8
Forecasting Strategies

So far, we have mainly been focusing on the first two components of the time series analysis workflow—data preprocessing and descriptive analysis. Starting from this chapter, we will shift gear and move on to the third and last component of the analysis—the forecast. Before we dive into different forecasting models in the upcoming chapters, we will introduce the main elements of the forecasting workflow. This includes approaches for training a forecasting model, performance evaluation, and benchmark methods. This will provide you with a set of tools for designing and building a forecasting model according to the goal of the analysis.

This chapter covers the following topics:

- Training and testing approaches for a forecasting model
- Performance evaluation methods and error measurement matrices
- Benchmark methods
- Quantifying forecast uncertainty with confidence intervals and simulation

Technical requirement

The following packages will be used in this chapter:

- **forecast**: Version 8.5 and above
- **TSstudio**: Version 0.1.4 and above
- **plotly**: Version 4.8 and above

You can access the codes for this chapter from the following link:

`https://github.com/PacktPublishing/Hands-On-Time-Series-Analysis-with-R/tree/master/Chapter08`

The forecasting workflow

Traditional time series forecasting follows the same workflow as most of the fields of predictive analysis, such as regression or classification, and typically includes the following steps:

1. **Data preparation**: Here, we prepare the data for the training and testing process of the model. This step includes splitting the series into training (in-sample) and testing (out-sample) partitions, creating new features (when applicable), and applying a transformation if needed (for example, log transformation, scaling, and so on).
2. **Train the model**: Here, we used the training partition to train a statistical model. The main goal of this step is to utilize the training set to train, tune, and estimate the model coefficients that minimize the selected error criteria (later on in this chapter, we will discuss common error metrics in detail). The fitted values and the model estimation of the training partition observations will be used later on to evaluate the overall performance of the model.
3. **Test the model**: Here, we utilize the trained model to forecast the corresponding observations of the testing partition. The main idea here is to evaluate the performance of the model with a new dataset (that the model did not *see* during the training process).
4. **Model evaluation**: Last but not least, after the model was trained and tested, it is time to evaluate the overall performance of the model on both the training and testing partitions.

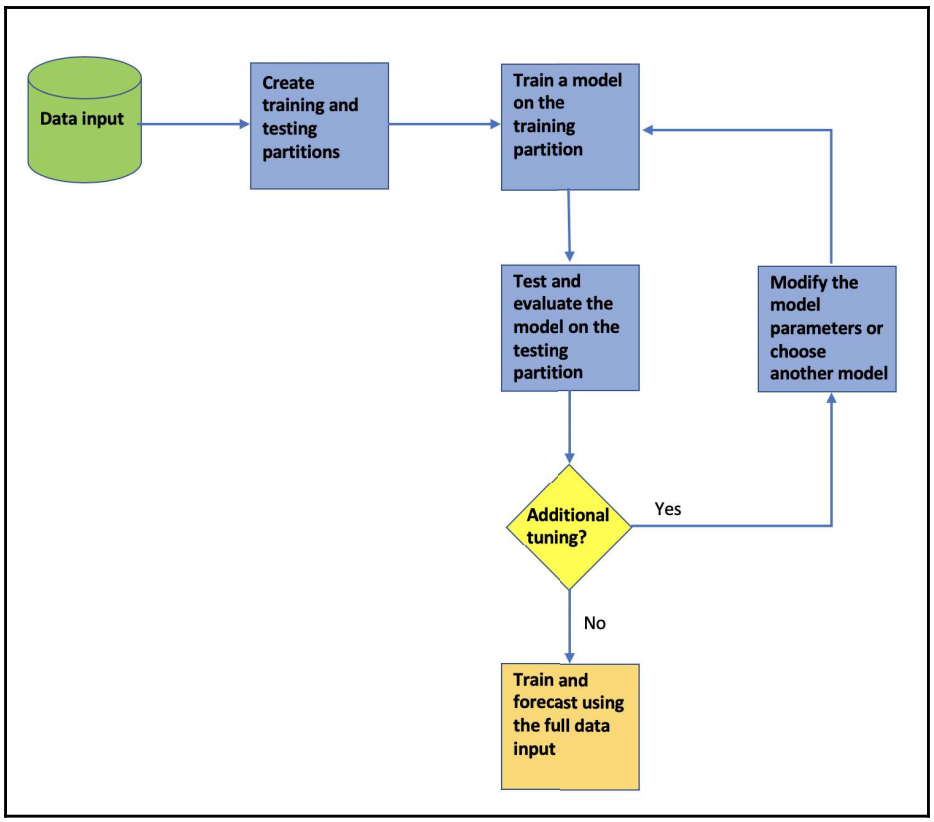
Based on the evaluation process of the model, if the model meets a certain threshold or criteria, then we either retain the model using the full series in order to generate the final forecast or select a new training parameter/different model and repeat the training process.

One of the main pitfalls when training a statistical model is overfitting the model to the training set. Overfitting occurs when the model is *overlearning* the data that the model trained with and fails to generalize the model's performance on the training set on other datasets. One of the main signs for overfitting is a high ratio between the error rate on the testing set and training set (for example, a high error on the training partition versus a low error on the training partition). Therefore, the evaluation of the model's performance on both the training and testing partitions is essential to identify overfitting. Generally, overfitting is caused by incorrectly tuning the model parameters or by using a model with high complexity.

On the other hand, this process has its own unique characteristics, which distinguish it from other predictive fields:

- The training and testing partitions must be ordered in chronological order, as opposed to random sampling.
- Typically, once we have trained and tested the model using the training and testing partitions, we will retrain the model with all of the data (or at least the most recent observation in chronological order). At first glance, this might be shocking and terrifying for people with a background in traditional machine learning or supervised learning modeling, as typically it leads to overfitting and other problems. We will discuss the reason behind this and how to avoid overfitting later.

The following diagram demonstrates the forecasting workflow:



Now, we will take a look at the different training approaches.

Training approaches

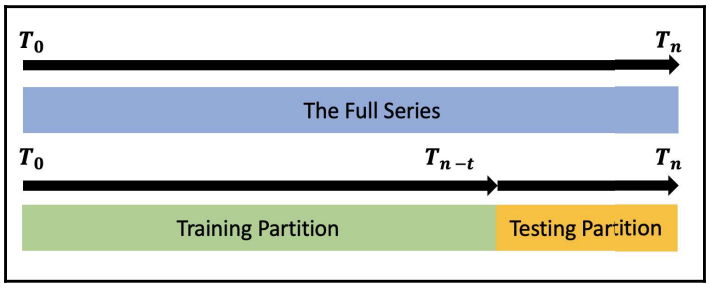
One of the core elements of the forecasting workflow is the model training process. The quality of the model's training will have a direct impact on the forecast output. The main goals of this process are as follows:

- Formalize the relationship of the series with other factors, such as seasonal and trend patterns, correlation with past lags, and external variables in a predictive manner
- Tune the model parameters (when applicable)
- The model is scalable on new data, or in other words, avoids overfitting

As we mentioned previously, prior to the training process, the series is split into training and testing partitions, where the model is being trained on the training partition and tested on the testing partition. These partitions must be in chronological order, regardless of the training approach that has been used. The main reason for this is that most of the time series models establish a mathematical relationship between the series in terms of its past lags and error terms.

Training with single training and testing partitions

One of the most common training approaches is using single training and testing (or single out-of-sample) partitions. This simplistic approach is based on splitting the series into training and testing partitions (or in-sample and out-sample partitions, respectively), training the model on the training partition, and testing its performance on the testing set:



As you can see in the preceding diagram, the training and testing partitions are ordered and organized in chronological order. This approach has a single parameter—the length of the out-of-sample (or the length of the testing partition). Typically, the length of the testing partition is derived from the following rules of thumb:

- The length of the testing partition should be up to 30% of the total length of the series in order to have enough observation data for the training process.
- The length of the forecasting horizon (as long it is not violating the previous term). This is mainly related to the fact that the level of uncertainty increases as the forecast horizon increases. Therefore, aligning the testing set to the forecast horizon could, potentially, provide a closer estimate of the forecast's expected error.

For example, if we have a monthly series with 72 observations (or 6 years) and the goal is to forecast the next year (or 12 months), it would make sense to use the first 60 observations for training and test the performance using the last 12 observations. Creating partitions in R can be done manually with the `window` function from the **stats** package. For instance, let's split the `USgas` series into partitions, leaving the last 12 observations of the series as the testing partition and the rest as training:

1. First, let's load the `USgas` series from the **TSstudio** package into the environment:

```
library(TSstudio)

data(USgas)
```

2. We can observe the main characteristics of the `USgas` series with the `ts_info` function:

```
ts_info(USgas)
```

We get the following output:

```
##  The USgas series is a ts object with 1 variable and 227
observations
##  Frequency: 12
##  Start time: 2000 1
##  End time: 2018 11
```

3. Let's use the `window` function to split the series into training and testing partitions:

```
train <- window(USgas,
                start = time(USgas)[1],
```

```
                 end = time(USgas)[length(USgas) - 12])

test <- window(USgas,
               start = time(USgas)[length(USgas) - 12 + 1],
               end = time(USgas)[length(USgas)])
```

4. The summary of the testing and testing partition can be seen in the following output:

```
ts_info(train)
```

We get the following output:

```
##  The train series is a ts object with 1 variable and 215
observations
##  Frequency: 12
##  Start time: 2000 1
##  End time: 2017 11
```

5. Now, let's take a look at the `test` partition:

```
ts_info(test)
```

We get the following output:

```
##  The test series is a ts object with 1 variable and 12
observations
##  Frequency: 12
##  Start time: 2017 12
##  End time: 2018 11
```

6. Alternatively, the `ts_split` function from the **TSstudio** package provides a customized way for creating training and testing partitions for time series data:

```
# The sample.out argument set the size of the testing partition
(and therefore the training partition)
USgas_partitions <- ts_split(USgas, sample.out = 12)

train <- USgas_partitions$train
test <- USgas_partitions$test
```

7. You can observe from the following output that we received the same execute results that we received previously:

```
ts_info(train)
```

We get the following output:

```
##  The train series is a ts object with 1 variable and 215
observations
##  Frequency: 12
##  Start time: 2000 1
##  End time: 2017 11
```

8. Now, let's look at the `test` partition:

```
ts_info(test)
```

We get the following output:

```
##  The test series is a ts object with 1 variable and 12
observations
##  Frequency: 12
##  Start time: 2017 12
##  End time: 2018 11
```

The simplicity of this method is its main advantage, as it is fairly fast to train and test a model while using (relatively) *cheap* compute power. On the other hand, it isn't possible to come to a conclusion about the stability and scalability of the model's performance based on a single test unit. It is feasible that a model, only by chance, will have relatively good performance on the testing set but do poorly on the actual forecast as it isn't stable over time. One way to mitigate that risk is with the backtesting approach, which is based on training a model with multiple training and testing partitions.

Forecasting with backtesting

The backtesting approach for training a forecasting model is an advanced version of the single out-of-sample approach we saw previously. It is based on the use of a rolling window to split the series into multiple pairs of training and testing partitions. A basic backtesting training process includes the following steps:

1. **Data preparation**: Create multiple pairs of training and testing partitions.
2. **Train a model**: This is done on each one of the training partitions.
3. **Test the model**: Score its performance on the corresponding testing partitions.

4. **Evaluate the model**: Evaluate the model's accuracy, scalability, and stability based on the testing score. Based on the evaluation, you would do one of the following:
    - Generate the final forecast to check whether the model score meets a specific threshold or criteria
    - Apply additional tuning and optimization for the model and repeat the training and evaluations steps

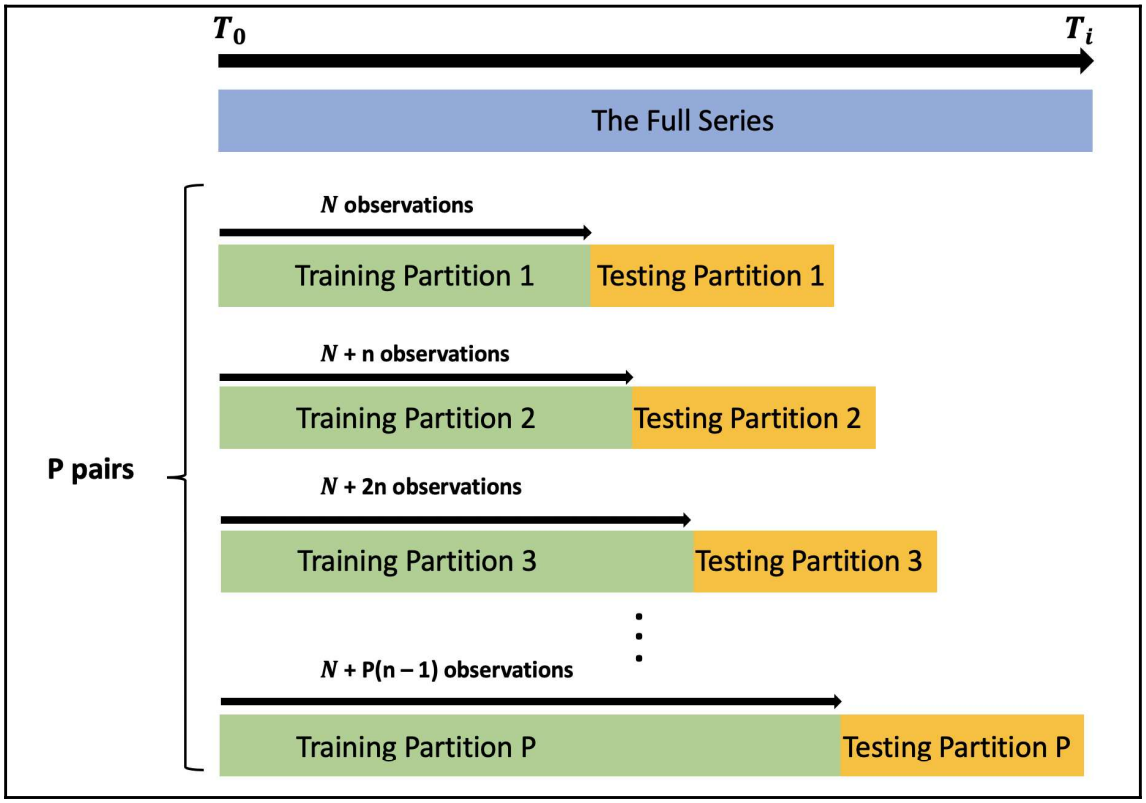
The use of scoring methodology allows us to assess the model's stability by examining the model's error rate on the different testing sets. We would consider a model as stable whenever the model's error distribution on the testing sets is fairly narrow. In this case, the error rate of the actual forecast should be within the same range of the testing sets (assuming there are no abnormal events that impact the forecast error rate).

This method is, conceptually, similar to the cross-validation approach for training machine learning models. The main distinction between the two approaches is related to how their partitions are set. While the backtesting partitions are ordered chronologically, the ones of the cross-validation approach are based on random sampling.

