
models.append(('ABR', AdaBoostRegressor()))
models.append(('GBR', GradientBoostingRegressor()))
# Bagging methods
models.append(('RFR', RandomForestRegressor()))
models.append(('ETR', ExtraTreesRegressor()))
```

17 We could have chosen RMSE as the evaluation metric; however, R^2 was chosen as the evaluation metric given that we already used RMSE as the evaluation metric in the previous case studies.

The Python code for the *k*-fold analysis step is similar to that of previous case studies. Readers can also refer to the Jupyter notebook of this case study in the code repository for more details. Let us look at the performance of the models in the training set.



The nonlinear models perform better than the linear models, which means that there is a nonlinear relationship between the risk tolerance and the variables used to predict it. Given random forest regression is one of the best methods, we use it for further grid search.

5. Model tuning and grid search

As discussed in Chapter 4, random forest has many hyperparameters that can be tweaked while performing the grid search. However, we will confine our grid search to number of estimators (`n_estimators`) as it is one of the most important hyperparameters. It represents the number of trees in the random forest model. Ideally, this should be increased until no further improvement is seen in the model:

```
# 8. Grid search : RandomForestRegressor
'''
n_estimators : integer, optional (default=10)
    The number of trees in the forest.
'''
param_grid = {'n_estimators': [50,100,150,200,250,300,350,400]}
model = RandomForestRegressor()
kfold = KFold(n_splits=num_folds, random_state=seed)
grid = GridSearchCV(estimator=model, param_grid=param_grid, scoring=scoring, \
  cv=kfold)
grid_result = grid.fit(X_train, Y_train)
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
```

```
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
```

Output

```
Best: 0.738632 using {'n_estimators': 250}
```

Random forest with number of estimators as 250 is the best model after grid search.

6. Finalize the model

Let us look at the results on the test dataset and check the feature importance.

6.1. Results on the test dataset. We prepare the random forest model with the number of estimators as 250:

```
model = RandomForestRegressor(n_estimators = 250)
model.fit(X_train, Y_train)
```

Let us look at the performance in the training set:

```
from sklearn.metrics import r2_score
predictions_train = model.predict(X_train)
print(r2_score(Y_train, predictions_train))
```

Output

```
0.9640632406817223
```

The R^2 of the training set is 96%, which is a good result. Now let us look at the performance in the test set:

```
predictions = model.predict(X_validation)
print(mean_squared_error(Y_validation, predictions))
print(r2_score(Y_validation, predictions))
```

Output

```
0.007781840953471237
0.7614494526639909
```

From the mean squared error and R^2 of 76% shown above for the test set, the random forest model does an excellent job of fitting the risk tolerance.

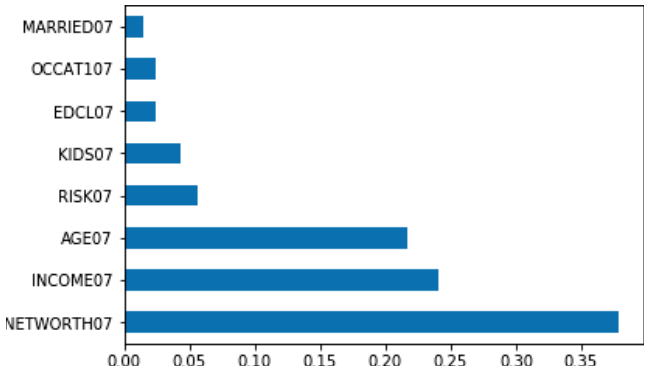
6.2. Feature importance and features intuition

Let us look into the feature importance of the variables within the random forest model:

```
import pandas as pd
import numpy as np
model = RandomForestRegressor(n_estimators= 200,n_jobs=-1)
model.fit(X_train,Y_train)
#use inbuilt class feature_importances of tree based classifiers
#plot graph of feature importances for better visualization
```

```
feat_importances = pd.Series(model.feature_importances_, index=X.columns)
feat_importances.nlargest(10).plot(kind='barh')
plt.show()
```

Output



In the chart, the x-axis represents the magnitude of the importance of a feature. Hence, income and net worth, followed by age and willingness to take risk, are the key variables in determining risk tolerance.

6.3. Save model for later use. In this step we save the model for later use. The saved model can be used directly for prediction given the set of input variables. The model is saved as *finalized_model.sav* using the `dump` module of the pickle package. This saved model can be loaded using the `load` module.

Let's save the model as the first step:

```
# Save Model Using Pickle
from pickle import dump
from pickle import load

# save the model to disk
filename = 'finalized_model.sav'
dump(model, open(filename, 'wb'))
```

Now let's load the saved model and use it for prediction:

```
# load the model from disk
loaded_model = load(open(filename, 'rb'))
# estimate accuracy on validation set
predictions = loaded_model.predict(X_validation)
result = mean_squared_error(Y_validation, predictions)
print(r2_score(Y_validation, predictions))
print(result)
```

```
Output

    0.7683894847939692
    0.007555447734714956
```

7. Additional step: robo-advisor dashboard

We mentioned the robo-advisor dashboard in the beginning of this case study. The robo-advisor dashboard performs an automation of the portfolio management process and aims to overcome the problem of traditional risk tolerance profiling.

Python Code for Robo-Advisor Dashboard

This robo-advisor dashboard is built in Python using the plotly dash package. Dash is a productive Python framework for building web applications with good user interfaces. The code for the robo-advisor dashboard is added to the code repository for this book. The code is in a Jupyter notebook called "Sample Robo-advisor". A detailed description of the code is outside the scope of this case study. However, the codebase can be leveraged for creation of any new machine learning–enabled dashboard.

The dashboard has two panels:

- Inputs for investor characteristics
- Asset allocation and portfolio performance

Input for investor characteristics. Figure 5-7 shows the input panel for the investor characteristics. This panel takes all the input regarding the investor's demographic, financial, and behavioral attributes. These inputs are for the predicted variables we used in the risk tolerance model created in the preceding steps. The interface is designed to input the categorical and continuous variables in the correct format.

Once the inputs are submitted, we leverage the model saved in "6.3. Save model for later use" on page 137. This model takes all the inputs and produces the risk tolerance of an investor (refer to the `predict_riskTolerance` function of the "Sample Robo-advisor" Jupyter notebook in the code repository for this book for more details). The risk tolerance prediction model is embedded in this dashboard and is triggered once the "Calculate Risk Tolerance" button is pressed after submitting the inputs.

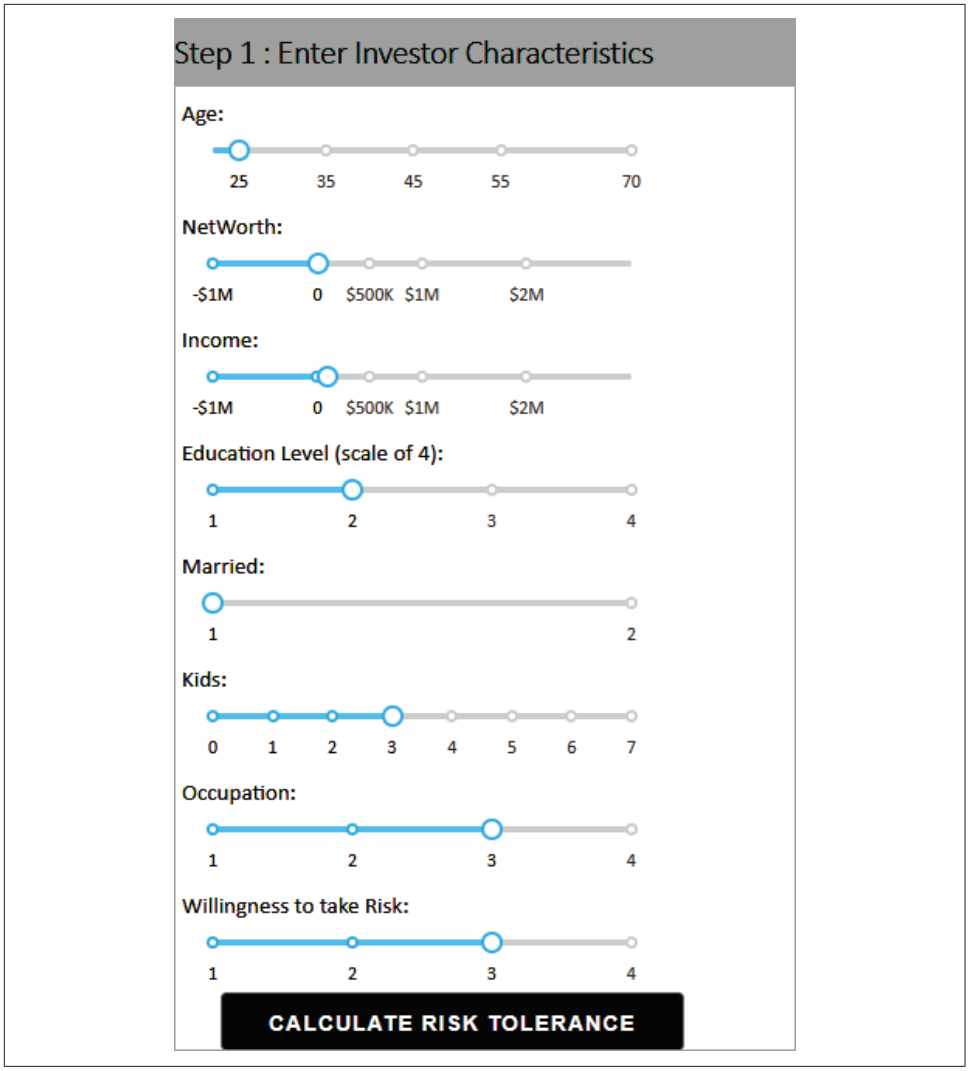


Figure 5-7. Robo-advisor input panel

7.2 Asset allocation and portfolio performance. Figure 5-8 shows the "Asset Allocation and Portfolio Performance" panel, which performs the following functionalities:

- Once the risk tolerance is computed using the model, it is displayed on the top of this panel.
- In the next step, we pick the assets for our portfolio from the dropdown.
- Once the list of assets are submitted, the traditional mean-variance portfolio allocation model is used to allocate the portfolio among the assets selected. Risk

tolerance is one of the key inputs for this process. (Refer to the `get_asset_allocation` function of the "Sample Robo-advisor" Jupyter notebook in the code repository for this book for more details.)

- The dashboard also shows the historical performance of the allocated portfolio for an initial investment of *$100*.

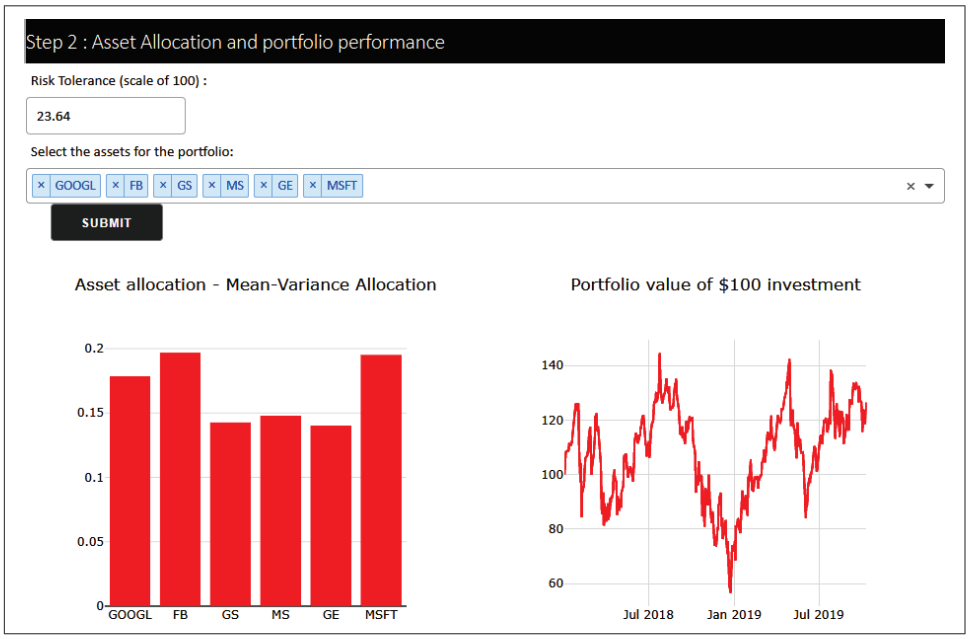


Figure 5-8. Robo-advisor asset allocation and portfolio performance panel

Although the dashboard is a basic version of the robo-advisor dashboard, it performs end-to-end asset allocation for an investor and provides the portfolio view and historical performance of the portfolio over a selected period. There are several potential enhancements to this prototype in terms of the interface and underlying models used. The dashboard can be enhanced to include additional instruments and incorporate additional features such as real-time portfolio monitoring, portfolio rebalancing, and investment advisory. In terms of the underlying models used for asset allocation, we have used the traditional mean-variance optimization method, but it can be further enhanced to use the allocation algorithms based on machine learning techniques such as eigen-portfolio, hierarchical risk parity, or reinforcement learning–based models, described in Chapters 7, 8 and 9, respectively. The risk tolerance model can be further enhanced by using additional features or using the actual data of the investors rather than using data from the Survey of Consumer Finances.

Conclusion

In this case study, we introduced the regression-based algorithm applied to compute an investor's risk tolerance, followed by a demonstration of the model in a robo-advisor setup. We showed that machine learning models might be able to objectively analyze the behavior of different investors in a changing market and attribute these changes to variables involved in determining risk appetite. With an increase in the volume of investors' data and the availability of rich machine learning infrastructure, such models might prove to be more useful than existing manual processes.

