CHAPTER 11

Forecasting Stock and Commodity Prices

In the chapters so far, we covered a variety of concepts and solved diverse real-world problems. In this chapter, we will dive into forecast/prediction use cases. Predictive analytics or modeling involves concepts from data mining, advanced statistics, Machine Learning, and so on to model historical data to forecast future events. Predictive modeling has use cases across domains such as financial services, healthcare, telecommunications, etc.

There are a number of techniques developed over the years to understand temporal data and model patterns to score future events and behaviors. *Time series analysis* forms the descriptive aspect of such data and the understanding helps in modeling and forecasting the same. Traditional approaches like *regression analysis* (see Chapter 6) and *Box-Jenkins methodologies* have been deeply studied and applied over the years. More recently, improvements in computation and Machine Learning algorithms have seen Machine Learning techniques like *neural networks* or to be more specific, Deep Learning, making a headway in forecasting use cases with some amazing results.

This chapter discusses forecasting using stock and commodity price datasets. We will utilize traditional time series models as well as deep learning models like recurrent neural networks to forecast the prices. Through this chapter, we will cover the following topics:

- Brief overview of time series analysis
- Forecasting commodity pricing using traditional approaches like ARIMA
- Forecasting stock prices with newer deep learning approaches like RNNs and LSTMs

The code samples, jupyter notebooks, and sample datasets for this chapter is available in the GitHub repository for this book at `https://github.com/dipanjanS/practical-machine-learning-with-python` under the directory/folder for Chapter 11.

Time Series Data and Analysis

A time series is a sequence of observed values in a time-ordered manner. The observations are recorded at equally spaced time intervals. Time series data is available and utilized in the fields of statistics, economics, finance, weather modeling, pattern recognition, and many more.

Time series analysis is the study of underlying structure and forces that produced observations. It provides descriptive frameworks to analyze characteristics of data and other meaningful statistics. It also provides techniques to then utilize this to fit a model to forecast, monitor, and control.

There is another school of thought that separates the descriptive and modeling components of time series. Here, time series analysis typically is concerned with only the descriptive analysis of time series data to understand various components and underlying structure. The modeling aspect that utilizes time series

© Dipanjan Sarkar, Raghav Bali and Tushar Sharma 2018
D. Sarkar et al., *Practical Machine Learning with Python*, https://doi.org/10.1007/978-1-4842-3207-1_11

for prediction/forecasting use cases is termed as *time series forecasting*. Though in general, both schools of thought utilize the same set of tools and techniques. It is more about how the concepts are grouped for a structured learning and application.

■ **Note** Time series and its analysis is a complete field of study on its own and we merely discuss certain concepts and techniques for our use cases here. This chapter is by no means a complete guide on time series and its analysis.

Time series can be analyzed both in frequency and time domains. The frequency domain analysis includes spectral and wavelet analysis techniques, while time domain includes auto- and cross-correlation analysis. In this chapter, we primarily focus on time series forecasting in the time domain with a brief discussion around descriptive characteristics of time series data. Moreover, we concentrate on univariate time series analysis (time is an implicit variable).

To better understand concepts related to time series, we utilize a sample dataset. The following snippet loads a web site visit data with a daily frequency using `pandas`. You can refer to the jupyter notebook `notebook_getting_started_time_series.ipynb` for the necessary code snippets and examples. The dataset is available at `http://openmv.net/info/website-traffic`.

```
In [1] : import pandas as pd
    ...:
    ...: #load data
    ...: input_df = pd.read_csv(r'website-traffic.csv')
    ...:
    ...: input_df['date_of_visit'] = pd.to_datetime(input_df.MonthDay.\
    ...:                                         str.cat( input_df.Year.astype(str),
    ...:                                         sep=' '))
```

The snippet first creates a new attribute called the `date_of_visit` using a combination of Day, Month, and Year values available in the base dataframe. Since the dataset is about web site visits per day, the variable of interest is the visit count for the day with the time dimension, i.e. `date_of_visit` being the implicit one. The output plot of visits per day is shown in Figure 11-1.

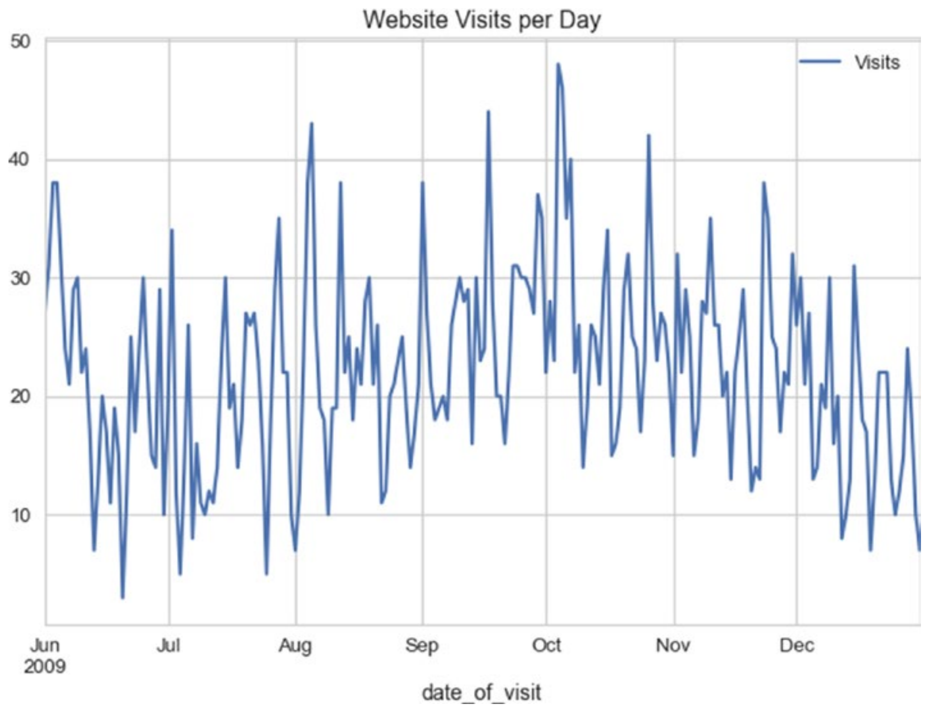


Figure 11-1. *Web site visits per day*

Time Series Components

The time series at hand is data related to web site visits per day for a given web site. As mentioned earlier, time series analysis deals with understanding the underlying structure and forces that result in the series as we see it. Let's now try to deconstruct various components that make up the time series at hand. A time series is said to be comprised of the following three major components:

- **Seasonality**: These are the periodic fluctuations in the observed data. For example, weather patterns or sale patterns.
- **Trend**: This is the increasing or decreasing behavior of the series with time. For example, population growth patterns.
- **Residual**: This is the remaining signal after removing the seasonality and trend signals. It can be further decomposed to remove the noise component as well.

It is interesting to note that most real-world time series data have a combination or all of these components available. Yet, it is mostly the noise that's always apparently present, with trend and seasonality being optional in certain cases. In the following snippet, we utilize `statsmodels` to decompose our web site visit time series into its three constituents and then plot the same.

```
In [2] : from statsmodels.tsa.seasonal import seasonal_decompose
    ...:
    ...: # extract visits as series from the dataframe
    ...: ts_visits = pd.Series(input_df.Visits.values,
    ...:                          index=pd.date_range(
    ...:                                          input_df.date_of_visit.min(),
    ...:                                          input_df.date_of_visit.max(),
    ...:                                          freq='D')
    ...:                          )
    ...:
    ...:
    ...: deompose = seasonal_decompose(ts_visits.interpolate(),
    ...:                                freq=24)
    ...: deompose.plot()
```

We first create a pandas `Series` object with additional care taken to set the frequency of the time series index. It is important to note that `statsmodels` has a number of time series modeling modules available and they rely on underlying data structures (such as `pandas, numpy`, etc.) to specify the frequency of the time series. In this case, since the data is at a daily level, we set the frequency of `ts_visits` object to 'D' denoting a daily frequency. We then simply use the `seasonal_decompose()` function from `statsmodels` to get the required constituents. The decomposed series is shown in the plot in Figure 11-2.

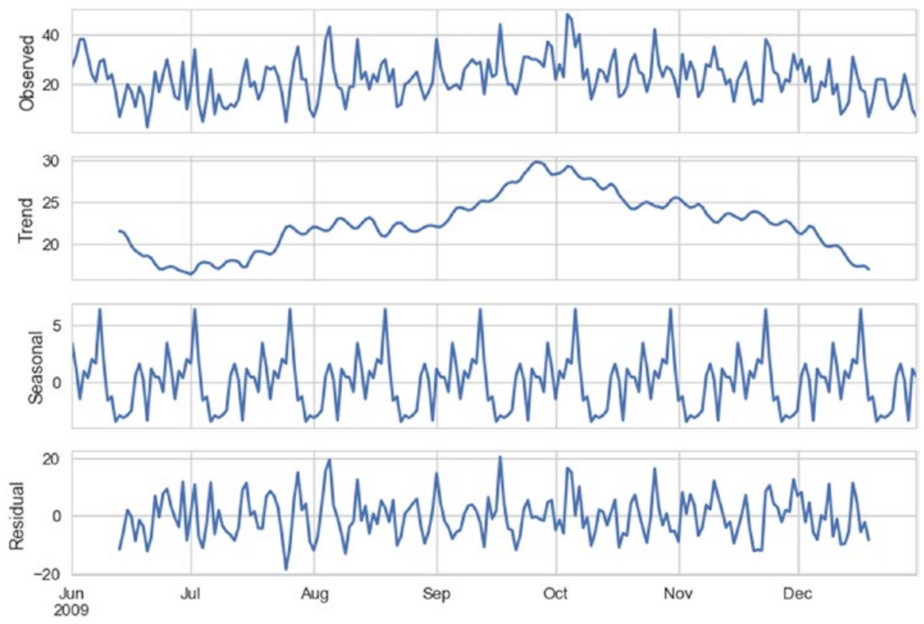


Figure 11-2. *Web site visit time series and its constituent signals*

It is apparent from Figure 11-2, the time series at hand has both upward and downward trends in it. It shows a gradual increasing trend until October, post which it starts a downward behavior. The series certainly has a monthly periodicity or seasonality to it. The remaining signal is what is marked as residual in Figure 11-2.

Smoothing Techniques

As discussed in the previous chapters, preprocessing the raw data depends upon the data as well as the use case requirements. Yet there are certain standard preprocessing techniques for each type of data. Unlike the datasets we have seen so far, where we consider each observation to be independent of other (past or future) observations, time series have inherent dependency on historical observations. As seen from the decomposition of web site visits series, there are multiple factors impacting each observation. It is an inherent property of time series data to have random variation to it apart from its other constituents. To better understand, model, and utilize time series for prediction related tasks, we usually perform a preprocessing step better termed as *smoothing*. Smoothing helps reduce the effect of random variation and helps clearly reveal the seasonality, trend, and residual components of the series. There are various methods to smooth out a time series. They are broadly categorized as follows.