The main setting parameters of a backtesting training model are as follows:

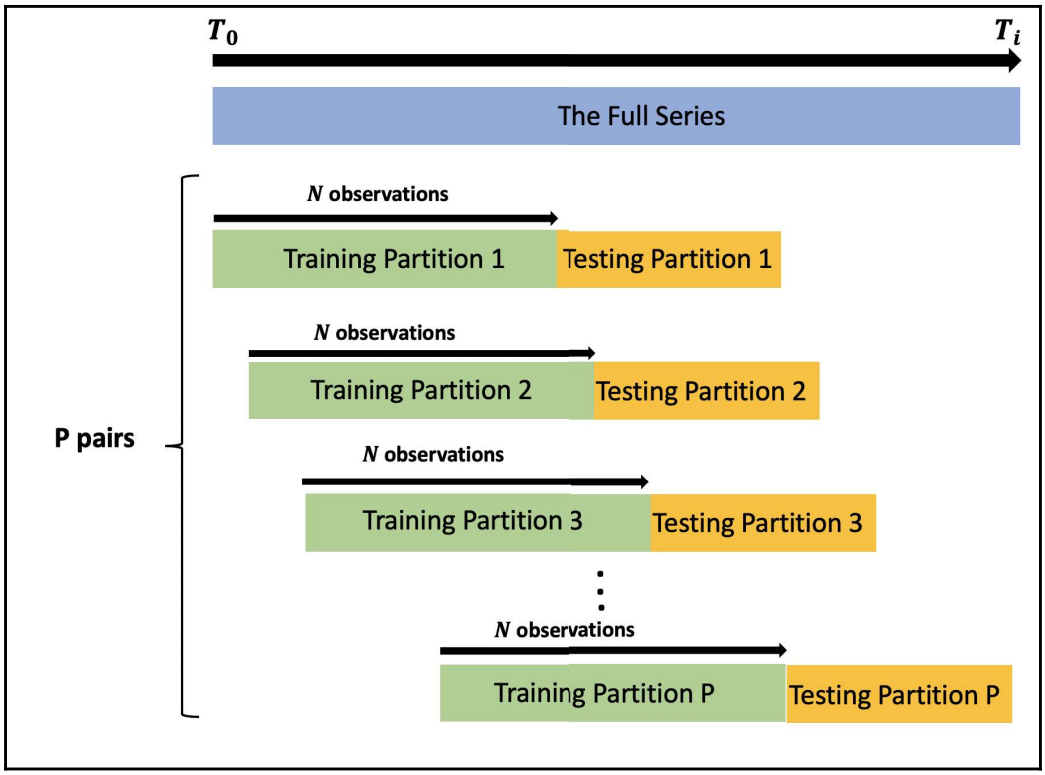
- **The length of the training partitions**: It is set by the window setting. There are two common types of rolling windows:
    - **Expanding window**: Using the first *N* observations as the initial training partition, add the following *n* observations to create the following training partition. The length of the training partition grows as the window rolls over the series.
    - **Sliding window**: Set the first *N* observations as the initial training partition and shift the window by *n* observations to create the following training partition. The length of the training partitions remains the same as the window rolls over the series:
        - The length of the testing partitions—a constant value, typically aligned with the length of the forecasting horizon (under the limitation of the minimum number of observations required to train the model)
        - The space between each training partition—defines the pace of the rolling window
        - The number of training and testing partitions

The following diagram demonstrates the structure of the backtesting with an expanding window:



As you can see in the preceding diagram, all of the training partitions of the expanding window method start at the same index point, T_0, where each training partition is a combination of the previous training partition with the following *n* observations (where *n* is a constant that represents the expanded rate of the window). It would make sense to use this method when the series has a strong seasonal pattern and stable trend. In this case, the first observations of the series could, potentially, have predictive information that can be utilized by the model. The downside of this method is training the model with different length partitions, as typically a model tends to *learn* more and therefore perform better when more data is available. Therefore, we may observe that the performance of models that are trained on the latest partitions perform better than the ones that are trained with the first partitions.

This bias does not exist in the second training approach—the sliding window, as all of the training partitions, are of the same length, as you can see in the following diagram:



It would make more sense to use the sliding window whenever the input series has structural change or high volatility, or when most of the predictive power is linked to the most recent history (or high correlation with the most recent lags of the series). For example, as we saw in the previous chapter, monthly crude oil prices have a strong relationship with the most recent lags of the series, and the far history doesn't contain powerful predictive information about the series' future direction.

While the backtesting approach provides us with intuitive information about the stability and scalability of the model, it comes with a higher computation cost compared to the single training and testing partition approach. Hence, this trade-off between the two approaches should be taken into consideration when selecting the training approach.

Forecast evaluation

The primary goal of the evaluation step is to assess the ability of the trained model to forecast (or based on other criteria) the future observations of the series accurately. This process includes doing the following:

- **Residual analysis**: This focuses on the quality of the model, with fitted values in the training partition
- **Scoring the forecast**: This is based on the ability of the model to forecast the actual values of the testing set

Residual analysis

Residual analysis tests how well the model captured and identified the series patterns. In addition, it provides information about the residuals distributions, which are required to build confidence intervals for the forecast. The mathematical definition of a residual is the difference between the actual observation and the corresponding fitted value of the model, which was observed in the training process, or as the following equation:

$$\epsilon_t = Y_t - \hat{Y}_t$$

Here, ϵ_t, Y_t, and $\hat{Y}_t$ represent the residual, actual, and fitted values, respectively, at time t.

This process includes the use of data visualizations tools and statistical test to assess the following:

- **Test the goodness of the fit against the actual values**: You do this by plotting the residuals values over time in chronological order. The plot indicates how well the model was able to capture the oscillation of the series in the training partition. Residuals with random oscillation around the zero and with constant variation (white noise) indicate that the model is able to capture the majority of the series variation. On the other hand, if the residual oscillation does not have the white noise characteristics, it is an indication that the model failed to capture the series patterns. Here are some potential interpretations of the possible output:
    - **All or most of the residuals are above the zero lines**: This is an indication that the model tends to underestimate the actual values
    - **All or most of the residuals are below the zero lines**: This is an indication that the model tends to overestimate the actual values

- **Random spikes**: This is an indication for potential outliers in the training partition:
    - **The residual autocorrelation**: This indicates how well the model was able to capture the patterns of the series. Non-correlated lags indicate that there were no patterns that the model did not capture. Similarly, the existence of correlated lags indicates patterns that the model did not capture.
    - **The residuals distribution**: This is required to conclude about the reliability of the forecast confidence interval. If the residuals are not normally distributed, we cannot use it to create confidence intervals as it is based on the assumption that the residuals are normally distributed.

- To demonstrate the residual analysis process, we will train an ARIMA model on the training partition we created earlier for the `USgas` series. Don't worry if you're not familiar with the ARIMA model—we will discuss it in detail in `Chapter 11`, *Forecasting with the ARIMA Model*. We will train the model with the `auto.arima` function from the **forecast** package:

```
library(forecast)

 md <- auto.arima(train)
```

To examine the residuals, we will use the `checkresiduals` function from the **forecast** package, which returns the following four outputs:

- Time series plot of the residuals
- ACF plot of the residuals
- Distribution plot of the residuals
- The output of the Ljung-Box test

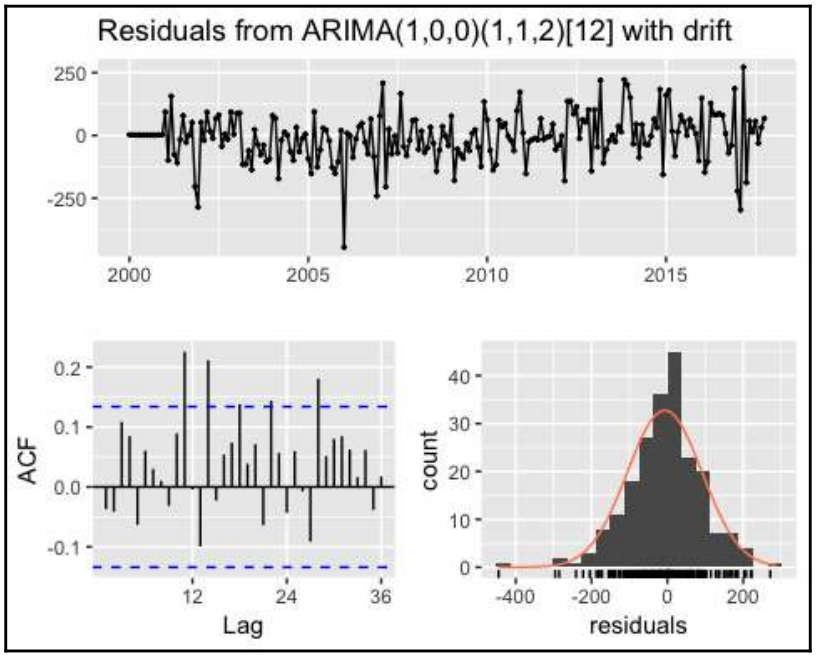
The Ljung-Box test is a statistical method for testing whether the autocorrelation of a series (which, in this case, is the residuals) is different from zero and uses the following hypothesis:

- H_0: The level of correlation between the series and its lag is equal to zero, and therefore the series observations are independent
- H_1: The level of correlation between the series and its lag is different from zero, and therefore the series observations are not independent

Let's use the `checkresiduals` function to evaluate the trained model's performance on the training partition:

```
checkresiduals(md)
```

We get the following residuals from the ARIMA model:



The output is as follows:

```
##
##  Ljung-Box test
##
## data:  Residuals from ARIMA(1,0,0)(1,1,2)[12] with drift
## Q* = 48.345, df = 19, p-value = 0.0002287
##
## Model df: 5.   Total lags used: 24
```

Starting with the output of the Ljung-Box test output, you will notice that, based on the *P*-value results, we can reject the null hypothesis with a level of significate of 0.01. Hence, there is an indication that the correlation between the residual series and its lags are different from zero. The ACP plot provides additional support for that as well. This indicates that the model did not fully capture all of the series patterns, and you may want to modify the model tuning parameters. The residual time series plot oscillates around the *x*-axis, with the exception of a few residuals, which cross the value of $\pm$ 250. This could indicate that some outliers occur during these periods, and you should check those data points in the series (later on in this chapter, we will look at a more intuitive method to compare the fitted values with the actual values using the `test_forecast` function from the **TSstudio** package). Last but not least is the distribution plot of the residuals, which seem to be a fairly good representation of a normal distribution.

In an ideal world, you should end this process with white noise and independent residuals. Yet, in reality, in some cases, it will be harder to achieve this goal due to the series' structure (outliers, structural breaks, and so on), and you may have to select a model that brings you closer to this goal.

Scoring the forecast

Once you finalize the model tuning, it is time to test the ability of the model to predict observations that the model didn't see before (as opposed to the fitted values that the model saw throughout the training process). The most common method for evaluating the success of the forecast in order to predict the actual values is to use accuracy or error metrics. The most common method for evaluating the forecast's success is to predict the actual values with the use of an error metric to quantify the forecast's overall accuracy. The selection of a specific error metric depends on the forecast accuracy's goals. An example of common error metrics are as follows:

- **Mean Squared Error (MSE)**: This quantifies the average squared distance between the actual and forecasted values:

$$MSE = \frac{1}{n}\sum_{t=1}^{n}(Y_t - \hat{Y}_t)^2$$

The squared effect of the error prevents positive and negative values from cancelling each other out and panelize the error score as the error rate increases.

- **Root Mean Squared Error (RMSE)**: This is the root of the average squared distance of the actual and forecasted values:

$$RMSE = \sqrt{MSE} = \sqrt{\frac{1}{n}\sum_{t=1}^{n}(Y_t - \hat{Y}_t)^2}$$

Like *MSE*, the *RMSE* has a large error rate due to the squared effect and is therefore sensitive to outliers.

- **Mean Absolute Error (MAE)**: This measures the absolute error rate of the forecast:

$$MAE = \frac{|Y_t - \hat{Y}_t|}{n}$$

Similarly to *MSE* and *RMSE*, this method can only have positive values. This is so that is can avoid the cancellation of positive and negative values. On the other hand, there is no error penalization, and therefore this method is not sensitive to outliers.

- **Mean Absolute Percentage Error (MAPE)**: This measures the average percentage absolute error:

$$MAPE = \frac{1}{n}\sum_{t=1}^{n}|\frac{Y_t - \hat{Y}_t}{Y_t}|$$

It is easier to compare, benchmark, or communicate with non-technical people due to the percentage representative.

For example, let's use the model we trained earlier to forecast the 12 observations we left for testing and score its performance. We will use the `forecast` function from the **forecast** package to forecast the following `12` mounts (with respect to the end point of the training partition):

```
fc <- forecast(md, h = 12)
```

Now that we've assigned the forecast to the `fc` object, we will use the accuracy function from the **forecast** package to score the model's performance with respect to the actual values in the testing partition:

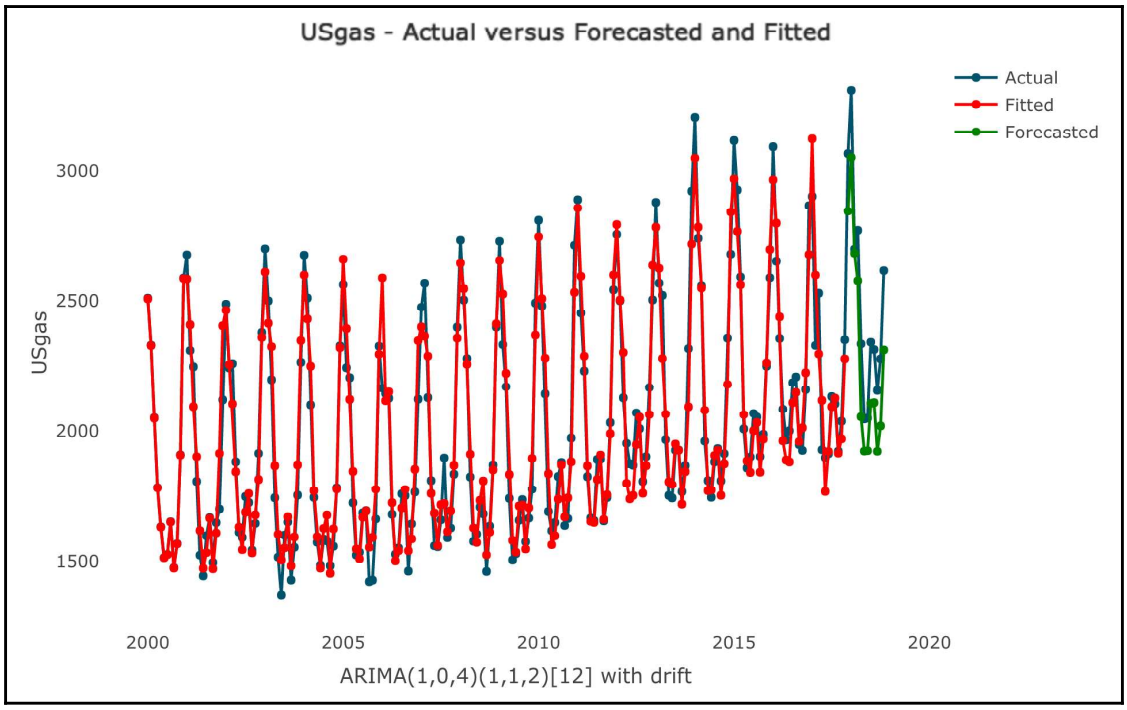
```
accuracy(fc, test)
##                      ME      RMSE       MAE       MPE     MAPE      MASE
 ## Training set  -7.049742  98.39066  72.24738 -0.584141 3.525916
0.6760411
 ## Test set     192.851978 208.01192 192.85198  7.840542 7.840542
1.8045758
 ##                     ACF1 Theil's U
 ## Training set -0.03702499        NA
 ## Test set     -0.29130569  0.658966
```

The `accuracy` function, which we will use intensively in the upcoming chapters, returns several error metrics for both the fitted values (the `Training set` row) and the actual forecast (the `Test set` row). You will notice that the model MAPE results are 3.52% and 7.84% on the training and testing partitions, respectively. A higher error rate on the testing partition compared to the training partition should not come as a surprise, as typically the model saw the training partition data throughout the training process. A fairly low error rate in the training set, along with the high error rate in the testing set, is a clear indication of overfitting in the model.

An alternative approach to evaluating the fit of the model on both the training and testing is with the `test_forecast` function from the **TSstudio** package. This function visualizes the actual series, the fitted values on the training partition, and the forecasted values on the testing set. Hovering over the fitted or forecasted values makes a textbox pop up with the RMSE and MAPE results on both the training and testing partitions (which, unfortunately, cannot transfer to the book itself):

```
test_forecast(actual = USgas,
              forecast.obj = fc,
              test = test)
```

The output is as follows:



It is easier and faster to identify insights about the goodness of the fit of both the fitted and forecasted values when plotting those values against the actual values of the series. For instance, you will immediately notice that the residual peak during 2006 is caused by outliers (or lower consumption than the normal pattern of the series). In addition, the actual forecast missed the 2018 yearly peak. Those insights cannot be observed with error metrics.

Forecast benchmark

According to the error metrics, the trained model scored a MAPE of 7.84% or RMSE of 208.01. How can we assess whether these results are too high or low? The most common method is to benchmark the model's performance to some baseline forecast or to some legacy method that we wish to replace. A popular benchmark approach would be to use a simplistic forecasting approach as a baseline. For instance, let's forecast the series with a naive approach and use it as a benchmark for the previous forecast we created with the ARIMA model.