We saw that there is a nonlinear relationship between the variables and the risk tolerance. We analyzed the feature importance and found that results of the case study are quite intuitive. Income and net worth, followed by age and willingness to take risk, are the key variables to deciding risk tolerance. These variables have been considered key variables to model risk tolerance across academic and industry literature.

Through the robo-advisor dashboard powered by machine learning, we demonstrated an effective combination of data science and machine learning implementation in wealth management. Robo-advisors and investment managers could leverage such models and platforms to enhance the portfolio management process with the help of machine learning.

Case Study 4: Yield Curve Prediction

A *yield curve* is a line that plots yields (interest rates) of bonds having equal credit quality but differing maturity dates. This yield curve is used as a benchmark for other debt in the market, such as mortgage rates or bank lending rates. The most frequently reported yield curve compares the 3-months, 2-years, 5-years, 10-years, and 30-years U.S. Treasury debt.

The yield curve is the centerpiece in a fixed income market. Fixed income markets are important sources of finance for governments, national and supranational institutions, banks, and private and public corporations. In addition, yield curves are very important to investors in pension funds and insurance companies.

The yield curve is a key representation of the state of the bond market. Investors watch the bond market closely as it is a strong predictor of future economic activity and levels of inflation, which affect prices of goods, financial assets, and real estate. The slope of the yield curve is an important indicator of short-term interest rates and is followed closely by investors.

Hence, an accurate yield curve forecasting is of critical importance in financial applications. Several statistical techniques and tools commonly used in econometrics and finance have been applied to model the yield curve.

In this case study we will use supervised learning–based models to predict the yield curve. This case study is inspired by the paper *Artificial Neural Networks in Fixed Income Markets for Yield Curve Forecasting* by Manuel Nunes et al. (2018).

In this case study, we will focus on:

- Simultaneous modeling (producing multiple outputs at the same time) of the interest rates.
- Comparison of neural network versus linear regression models.
- Modeling a time series in a supervised regression–based framework.
- Understanding the variable intuition and feature selection.

Overall, the case study is similar to the stock price prediction case study presented earlier in this chapter, with the following differences:

- We predict multiple outputs simultaneously, rather than a single output.
- The predicted variable in this case study is not the return variable.
- Given that we already covered time series models in case study 1, we focus on artificial neural networks for prediction in this case study.

Blueprint for Using Supervised Learning Models to Predict the Yield Curve

1. Problem definition

In the supervised regression framework used for this case study, three tenors (1M, 5Y, and 30Y) of the yield curve are the predicted variables. These tenors represent short-term, medium-term, and long-term tenors of the yield curve.

We need to understand what affects the movement of the yield curve and hence incorporate as much information into our model as we can. As a high-level overview, other than the historical price of the yield curve itself, we look at other correlated variables that can influence the yield curve. The independent or predictor variables we consider are:

- *Previous value of the treasury curve for different tenors*. The tenors used are 1-month, 3-month, 1-year, 2-year, 5-year, 7-year, 10-year, and 30-year yields.

- *Percentage of the federal debt* held by the public, foreign governments, and the federal reserve.
- *Corporate spread* on Baa-rated debt relative to the 10-year treasury rate.

The federal debt and corporate spread are correlated variables and can be potentially useful in modeling the yield curve. The dataset used for this case study is extracted from Yahoo Finance and FRED. We will use the daily data of the last 10 years, from 2010 onward.

By the end of this case study, readers will be familiar with a general machine learning approach to yield curve modeling, from gathering and cleaning data to building and tuning different models.

2. Getting started—loading the data and Python packages

2.1. Loading the Python packages. The loading of Python packages is similar to other case studies in this chapter. Refer to the Jupyter notebook of this case study for more details.

2.2. Loading the data. The following steps demonstrate the loading of data using Pandas's `DataReader` function:

```
# Get the data by webscraping using pandas datareader
tsy_tickers = ['DGS1MO', 'DGS3MO', 'DGS1', 'DGS2', 'DGS5', 'DGS7', 'DGS10',
               'DGS30',
               'TREAST', # Treasury securities held by the Federal Reserve ($MM)
               'FYGFDPUN', # Federal Debt Held by the Public ($MM)
               'FDHBFIN', # Federal Debt Held by International Investors ($BN)
               'GFDEBTN', # Federal Debt: Total Public Debt ($BN)
               'BAA10Y', # Baa Corporate Bond Yield Relative to Yield on 10-Year
              ]
tsy_data = web.DataReader(tsy_tickers, 'fred').dropna(how='all').ffill()
tsy_data['FDHBFIN'] = tsy_data['FDHBFIN'] * 1000
tsy_data['GOV_PCT'] = tsy_data['TREAST'] / tsy_data['GFDEBTN']
tsy_data['HOM_PCT'] = tsy_data['FYGFDPUN'] / tsy_data['GFDEBTN']
tsy_data['FOR_PCT'] = tsy_data['FDHBFIN'] / tsy_data['GFDEBTN']
```

Next, we define our dependent (Y) and independent (X) variables. The predicted variables are the rate for three tenors of the yield curve (i.e., 1M, 5Y, and 30Y) as mentioned before. The number of trading days in a week is assumed to be five, and we compute the lagged version of the variables mentioned in the problem definition section as independent variables using five trading day lag.

The lagged five-day variables embed the time series component by using a *time-delay approach*, where the lagged variable is included as one of the independent variables. This step reframes the time series data into a supervised regression–based model framework.

3. Exploratory data analysis. We will look at descriptive statistics and data visualization in this section.

3.1. Descriptive statistics. Let us look at the shape and the columns in the dataset:

```
dataset.shape
```

```
Output
```

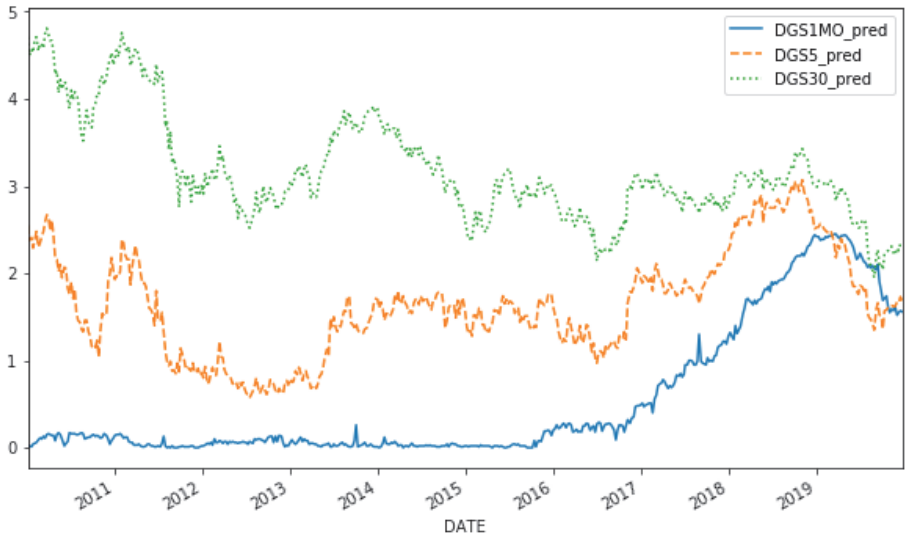
```
(505, 15)
```

The data contains around 500 observations with 15 columns.

3.2. Data visualization. Let us first plot the predicted variables and see their behavior:

```
Y.plot(style=['-','--',':'])
```

```
Output
```



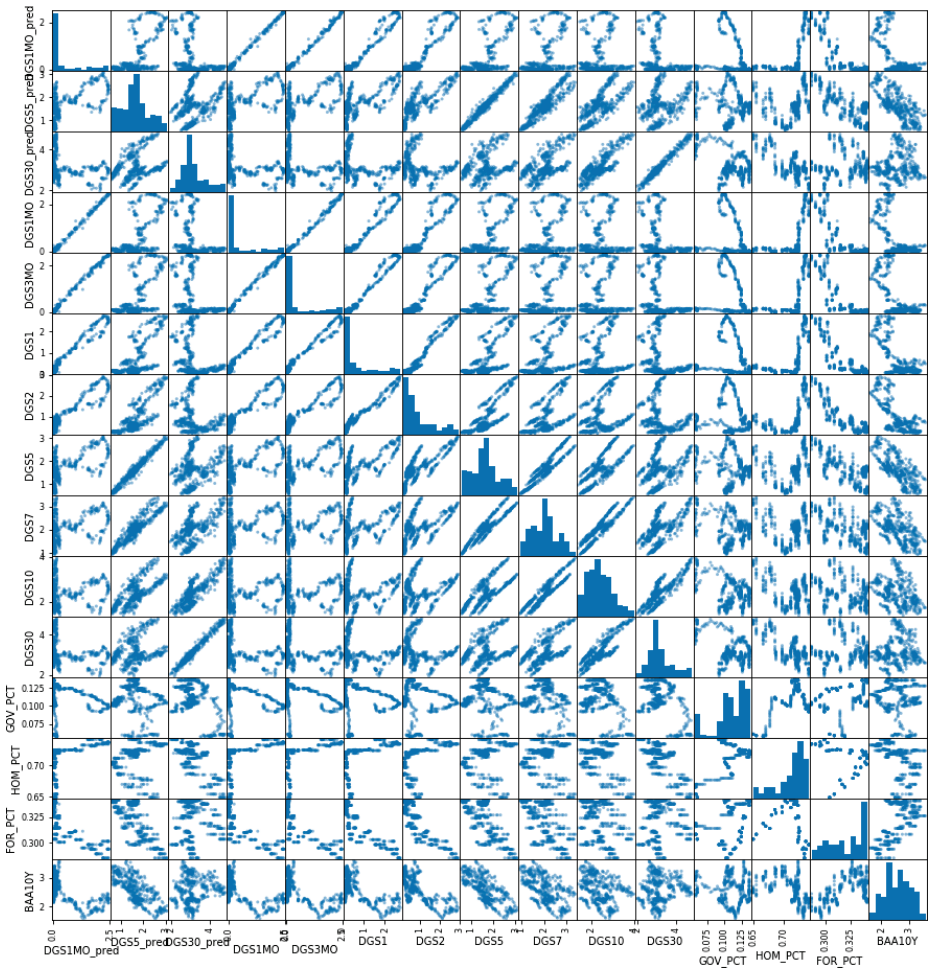
In the plot, we see that the deviation among the short-term, medium-term, and long-term rates was higher in 2010 and has been decreasing since then. There was a drop in the long-term and medium-term rates during 2011, and they also have been declining since then. The order of the rates has been in line with the tenors. However, for a few months in recent years, the *5Y* rate has been lower than the *1M* rate. In the time series of all the tenors, we can see that the mean varies with time, resulting in an upward trend. Thus these series are nonstationary time series.

In some cases, the linear regression for such nonstationary dependent variables might not be valid. However, we are using the lagged variables, which are also nonstationary as independent variables. So we are effectively modeling a nonstationary time series against another nonstationary time series, which might still be valid.

Next, we look at the scatterplots (a correlation plot is skipped for this case study as it has a similar interpretation to that of a scatterplot). We can visualize the relationship between all the variables in the regression using the scatter matrix shown below:

```
# Scatterplot Matrix
pyplot.figure(figsize=(15,15))
scatter_matrix(dataset,figsize=(15,16))
pyplot.show()
```

`Output`



Looking at the scatterplot (full-size version available on GitHub), we see a significant linear relationship of the predicted variables with their lags and other tenors of the yield curve. There is also a linear relationship, with negative slope between 1M, 5Y rates versus corporate spread and changes in foreign government purchases. The 30Y rate shows a linear relationship with these variables, although the slope is negative. Overall, we see a lot of linear relationships, and we expect the linear models to perform well.

4. Data preparation and analysis

We performed most of the data preparation steps (i.e., getting the dependent and independent variables) in the preceding steps, and so we'll skip this step.

5. Evaluate models

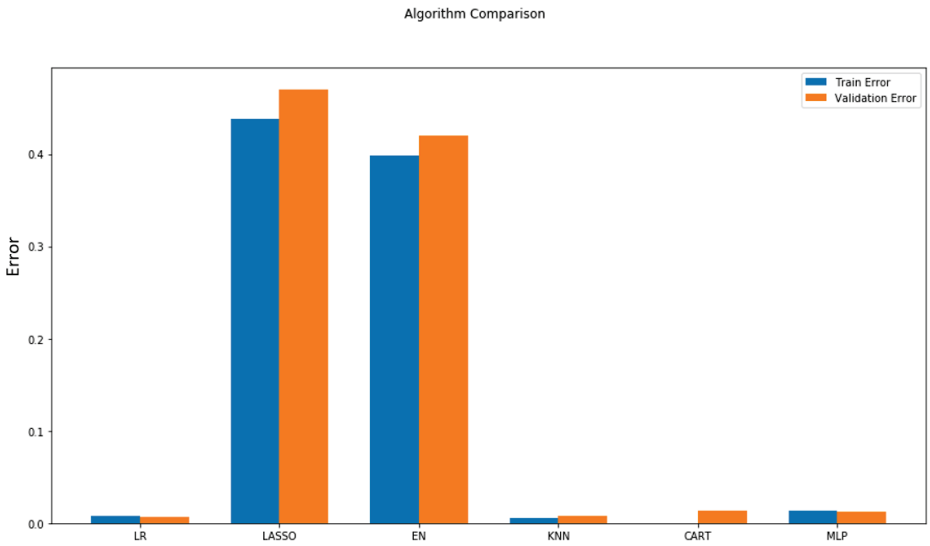
In this step we evaluate the models. The Python code for this step is similar to dthat in case study 1, and some of the repetitive code is skipped. Readers can also refer to the Jupyter notebook of this case study in the code repository for this book for more details.

5.1. Train-test split and evaluation metrics. We will use 80% of the dataset for modeling and use 20% for testing. We will evaluate algorithms using the mean squared error metric. All the algorithms use default tuning parameters.

5.2. Compare models and algorithms. In this case study, the primary purpose is to compare the linear models with the artificial neural network in yield curve modeling. So we stick to the linear regression (LR), regularized regression (LASSO and EN), and artificial neural network (shown as MLP). We also include a few other models such as KNN and CART, as these models are simpler with good interpretation, and if there is a nonlinear relationship between the variables, the CART and KNN models will be able to capture it and provide a good comparison benchmark for ANN.

Looking at the training and test error, we see a good performance of the linear regression model. We see that lasso and elastic net perform poorly. These are regularized regression models, and they reduce the number of variables in case they are not important. A decrease in the number of variables might have caused a loss of information leading to poor model performance. KNN and CART are good, but looking closely, we see that the test errors are higher than the training error. We also see that the performance of the artificial neural network (MLP) algorithm is comparable to the linear regression model. Despite its simplicity, the linear regression is a tough benchmark to beat for one-step-ahead forecasting when there is a significant linear relationship between the variables.

Output



6. Model tuning and grid search.

Similar to case study 2 of this chapter, we perform a grid search of the ANN model with different combinations of hidden layers. Several other hyperparameters such as learning rate, momentum, activation function, number of epochs, and batch size can be tuned during the grid search process, similar to the steps mentioned below.