Moving Average

Instead of taking an average of complete time series (which we do in cases of non-temporal data) to summarize, moving average makes use of a rolling windowed approach. In this case, we compute the mean of each successive smaller windows of past data to smoothen out the impact of random variation.
The following is a general formula for moving average calculation.

$$MA_t = \frac{x_t + x_{t-1} + x_{t-2} + \ldots + x_{t-n}}{n}$$

Where, MA_t is the moving average for time period t, x_t, x_{t-1} and so on denote observed values at particular time periods, and n is the window size. For example, the following snippet calculates the moving average for visits with a window size of 3.

```
In [3] : # moving average
    ...: input_df['moving_average'] = input_df['Visits'].rolling(window=3,
    ...:                                                          center=False).mean()
    ...:
    ...: print(input_df[['Visits','moving_average']].head(10))
    ...:
    ...: plt.plot(input_df.Visits,'-',color='black',alpha=0.3)
    ...: plt.plot(input_df.moving_average,color='b')
    ...: plt.title('Website Visit and Moving Average Smoothening')
    ...: plt.legend()
    ...: plt.show()
```

The moving average calculated using a window size 3 has the following results. It should be pretty clear that for a window size of 3, the first two observations would not have any moving averages available, hence the `NaN`.

```
Out[3]:
   Visits  moving_average
0      27             NaN
1      31             NaN
2      38       32.000000
3      38       35.666667
4      31       35.666667
```

```
5       24        31.000000
6       21        25.333333
7       29        24.666667
8       30        26.666667
9       22        27.000000
```

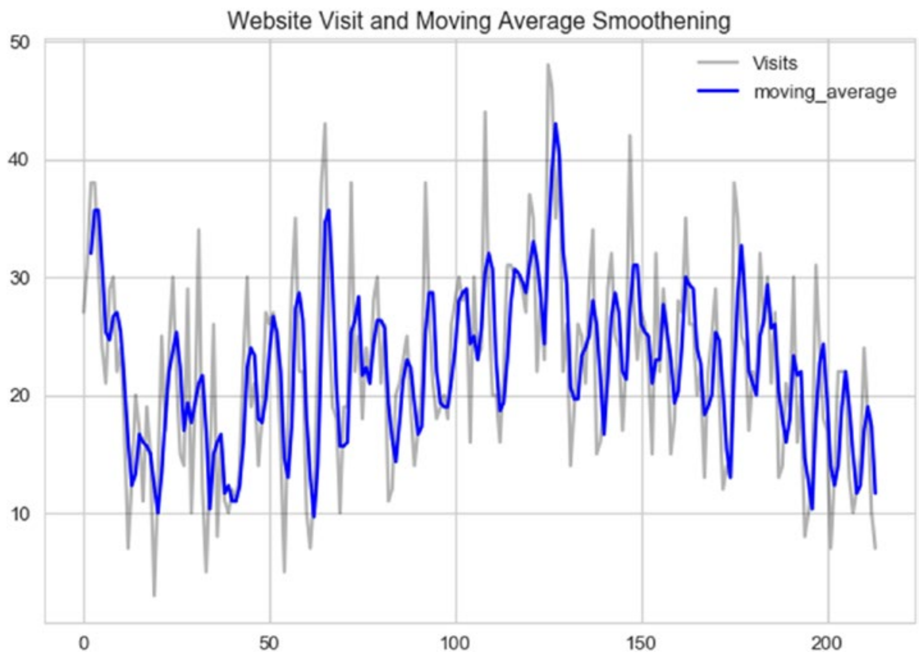


Figure 11-3. *Smoothening using moving average*

The plot in Figure 11-3 shows the smoothened visits time series. The smoothened series captures the overall structure of the original series all the while reducing the random variation in it. You are encouraged to explore and experiment with different window sizes and compare the results.

Depending on the use case and data at hand, we also try different variations of moving average like *centered moving average, double moving average,* and so on apart from different window sizes. We will utilize some of these concepts when we deal with actual use cases in the coming sections of the chapter.

Exponential Smoothing

Moving average based smoothening is effective yet it is a pretty simple preprocessing technique. In case of moving average, all past observations in the window are given equal weight. Unlike the previous method, exponential smoothening techniques apply exponentially decreasing weights to older observations. In simple words, exponential smoothening methods give more weight to recent past observations as compared to older observations. Depending on the level of smoothening required, there may be one or more smoothening parameters to set in case of exponential smoothening.

Exponential smoothening is also called *exponentially weighted moving average* or EWMA for short. Single exponential smoothening is one of the simplest to get started with. The general formula is given as.

$$E_t = \alpha y_{t-1} + (1-\alpha) E_{t-1}$$

Where, E_t is the t^{th} smoothened observation, y is the actual observed value at t-1 instance, and α is smoothing constant between 0 and 1. There are different methods to bootstrap the value of E_2 (the time period from which smoothing begins). It can be done by setting it to y_1 or an average of first n time periods and so on. Also, the value of α determines how much of the past is accounted for. A value closer to 1 dampens out the past observations quickly while values closer to 0 dampen out slowly. The following snippet uses the pandas `ewm()` function to calculate the smoothened series for visits. The parameter `halflife` is used to calculate α in this case.

```
In [4] : input_df['ewma'] = input_df['Visits'].ewm(halflife=3,
    ...:                                             ignore_na=False,
    ...:                                             min_periods=0,
    ...:                                             adjust=True).mean()
    ...:
    ...: plt.plot(input_df.Visits,'-',color='black',alpha=0.3)
    ...: plt.plot(input_df.ewma,color='g')
    ...: plt.title('Website Visit and Exponential Smoothening')
    ...: plt.legend()
    ...: plt.show()
```

The plot depicted in Figure 11-4 showcases the EWMA smoothened series along with the original one.

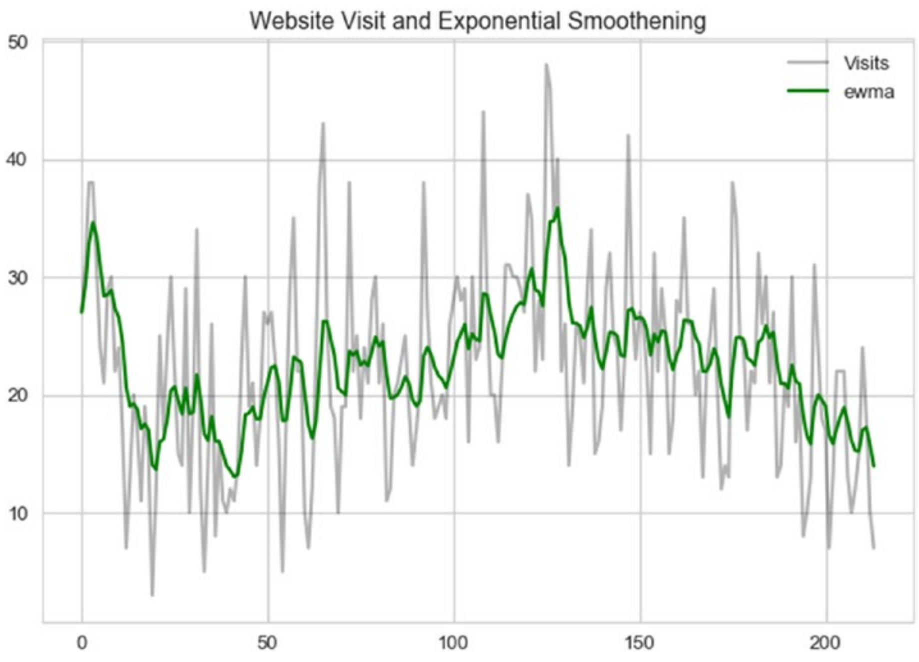


Figure 11-4. *Smoothening using EWMA*

In the coming sections, we will apply our understanding of time series, preprocessing techniques, and so on to solve stock and commodity price forecasting problems using different forecasting methods.

Forecasting Gold Price

Gold, the yellow shiny metal, has been the fancy of mankind since ages. From making jewelry to being used as an investment, gold covers a huge spectrum of use cases. Gold, like other metals, is also traded on the commodities indexes across the world. For better understanding time series in a real-world scenario, we will work with gold prices collected historically and predict its future value. Let's begin by first formally stating the problem statement.

Problem Statement

Metals such as gold have been traded for years across the world. Prices of gold are determined and used for trading the metal on commodity exchanges on a daily basis using a variety of factors. Using this daily price-level information *only*, our task is to predict future price of gold.

Dataset

For any problem, first and foremost is the data. Stock and Commodity exchanges do a wonderful job of storing and sharing daily level pricing data. For the purpose of this use case, we will utilize gold pricing from Quandl. Quandl is a platform for financial, economic, and alternative datasets. You can refer to the jupyter notebook `notebook_gold_forecast_arima.ipynb` for the necessary code snippets and examples.

To access publicly shared datasets on Quandl, we can use the `pandas-datareader` library as well as `quandl` (library from Quandl itself). For this use case, we will depend upon `quandl`. Kindly install the same using `pip` or `conda`. The following snippet shows a quick one-liner to get your hands on gold pricing information since 1970s.

```
In [5]: import quandl
   ...: gold_df = quandl.get("BUNDESBANK/BBK01_WT5511", end_date="2017-07-31")
```

The `get()` function takes the stock/commodity identifier as first parameter followed by the date until which we need the data. Note that not all datasets are public, for some of them, API access must be obtained.

Traditional Approaches

Time series analysis and forecasting have been long studied in detail. There are matured and extensive set of modeling techniques available for the same. Out of the many, the following are a few most commonly used and explored techniques:

- Simple moving average and exponential smoothing based forecasting
- Holt's, Holt-Winter's Exponential Smoothing based forecasting
- Box-Jenkins methodology (AR, MA, ARIMA, S-ARIMA, etc.)

■ **Note** Causal or cross-sectional forecasting/modeling is where the target variable has a relationship with one or more predictor variables, example regression models (see Chapter 6). Time series forecasting is about forecasting variable(s) that is changing over time. Both these techniques are grouped under quantitative techniques.

As mentioned, there are quite a handful of techniques available, each a deep topic of research and study. For the scope of this section and chapter, we will focus upon *ARIMA* models (from the Box-Jenkin's methodology) to forecast gold prices. Before we move ahead and discuss ARIMA, let's look at a few key concepts.

Key Concepts

- **Stationarity**: One the key assumptions behind the ARIMA models we will be discussing next. Stationarity refers to the property where for a time series its mean, variance, and autocorrelation are time invariant. In other words, mean, variance, and autocorrelation do not change with time. For instance, a time series having an upward (or downward) trend is a clear indicator of a non-stationarity because its mean would change with time (see web site visit data example in the previous section).
- **Differencing**: One of the methods of stationarizing series. Though there can be other transformations, differencing is widely used to stabilize the mean of a time series. We simply compute difference between consecutive observations to obtain a differenced series. We can then apply different tests to confirm if the resulting series is stationary or not. We can also perform *second order differencing, seasonal differencing,* and so on, depending on the time series at hand.
- **Unit Root Tests**: Statistical tests that help us understand if a given series is stationary or not. *The Augmented Dickey Fuller* test begins with a null hypothesis of series being non-stationary, while *Kwiatkowski-Phillips-Schmidt-Shin* test or KPSS has a null hypothesis that the series is stationary. We then perform a regression fit to reject or fail to reject the null hypothesis.