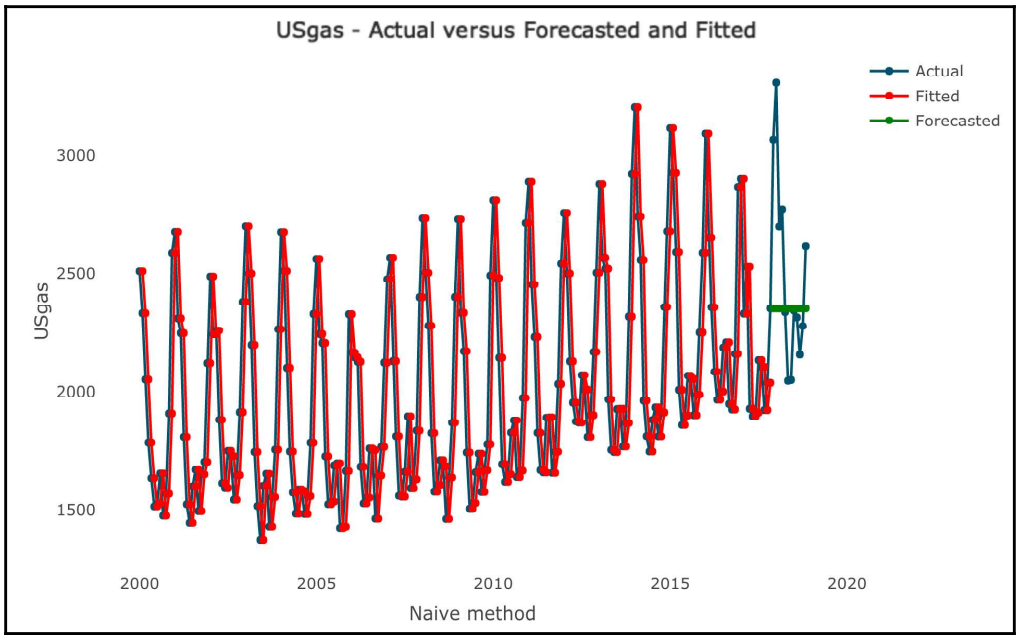
A simple naive approach typically assumes that the most recently observed value is the true representative of the future. Therefore, it will continue with the last value to infinity (or as the horizon of the forecast). We can create a naive forecast with the `naive` function from the **forecast** package and use the training set as the model input:

```
library(forecast)
```

We can review the performance of the model on the training and testing partitions using the following `test_forecast` function:

```
naive_model <- naive(train, h  = 12)
test_forecast(actual = USgas,
              forecast.obj = naive_model,
              test = test)
```

We get the following output:



Similarly, we can utilize the `accuracy` function to evaluate the model's performance on both the training and testing partitions:

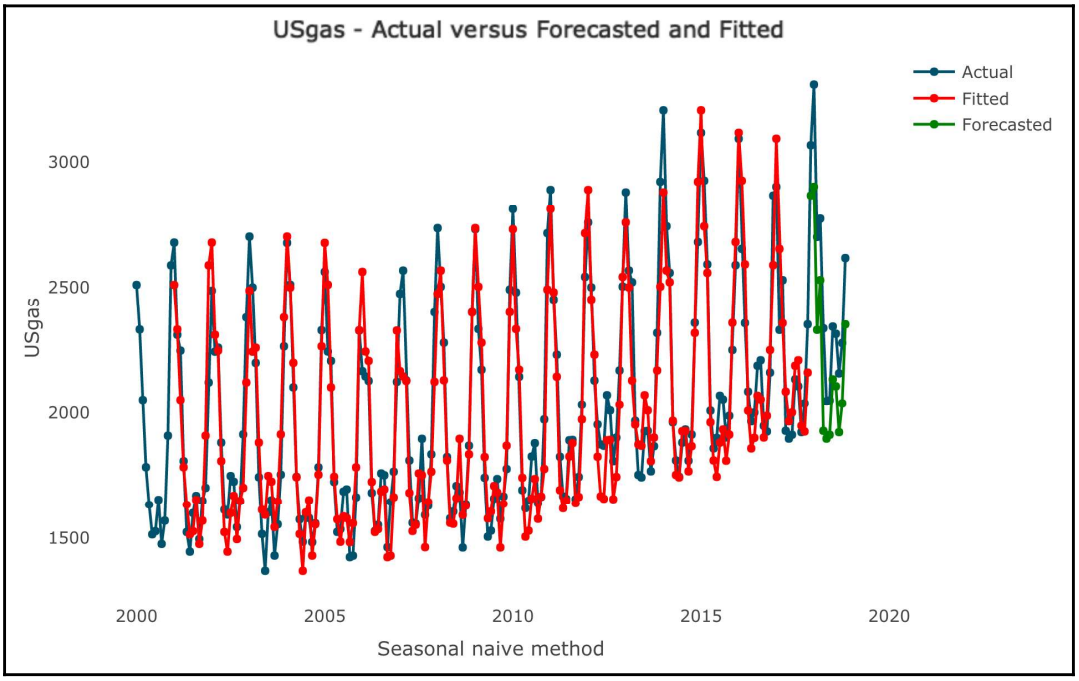
```
accuracy(naive_model, test)
##                      ME     RMSE      MAE         MPE     MAPE     MASE
 ## Training set  -2.223474 281.3634 225.4507 -0.9763903 10.96819 2.109612
 ## Test set     432.466667 580.8980 432.4667 15.6592587 15.65926 4.046725
```

```
##                     ACF1 Theil's U
## Training set 0.3893404        NA
## Test set     0.6063137  1.637631
```

In the case of the naive model, there is no training process, and the fitted values are set as the actual values (as you can see from the preceding plot). Since `USgas` has a strong seasonal pattern, it would make sense to use a seasonal naive model that takes into account seasonal variation. `snaive_model` from the **forecast** package uses the last seasonal point as a forecast of all of the corresponding seasonal observations. For example, if we are using monthly series, the value of the most recent January in the series will be used as the point forecast for all future January months:

```
snaive_model <- snaive(train, h = 12)
 test_forecast(actual = USgas,
               forecast.obj = snaive_model,
               test = test)
```

The output is shown in the following screenshot:



Let's use the `accuracy` function once more to review the seasonal naive performance on the training and testing partitions:

```
accuracy(snaive_model, test)
##                      ME     RMSE      MAE       MPE     MAPE     MASE
## Training set  20.69505 138.1953 106.8683 0.8499305 5.216692 1.000000
## Test set     242.78333 260.0920 242.7833 9.7342130 9.734213 2.271799
##                   ACF1 Theil's U
## Training set 0.4089415        NA
## Test set     0.1542089 0.7419918
```

It seems that the seasonal naive model has a better fit for the type of series we are forecasting, that is, `USgas`, due to its strong seasonal pattern (compared to the naive model). Therefore, we will use it as a benchmark for the ARIMA model. By comparing both the MAPE and RMSE of the two models in the testing partition, it is clear that the ARIMA model provides a lift (in terms of accuracy) with respect to the benchmark model:

Model	**MAPE**	**RMSE**
ARIMA	7.84%	208.01
snaive	9.73%	250.09

Finalizing the forecast

Now that the model has been trained, tested, tuned (if required), and evaluated successfully, we can move forward to the last step and finalize the forecast. This step is based on recalibrating the model's weights or coefficients with the full series. There are two approaches to using the model parameter setting:

- If the model was tuned manually, you should use the exact tuning parameters that were used on the trained model
- If the model was tuned automatically by an algorithm (such as the `auto.arima` function we used previously), you can do either of the following:
    - Extract the parameter setting that was used by with the training partition
    - Let the algorithm retune the model parameters using the full series, under the assumption that the algorithm has the ability to adjust the model parameters correctly when training the model with new data

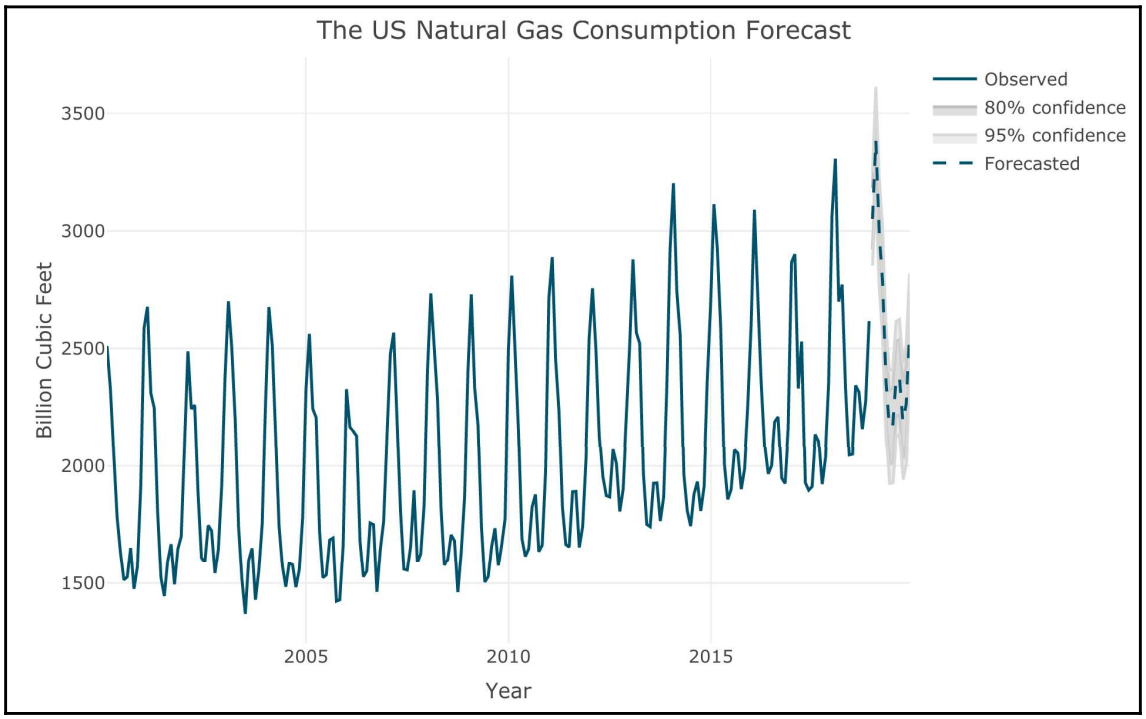
The use of algorithms to automate the model tuning process is recommended when the model's ability to tune the model is tested with backtesting. This allows you to review whether the algorithm has the ability to adjust the model parameters correctly, based on the backtesting results. For simplicity reasons, we will keep using the `auto.arima` model to train the final model:

```
md_final <- auto.arima(USgas)

fc_final <- forecast(md_final, h = 12)
```

We will use the `plot_forecast` function from the **TSstudio** package to plot the final forecast:

```
plot_forecast(fc_final,
              title = "The US Natural Gas Consumption Forecast",
              Xtitle = "Year",
              Ytitle = "Billion Cubic Feet")
```

The output is as follows:



Handling forecast uncertainty

The main goal of the forecasting process, as we saw previously, is to minimize the level of uncertainty around the future values of the series. Although we cannot completely eliminate this uncertainty, we can quantify it and provide some range around the point estimate of the forecast (which is nothing but the model's expected value of each point in the future). This can be done by using either the confidence interval (or a credible interval, when using the Bayesian model) or by using simulation.

Confidence interval

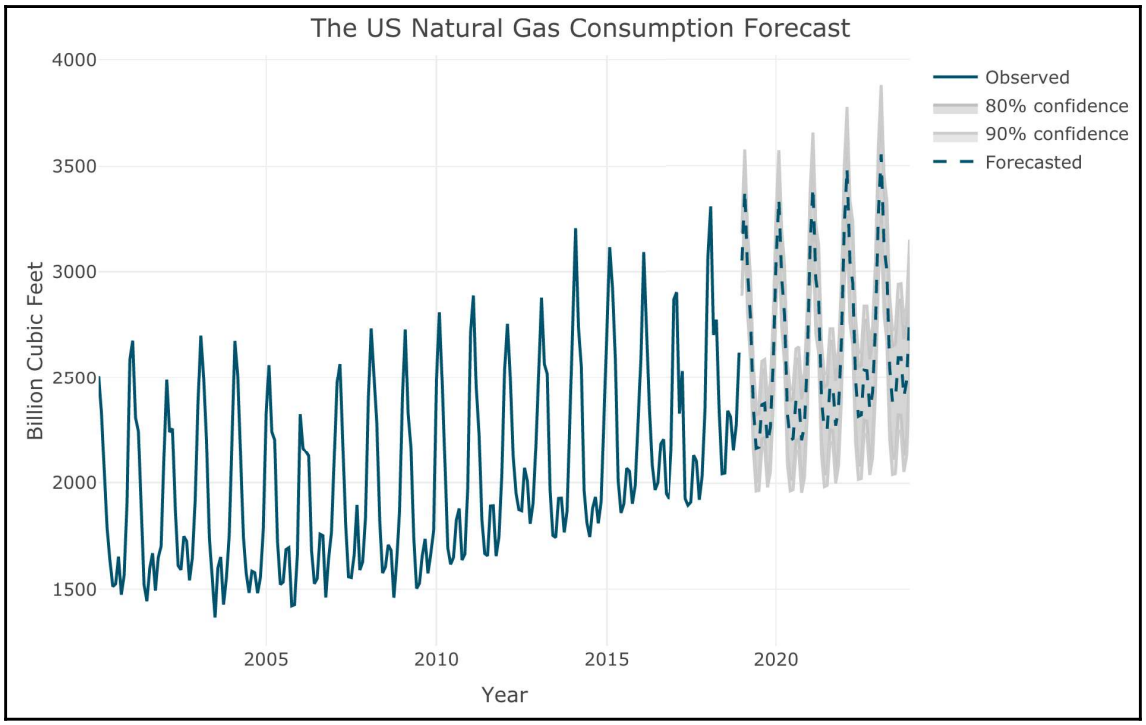
The confidence interval is a statistical approximation method that's used to express the range of possible values that contain the true value with some degree of confidence (or probability). There are two parameters that determine the range of confidence interval:

- The level of confidence or the probability that the true value will be in that range. The higher the level of confidence is, the wider the interval range.
- The estimated standard deviation of the forecast at time $T+i$, where T represents the length of the series and i represents the i forecasted value. The lower the error rate, the shorter the range of the prediction interval.

By default, the `forecast` function generates a prediction interval with a level of confidence of 80% and 95%, but you can modify it using the level argument. For example, let's use the trained model's `md_final` and `forecast` functions for the next 60 months using the prediction interval with confidence levels of 80% and 90%:

```
fc_final2 <- forecast(md_final,
                      h = 60,
                      level = c(80, 90))

plot_forecast(fc_final2,
              title = "The US Natural Gas Consumption Forecast",
              Xtitle = "Year",
              Ytitle = "Billion Cubic Feet")
```

The output is shown in the following screenshot:



Simulation

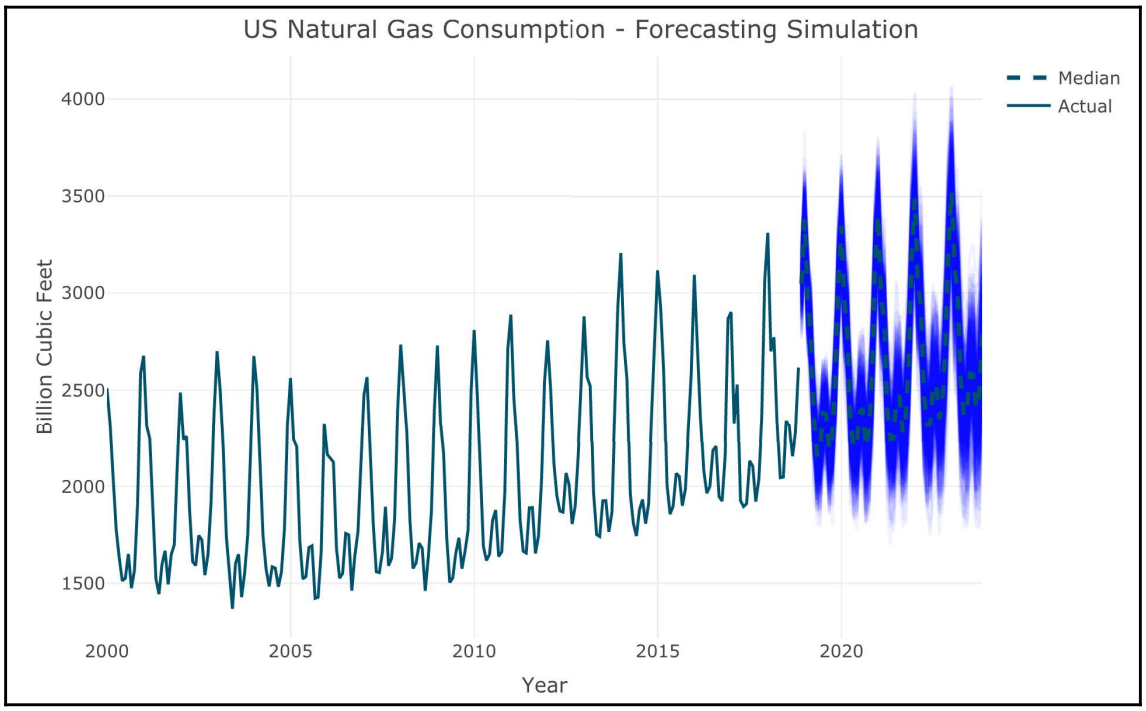
An alternative approach is to use the model distribution to simulate possible paths for the forecast. This method can only be used when the model distribution is available. The `forecast_sim` function from the **TSstudio** package provides a built-in function for simulating possible forecasting paths. This estimate can be used to calculate the forecast point estimate (for example, using the mean or median of all of the paths), or to calculate probabilities of getting different values. We will feed the same model to the function and run `100` iterations:

```
fc_final3 <- forecast_sim(model = md_final,
                          h = 60,
                          n = 500)
```

The output of the preceding function contains all of the calculate simulations and the simulated paths. Let's extract the simulation plot (and use the **plotly** package to add titles for the plot):

```
library(plotly)

fc_final3$plot %>%
  layout(title = "US Natural Gas Consumption - Forecasting Simulation",
         yaxis = list(title = "Billion Cubic Feet"),
         xaxis = list(title = "Year"))
```

The output is as follows:



Now, let's take a look at the horse race approach.