```
'''
hidden_layer_sizes : tuple, length = n_layers - 2, default (100,)
    The ith element represents the number of neurons in the ith
    hidden layer.
'''
param_grid={'hidden_layer_sizes': [(20,), (50,), (20,20), (20, 30, 20)]}
model = MLPRegressor()
kfold = KFold(n_splits=num_folds, random_state=seed)
grid = GridSearchCV(estimator=model, param_grid=param_grid, scoring=scoring, \
  cv=kfold)
grid_result = grid.fit(X_train, Y_train)
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

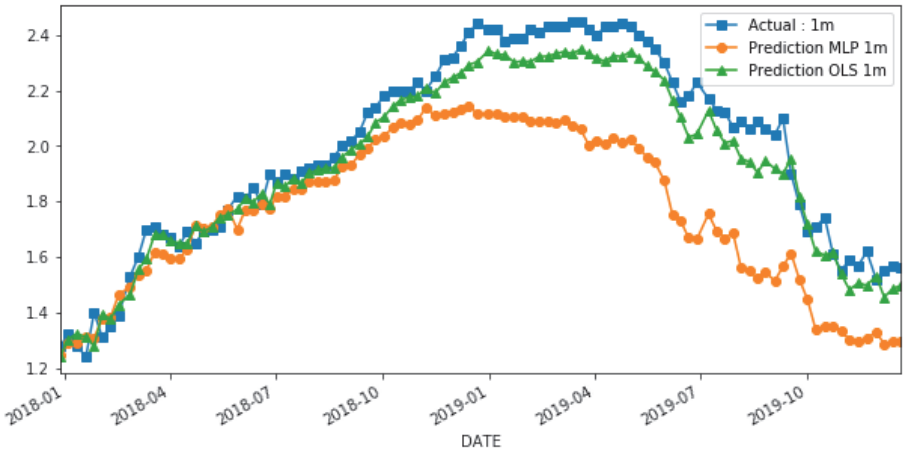
Output

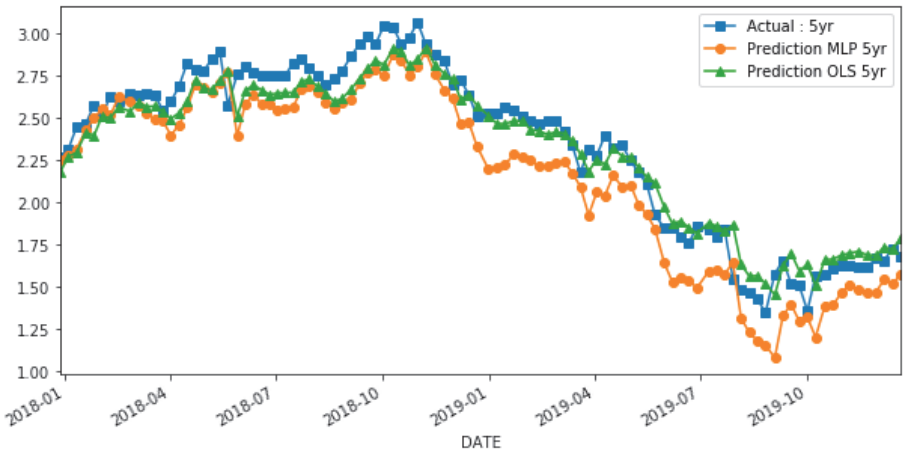
```
Best: -0.018006 using {'hidden_layer_sizes': (20, 30, 20)}
-0.036433 (0.019326) with: {'hidden_layer_sizes': (20,)}
```