ARIMA

The Box-Jenkin's methodology consists of a wide range of statistical models which are widely used to model time series for forecasting. For this section, we will be concentrating on one such model called as ARIMA.

ARIMA stands for *Auto Regressive Integrated Moving Average* model. Sounds pretty complex, right? Let's look at the basics and constituents of this model and then build on our understanding to forecast gold prices.

- **Auto Regressive or AR Modeling**: A simple linear regression model where current observation is regressed upon one or more prior observations. The model is denoted as.

$$X_t = \delta + \theta_1 X_{t-1} + \ldots + \theta_p X_{t-p} + \varepsilon_t$$

where, X_t is the observation at time t, ε_t is the noise and

$$\delta = \left(1 - \sum_{i=1}^{p} \theta_i\right)\mu$$

the dependency on prior values is denoted by p or the order of AR model.

- **Moving Average or MA Modeling:** Is again essentially a linear regression model that models the impact of noise/error from prior observations to current one. The model is denoted as

$$X_t = \mu + \varepsilon_t - \varphi_1 \varepsilon_{t-1} + \ldots + \varphi_q \varepsilon_{t-q}$$

where, μ is the series mean, ε_i are the noise terms, and q is the order of the model.

The AR and MA models were known long before Box-Jenkin's methodology was presented. Yet this methodology presented a systematic approach to identify and apply these models for forecasting.

■ **Note** Box, Jenkins, and Reinsel presented this methodology in their book titled *Time Series Analysis: Forecasting and Control.* You are encouraged to go through it for a deeper understanding.

The ARIMA model is a logical progression and combination of the two models. Yet if we combine AR and MA with a differenced series, what we get is called as *ARIMA(p,d,q)* model.
where,

- **p** is the order of Autoregression
- **q** is the order of Moving average
- **d** is the order of differencing

Thus, for a stationary time series ARIMA models combine autoregressive and moving average concepts to model the behavior of a long running time series and helps in forecasting. Let's now apply these concepts to model gold price forecasting.

Modeling

While describing the dataset, we extracted the gold price information using `quandl`. Let's first plot and see how this time series looks. The following snippet uses the `pandas` to plot the same.

```
In [6]: gold_df.plot(figsize=(15, 6))
   ...: plt.show()
```

The plot in Figure 11-5 shows a general upward trend with sudden rise in the 1980s and then near 2010.

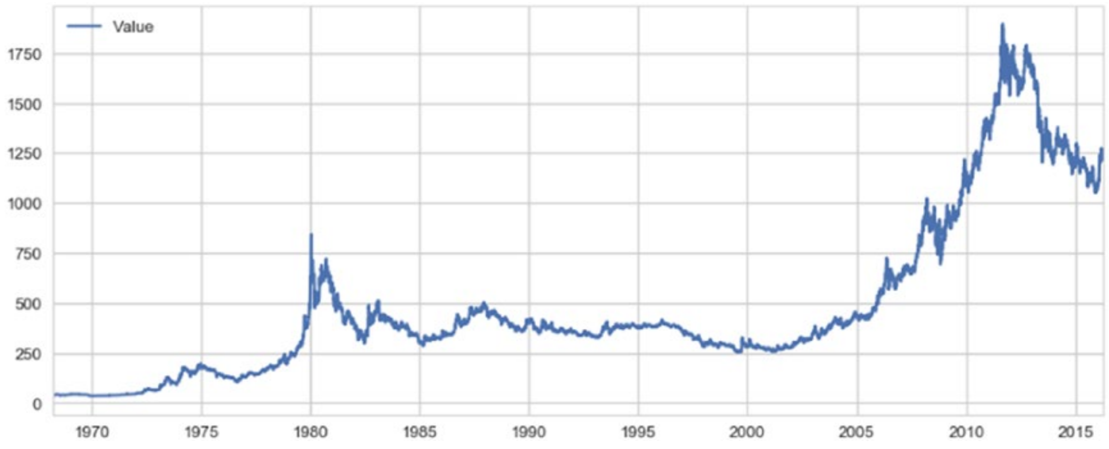


Figure 11-5. *Gold prices over the years*

Since stationarity is one of the primary assumptions of ARIMA models, we will utilize *Augmented Dickey Fuller* test to check our series for *stationarity*. The following snippet helps us calculate the AD Fuller test statistics and plot rolling characteristics of the series.

```
In [7]: # Dickey Fuller test for Stationarity
   ...: def ad_fuller_test(ts):
   ...: dftest = adfuller(ts, autolag='AIC')
   ...: dfoutput = pd.Series(dftest[0:4], index=['Test Statistic',
   ...:                                                 'p-value',
   ...:                                                 '#Lags Used',
   ...:                                                 'Number of Observations Used'])
   ...: for key,value in dftest[4].items():
   ...:         dfoutput['Critical Value (%s)'%key] = value
   ...: print(dfoutput)
   ...:
   ...: # Plot rolling stats for a time series
   ...: def plot_rolling_stats(ts):
   ...: rolling_mean = ts.rolling(window=12,center=False).mean()
   ...: rolling_std = ts.rolling(window=12,center=False).std()
   ...:
   ...: #Plot rolling statistics:
   ...: orig = plt.plot(ts, color='blue',label='Original')
   ...: mean = plt.plot(rolling_mean, color='red', label='Rolling Mean')
   ...: std = plt.plot(rolling_std, color='black', label = 'Rolling Std')
   ...: plt.legend(loc='best')
   ...: plt.title('Rolling Mean & Standard Deviation')
   ...: plt.show(block=False)
```

If the test statistic of AD Fuller test is less than the critical value(s), we reject the null hypothesis of non-stationarity. The AD Fuller test is available as part of the `statsmodel` library. Since it is quite evident that our original series of gold prices is non-stationary, we will perform a log transformation and see if we are able to obtain stationarity. The following snippet uses the rolling stats plots and AD Fuller tests to check the same.

```
In [8]: log_series = np.log(gold_df.Value)
   ...:
   ...: ad_fuller_test(log_series)
   ...: plot_rolling_stats(log_series)
```

The test statistic of -1.8 is greater than either of critical values, hence we fail to reject the null hypothesis, i.e., the series is non-stationary even after log transformation. The output and plot depicted in Figure 11-6 confirm the same.

```
Test Statistic                    -1.849748
p-value                            0.356057
#Lags Used                        29.000000
Number of Observations Used    17520.000000
Critical Value (1%)               -3.430723
Critical Value (5%)               -2.861705
Critical Value (10%)              -2.566858
dtype: float64
```

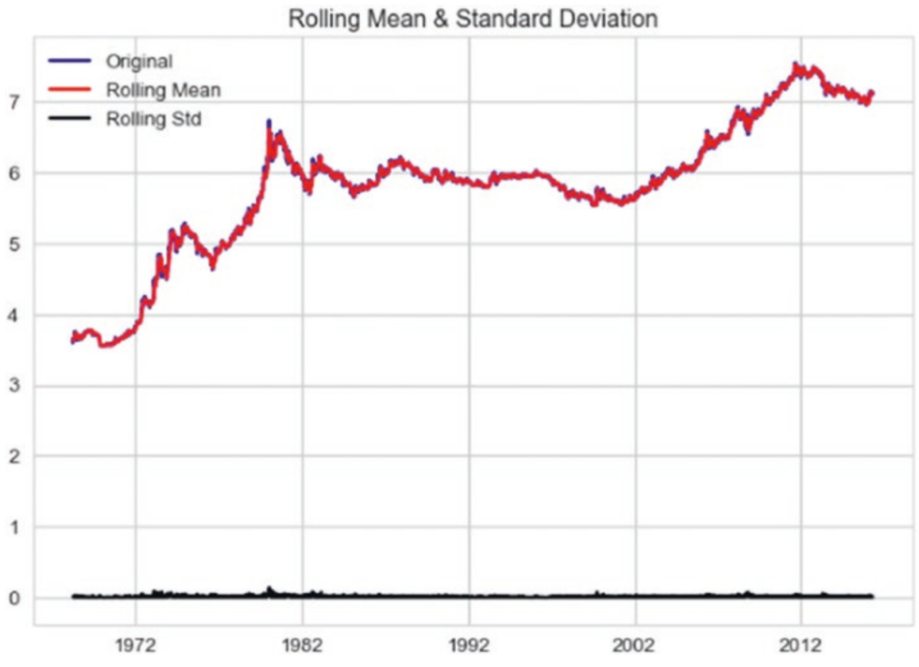


Figure 11-6. *Rolling mean and standard deviation plot for log transformed gold price*

The plot points out a time varying mean of the series and hence the non-stationarity. As discussed in the key concepts, differencing a series helps in achieving stationarity. In the following snippet, we prepare a first order differenced log series and perform the same tests.

```
In [9]: log_series_shift = log_series - log_series.shift()
   ...: log_series_shift = log_series_shift[~np.isnan(log_series_shift)]
   ...:
   ...: ad_fuller_test(log_series_shift)
   ...: plot_rolling_stats(log_series_shift)
```

The test statistic at -23.91 is lower than even 1% critical value, thus we reject the null hypothesis for AD Fuller test. The following are the test results.

```
Test Statistic                    -23.917175
p-value                             0.000000
#Lags Used                         28.000000
Number of Observations Used     17520.000000
Critical Value (1%)                -3.430723
Critical Value (5%)                -2.861705
Critical Value (10%)               -2.566858
dtype: float64
```

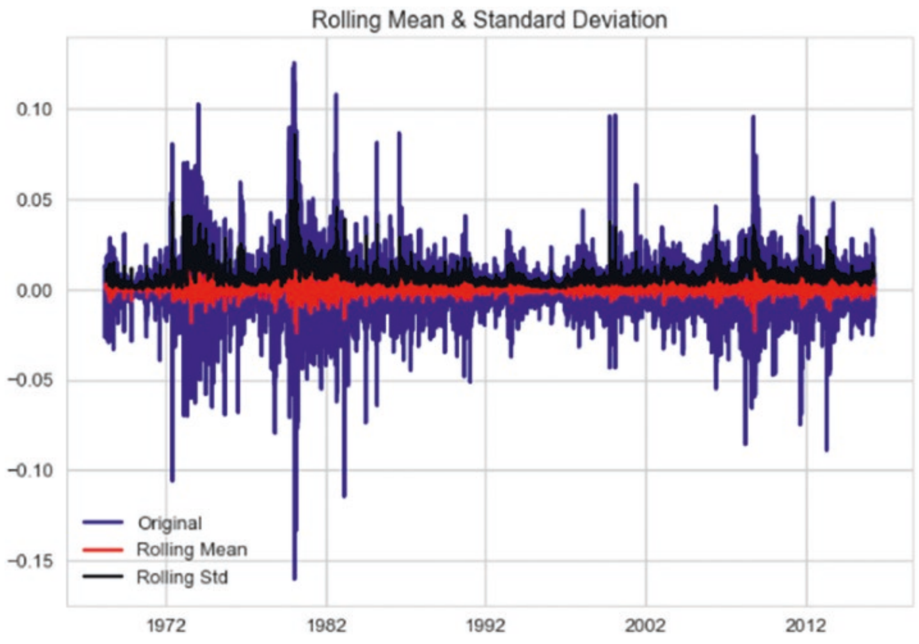


Figure 11-7. *Rolling mean and standard deviation plot for log differenced gold price series*

This exercise points us to the fact that we need to use a log differenced series for ARIMA to model the dataset at hand. See Figure 11-7. Yet we still need to figure out the order of autoregression and moving average components, i.e., ***p*** and ***q***.