Horse race approach

Last but not least, we will end this chapter with a robust forecasting approach that combines what we've learned so far in this chapter. The *horse race* approach is based on training, testing, and evaluating multiple forecasting models and selecting the model that performs the best on the testing partitions. In the following example, we will apply horse racing between seven different models (we will review the models in the upcoming chapters; for now, don't worry if you are not familiar with them) using six periods of backtesting. The `ts_backtesting` function from the **TSstudio** package conducts the full process of training, testing, evaluating, and then forecasting, using the model that performed the best on the backtesting testing partitions. By default, the model will test the following models:

- `auto.arima`: Automated ARIMA model
- `bsts`: Bayesian structural time series model
- `ets`: Exponential smoothing state space model
- `hybrid`: An ensemble of multiple models
- `nnetar`: Neural network time series model
- `tbats`: Exponential smoothing state space model, along with Box-Cox transformation, trend, ARMA errors, and seasonal components
- `HoltWinters`: Holt-Winters filtering

Before we run the function, let's set the seed value with the `set.seed` function so that we're able to reproduce the results:

```
set.seed(1234)
```

Now, let's run the `ts_backtesting` function and see its output:

```
USgas_forecast <- ts_backtesting(ts.obj = USgas,
                                 periods = 6,
                                 models = "abehntw",
                                 error = "MAPE",
                                 window_size = 12,
                                 h = 60,
                                 plot = FALSE)
```

The output of the `ts_backtesting` function is as follows:

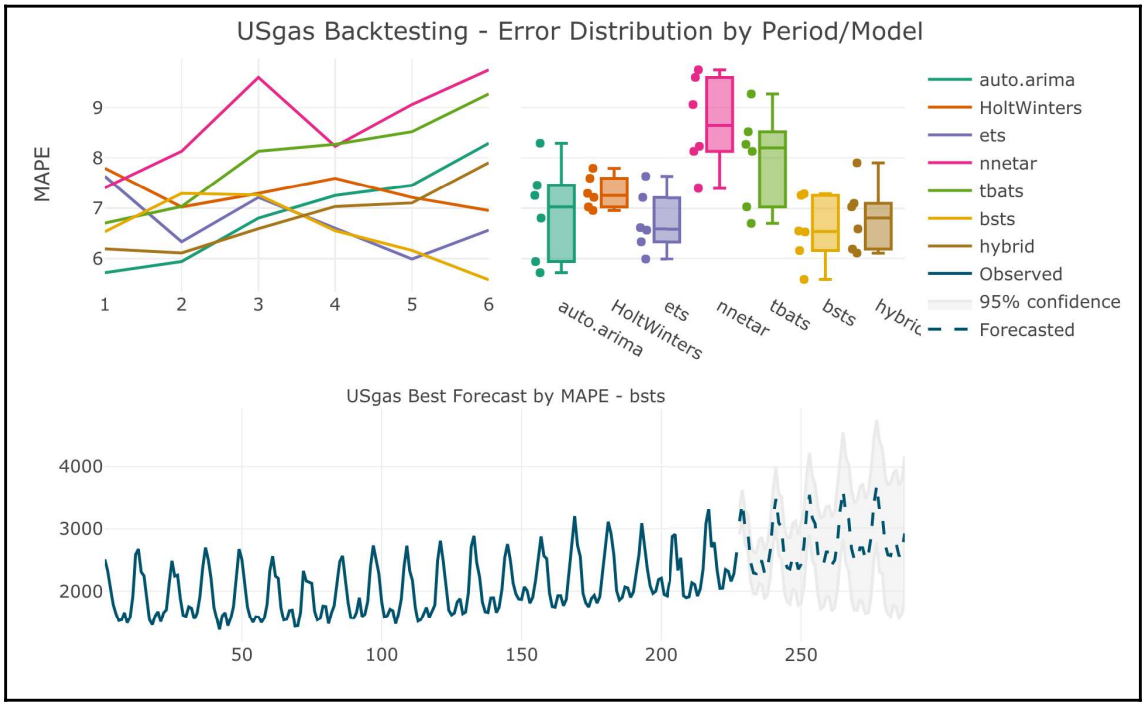
```
##    Model_Name  avgMAPE    sdMAPE  avgRMSE    sdRMSE
## 1        bsts 6.341667 0.5825261 184.9650 19.414553
## 2         ets 6.721667 0.5984786 176.9167 12.839003
## 3  auto.arima 6.908333 0.9675416 192.2400 17.169408
```

```
## 4      hybrid 6.930000 0.6554998 190.9467  7.320794
## 5 HoltWinters 7.308333 0.3264608 204.8283  8.812086
## 6       tbats 7.986667 0.9594721 215.8633 18.279782
## 7      nnetar 8.508333 1.2979587 268.6117 23.842204
```

The model provides a leaderboard (as we can see in the preceding output) that's ordered based on the error criteria that's set. In this case, the `bsts` model had the lowest error rate, and therefore the function recommended that we use it (although all of the models and their forecasts are available for extraction from the function output). We can plot the error rate and the suggested forecast using the model stored in the output object:

```
USgas_forecast$summary_plot
```

The output is shown as follows:



Those error plots provide an informative overview of the performance of each model. We consider a model good if the error distribution is both narrow and low with respect to the rest of the tested models (such as the `bsts` model).

Summary

The training process of a forecasting model is the final step of the time series analysis. The focus of this chapter was to introduce the principle of the forecasting workflow. As we saw, there are several methods that we can use to train a forecasting model, and the method selection process should align with the forecasting goals and available resources. In the following chapters, you will see these applications in practice.

In the next chapter, we will use the applications of the linear regression model to forecast time series data.

9
Forecasting with Linear Regression

The linear regression model is one of the most common methods for identifying and quantifying the relationship between a dependent variable and a single (univariate linear regression) or multiple (multivariate linear regression) independent variables. This model has a wide range of applications, from causal inference to predictive analysis and, in particular, time series forecasting.

The focus of this chapter is on methods and approaches for forecasting time series data with linear regression. That includes methods for decomposing and forecasting the series components (for example, the trend and seasonal patterns), handling special events (such as outliers and holidays), and using external variables as regressors.

This chapter covers the following topics:

- Forecasting approaches with linear regression models
- Extracting and estimating the series components
- Handling structural breaks, outliers, and special events
- Forecasting series with multiseasonality

Technical requirement

The following packages will be used in this chapter:

- **TSstudio**: Version 0.1.4 and above
- **plotly**: Version 4.8 and above
- **dplyr**: Version 0.8.1 and above
- **lubridate**: Version 1.7.4 and above
- **forecast**: Version 8.5 and above

You can access the codes for this chapter from the following link:

`https://github.com/PacktPublishing/Hands-On-Time-Series-Analysis-with-R/tree/master/Chapter09`

The linear regression

The primary usage of the linear regression model is to quantify the relationship between the dependent variable Y (also known as the response variable) and the independent variable/s X (also known as the predictor, driver, or regressor variables) in a linear manner. In other words, the model expresses the dependent variable as a linear combination of the independent variables. A linear relationship between the dependent and independent variables can be generalized by the following equations:

- In the case of a single independent variable, the equation is as follows:

$$Y_i = \beta_0 + \beta_1 * X_{1,i} + \epsilon_i$$

- For n independent variables, the equation looks as follows:

$$Y_i = \beta_0 + \beta_1 X_{1,i} + \beta_2 X_{2,i} + \ldots + \beta_n X_{n,i} + \epsilon_i$$

The model variables for these equations are as follows:

- i represents the observations index, $i = 1,..., N$
- Y_i represents the i observation of the dependent variable
- $X_{j,i}$ represents the i value of the j independent variable, where $j = 1,..., n$
- β_0 represents the value of the constant term (or intercept)
- β_j represents the corresponded parameters (or coefficients) of the j independent variables
- ε_i defines the error term, which is nothing but all the information that was not captured by independent variables for the i observation

The term linear, in the context of regression, referred to the model coefficients, which must follow a linear structure (as this allows us to construct a linear combination from the independent variables). On the other hand, the independent variables can follow both a linear and non-linear formation.

Assuming the preceding equations represent the true nature of the linear relationship between the dependent and independent variables, then the linear regression model provides an estimation for those coefficients (that is, $\beta_0, \beta_1, \ldots, \beta_n$), which can be formalized by the following equations:

- For the univariate linear regression model, the equation is as follows:

$$\hat{Y}_i = \hat{\beta}_0 + \hat{\beta}_1 X_{1,i}$$

- For the multivariate linear regression model, the equation is as follows:

$$\hat{Y}_i = \hat{\beta}_0 + \hat{\beta}_1 X_{1,i} + \hat{\beta}_2 X_{2,i} + \ldots + \hat{\beta}_n X_{n,i}$$

The variables for these equations are as follows:

- i represents the observation's index, $i = 1,\ldots, N$
- $\hat{Y}_i$ represents the estimate of the dependent variable i observation
- $X_{j,i}$ represents the i value of the j independent variable, where $j = 1,\ldots, n$
- $\hat{\beta}_0$ represents the estimate of the constant term (or intercept)
- $\hat{\beta}_1, \ldots, \hat{\beta}_n$ are the estimate of the corresponded parameters (or coefficients) of the n independent variables

The estimation of the model's coefficients is based on the following two steps:

- Define a cost function (also known as loss function)—setting some error metric to minimize
- Apply mathematical optimization for minimizing the cost function

The most common estimation approach is applying the **Ordinary Least Squares** (**OLS**) method as an optimization technique for minimizing the residuals sum of squares ($\sum_{i=1}^{N} \hat{\epsilon_i^2}$).

Squaring the residuals has two effects:

- It prevents positive and negative values canceling each other when summing them together
- The square effect provides an exponential penalization for residuals with longer distance, as their cost becomes higher

There are multiple estimation techniques besides the **OLS**, such as the maximum likelihood, method of moments, and Bayesian. Although those methods are not in the scope of the book, it is highly recommended to read and learn about alternative approaches.

Coefficients estimation with the OLS method

The OLS is a simple optimization method that is based on basic linear algebra and calculus, or matrix calculus. (This section is for general knowledge—if you are not familiar with matrix calculus you can skip this section.) The goal of the OLS is to identify the coefficients that minimize the residuals sum of squares. Suppose the residual of the *i* observation defines as the following:

$$\hat{\epsilon_i} = Y_i - \hat{Y_i}$$

We can then set the cost function with the following expression:

$$\sum_{i=1}^{N} \hat{\epsilon_i^2} = (Y_1 - \hat{Y_1})^2 + (Y_2 - \hat{Y_2})^2 + \ldots + (Y_n - \hat{Y_n})^2$$

Before applying the OLS method for minimizing the residuals sum of squares, for simplicity reasons, we will transform the representative of the cost function into a matrix formation:

$$Y = \begin{bmatrix} Y_1 \\ Y_2 \\ . \\ . \\ . \\ Y_N \end{bmatrix}, \quad \mathbf{X} = \begin{bmatrix} 1 & X_{1,1} & . & . & . & X_{1,n} \\ . & & & & & \\ . & & & & & \\ . & & & & & \\ 1 & X_{N,1} & . & . & . & X_{N,n} \end{bmatrix}, \quad \beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ . \\ . \\ . \\ \beta_n \end{bmatrix}, \quad \epsilon = Y - X\beta = \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ . \\ . \\ . \\ \epsilon_N \end{bmatrix}$$

Those sets of matrices represent the following:

- Vector Y (or $N \times 1$ matrix), representing a dependent variable with N observations
- X, a $N \times n + 1$ dimensions matrix, representing the corresponding n independent variables and a scalar of 1's for the intercept component (β_0)

- β, an $(n+1) \times 1$ dimensions matrix, representing the model coefficients
- ϵ, an $N \times 1$ dimensions matrix, representing the corresponding error (or the difference between) the actual value Y and its estimate βX

The residual term $\hat{\epsilon_t}$ should not be confused with the error term ϵ_i. While the first represents the difference between the series Y and its estimate $\hat{Y}$, the second (error term) represents the difference between the series and its expected value

Let's set the cost function using the matrix form as we defined it previously:

$$\sum \epsilon^2 = \sum \epsilon^T \epsilon$$

We will now start to expand this expression by using the formula of ϵ as we outlined previously:

$$\epsilon^T \epsilon = (Y - X\beta)^T (Y - X\beta)$$

Next, we will multiply the two components (ϵ^T and ϵ) and open the brackets:

$$\epsilon^T \epsilon = Y^T Y - 2Y^T X\beta + \beta^T X^T X\beta$$

Since our goal is to find the β that minimizes this equation, we will differentiate the equation with respect to β and then set it to zero:

$$\frac{\partial \epsilon^T \epsilon}{\partial \beta} = \frac{\partial (Y^T Y - 2Y^T X\beta + \beta^T X^T X\beta)}{\partial \beta} = 0$$

Solving this equation will yield the following output:

$$X^T X\beta = X^T Y$$

Manipulating this equation allows us to extract $\hat{\beta}$ matrix, the estimate of the coefficient matrix β:

$$\hat{\beta} = (X^T X)^{-1} X^T Y$$

Note that we changed the notation of β to $\hat{\beta}$ on the final output as it represents the estimate of the true value of β.

The key properties of the OLS coefficients estimation are as follows:

- The main feature of the OLS coefficients estimation method is the unbiasedness of the estimation of the coefficients with respect to the actual values. In other words, for any given, $\hat{\beta}_i$, $E(\hat{\beta}_i) = \beta_i$
- The sample regression line will always go through the mean of *X* and *Y*
- The mean of $\hat{Y}$, the OLS estimation of the dependent variable Y, is equal to $\overline{Y}$, the mean the dependent variable
- The mean of the residuals vector $\hat{\epsilon}$ is equal to zero, or $\frac{\sum_{i=1}^{N} \hat{\epsilon}_i}{N} = 0$

The OLS assumptions

The OLS model's main assumptions are the following:

- The model coefficients must follow a linear structure (for example, $Y = \hat{\beta}_0 + \hat{\beta}_1 e^X$ is a linear model but $Y = \hat{\beta}_0 + X_1^{\hat{\beta}_1}$ is not).
- There is no perfect collinearity between independent variables $X_1, X_2, \ldots, X_n$. In other words, none of the independent variables are a linear combination of any of the other independent variables.
- All the independent variables must be a non-zero variance (or non-constant).
- The error term ϵ, conditioned on the matrix of independent variables *X*, is an **independent and identically distributed** (**i.i.d**) variable with mean *0* and constant variance σ^2.
- Both the dependent and independent variables draw from the population in a random sample. This assumption does not hold when regressing time series data, as typically the observations have some degree of correlation. Therefore, this assumption is relaxed when regressing time series data.

The OLS assumption about the error term ϵ—i.i.d. with mean *0* and variance σ^2 is in many cases violated when working with time series data, as, typically, some degree of correlation exists in the model residuals. This is mainly an indication that the regression model didn't capture all the patterns or information of the series. In `Chapter 11`, *Forecasting with ARIMA Models*, we will introduce a method for handling this type of case, by modeling the error term with ARIMA model.

Forecasting with linear regression

The linear regression model, unlike the traditional time series models such as the **ARIMA** or **Holt-Winters**, was not designed explicitly to handle and forecast time series data. Instead, it is a generic model with a wide range of applications from causal inference to predictive analysis.

Therefore, forecasting with a linear regression model is mainly based on the following two steps:

1. Identifying the series structure, key characteristics, patterns, outliers, and other features
2. Transforming those features into input variables and regressing them with the series to create a forecasting model

The core features of a linear regression forecasting model are the trend and seasonal components. The next section focuses on identifying the series trend and seasonal components and then transforming them into input variables of the regression model.