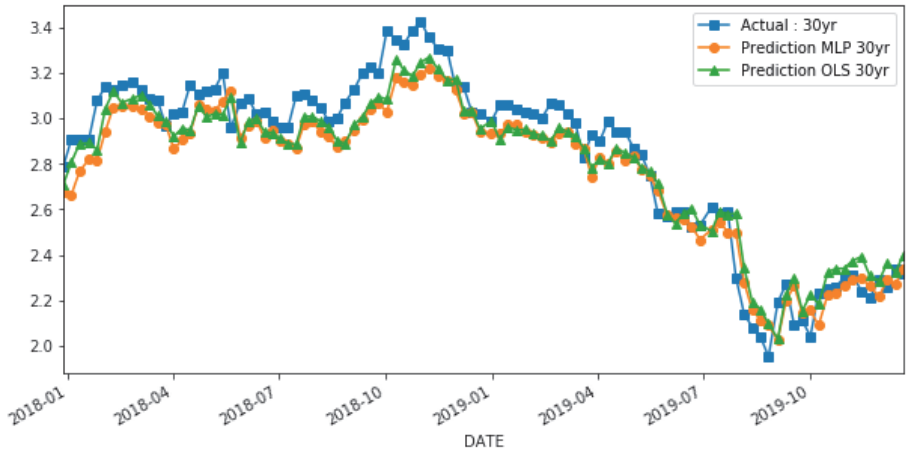
```
-0.020793 (0.007075) with: {'hidden_layer_sizes': (50,)}
-0.026638 (0.010154) with: {'hidden_layer_sizes': (20, 20)}
-0.018006 (0.005637) with: {'hidden_layer_sizes': (20, 30, 20)}
```

The best model is the model with three layers, with 20, 30, and 20 nodes in each hidden layer, respectively. Hence, we prepare a model with this configuration and check its performance on the test set. This is a crucial step, as a greater number of layers may lead to overfitting and have poor performance in the test set.

Prediction comparison. In the last step we look at the prediction plot of actual data versus the prediction from both linear regression and ANN models. Refer to the Jupyter notebook of this case study for the Python code of this section.







Looking at the charts above, we see that the predictions of the linear regression and ANN are comparable. For 1M tenor, the fitting with ANN is slightly poor compared to the regression. However, for 5Y and 30Y tenors the ANN performs as well as the regression model.

Conclusion

In this case study, we applied supervised regression to the prediction of several tenors of yield curve. The linear regression model, despite its simplicity, is a tough benchmark to beat for such one-step-ahead forecasting, given the dominant characteristic of the last available value of the variable to predict. The ANN results in this case study are comparable to the linear regression models. An additional benefit of ANN is that it is more flexible to changing market conditions. Also, ANN models can be enhanced by performing grid search on several other hyperparameters and the option of incorporating recurrent neural networks, such as LSTM.

Overall, we built a machine learning–based model using ANN with an encouraging outcome, in the context of fixed income instruments. This allows us to perform predictions using historical data to generate results and analyze risk and profitability before risking any actual capital in the fixed income market.

Chapter Summary

In "Case Study 1: Stock Price Prediction" on page 95, we covered a machine learning and time series–based framework for stock price prediction. We demonstrated the significance of visualization and compared time series against the machine learning

models. In "Case Study 2: Derivative Pricing" on page 114, we explored the use of machine learning for a traditional derivative pricing problem and demonstrated a high model performance. In "Case Study 3: Investor Risk Tolerance and Robo-Advisors" on page 125, we demonstrated how supervised learning models can be used to model the risk tolerance of investors, which can lead to automation of the portfolio management process. "Case Study 4: Yield Curve Prediction" on page 141 was similar to the stock price prediction case study, providing another example of comparison of linear and nonlinear models in the context of fixed income markets.

We saw that time series and linear supervised learning models worked well for asset price prediction problems (i.e., case studies 1 and 4), where the predicted variable had a significant linear relationship with its lagged component. However, in derivative pricing and risk tolerance prediction, where there are nonlinear relationships, ensemble and ANN models performed better. Readers who are interested in implementing a case study using supervised regression or time series models are encouraged to understand the nuances in the variable relationships and model intuition before proceeding to model selection.

Overall, the concepts in Python, machine learning, time series, and finance presented in this chapter through the case studies can used as a blueprint for any other supervised regression–based problem in finance.

Exercises

- Using the concepts and framework of machine learning and time series models specified in case study 1, develop a predictive model for another asset class—currency pair (EUR/USD, for example) or bitcoin.
- In case study 1, add some technical indicators, such as trend or momentum, and check the enhancement in the model performance. Some of the ideas of the technical indicators can be borrowed from "Case Study 3: Bitcoin Trading Strategy" on page 179 in Chapter 6.
- Using the concepts in "Case Study 2: Derivative Pricing" on page 114, develop a machine learning–based model to price American options.
- Incorporate multivariate time series modeling using a variant of the ARIMA model, such as VARMAX, for rates prediction in the yield curve prediction case study and compare the performance against the machine learning–based models.
- Enhance the robo-advisor dashboard presented in "Case Study 3: Investor Risk Tolerance and Robo-Advisors" on page 125 to incorporate instruments other than equities.