Building an ARIMA model requires some experience and intuition as compared other models. Identifying *p, d,* and *q* parameters of the model can be done using different methods, though arriving at the right set of numbers is dependent both upon requirements and experience.

One of the commonly used methods is the plotting of ACF and PACF plots to determine *p* and q values. **ACF** or Auto Correlation Function plot and **PACF** or the Partial Auto Correlation Function plot helps us narrow down the search space of determining the p and q values with a few caveats. There are certain rules or heuristics developed over the years to best utilize these plots and thus these do not guarantee the best possible values.

The ACF plot helps us understand the correlation of an observation with its lag (or previous value). The ACF plot is used to determine the MA order, i.e. *q*. The value at which ACF drops is the order of the MA model.

On the same lines, PACF points toward correlation between an observation and a specific lagged value, excluding effect of other lags. The value at which PACF drops points toward the order of AR model or the *p* in ARIMA(p,d,q).

■ **Note** Further details on ACF and PACF is available at `http://www.itl.nist.gov/div898/handbook/eda/section3/autocopl.htm`.

Let's again utilize `statsmodels` to generate ACF and PACF plots for our series and try to determine *p* and *q* values. The following snippet uses the log differenced series to generate the required plots.

```
In [10]: fig = plt.figure(figsize=(12,8))
    ...: ax1 = fig.add_subplot(211)
    ...: fig = sm.graphics.tsa.plot_acf(log_series_shift.squeeze(), lags=40, ax=ax1)
    ...: ax2 = fig.add_subplot(212)
    ...: fig = sm.graphics.tsa.plot_pacf(log_series_shift, lags=40, ax=ax2)
```

The output plots (Figure 11-8) show a sudden drop at lag 1 for both ACF and PACF, thus pointing toward possible values of q and p to be 1 each, respectively.

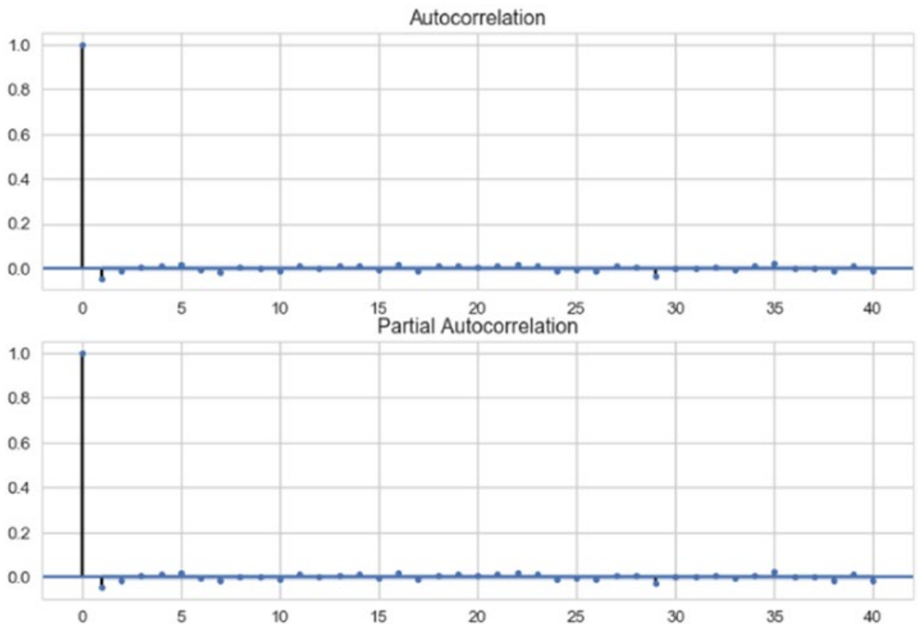


Figure 11-8. *ACF and PACF plots*

The ACF and PACF plot also help us understand if a series is stationary or not. If a series has gradually decreasing values for ACF and PACF, it points toward non-stationarity property in the series.

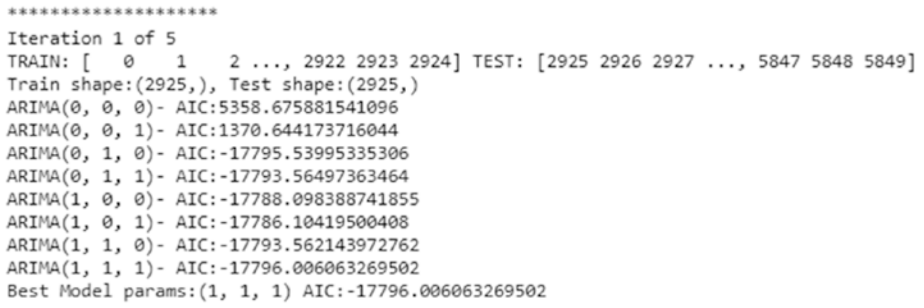
■ **Note** Identifying the p, d, and q values for any ARIMA model is as much science as it is art. More details on this are available at `https://people.duke.edu/~rnau/411arim.htm`.

Another method to derive the *p, d, q* parameters is to perform a grid search of the parameter space. This is more in tune with the Machine Learning way of hyperparameter tuning. Though `statsmodels` does not provide such a utility (for obvious reasons though), we can write our own utility to identify the best fitting model. Also, pretty much like any other Machine Learning/Data Science use case, we need to split our dataset into train and test sets. We utilize `scikit-learn`'s `TimeSeriesSplit` utility to help us get proper training and testing sets.

We write a utility function `arima_grid_search_cv()` to grid search and cross validate the results using the gold prices at hand. The function is available in `arima_utils.py` module for reference. The following snippet performs a five-fold cross validation with auto-ARIMA to find the best fitting model.

```
In [11]: results_dict = arima_gridsearch_cv(gold_df.log_series,cv_splits=5)
```

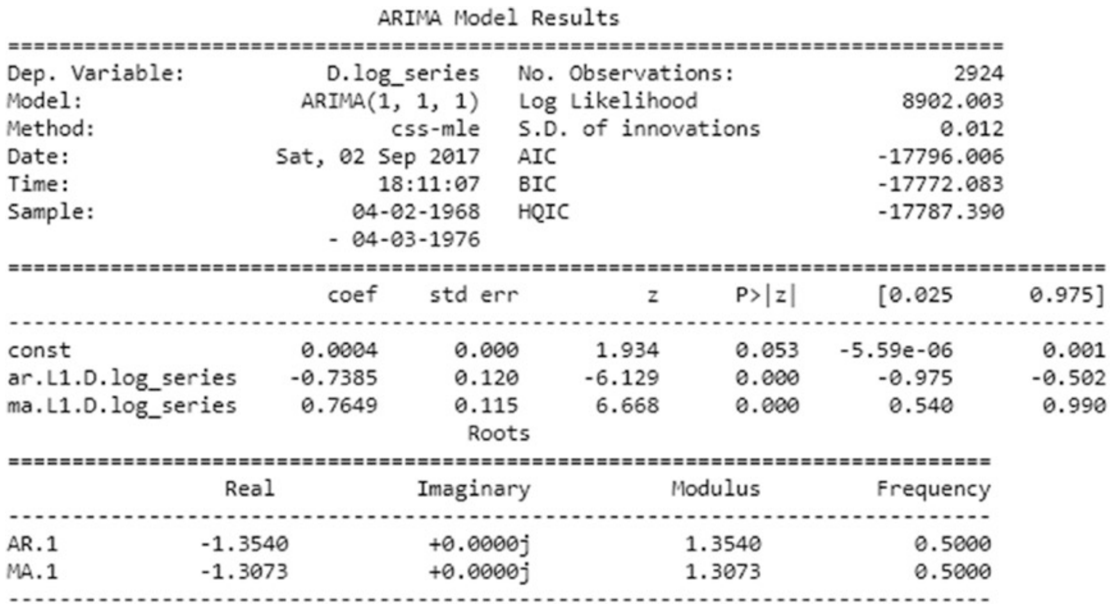
Note that we are passing the log transformed series as input to the `arima_gridsearch_cv()` function. As we saw earlier, the log differenced series was what helped us achieve stationarity, hence we use the log transformation as our starting point and fit an ARIMA model with d set to 1. The function call generates a detailed output for each train-test split (we have five of them lined up), each iteration performing a grid search over *p, d,* and *q*. Figure 11-9 shows the output of the first iteration, where the training set included only 2924 observations.



```
********************
Iteration 1 of 5
TRAIN: [   0    1    2 ..., 2922 2923 2924] TEST: [2925 2926 2927 ..., 5847 5848 5849]
Train shape:(2925,), Test shape:(2925,)
ARIMA(0, 0, 0)- AIC:5358.675881541096
ARIMA(0, 0, 1)- AIC:1370.644173716044
ARIMA(0, 1, 0)- AIC:-17795.53995335306
ARIMA(0, 1, 1)- AIC:-17793.56497363464
ARIMA(1, 0, 0)- AIC:-17788.098388741855
ARIMA(1, 0, 1)- AIC:-17786.10419500408
ARIMA(1, 1, 0)- AIC:-17793.562143972762
ARIMA(1, 1, 1)- AIC:-17796.006063269502
Best Model params:(1, 1, 1) AIC:-17796.006063269502
```

Figure 11-9. *Auto ARIMA*

Similar to our findings using ACF-PACF, auto ARIMA suggests that the best fitting model is ARIMA(1,1,1) based upon the AIC criteria. Note that AIC or *Akaike Information Criterion* measures the goodness of fit and parsimony. It is a relative metric and does not point toward quality of models in an absolute sense, i.e., if all models being compared are poor, AIC will not be able to point that out. Thus, AIC should be used as a heuristic. A low value points toward a better fitting model. The following is the summary generated by the ARIMA(1,1,1) fit, see Figure 11-10.



```
                             ARIMA Model Results
==============================================================================
Dep. Variable:            D.log_series   No. Observations:                 2924
Model:                  ARIMA(1, 1, 1)   Log Likelihood                8902.003
Method:                        css-mle   S.D. of innovations              0.012
Date:                 Sat, 02 Sep 2017   AIC                         -17796.006
Time:                         18:11:07   BIC                         -17772.083
Sample:                     04-02-1968   HQIC                        -17787.390
                          - 04-03-1976
=======================================================================================
                          coef    std err          z      P>|z|      [0.025      0.975]
---------------------------------------------------------------------------------------
const                   0.0004      0.000      1.934      0.053   -5.59e-06      0.001
ar.L1.D.log_series     -0.7385      0.120     -6.129      0.000      -0.975     -0.502
ma.L1.D.log_series      0.7649      0.115      6.668      0.000       0.540      0.990
                                    Roots
=============================================================================
                 Real           Imaginary           Modulus         Frequency
-----------------------------------------------------------------------------
AR.1          -1.3540           +0.0000j            1.3540            0.5000
MA.1          -1.3073           +0.0000j            1.3073            0.5000
-----------------------------------------------------------------------------
```

Figure 11-10. *Summary of ARIMA(1,1,1)*

The summary is quite self-explanatory. The top section shows details about the training sample, AIC, and other metrics. The middle section talks about the coefficients of the fitted mode. In case of ARIMA(1,1,1) for iteration 1, both AR and MA coefficients are statistically significant. The Forecast Plot for iteration 1 with ARIMA(1,1,1) fitted is shown in Figure 11-11.