Forecasting the trend and seasonal components

In `Chapter 5`, *Decomposition of Time Series Data*, we introduced the series structural components: the trend, cycle, seasonal and irregular components, and methods for decomposing them with the decompose function.

Recall that the decomposition of a series can be defined by one of the following expressions:

- $Y_t = T_t + S_t + C_t + I_t$, when the series has an additive structure
- $Y_t = T_t \times S_t \times C_t \times I_t$, when the series has a multiplicative structure

The explanation is as follows:

- **Trend**: Represents the series' growth over time after adjusting and removing the seasonal effects
- **Seasonal**: A recurring cyclical pattern that derived directly from the series frequency units (for example, the month of the year for a series with a monthly frequency)
- **Cycle**: A cyclical pattern that is not related to the series frequency unit
- **Irregular**: Any other patterns that are not captured by the trend, seasonal, and cycle components

For the sake of simplicity, we can drop the cycle component as it typically merged into the trend component (or ignored the cycle component). Therefore, we can update and replace the preceding equations with the following:

- $Y_t = T_t + S_t + I_t$, when the series has an additive structure
- $Y_t = T_t \times S_t \times I_t$, when the series has a multiplicative structure

We can now transform those equations for a linear regression model, by modifying the equations notation:

$$Y_t = \beta_0 + \beta_1 T_t + \beta_2 S_t + \epsilon_t$$

Where:

- Y represents a time series with n observations
- T, an independent variable with n observations, represents the series trend component
- S, an independent variable with n observations, represents the series seasonal component
- ϵ_t, the regression error term, represents the irregular component or any pattern that is not captured by the series trend and seasonal component
- β_0, β_1, and β_2, represent the model intercept, and coefficients of the trend and seasonal components, respectively

For the sake of convenience and in the context of working with time series, we will change the observations notation from *i* to *t*, as it represents the time dimension of the series.

The transformation of a series with a multiplicative structure into linear regression formation required a transformation of the series into an additive structure. This can be done by applying *log* transformation for both sides of the equations:

$$\log(Y_t) = \log(T_t) + \log(S_t) + \log(l_t)$$

Once the series transformed into an additive structure, the transformation to a linear regression formation is straightforward and follows the same process as described previously:

$$\log(Y_t) = \beta_0 + \beta_1 \log(T_t) + \beta_2 \log(S_t) + \epsilon_t$$

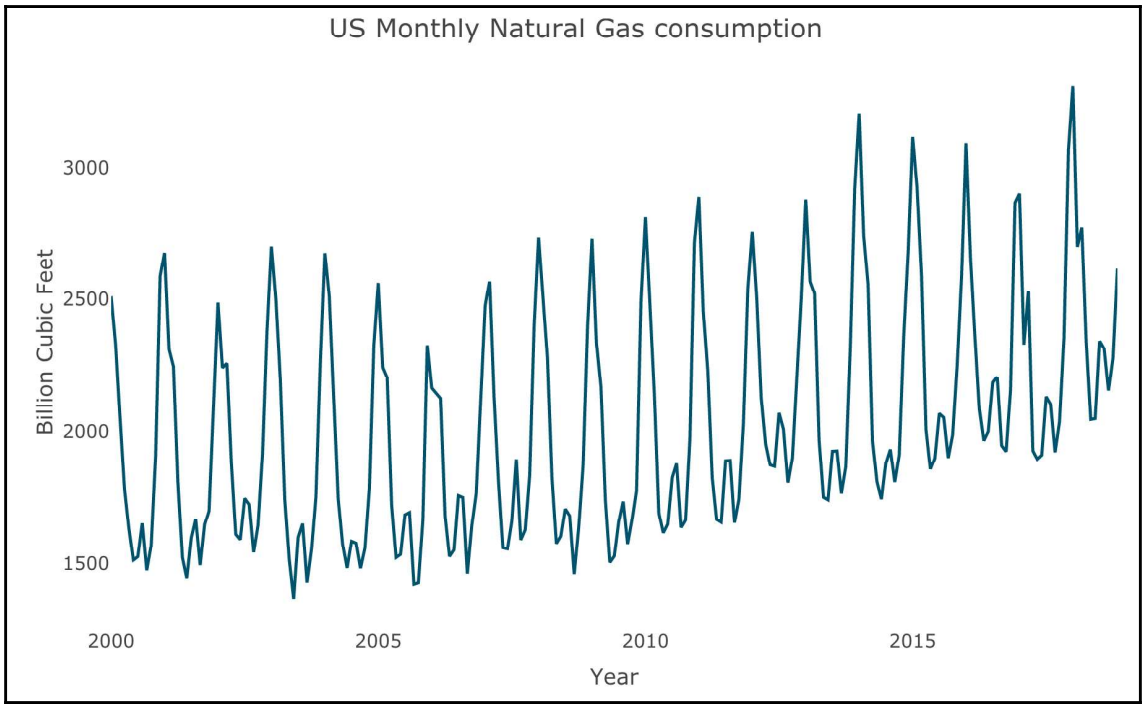
Features engineering of the series components

Before creating the regression inputs that represent the series trend and seasonal components, we first have to understand their structure. In the following examples, we will demonstrate the process of creating new features from existing series using the `USgas` series.

Let's load the series from the **TSstudio** package again and plot it with the `ts_plot` function:

```
library(TSstudio)

data(USgas)

ts_plot(USgas,
        title = "US Monthly Natural Gas consumption",
        Ytitle = "Billion Cubic Feet",
        Xtitle = "Year")
```

The output is shown as follows:



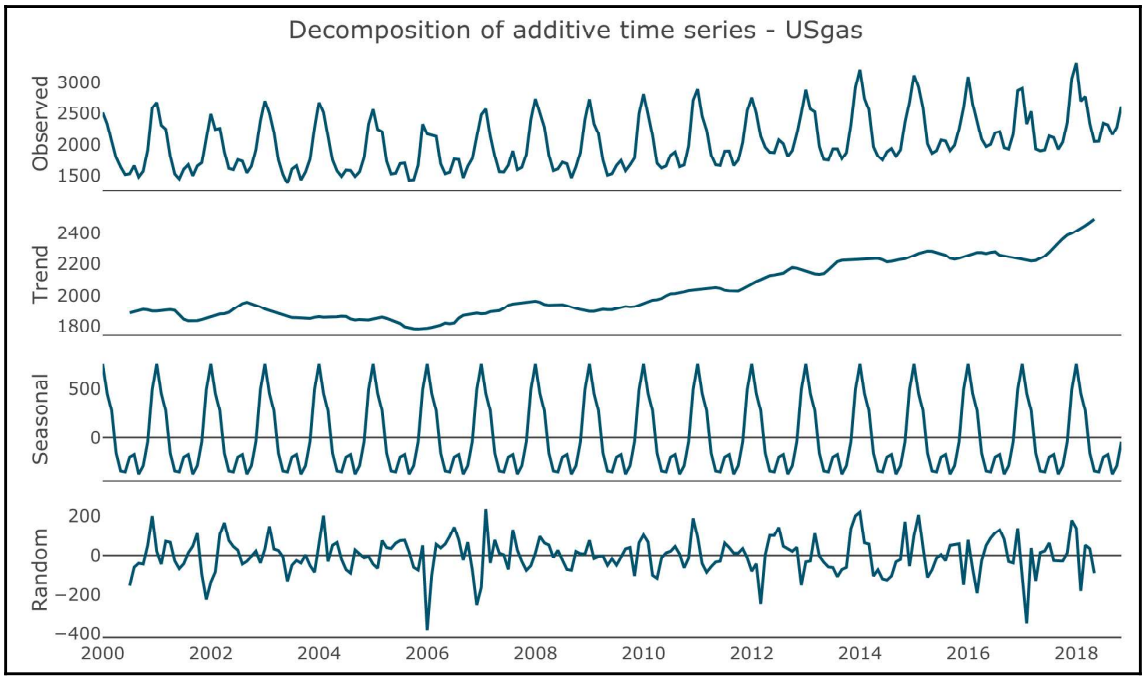
Also, let's review the main characteristics of the series using the `ts_info` function:

```
ts_info(USgas)
##  The USgas series is a ts object with 1 variable and 227 observations
##  Frequency: 12
##  Start time: 2000 1
##  End time: 2018 11
```

As you can see in the series plot, and as we saw on the previous chapters, `USgas` is a monthly series with a strong monthly seasonal component and fairly stable trend line. We can explore the series components structure with the `ts_decompose` function further:

```
ts_decompose(USgas)
```

The output is shown as follows:



You can see in the preceding plot that the trend of the series is fairly flat between **2000** and **2010**, and has a fairly linear growth moving forward. Therefore, the overall trend between **2000** and **2018** is not strictly linear. This is an important insight that will help us to define the trend input for the regression model.

Before using the `lm` function, the built-in R linear regression function from the **stats** package, we will have to transform the series from a `ts` object to a `data.frame` object. Therefore, we will utilize the `ts_to_prophet` function from the **TSstudio** package:

```
USgas_df <- ts_to_prophet(USgas)
```

The function transforms the `ts` object into two columns of `data.frame`, where the two columns represent the time and numeric components of the series, respectively:

```
head(USgas_df)
```

We get the following output:

```
##           ds      y
## 1 2000-01-01 2510.5
## 2 2000-02-01 2330.7
## 3 2000-03-01 2050.6
## 4 2000-04-01 1783.3
## 5 2000-05-01 1632.9
## 6 2000-06-01 1513.1
```

After we transform the series into a `data.frame` object, we can start to create the regression input features. The first feature we will create is the series trend. A basic approach for constructing the trend variable is by indexing the series observations in chronological order:

```
USgas_df$trend <- 1:nrow(USgas_df)
```

Regressing the series with the series index provides an estimate of the marginal growth from month to month, as the index is in chronological order with constant increments.

The second feature we want to create is the seasonal component. Since we want to measure the contribution of each frequency unit to the oscillation of the series, we will use a categorical variable for each frequency unit. In the case of the `USgas` series, the frequency units represent the months of the year, and, therefore, we will create a categorical variable with 12 categories, each category corresponding to a specific month of the year. We will use the `month` function from the **lubridate** package to extract the month of the year from the `ds` date variable:

```
library(lubridate)

USgas_df$seasonal <- factor(month(USgas_df$ds, label = T), ordered = FALSE)
```

We used the `factor` function to convert the output of the `month` function into no ordered categorical variable. Let's now review the data frame after adding the new features:

```
head(USgas_df)
```

We get the following output:

```
##           ds      y trend seasonal
## 1 2000-01-01 2510.5     1      Jan
## 2 2000-02-01 2330.7     2      Feb
## 3 2000-03-01 2050.6     3      Mar
## 4 2000-04-01 1783.3     4      Apr
## 5 2000-05-01 1632.9     5      May
## 6 2000-06-01 1513.1     6      Jun
```

Last but not least, before we start to regress the series with those features, we will split the series into a training and testing partition. We will set the last 12 months of the series as a testing partition:

```
h <- 12 # setting a testing partition length

train <- USgas_df[1:(nrow(USgas_df) - h), ]

test <- USgas_df[(nrow(USgas_df) - h + 1):nrow(USgas_df), ]
```

Now, after we created the training and testing data frames, let's review how the regression model captures each one of the components separately and all together.

Modeling the series trend and seasonal components

We will first model the series trend by regressing the series with the trend variable, on the training partition:

```
md_trend <- lm(y ~ trend, data = train)
```

We will use the `summary` function to review the model details:

```
summary(md_trend)
##
## Call:
## lm(formula = y ~ trend, data = train)
##
## Residuals:
##    Min     1Q Median     3Q    Max
```

```
## -537.6 -305.3 -150.1  317.1 1067.7
##
## Coefficients:
##              Estimate Std. Error t value             Pr(>|t|)
## (Intercept) 1772.2648    53.3781  33.202 < 0.0000000000000002 ***
## trend          2.1548     0.4285   5.029          0.00000105 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 390 on 213 degrees of freedom
## Multiple R-squared:  0.1061, Adjusted R-squared:  0.1019
## F-statistic: 25.29 on 1 and 213 DF,  p-value: 0.000001048
```

As you can see from the preceding regression output, the coefficient of the trend variable is statistically significant to a level of `0.001`. However, the adjusted R-squared of the regression is fairly low, which generally makes sense, as most of the series variation of the series is related to the seasonal pattern as we saw in the plots previously.

As you can note from the preceding regression output, the fourth column represents the *p*-value of each one of the model coefficients. The *p*-value provides the probability that we will reject the null hypothesis given it is actually true, or the type I error. Therefore, for the *p*-value smaller than α, the threshold value, we will reject the null hypothesis with a level of significance of α, where typical values of α are 0.1, 0.05, 0.01, and so on.

As always, it is recommended that you put some context to the numbers with data visualization. Therefore, we will use the model we created to predict the fitted values on the training partition and the forecasted values on the testing partition. The `predict` function from the **stats** package, as the name implies, predicts the values of an input data based on a given model.

We will use it to predict both the fitted and forecasted values of the trend model we trained before:

```
train$yhat <- predict(md_trend, newdata = train)

test$yhat <- predict(md_trend, newdata = test)
```

We will create a utility function that plots the series and the model output, utilizing the **plotly** package:

```
library(plotly)

plot_lm <- function(data, train, test, title = NULL){
 p <- plot_ly(data = data,
         x = ~ ds,
         y = ~ y,
         type = "scatter",
         mode = "line",
         name = "Actual") %>%
   add_lines(x =  ~ train$ds,
             y = ~ train$yhat,
             line = list(color = "red"),
             name = "Fitted") %>%
   add_lines(x =  ~ test$ds,
             y = ~ test$yhat,
             line = list(color = "green", dash = "dot", width = 3),
             name = "Forecasted") %>%
   layout(title = title,
          xaxis = list(title = "Year"),
          yaxis = list(title = "Billion Cubic Feet"),
          legend = list(x = 0.05, y = 0.95))
 return(p)
 }
```

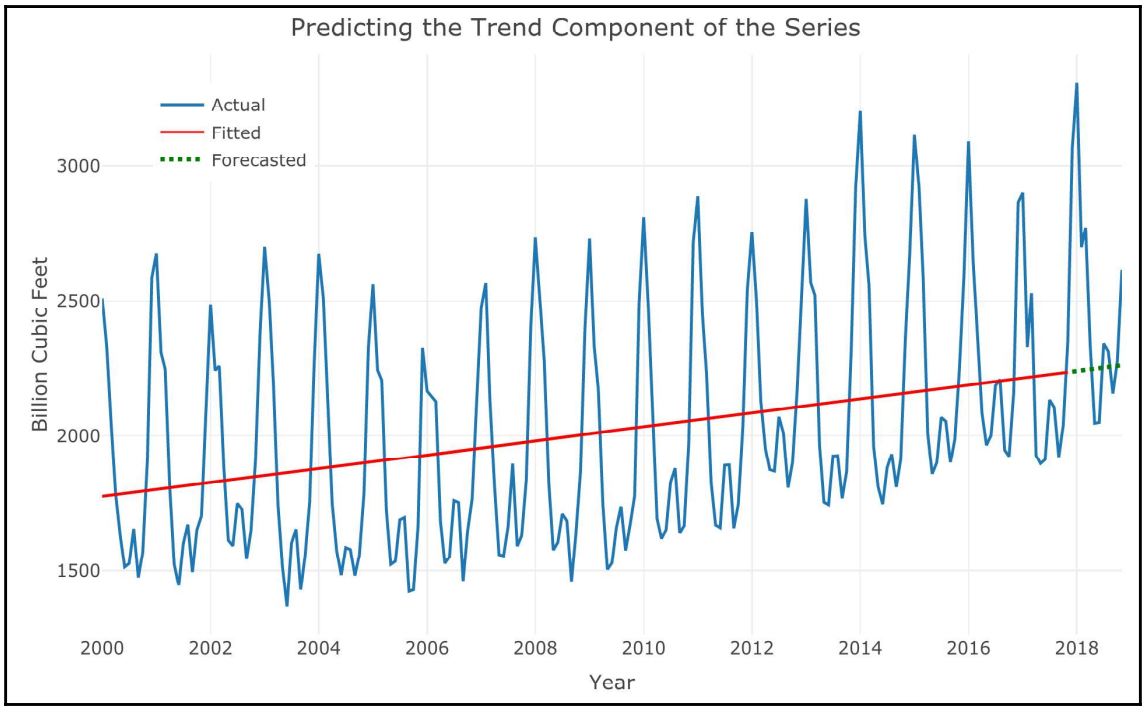
The function arguments are as follows:

- `data`: The input data, a `data.frame` object following the same structure as the one of the `USgas_df` (including the `yhat` variable)
- `train`: The corresponding training set that was used to train the model
- `test`: Likewise, the corresponding testing set that was used to evaluate the forecast model
- `title`: The plot `title`, by default, set to `NULL`

Let's set the inputs of the `plot_lm` function with the model output:

```
plot_lm(data = USgas_df,
         train = train,
         test = test,
         title = "Predicting the Trend Component of the Series")
```

The output is shown as follows:



Overall, the model was able to capture the general movement of the trend, yet a linear trend may fail to capture the structural break of the trend that occurred around 2010. Later on, in this chapter, we will see an advanced method to capture a non-linear trend.

Last but not least, for comparison analysis, we want to measure the model error rate both in the training and the testing sets:

```
mape_trend <- c(mean(abs(train$y - train$yhat) / train$y),
                mean(abs(test$y - test$yhat) / test$y))
mape_trend
```

We get the following output:

```
## [1] 0.1646270 0.1201788
```

The process of modeling and forecasting the seasonal component follows the same process as we applied with the trend, by regressing the series with the seasonal variable we created before:

```
md_seasonal <- lm(y ~ seasonal, data = train)
```

Let's review the model details:

```
summary(md_seasonal)
```

We get the following output:

```
##
## Call:
## lm(formula = y ~ seasonal, data = train)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -577.1 -141.1  -41.9  130.0  462.2
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   2742.4       45.3   60.52  < 2e-16 ***
## seasonalFeb   -279.4       64.1   -4.36  2.1e-05 ***
## seasonalMar   -474.5       64.1   -7.41  3.4e-12 ***
## seasonalApr   -900.2       64.1  -14.05  < 2e-16 ***
## seasonalMay  -1076.6       64.1  -16.80  < 2e-16 ***
## seasonalJun  -1095.2       64.1  -17.09  < 2e-16 ***
## seasonalJul   -936.3       64.1  -14.61  < 2e-16 ***
## seasonalAug   -906.5       64.1  -14.15  < 2e-16 ***
## seasonalSep  -1110.1       64.1  -17.32  < 2e-16 ***
## seasonalOct  -1019.3       64.1  -15.91  < 2e-16 ***
## seasonalNov   -766.0       64.1  -11.95  < 2e-16 ***
## seasonalDec   -258.1       65.0   -3.97  1.0e-04 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 192 on 203 degrees of freedom
## Multiple R-squared:  0.793,  Adjusted R-squared:  0.782
## F-statistic: 70.7 on 11 and 203 DF,  p-value: <2e-16
```

Since we regress the dependent variable with a categorical variable, the regression model creates coefficients for 11 out of the 12 categories, which are those embedded with the slope values. As you can see in the regression summary of the seasonal model, all the model's coefficients are statistically significant. Also, you can notice that the adjusted R-squared of the seasonal model is somewhat higher with respect to the trend model (`0.78` as opposed to `0.1`).

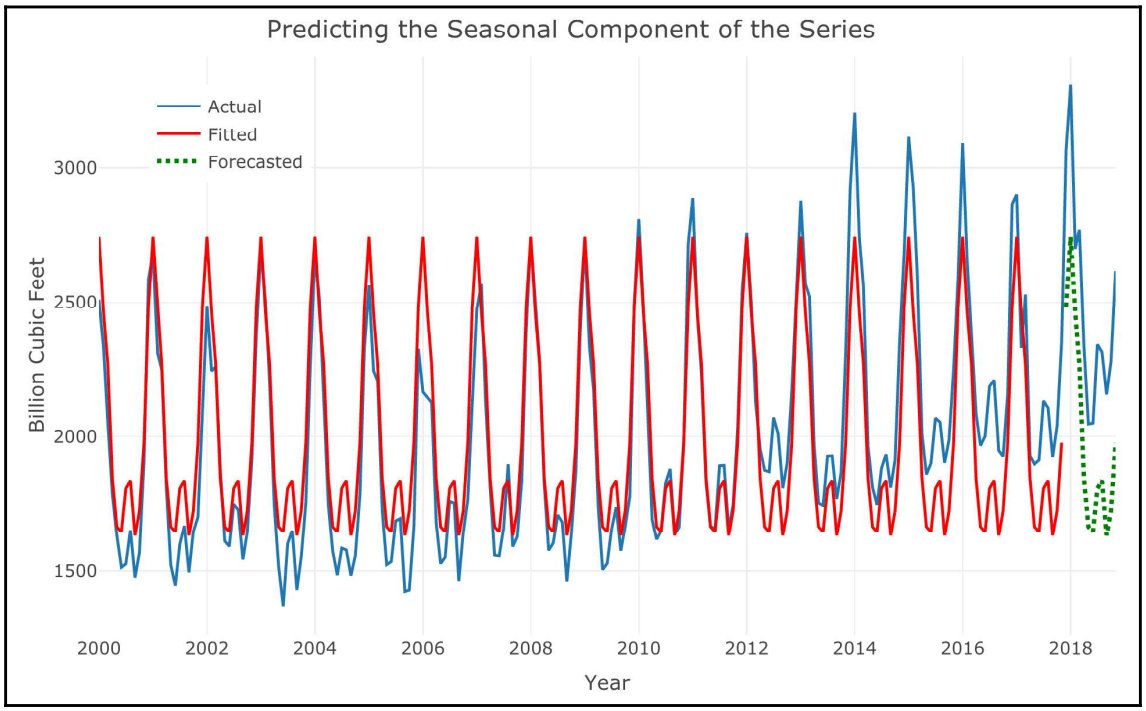
Before we plot the fitted model and forecast values with the `plot_lm` function, we will update the values of `yhat` with the `predict` function:

```
train$yhat <- predict(md_seasonal, newdata = train)
test$yhat <- predict(md_seasonal, newdata = test)
```

Now we can use the `plot_lm` function to visualize the fitted model and forecast values:

```
plot_lm(data = USgas_df,
        train = train,
        test = test,
        title = "Predicting the Seasonal Component of the Series")
```

The output is shown as follows:



As you can see in the preceding plot, the model is doing a fairly good job of capturing the structure of the series seasonal pattern. However, you can observe that the series trend is missing. Before we add both the trend and the seasonal components, we will score the model performance:

```
mape_seasonal <- c(mean(abs(train$y - train$yhat) / train$y),
                   mean(abs(test$y - test$yhat) / test$y))

mape_seasonal
```

We get the following output:

```
## [1] 0.07786439 0.19906796
```

The high error rate on the testing set is related to the trend component that was not included in the model. The next step is to join the two components into one model and to forecast the feature values of the series:

```
md1 <- lm(y ~ seasonal + trend, data = train)
```

Let's review the model summary after regressing the series with both the trend and seasonal components:

```
summary(md1)
```

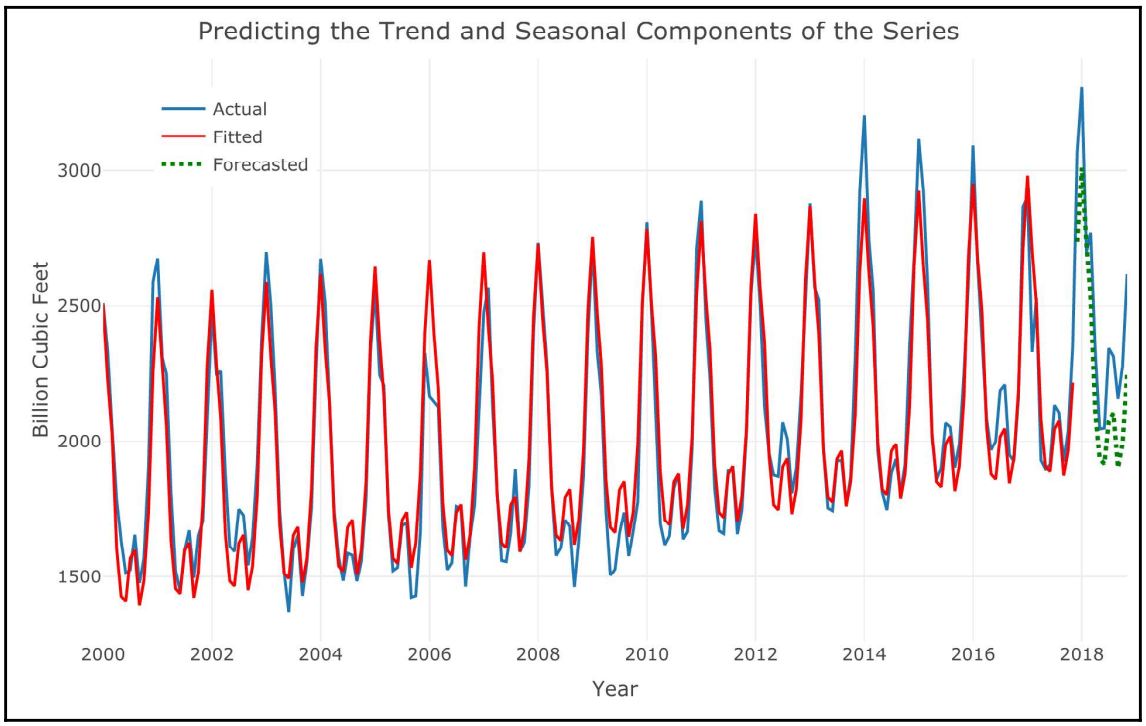
We get the following output:

```
##
## Call:
## lm(formula = y ~ seasonal + trend, data = train)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -506.7  -71.2  -13.8   79.0  328.5
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  2500.682     31.620   79.09  < 2e-16 ***
## seasonalFeb  -281.769     40.302   -6.99  3.9e-11 ***
## seasonalMar  -479.227     40.303  -11.89  < 2e-16 ***
## seasonalApr  -907.201     40.304  -22.51  < 2e-16 ***
## seasonalMay -1085.948     40.305  -26.94  < 2e-16 ***
## seasonalJun -1106.933     40.307  -27.46  < 2e-16 ***
## seasonalJul  -950.374     40.310  -23.58  < 2e-16 ***
## seasonalAug  -922.932     40.312  -22.89  < 2e-16 ***
## seasonalSep -1128.862     40.316  -28.00  < 2e-16 ***
## seasonalOct -1040.442     40.319  -25.81  < 2e-16 ***
## seasonalNov  -789.461     40.324  -19.58  < 2e-16 ***
## seasonalDec  -269.863     40.895   -6.60  3.6e-10 ***
## trend           2.347      0.133   17.64  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 121 on 202 degrees of freedom
## Multiple R-squared:  0.919,  Adjusted R-squared:  0.914
## F-statistic:  190 on 12 and 202 DF,  p-value: <2e-16
```

Regressing the series with both the trend and the seasonal components together provides additional lift to the adjusted R-squared of the model from `0.78` to `0.91`. This can be seen in the plot of the model output:

```
train$yhat <- predict(md1, newdata = train)
test$yhat <- predict(md1, newdata = test)

plot_lm(data = USgas_df,
        train = train,
        test = test,
        title = "Predicting the Trend and Seasonal Components of the
Series")
```

The output is shown as follows:



Let's measure the model's MAPE score on both the training and testing partitions:

```
mape_md1 <- c(mean(abs(train$y - train$yhat) / train$y),
              mean(abs(test$y - test$yhat) / test$y))

mape_md1
```

We get the following output:

```
## [1] 0.04501471 0.09192438
```

Regressing the series with both the trend and the seasonal components provides a significant lift in both the quality of fit of the model and with the accuracy of the model. However, when looking at the plot of the model fit and forecast, you can notice that the model trend is *too* linear and missing the structural break of the series trend. This is the point where adding a polynomial component for the model could potentially provide additional improvement for the model accuracy.

A simple technique to capture a non-linear trend is to add a polynomial component to the series trend in order to capture the trend curvature over time. We will use the `I` argument, which allows us to apply mathematical operations on any of the input objects. Therefore, we will use this argument to add a second degree of the polynomial for the trend input:

```
md2 <- lm(y ~ seasonal + trend + I(trend^2), data = train)
```

The summary of the model can be seen as follows:

```
summary(md2)
```

We get the following output:

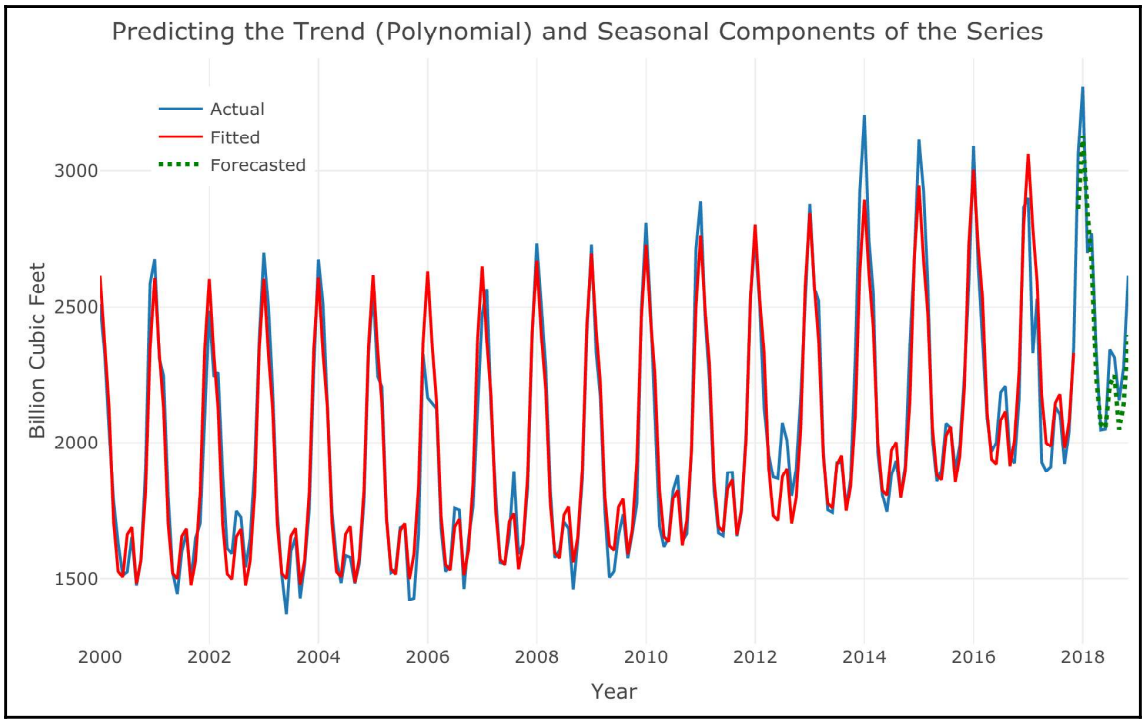
```
##
## Call:
## lm(formula = y ~ seasonal + trend + I(trend^2), data = train)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -466.5  -55.5   -5.1   60.1  309.5
##
## Coefficients:
##                Estimate  Std. Error t value Pr(>|t|)
## (Intercept)  2617.20174    33.00223   79.30  < 2e-16 ***
## seasonalFeb  -281.63380    36.26107   -7.77  4.0e-13 ***
## seasonalMar  -478.98651    36.26167  -13.21  < 2e-16 ***
## seasonalApr  -906.88591    36.26267  -25.01  < 2e-16 ***
## seasonalMay -1085.58755    36.26406  -29.94  < 2e-16 ***
## seasonalJun -1106.55810    36.26584  -30.51  < 2e-16 ***
```

```
## seasonalJul  -950.01422    36.26801  -26.19  < 2e-16 ***
## seasonalAug  -922.61703    36.27057  -25.44  < 2e-16 ***
## seasonalSep -1128.62207    36.27352  -31.11  < 2e-16 ***
## seasonalOct -1040.30714    36.27686  -28.68  < 2e-16 ***
## seasonalNov  -789.46112    36.28061  -21.76  < 2e-16 ***
## seasonalDec  -263.18413    36.80761   -7.15  1.6e-11 ***
## trend          -0.89541     0.48054   -1.86    0.064 .
## I(trend^2)      0.01501     0.00215    6.97  4.5e-11 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 109 on 201 degrees of freedom
## Multiple R-squared:  0.934,  Adjusted R-squared:  0.93
## F-statistic:  220 on 13 and 201 DF,  p-value: <2e-16
```

Adding the second-degree polynomial to the regression model did not lead to a significant improvement of the goodness of fit of the model. On the other model, as you can see in the following model output plot, this simple change in the model structure allows us to capture the structural break of the trend over time:

```
train$yhat <- predict(md2, newdata = train)
test$yhat <- predict(md2, newdata = test)


 plot_lm(data = USgas_df,
         train = train,
         test = test,
         title = "Predicting the Trend (Polynomial) and Seasonal Components
of the Series")
```

The output is shown as follows:



As we can see from the model following the MAPE score, the model accuracy significantly improved from adding the polynomial trend to the regression model, as the error on the testing set dropped from 9.2% to 4.5%:

```
mape_md2 <- c(mean(abs(train$y - train$yhat) / train$y),
              mean(abs(test$y - test$yhat) / test$y))

mape_md2
```

We get the following output:

```
## [1] 0.03706897 0.04559134
```

The tslm function

So far, we have seen the manual process of transforming a `ts` object to a linear regression forecasting model format. The `tslm` function from the **forecast** package provides a built-in function for transforming a `ts` object into a linear regression forecasting model. Using the `tslm` function, you can set the regression component along with other features.