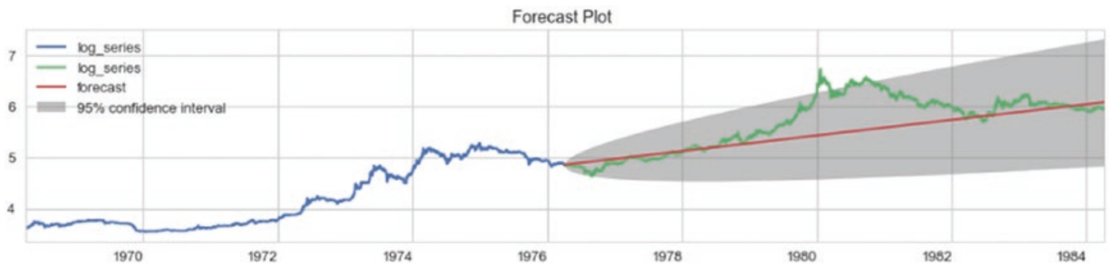


Figure 11-11. *The forecast plot for ARIMA(1,1,1)*

As evident from the plot in Figure 11-11, the model captures the overall upward trend though it misses out on the sudden jump in values around 1980. Yet it seems to give a pretty nice idea of what can be achieved using this methodology. The `arima_gridsearch_cv()` function produces similar statistics and plots for five different train-test splits. We observe that ARIMA(1,1,1) provides us decent enough fit, although we can define additional performance and error criteria to select a particular model.

In this case, we generated forecast for time periods for which we already had data. This helps us in visualizing and understanding how the model is performing. This is also called as *back testing*. Out of sample forecasting is also supported by `statsmodels` through its `forecast()` method. Also, the plot in Figure 11-11 showcases values in the transformed scale, i.e. log scale. Inverse transformation can be easily applied to get data back in original form.

You should also note that commodity prices are impacted by a whole lot of other factors like global demand, economic conditions like recession and so on. Hence, what we showcased here was in certain ways a naïve modeling of a complex process. We would need more features and attributes to have sophisticated forecasts.

Stock Price Prediction

Stocks and financial instrument trading is a lucrative proposition. Stock markets across the world facilitate such trades and thus wealth exchanges hands. Stock prices move up and down all the time and having ability to predict its movement has immense potential to make one rich.

Stock price prediction has kept people interested from a long time. There are hypothesis like the *Efficient Market Hypothesis,* which says that it is almost impossible to beat the market consistently and there are others which disagree with it.

There are a number of known approaches and new research going on to find the magic formula to make you rich. One of the traditional methods is the time series forecasting, which we saw in the previous section. *Fundamental analysis* is another method where numerous performance ratios are analyzed to assess a given stock. On the emerging front, there are *neural networks, genetic algorithms,* and *ensembling techniques*.

■ **Note** Stock price prediction (along with gold price prediction in the previous section) is an attempt to explain concepts and techniques to model real-world data and use cases. This chapter is by no means and extensive guide to algorithmic trading. *Algorithmic trading* is a complete field of study on its own and you may explore it further. Knowledge from this chapter alone would not be sufficient to perform trading of any sort and is beyond both the scope and intent of this book.

In this section, we learn how to apply *recurrent neural networks* (RNNs) to the problem of stock price prediction and understand the intricacies.

Problem Statement

Stock price prediction is the task of forecasting the future value of a given stock. Given the historical daily close price for S&P 500 Index, prepare and compare forecasting solutions.

S&P 500 or *Standard and Poor's 500* index is an index comprising of 500 stocks from different sectors of US economy and is an indicator of US equities. Other such indices are the Dow 30, NIFTY 50, Nikkei 225, etc. For the purpose of understanding, we are utilizing S&P500 index, concepts, and knowledge can be applied to other stocks as well.

Dataset

Similar to gold price dataset, the historical stock price information is also publicly available. For our current use case, we will utilize the `pandas_datareader` library to get the required S&P 500 index history using Yahoo Finance databases. We will utilize the *closing price* information from the dataset available though other information such as opening price, adjusted closing price, etc., are also available.

We prepare a utility function `get_raw_data()` to extract required information in a `pandas` dataframe. The function takes index ticker name as input. For S&P 500 index, the ticker name is ^*GSPC*. The following snippet uses the utility function to get the required data.

```
In [1]: sp_df = get_raw_data('^GSPC')
   ...: sp_close_series = sp_df.Close
   ...: sp_close_series.plot()
```

The plot for closing price is depicted in Figure 11-12.

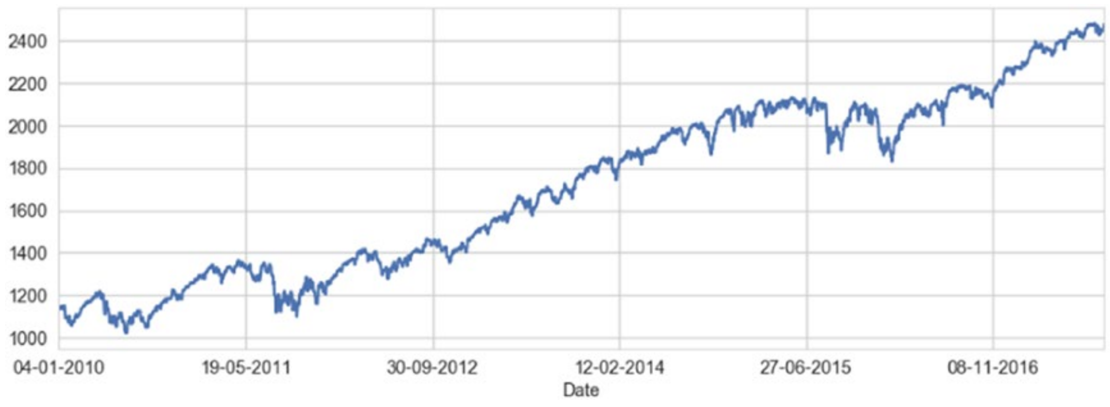


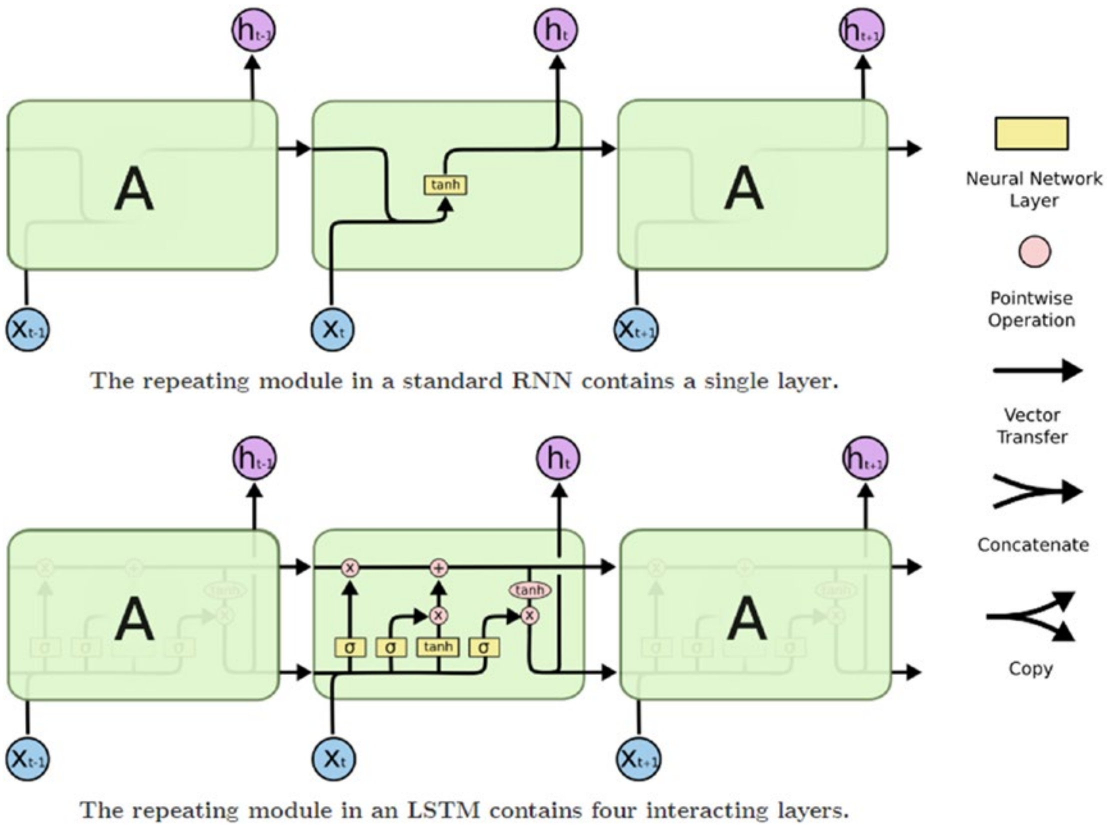
Figure 11-12. *The S&P 500 index*

The plot in Figure 11-12 shows that we have closing price information available since 2010 up until recently.

Kindly note that the same information is also available through `quandl` (which we used in the previous section to get gold price information). You may use the same to get this data as well.

Recurrent Neural Networks: LSTM

Artificial neural networks are being employed to solve a number of use cases in a variety of domains. Recurrent neural networks are a class of neural networks with capabilities of modeling sequential data. LSTMs or *Long Short Term Memory* is an RNN architecture that's useful for modeling arbitrary intervals of information. RNNs, particularly LSTMs were discussed in Chapter 1 while a practical use case was explored in Chapter 7 for analyzing textual data from movie reviews for sentiment analysis.



h_{t-1}
h_t
h_{t+1}
A
tanh
A
X_{t-1}
X_t
X_{t+1}
The repeating module in a standard RNN contains a single layer.
σ
σ
tanh
σ
tanh
The repeating module in an LSTM contains four interacting layers.
Neural Network Layer
Pointwise Operation
Vector Transfer
Concatenate
Copy

Figure 11-13. *Basic structure of RNN and LSTM units. (Source: Christopher Olah's blog: colah.github.io)*

As a quick refresher, Figure 11-13 points toward the general architecture of an RNN along with the internals of a typical LSTM unit. LSTM comprises of three major gates—the *input, output, and forget* gates. These gates work in tandem to learn and store long and short term sequence related information. For more details, refer to *Advanced Supervised Deep Learning Models, Chapter 7*.

For stock price prediction, we will utilize LSTM units to implement an RNN model. RNNs are typically useful in sequence modeling applications; some of them are as follows:

- **Sequence classification** tasks like sentiment analysis of a given corpus (see Chapter 7 for the detailed use case)
- **Sequence tagging** tasks like POS tagging a given sentence
- **Sequence mapping** tasks like speech recognition

Unlike traditional forecasting approaches (like ARIMA), which require preprocessing of time series information to conform to stationarity and other assumptions along with parameter identification (***p, d, q,*** for instance), neural networks (particularly RNNs) impose far fewer restrictions.

Since stock price information is also a time series data, we will explore the application of LSTMs to this use case and generate forecasts. There are a number of ways this problem can be modeled to forecast values. The following sections covers two such approaches.

Regression Modeling

We introduced regression modeling in Chapter 6 to analyze bike demand based on certain predictor variables. In essence, regression modeling refers to the process of investigating relationship between dependent and independent variables.

To model our current use case as a regression problem, we state that the stock price at timestamp t+1 (dependent variable) is a function of stock price at timestamps t, t -1, t -2, ..., t -n. Where n is the past window of stock prices.

Now that we have a framework defined on how we would model our time series, we need to transform our time series data into windowed form. See Figure 11-14.

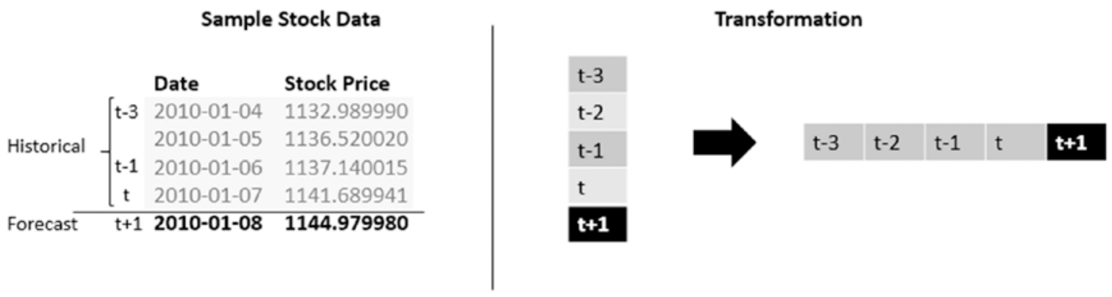


Figure 11-14. *Transformation of stock price time series into windowed format*

The windowed transformation is outlined in Figure 11-15 where a window size of 4 is used. The value at time t+1 is forecasted using past four values. We have data for the S&P 500 index since 2010, hence we would apply this windowed transformation in a rolling fashion and create multiple such sequences. Thus, if we have a time series of length *M* and a window size of *n,* there would be *M-n-1* total windows generated.

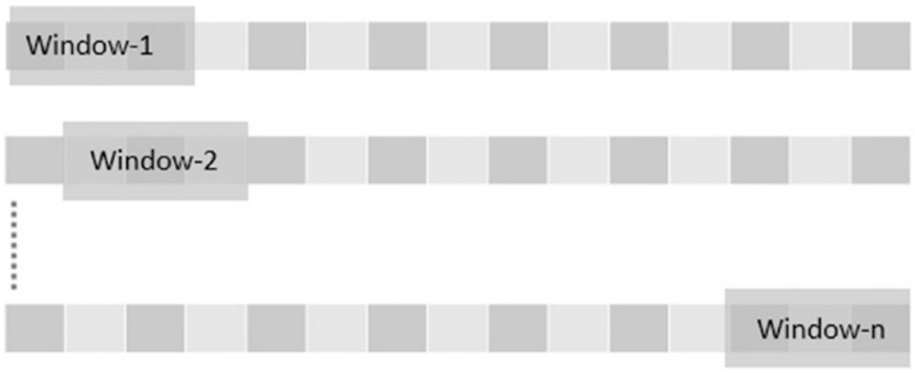


***Figure 11-15.** Rolling/sliding windows from original time series*

For the hands-on examples in this section, you can refer to the jupyter notebook `notebook_stock_prediction_regression_modeling_lstm.ipynb` for the necessary code snippets and examples. For using LSTMs to model our time series, we need to apply one more level of transformation to be able to input our data. LSTMs accept 3D tensors as input, we transform each of the windows (or sequences) in **(N, W, F)** format. Here, ***N*** is the *number of samples* or windows from the original time series, **W** is the *size of each window* or the number of historical time steps and ***F*** is the *number of features* per time step. In our case, as we are only using the closing price, ***F*** is equal to 1, ***N*** and ***W*** are configurable. The following function performs the windowing and 3D tensor transformations using `pandas` and `numpy`.

```
def get_reg_train_test(timeseries,sequence_length= 51,
                   train_size=0.9,roll_mean_window=5,
                   normalize=True,scale=False):
    # smoothen out series
    if roll_mean_window:
        timeseries = timeseries.rolling(roll_mean_window).mean().dropna()

    # create windows
    result = []
    for index in range(len(timeseries) - sequence_length):
        result.append(timeseries[index: index + sequence_length])

    # normalize data as a variation of 0th index
    if normalize:
        normalised_data = []
        for window in result:
            normalised_window = [((float(p) / float(window[0])) - 1) \
                                   for p in window]
            normalised_data.append(normalised_window)
        result = normalised_data

    # identify train-test splits
    result = np.array(result)
    row = round(train_size * result.shape[0])

    # split train and test sets
    train = result[:int(row), :]
    test = result[int(row):, :]
    # scale data in 0-1 range
    scaler = None
    if scale:
        scaler=MinMaxScaler(feature_range=(0, 1))
        train = scaler.fit_transform(train)
        test = scaler.transform(test)

    # split independent and dependent variables
    x_train = train[:, :-1]
    y_train = train[:, -1]

    x_test = test[:, :-1]
    y_test = test[:, -1]

    # Transforms for LSTM input
    x_train = np.reshape(x_train, (x_train.shape[0],
                                   x_train.shape[1],
                                   1))
    x_test = np.reshape(x_test, (x_test.shape[0],
                                 x_test.shape[1],
                                 1))

    return x_train,y_train,x_test,y_test,scaler
```

The function get_reg_train_test() also performs a number of other optional preprocessing steps. It allows us to *smoothen* the time series using rolling mean before the windowing is applied. We can also *normalize* the data as well as *scale* based on requirements. Neural networks are sensitive to input values and it is generally advised to *scale* inputs before training the network. For this use case, we will utilize the *normalization* of the time series wherein, for each window, every time step is the percentage change from the first value in that window (we could also use scaling or both and repeat the process).

For our case, we begin with a window size of six days (you can experiment with smaller or larger windows and observe the difference). The following snippet uses the get_reg_train_test() function with normalization set to true.

```
In [2] : WINDOW = 6
    ...: PRED_LENGTH = int(window/2)
    ...: x_train,y_train,x_test,y_test,scaler = get_reg_train_test(sp_close_series,
    ...:                                                          sequence_length=WINDOW +1,
    ...:                                                          roll_mean_window=None,
    ...:                                                          normalize=True,
    ...:                                                          scale=False)
```

This snippet creates a seven-day window that is comprised of six days of historical data (x_train) and one-day forecast y_train. The shapes of the train and test variables are as follows.

```
In [3] : print("x_train shape={}".format(x_train.shape))
    ...: print("y_train shape={}".format(y_train.shape))
    ...: print("x_test shape={}".format(x_test.shape))
    ...: print("y_test shape={}".format(y_test.shape))
```

```
x_train shape=(2516, 6, 1)
y_train shape=(2516,)
x_test shape=(280, 6, 1)
y_test shape=(280,)
```

The `x_train` and `x_test` tensors conform to the **(N, W, F)** format we discussed earlier and is required for input to our RNN. We have 2516 sequences in our training set, each with six time steps and one value to forecast. Similarly, we have 280 sequences in our test set.

Now that we have our datasets preprocessed and ready, we build up an RNN network using `keras`. The `keras` framework provides us high level abstractions to work with neural networks over `theano` and `tensorflow` backends. The following snippet showcases the model prepared using the `get_reg_model()` function.

```
In [4]: lstm_model = get_reg_model(layer_units=[50,100],
   ...:                            window_size=window)
```

The generated LSTM model architecture has two hidden `LSTM` layers stacked over each other with first one having 50 LSTM units and the second one having 100. The output layer is a `Dense` layer with linear activation function. We use *mean squared error* as our loss function to optimize upon. Since we are stacking LSTM layers, we need to set `return_sequences` to true in order for the subsequent layer to get the required values. As is evident, `keras` abstracts most of the heavy lifting and makes it pretty intuitive to build even complex architectures with just a few lines of code.

The next step is to train our LSTM network. We use batch size of 16 with 20 epochs and a validation set of 5%. The following snippet uses the `fit()` function to train the model.

```
In [5]: # use early stopping to avoid overfitting
   ...: callbacks = [keras.callbacks.EarlyStopping(monitor='val_loss',
   ...:                                            patience=2,
   ...:                                            verbose=0)]
   ...: lstm_model.fit(x_train, y_train,
   ...:                 epochs=20, batch_size=16,
   ...:                 verbose=1,validation_split=0.05,
   ...:                 callbacks=callbacks)
```

The model generates information regarding training and validation loss for every epoch it runs. The callback for stopping enables us to stop the training if there is no further improvement observed for two consecutive epochs. We start with a batch size of 16; you may experiment with larger batch sizes and observe the difference.

Once the model is fit, the next step is to forecast using the `predict()` function. Since we have modeled this as a regression problem with a fixed window size, we would generate forecasts for every sequence. To do so, we write another utility function called `predict_reg_multiple()`. This function takes the *lstm* model, windowed dataset, window and prediction lengths as input parameters to return a list of predictions for every input window. The `predict_reg_multiple()` function works as follows.

1. For every sequence in the list of windowed sequences, repeat Steps a-c:
    a. Use keras's `predict()` function to generate one output value.
    b. Append this output value to the end of the input sequence and remove the first value to maintain the window size.
    c. Then repeat this process (Steps a and b) until the required prediction length is achieved.
2. The function utilizes predicted values to forecast subsequent ones.