We will now repeat the previous example and forecast the last 12 observations of the `USgas` series with the `tslm` function using the trend, square of the trend, and the seasonal components. First, let's split the series to training and testing partitions using the `ts_split` function:

```
USgas_split <- ts_split(USgas, sample.out = h)

train.ts <- USgas_split$train

test.ts <- USgas_split$test
```

Next, we will apply the same formula we used to create the preceding `md2` forecasting model using the `tslm` function:

```
library(forecast)

md3 <- tslm(train.ts ~ season + trend + I(trend^2))
```

Let's now review `md3`, the output of the `tslm` function, and compare it with the output of `md2`:

```
summary(md3)
```

We get the following output:

```
##
## Call:
## tslm(formula = train.ts ~ season + trend + I(trend^2))
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -466.52  -55.46   -5.13   60.06  309.53
##
## Coefficients:
##                 Estimate   Std. Error t value             Pr(>|t|)
## (Intercept)  2617.201742    33.002235  79.304 < 0.0000000000000002 ***
## season2      -281.633803    36.261068  -7.767    0.000000000000404 ***
## season3      -478.986514    36.261672 -13.209 < 0.0000000000000002 ***
## season4      -906.885912    36.262671 -25.009 < 0.0000000000000002 ***
```

```
## season5     -1085.587550    36.264062 -29.936 < 0.0000000000000002 ***
## season6     -1106.558097    36.265843 -30.512 < 0.0000000000000002 ***
## season7      -950.014218    36.268012 -26.194 < 0.0000000000000002 ***
## season8      -922.617026    36.270570 -25.437 < 0.0000000000000002 ***
## season9     -1128.622074    36.273520 -31.114 < 0.0000000000000002 ***
## season10    -1040.307142    36.276865 -28.677 < 0.0000000000000002 ***
## season11     -789.461118    36.280610 -21.760 < 0.0000000000000002 ***
## season12     -263.184126    36.807605  -7.150    0.000000000015734 ***
## trend          -0.895408     0.480540  -1.863               0.0639 .
## I(trend^2)      0.015010     0.002155   6.966    0.000000000045450 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 108.8 on 201 degrees of freedom
## Multiple R-squared:  0.9344, Adjusted R-squared:  0.9301
## F-statistic: 220.1 on 13 and 201 DF,  p-value: < 0.00000000000000022
```

As you can observe from the preceding output, both models (`md2` and `md3`) are identical.

There are several advantages to using the `tslm` function, as opposed to manually setting a regression model for the series with the `lm` function:

- Efficiency—does not require transforming the series to a `data.frame` object and feature engineering
- The `output` object supports all the functionality of the `forecast` (such as the `accuracy` and `checkresiduals` functions) and **TSstudio** packages (such as the `test_forecast` and `plot_forecast` functions)

Modeling single events and non-seasonal events

In some cases, time series data may contain unusual patterns that are either re-occurring over time or not. The following are examples of such events:

- **Outliers**: A single event or events that are out of the normal patterns of the series.
- **Structural break**: A significant event that changes the historical patterns of the series. A common example is a change in the growth of the series.
- **Non-seasonal re-occurring events**: An event that repeats from cycle to cycle, but the time at which they occur changes from cycle to cycle. A common example of such an event is the Easter holidays, which occur every year around March/April.

By not expressing in the regression model, this type of events will bias the estimated coefficients, as the model will weight those types of events along with the regular events of the series. The use of hot encoding, binary, or flag variables could help the model to either ignore this type of events or adjust the model coefficients accordingly.

For instance, you can observe in the decompose plot of the `USgas` series shown previously that the series trend had a structural break around the year 2010. While growth before the year 2010 was relatively flat, the slope of the trend changed afterwards, with positive growth. In this case, we can use a binary variable that equals zero for observations before the year 2010 and one year afterwards.

Regressing a `tslm` model with external variables requires a separated `data.frame` object with the corresponding variables. The following example demonstrates the creation process of an external binary variable that equals `0` before the year `2010` and `1` afterward, using the `USgas_df` table:

```
r <- which(USgas_df$ds == as.Date("2014-01-01"))

USgas_df$s_break <- ifelse(year(USgas_df$ds) >= 2010, 1, 0)

USgas_df$s_break[r] <- 1
```

We will now use the new feature to remodel the `USgas` series:

```
md3 <- tslm(USgas ~ season + trend + I(trend^2) + s_break, data = USgas_df)
```

Let's use the `summary` function to review the model output:

```
summary(md3)
```

```
##
## Call:
## tslm(formula = USgas ~ season + trend + I(trend^2) + s_break,
##     data = USgas_df)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -461.4  -55.9   -6.4   67.4  285.8
##
## Coefficients:
##                Estimate  Std. Error t value Pr(>|t|)
## (Intercept)  2647.44816    32.30603   81.95  < 2e-16 ***
## season2      -298.73237    35.05548   -8.52  2.9e-15 ***
## season3      -481.80356    35.05753  -13.74  < 2e-16 ***
## season4      -910.06095    35.06096  -25.96  < 2e-16 ***
## season5     -1094.53085    35.06576  -31.21  < 2e-16 *
```

```
## season6     -1114.15537    35.07195  -31.77  < 2e-16 ***
## season7      -950.33977    35.07952  -27.09  < 2e-16 ***
## season8      -925.83668    35.08849  -26.39  < 2e-16 ***
## season9     -1129.21978    35.09886  -32.17  < 2e-16 ***
## season10    -1039.14697    35.11065  -29.60  < 2e-16 ***
## season11     -783.59194    35.12386  -22.31  < 2e-16 ***
## season12     -256.28337    35.59344   -7.20  1.0e-11 ***
## trend          -1.67443     0.46052   -3.64  0.00035 ***
## I(trend^2)      0.01678     0.00188    8.91  2.4e-16 ***
## s_break        74.57388    28.93187    2.58  0.01063 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 108 on 212 degrees of freedom
## Multiple R-squared:  0.939,  Adjusted R-squared:  0.935
## F-statistic:  234 on 14 and 212 DF,  p-value: <2e-16
```

As can see in the summary of the preceding model, the structural break variable is statistically significant, with a level of `0.01`. Likewise, in the case of outliers or holidays, hot encoding can be applied by setting a binary variable that equals `1` whenever an outlier or non-seasonal re-occurring event occurs, and `0` otherwise.

Note that, once you have trained a forecasting model with the `tslm` function with the use of external variables, you will have to produce the future values of those variables as they are going to be used as input of the forecast.

Forecasting a series with multiseasonality components – a case study

One of the main advantages of the regression model, as opposed to the traditional time series models such as ARIMA or Holt-Winters, is that it provides a wide range of customization options and allows us to model and forecast complex time series data such as series with multiseasonality.

In the following examples, we will use the `UKgrid` series to demonstrate the forecasting approach of a multiseasonality series with a linear regression model.

The UKgrid series

The `UKgrid` series represents the national grid demand for electricity in the UK, and it is available in the **UKgrid** package. This series represents a high-frequency time series data with half-hourly frequency. We will utilize the `extract_grid` function from the **UKgrid** package to define the series, main characteristics (for example, data format, variables, frequency, and so on). This transformation function allows us to aggregate the series frequency from half-hourly to a lower frequency such as hourly, daily, or monthly. As our goal here is to forecast the daily demand in the next 365 days, we will set the series to daily frequency using the `data.frame` structure:

```
library(UKgrid)

UKdaily <- extract_grid(type = "data.frame",
                        columns = "ND",
                        aggregate = "daily")
```

We will use the `head` function to review the series variables:

```
head(UKdaily)
```

We get the following output:

```
##    TIMESTAMP      ND
## 1 2011-01-01 1671744
## 2 2011-01-02 1760123
## 3 2011-01-03 1878748
## 4 2011-01-04 2076052
## 5 2011-01-05 2103866
## 6 2011-01-06 2135202
```

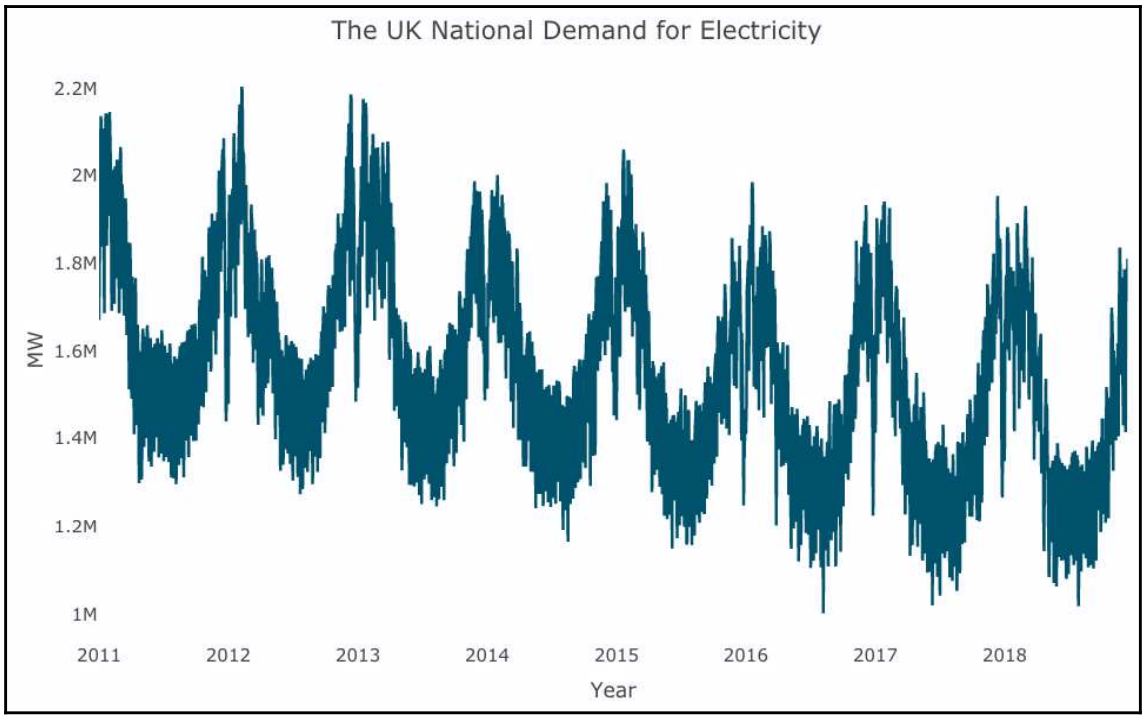
As you can see, this series has two variables:

- `TIMESTAMP`: A date object used as the series timestamp or index
- `ND`: The net demand of electricity

We will use the `ts_plot` function to plot and review the series structure:

```
ts_plot(UKdaily,
        title = "The UK National Demand for Electricity",
        Ytitle = "MW",
        Xtitle = "Year")
```

The following shows the output:



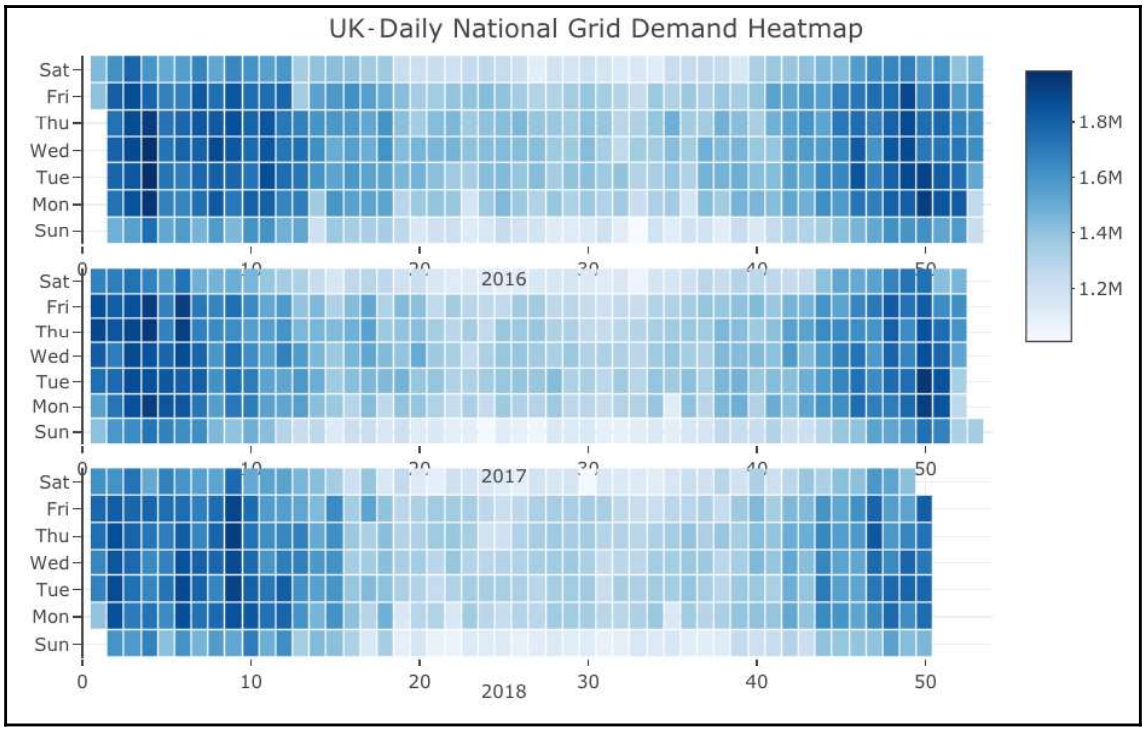
As you can see in the preceding plot, the series has a clear downtrend and has a string seasonal pattern. As we saw in `Chapter 6`, *Seasonality Analysis*, this series has multiple seasonality patterns:

- **Daily**: A cycle of 365 days a year
- **Day of the week**: A 7-day cycle
- **Monthly**: Effected from the weather

Evidence for those patterns can be seen in the following heatmap of the series since 2016 using the `ts_heatmap` function from the **TSstudio** package:

```
ts_heatmap(UKdaily[which(year(UKdaily$TIMESTAMP) >= 2016),],
           title = "UK the Daily National Grid Demand Heatmap")
```

The following shows the output:



As you can see in the series heatmap, the overall demand increases throughout the winter weeks (for example, calendar weeks 1 to 12 and weeks 44 to 52). In addition, you can observe the change of the series during the days of the weeks, as the demand increases during the working days of the week, and decreases over the weekend.

Preprocessing and feature engineering of the UKdaily series

In order to capture the seasonal components of the series, we will set the series as daily frequency and create the following two features:

- Day of the week indicator
- Month of the year indicator

In addition, as it is reasonable to assume (we will confirm this assumption with the ACF function once we have transformed the series into a `ts` object) that the series has a strong correlation with the seasonal lags, we will create a lag variable with a lag of 365 observations. We will utilize the **dplyr** package for creating those features:

```
library(dplyr)

UKdaily <- UKdaily %>%
  mutate(wday = wday(TIMESTAMP, label = TRUE),
         month = month(TIMESTAMP, label = TRUE),
         lag365 = dplyr::lag(ND, 365)) %>%
  filter(!is.na(lag365)) %>%
  arrange(TIMESTAMP)
```

Recall that the cost of using a lag variable with a length of *N* is the loss of the first *N* observations (as the lags of those observations cannot be generated from the series). Therefore, we used the filter functions for removing the rows of the table that the `lag365` variable is missing (that is, the first 365 observations).