The function is available in the script `lstm_utils.py`. The following snippet uses the `predict_reg_multiple()` function to get predictions on the test set.

```
In [6] : test_pred_seqs = predict_reg_multiple(lstm_model,
    ...:                                        x_test,
    ...:                                        window_size=WINDOW,
    ...:                                        prediction_len=PRED_LENGTH)
```

To analyze the performance, we will calculate the RMSE for the fitted sequence. We use sklearn's `metrics` module for the same. The following snippet calculates the RMSE score.

```
In [7] : test_rmse = math.sqrt(mean_squared_error(y_test[1:],
    ...:                                                np.array(test_pred_seqs).\
    ...:                                                flatten()))
    ...: print('Test Score: %.2f RMSE' % (test_rmse))
Test Score: 0.01 RMSE
```

The output is an RMSE of 0.01. As an exercise, you may compare RMSE with different window sizes and prediction lengths and observe the overall model performance.

To visualize our predictions, we plot the predictions against the normalized testing data. We use the function `plot_reg_results()`, which is also available for reference in `lstm_utils.py`. The following snippet generates the required plot using the same function.

```
In [8]: plot_reg_results(test_pred_seqs,y_test,prediction_len=PRED_LENGTH)
```

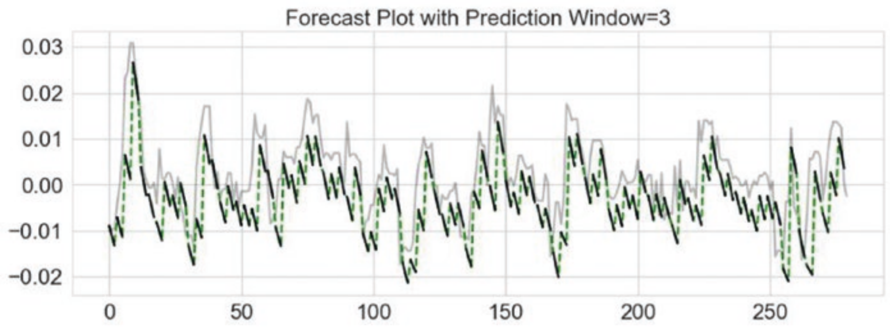


Figure 11-16. *Forecast Plot with LSTM, window size 6 and prediction length 3*

In Figure 11-16, the gray line is the original/true test data (normalized) and the black lines denote the predicted/forecast values in three-day periods. The dotted line is used to explain the overall flow of the predicted series. As is evident, the forecasts are off the mark to some extent from the actual data trends yet they seem to have some similarity to the actual data.

Before we conclude this section, there are a few important points to be kept in mind. LSTMs are vastly powerful units with memory to store and use past information. In our current scenario, we utilized a windowed approach with a stacked LSTM architecture (two LSTM layers in our model). This is also termed as *many-to-one* architecture where multiple input values are used to generate single output. Another important point here is that the window size along with other hyperparameters of the network (like epochs, batch size, LSTM units, etc.) have an impact on the final results (this is left as an exercise for you to explore). Thus, we should be careful before deploying such models in production.

Sequence Modeling

In the previous section we modeled our time series like a regression use case. In essence, the problem formulation though utilized past window to forecast, it did not use the time step information. In this section, we will solve the same stock prediction problem using LSTMs by modeling it as a sequence. For the hands-on examples in this section, you can refer to the jupyter notebook `notebook_stock_prediction_sequence_modeling_lstm.ipynb` for the necessary code snippets and examples.

Recurrent neural networks are naturally suited for sequence modeling tasks like machine translation, speech recognition and so on. RNNs utilize memory (unlike normal feed forward neural networks) to keep track of context and utilize the same to generate outputs. In general feed forward neural networks assume inputs are independent of each other. This independence may not hold in many scenarios (such as time series data). RNNs apply same transformations to each element of the sequence, with outcomes being dependent upon previous values.

In case of our stock price time series, we would like to model it now as sequence where value at each time step is a function of previous values. Unlike the regression-like modeling, here we do not divide the time series into windows of fixed sizes, rather we would utilize the LSTMs to learn from the data and determine which past values to utilize for forecasting.

To do so, we need to perform certain tweaks to the way we processed our data in the previous case and also how we built our RNN. In the previous section, we utilized the **(N,W,F)** format as input. In the current setting, the format remain the same with the following changes.

- **N (number of sequences)**: This will be set to 1 since we are dealing with only one stock's price information.
- **W (length of sequence)**: This will be set to total number of days worth of price information we have with us. Here we use the whole series as one big sequence.
- **F (features per timestamp)**: This is again 1, as we are only dealing with closing stock value per timestamp.

Talking about the output, in the previous section we had one output for every window/sequence in consideration. While modeling our data as a sequence/time series we expect our output to be a sequence as well. Thus, the output is also a 3D tensor following the same format as the input tensor.

We write a small utility function `get_seq_train_test()` to help us scale and generate train and test datasets out of our time series. We use a 70-30 split in this case. We then use `numpy` to reshape our time series into 3D tensors. The following snippet utilizes the `get_seq_train_test()` function to do the same.

```
In [1]: train,test,scaler = get_seq_train_test(sp_close_series,
   ...:                                        scaling=True,
   ...:                                        train_size=TRAIN_PERCENT)
   ...:
   ...: train = np.reshape(train,(1,train.shape[0],1))
   ...: test = np.reshape(test,(1,test.shape[0],1))
   ...:
   ...: train_x = train[:,:-1,:]
   ...: train_y = train[:,1:,:]
   ...:
   ...: test_x = test[:,:-1,:]
   ...: test_y = test[:,1:,:]
   ...:
   ...: print("Data Split Complete")
   ...:
   ...: print("train_x shape={}".format(train_x.shape))
   ...: print("train_y shape={}".format(train_y.shape))
   ...: print("test_x shape={}".format(test_x.shape))
   ...: print("test_y shape={}".format(test_y.shape))
Data Split Complete
train_x shape=(1, 1964, 1)
train_y shape=(1, 1964, 1)
test_x shape=(1, 842, 1)
test_y shape=(1, 842, 1)
```

Having prepared the datasets, let we'll now move onto setting up the RNN network. Since we are planning on generating a sequence as output as opposed to a single output in the previous case, we need to tweak our network architecture.

The requirement in this case is to apply similar transformations/processing for every time step and be able to get output for every input timestamp rather than waiting for the whole sequence to be processed. To enable such scenarios, keras provides a wrapper over dense layers called TimeDistributed. This wrapper applies the same task to every time step and provides hooks to get output after each such time step. We use TimeDistributed wrapper over Dense layer to get output from each of the time steps being processed. The following snippet showcases get_seq_model() function to generate the required model.

```
def get_seq_model(hidden_units=4,input_shape=(1,1),verbose=False):
    # create and fit the LSTM network
    model = Sequential()

    # input shape = timesteps*features
    model.add(LSTM(input_shape=input_shape,
                   units = hidden_units,
                   return_sequences=True
    ))

    # TimeDistributedDense uses the processing for all time steps.
    model.add(TimeDistributed(Dense(1)))
    start = time.time()

    model.compile(loss="mse", optimizer="rmsprop")

    if verbose:
        print("> Compilation Time : ", time.time() - start)
        print(model.summary())

    return model
```

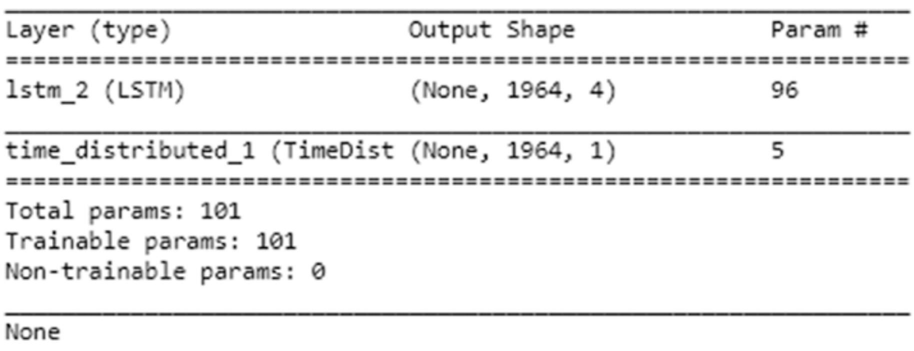
This function returns a single hidden layer RNN network with four LSTM units and a TimeDistributed Dense output layer. We again use *mean squared error* as our loss function.

■ **Note** TimeDistributed is a powerful yet tricky utility available through keras. You may explore more on this at https://github.com/fchollet/keras/issues/1029 and https://datascience.stackexchange.com/questions/10836/the-difference-between-dense-and-timedistributeddense-of-keras.

We have our dataset preprocessed and split into train and test along with a model object using the function `get_seq_model()`. The next step is to simply train the model using the `fit()` function. While modeling stock price information as a sequence, we are assuming the whole time series as one big sequence. Hence, while training the model, we set the batch size as 1 as there is only one stock to train in this case. The following snippet gets the model object and then trains the same using the `fit()` function.

```
In [2]: # get the model
   ...: seq_lstm_model = get_seq_model(input_shape=(train_x.shape[1],1),
   ...:                                verbose=VERBOSE)
   ...:
   ...: # train the model
   ...: seq_lstm_model.fit(train_x, train_y,
   ...:                        epochs=150, batch_size=1,
   ...:                        verbose=2)
```

This snippet returns a model object along with its summary. We also see the output of each of the 150 epochs while the model trains on the training data.



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_2 (LSTM)                (None, 1964, 4)           96
_________________________________________________________________
time_distributed_1 (TimeDist (None, 1964, 1)           5
=================================================================
Total params: 101
Trainable params: 101
Non-trainable params: 0
_________________________________________________________________
None
```

Figure 11-17. *RNN Summary*

Figure 11-17 shows the total parameters which the RNN tries to learn, a complete 101 of them. We urge you to explore the summary on the model prepared in the previous section, the results should surprise most (Hint: this model has far few parameters to learn!). This summary also points toward an important fact, the shape of the first LSTM layer. This clearly shows that the model expects the inputs to adhere to this shape (the shape of the training dataset) for training as well as predicting.

Since our test dataset is smaller (`shape: (1,842,1)`), we need some way to match the required shape. While modeling sequences with RNNs, it is a common practice to pad sequences in order to match a given shape. Usually in cases where there are multiple sequences to train upon (example, text generation), the size of the longest sequence is used and the shorter ones are padded to match it. We do so only for programmatic reasons and discard the padded values otherwise (see `keras` masking for more on this). The padding utility is available from the `keras.preprocessing.sequence` module. The following snippet pads the test dataset with 0s post the actual data (you can choose between pre-pad and post pad) and then uses the padded sequence to predict/forecast. We also calculate and print the RMSE score of the forecast.

```
In [3]: # Pad input sequence
   ...: testPredict = pad_sequences(test_x,
   ...:                                maxlen=train_x.shape[1],
   ...:                                padding='post',
   ...:                                dtype='float64')
   ...:
   ...: # forecast values
   ...: testPredict = seq_lstm_model.predict(testPredict)
   ...:
   ...: # evaluate performance
   ...: testScore = math.sqrt(mean_squared_error(test_y[0],
   ...:                         testPredict[0][:test_x.shape[1]]))
   ...: print('Test Score: %.2f RMSE' % (testScore))
Test Score: 0.07 RMSE
```

We can perform the same steps on the training set as well and check the performance. While generating the train and test datasets, the function `get_seq_train_test()` also returned the scaler object. We next use this scaler object to perform an inverse transformation to get the prediction values in the original scale. The following snippet performs inverse transformation and then plots the series.

```
In [4]: # inverse transformation
   ...: trainPredict = scaler.inverse_transform(trainPredict.\
   ...:                                         reshape(trainPredict.shape[1]))
   ...: testPredict = scaler.inverse_transform(testPredict.\
   ...:                                         reshape(testPredict.shape[1]))
   ...:
   ...: train_size = len(trainPredict)+1
   ...:
   ...: # plot the true and forecasted values
   ...: plt.plot(sp_close_series.index,
   ...:                 sp_close_series.values,c='black',
   ...:                 alpha=0.3,label='True Data')
   ...:
   ...: plt.plot(sp_close_series.index[1:train_size],
   ...:                 trainPredict,
   ...:                 label='Training Fit',c='g')
   ...:
   ...: plt.plot(sp_close_series.index[train_size+1:],
   ...:                 testPredict[:test_x.shape[1]],
   ...:                 label='Forecast')
   ...: plt.title('Forecast Plot')
   ...: plt.legend()
   ...: plt.show()
```

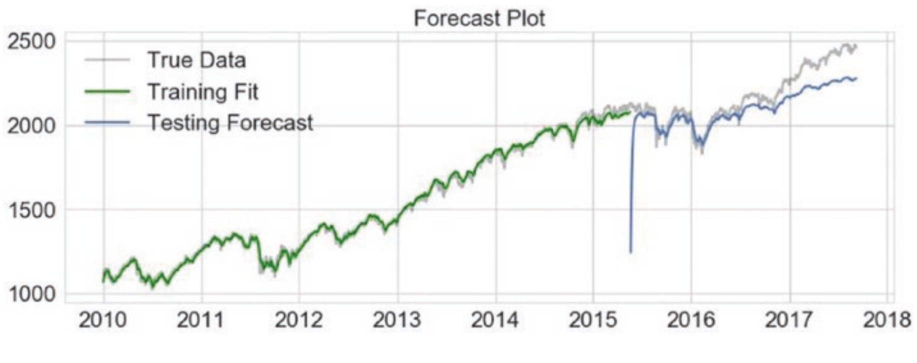


Figure 11-18. *Forecast for S&P 500 using LSTM based sequence modeling*

The forecast plot in Figure 11-18 shows a promising picture. We can see that the training fit is nearly perfect which is kind of expected. The testing performance or the forecast also shows decent performance. Even though the forecast deviates from the actual at places, the overall performance both in terms of RMSE and the fit seemed to have worked.

Through the use of `TimeDistributed` layer wrapper we achieved the goal of modeling this data as a time series. The model not just had a better performance in terms of overall fit, it required far less feature engineering and a much simpler model (in terms of number of training parameters). In this model, we also truly utilized the power of LSTMs by allowing it to learn and figure out what and how much of past information impacts the forecast (as compared to regression modeling case where we had restricted the window sizes).

Two important points before we conclude this section. First, both the models have their own advantages and disadvantages. The aim of this section was to chalk out potential ways of modeling a given problem. The actual usage mostly depends upon the requirements of the use case. Secondly and more importantly, either of the models is for learning/demonstration purposes. Actual stock price forecasting requires far more rigor and knowledge; we just scraped the tip of the iceberg.

Upcoming Techniques: Prophet

The Data Science landscape is ever evolving and new algorithms, tweaks and tools are coming up at a rapid pace. One such tool is called Prophet. This is a framework, open sourced by Facebook's Data Science team for analyzing and forecasting time series.

Prophet uses an additive model that can work with trending and seasonal data. The aim of this tool is to enable forecasting at scale. This is still in beta, yet has some really useful features. More on this is available at `https://facebookincubator.github.io/prophet/`. The research and intuition behind this tool is available in the paper available at `https://facebookincubator.github.io/prophet/static/prophet_paper_20170113.pdf`.

The installation steps are outlined on the web site and are straightforward through `pip` and `conda`. Prophet also uses `scikit` style APIs of `fit()` and `predict()` with additional utilities to better handle time series data. For the hands-on examples in this section, you can refer to the jupyter notebook `notebook_stock_prediction_fbprophet.ipynb` for the necessary code snippets and examples.

■ **Note** Prophet is still in beta and is undergoing changes. Also, its installation on Windows platform is known to cause issues. Kindly use `conda install` (steps mentioned on the web site) with Anaconda distribution to avoid issues.

Since we already have the S&P 500 index price information available in a dataframe/series. We now test how we can use this tool to forecast. We begin with converting the time series index into a column of its own (simply how prophet expects the data) followed by splitting the series into training and testing (90-10 split). The following snippet performs the required actions.

```
In [1] : # reset index to get date_time as a column
    ...: prophet_df = sp_df.reset_index()
    ...:
    ...: # prepare the required dataframe
    ...: prophet_df.rename(columns={'index':'ds','Close':'y'},inplace=True)
    ...: prophet_df = prophet_df[['ds','y']]
    ...:
```

```
    ...: # prepare train and test sets
    ...: train_size = int(prophet_df.shape[0]*0.9)
    ...: train_df = prophet_df.ix[:train_size]
    ...: test_df = prophet_df.ix[train_size+1:]
```

Once we have the datasets prepared, we create an object of the `Prophet` class and simply fit the model using `fit()` function. Kindly note that the model expects the time series value to be in a column named 'y' and timestamp in column named 'ds'. To make forecasts, prophet requires the set of dates for which we need to forecast. For this, it provides a clean utility called the `make_future_dataframe()`, which takes the number of days required for the forecast as input. The following snippet uses this dataframe to forecast values.

```
In [2] : # prepare a future dataframe
    ...: test_dates = pro_model.make_future_dataframe(periods=test_df.shape[0])
    ...:
    ...: # forecast values
    ...: forecast_df = pro_model.predict(test_dates)
```

The output from the `predict()` function is a dataframe that includes both in-sample predictions as well as forecasted values. The dataframe also includes the confidence interval values. All of this can be easily plotted using the `plot()` function of the model object. The following snippet plots the forecasted values against the original time series along with its confidence intervals.

```
In [3] : # plot against true data
    ...: plt.plot(forecast_df.yhat,c='r',label='Forecast')
    ...: plt.plot(forecast_df.yhat_lower.iloc[train_size+1:],
    ...:                 linestyle='--',c='b',alpha=0.3,
    ...:                 label='Confidence Interval')
    ...: plt.plot(forecast_df.yhat_upper.iloc[train_size+1:],
    ...:                 linestyle='--',c='b',alpha=0.3,
    ...:                 label='Confidence Interval')
    ...: plt.plot(prophet_df.y,c='g',label='True Data')
    ...: plt.legend()
    ...: plt.title('Prophet Model Forecast Against True Data')
    ...: plt.show()
```

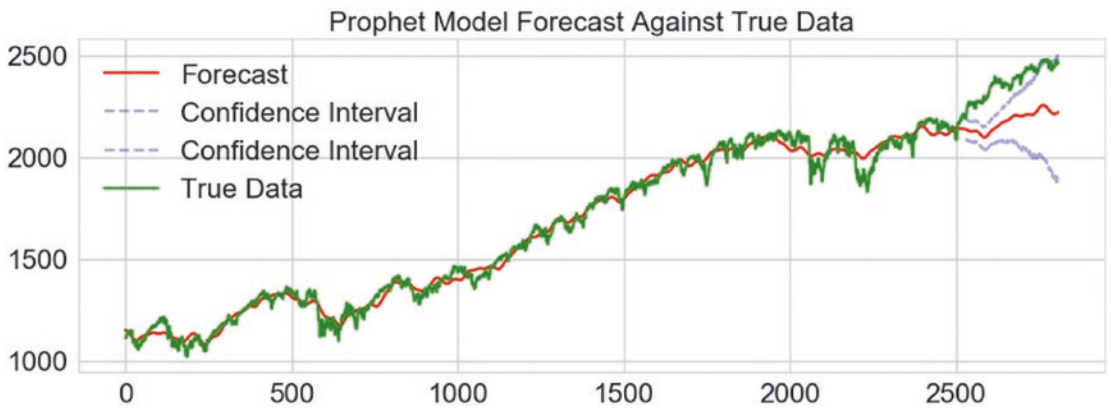


Figure 11-19. *Forecasts from prophet against true/observed values*

The model's forecasts are a bit off the mark (see Figure 11-19), but the exercise clearly demonstrates the possibilities here. The forecast dataframe provides even more details about seasonality, weekly trends, and so on. You are encouraged to explore this further. `Prophet` is based upon *Stan*. *Stan* is statistical modeling language/framework that provides algorithms exposed through interfaces for all major languages, including `python`. You may explore more on this at `http://mc-stan.org/`.

Summary

This chapter introduced the concepts of time series forecasting and analysis using stock and commodity price information. Through this chapter we covered the basic components of a time series along with common techniques for preprocessing such data. We then worked on the gold price prediction use case. This use case utilized the `quandl` library to get daily gold price information. We then discussed traditional time series analysis techniques and introduced key concepts related to Box-Jenkin's methodology and ARIMA in particular. We also discussed techniques for identification and transformation of non-stationary time series into one using AD Fuller tests, ACF and PACF plots. We modeled the gold price information using ARIMA based on statsmodel APIs while developing some key utility functions like `auto_arima()` and `arima_gridsearch_cv()`. Key insights and caveats were also discussed. The next section of the chapter introduced the stock price prediction use case. Here, we utilized `pandas_datareader` to get S&P 500 daily closing price information.

To solve this use case, we utilized RNN based models. Primarily we provided two alternative perspectives of formulating the forecasting problem, both using LSTMs. The first formulation closely imitated the regression concepts discussed in earlier chapters. A two-layer stacked LSTM network was used to forecast stock price information. The second perspective utilized `TimeDistributed` layer wrapper from `keras` to enable sequence modeling of the stock price information. Various utilities and key concepts were discussed while working on the use case. Finally, an upcoming tool (still in beta), `Prophet` from Facebook was discussed. The tool is made available by Facebook's Data Science team to perform forecasting at scale. We utilized the framework to quickly evaluate its performance on the same stock price information and shared the results. A multitude of techniques and concepts were introduced in this chapter, along with the intuition on how to formulate certain time series problems. Stay tuned for some more exciting use cases in the next chapter.