Let's review the structure of the `UKdaily` table after adding those new features:

```
str(UKdaily)
```

We get the following output:

```
## 'data.frame':    2540 obs. of  5 variables:
##  $ TIMESTAMP: Date, format: "2012-01-01" "2012-01-02" ...
##  $ ND       : int  1478868 1608394 1881072 1956360 1936635 1939424
1698505 1679311 1898593 1922898 ...
##  $ wday     : Factor w/ 7 levels "Sun","Mon","Tue",..: 1 2 3 4 5 6 7 1 2
3 ...
##  $ month    : Factor w/ 12 levels "Jan","Feb","Mar",..: 1 1 1 1 1 1 1 1
1 1 ...
##  $ lag365   : int  1671744 1760123 1878748 2076052 2103866 2135202
2121523 1861515 1837427 2093269 ...
```

As the `tslm` function input must be in a `ts` format (at least for the series), we will convert the series to a `ts` object. We will use the first timestamp of the series and the `year` and `yday` (the day of the year) functions from the **lubridate** package to set the object starting point:

```
start_date <- min(UKdaily$TIMESTAMP)

start <- c(year(start_date), yday(start_date))
```

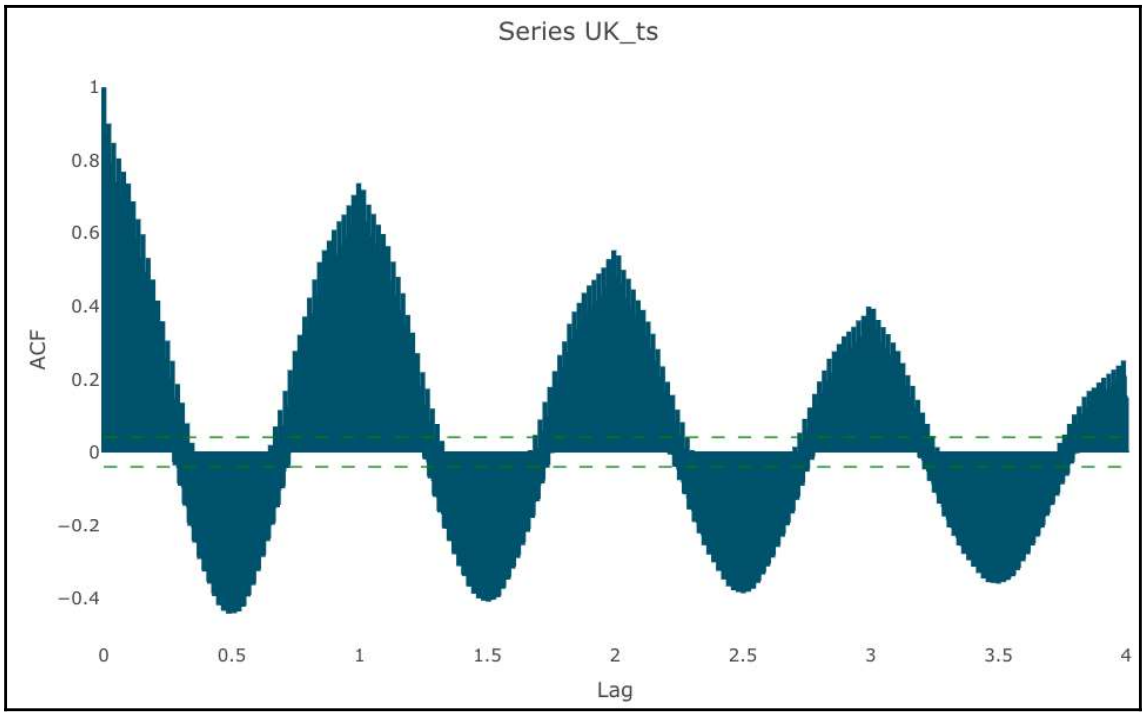
As we saw in `Chapter 3`, *The Time Series Object*, we will use the `ts` function from the **stats** package to set the `ts` object:

```
UK_ts <- ts(UKdaily$ND,
            start = start,
            frequency = 365)
```

After we transform the series into a `ts` object, we can go back and confirm the assumption we made about the correlation level between the series and its seasonal lags with the `ts_acf` function (an interactive version of the `acf` function from the **TSstudio** package). We will review the correlation of the series with its lags from the past four years:

```
ts_acf(UK_ts, lag.max = 365 * 4)
```

We get the following output:



The preceding ACF plot confirms our assumption, and the series has a strong relationship with the seasonal lags, in particular lag 365, the first lag.

As a side note, you can be sure that the series also has a strong correlation with the weekly lags (that is, lag 7, 14, 21, and so on). However, generally, it is not recommended that you use lags that are smaller than the forecast horizon (for example, lag 7, when the forecast horizon is 365), as you will have to forecast those lags as well to be able to use them as an input in the model. This involves additional effort, as you will have to forecast those lags. Furthermore, it may increase the forecast bias as we are using forecasted values as inputs.

Now, after we have created the new features for the model and set the `ts` object, we are ready to split the input series and the corresponding external features object we created (`UKdaily`) into a training and testing partition. As our goal is to forecast the next 365 observations, and the length of the series is large enough (2,540 observations), we can afford to set the testing partition to the length of the forecasting horizon—365 observations. We will set `h`, an indicator variable, to `365` and use it to define the partitions and, later on, the forecast horizon:

```
h <-  365
```

As before, we will split the series into a training and testing partitions with the `ts_split` function:

```
UKpartitions <- ts_split(UK_ts, sample.out = h)

train_ts <- UKpartitions$train
test_ts <- UKpartitions$test
```

In a similar manner, we have to split the features we created for the regression model (the seasonal and lag features) into a training and testing partition following the exact same order as we used for the corresponding `ts` object. We will use the `data.frame` index functionality to set the `UKdaily` table to training and testing partitions:

```
train_df <- UKdaily[1:(nrow(UKdaily) - h), ]
test_df <- UKdaily[(nrow(UKdaily) - h + 1):nrow(UKdaily), ]
```

Training and testing the forecasting model

After we have created the different features for the model, we are ready to start the training process of the forecasting model. We will use the training partition and start to train the following three models:

- **Baseline model**: Regressing the series with the seasonal and trend component using the built-in features of the `tslm` function. As we set the series frequency to `365`, the seasonal feature of the series refers to the daily seasonality.
- **Multiseasonal model**: Adding the day of the week and month of the year indicators for capturing the multiseasonality of the series.
- **A multiseasonal model with a seasonal lag**: Using, in addition to the seasonal indicators, the seasonal lag variable.

The comparison of these three models will be based on the following criteria:

- Performance of the model on the training and testing set using the MAPE score
- Visualizing the fitted and forecasted values versus the actual values of the series using the `test_forecast` function

Starting with a simplistic model (baseline model) will allow us to observe whether adding new features contributes to the model performance or whether we should avoid it, as adding more features or complexity to the model does not always yield better results.

We will start with the baseline model, regressing the series with its seasonal and trend components:

```
md_tslm1 <- tslm(train_ts ~ season + trend)
```

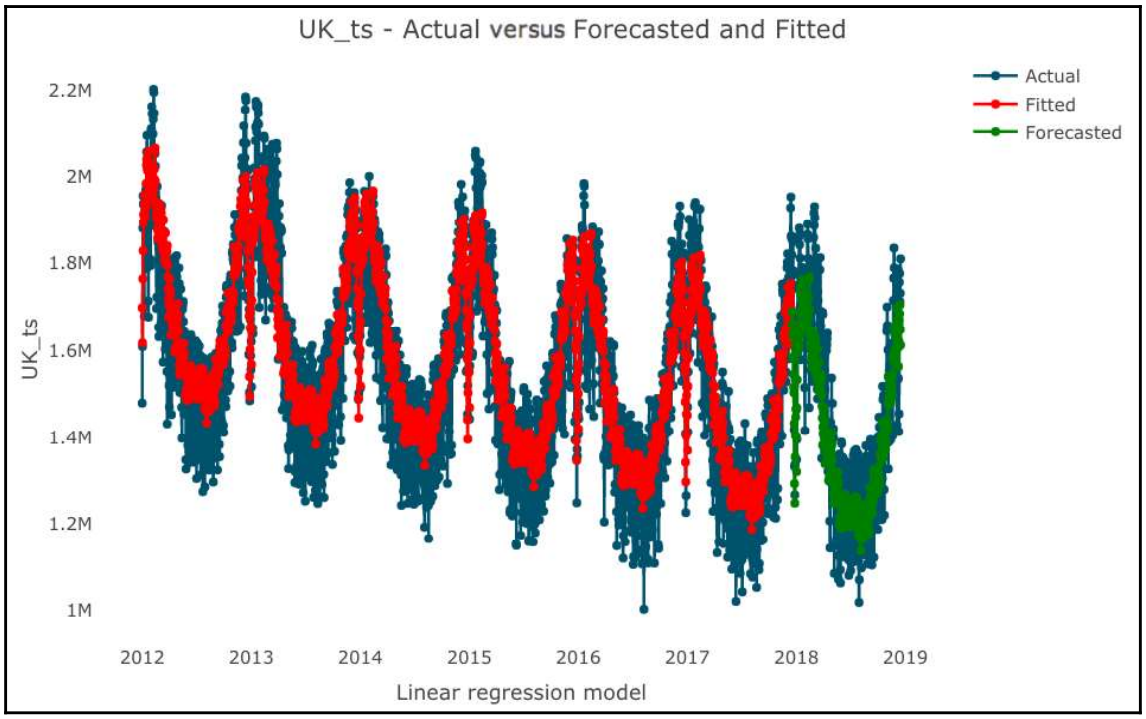
Next, we will utilize the trained model, `md_tslm1`, to forecast the next 365 days of the series, corresponding to the observations of the testing partition, using the `forecast` function:

```
fc_tslm1 <- forecast(md_tslm1, h = h)
```

Let's compare the model performance on the training and testing sets using the `test_forecast` function:

```
test_forecast(actual = UK_ts,
              forecast.obj = fc_tslm1,
              test = test_ts)
```

The output is shown as follows:



We can observe from the preceding performance plot that the baseline model is doing a great job of capturing both the series trend and the day of the year seasonality. On the other hand, it fails to capture the oscillation that related to the day of the week. The MAPE score of the model, as we can see in the output of the following `accuracy` function, is 6.09% and 7.77% on the training and testing partitions, respectively:

```
accuracy(fc_tslm1, test_ts)
```

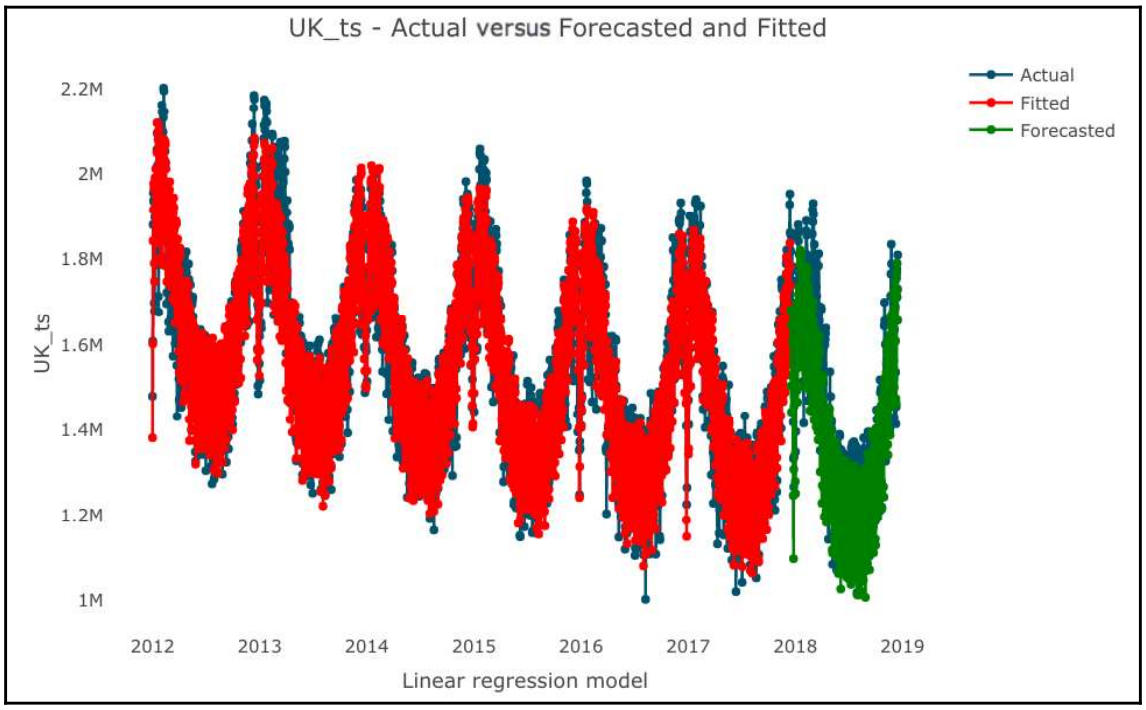
We get the following output:

```
##                                       ME     RMSE       MAE         MPE
## Training set    -0.0000000000007315552 112133.5  92602.47 -0.5399055
## Test set     48298.6969219034435809590 135141.0 113146.70  2.6201812
##                    MAPE      MASE     ACF1 Theil's U
## Training set 6.097994 0.7644557 0.502279        NA
## Test set     7.777603 0.9340533 0.508189  1.136721
```

We will now try improve the model accuracy, by adding the day of the week and month of the year features to the model:

```
md_tslm2 <- tslm(train_ts ~ season + trend + wday + month,
                 data = train_df)
```

As we are now using features from an external data source, we have to specify the input data with the `data` argument. We will now repeat the same process as before, by using a forecast with the trained model and evaluating the model performance:



Looking at the preceding performance plot of the second model, we can observe the contribution of the seasonal features on the forecast, as the second model was able to capture both the trend and the multiseasonal patterns of the series. This can also be observed on the model MAPE score, which dropped to 2.87% and 5.23% in the training and testing partitions, respectively:

```
accuracy(fc_tslm2, test_ts)
```

We get the following output:

```
##                                      ME     RMSE      MAE         MPE
## Training set    -0.000000000001671321 61384.46 45614.64 -0.1462402
## Test set     48475.302282246622780804 98603.29 77795.70  3.1031691
##                   MAPE      MASE      ACF1 Theil's U
## Training set 2.874880 0.3765599 0.7127754        NA
## Test set     5.236208 0.6422223 0.6565167 0.8053068
```

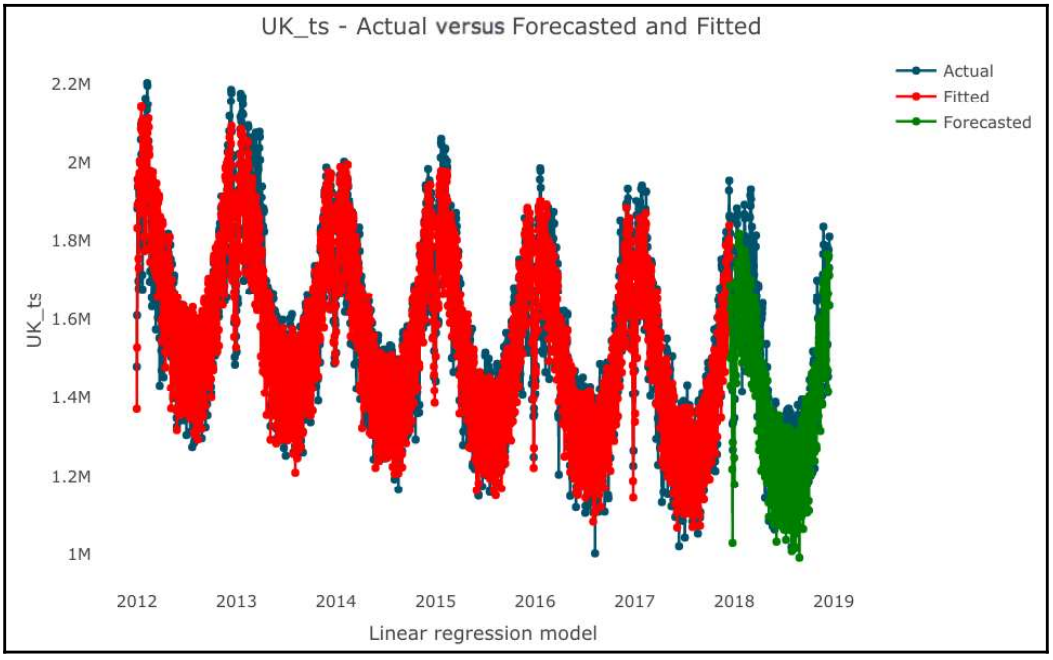
Last but not least, let's add the lag variable to the model, and repeat the same process as before:

```
md_tslm3 <- tslm(train_ts ~ season + trend + wday + month + lag365,
 data = train_df)

fc_tslm3 <- forecast(md_tslm3, h = h, newdata = test_df)
```

The performance of the third model can be seen in the following plot:

```
test_forecast(actual = UK_ts,
              forecast.obj = fc_tslm3,
              test = test_ts)
```

We get the plot as seen in the following graph:



It is hard to observe from the performance plot of the third model, if there is a significant difference from the second model. Therefore, let's review the MAPE score of the third model:

```
accuracy(fc_tslm3, test_ts)
```

We get the following output:

```
##                                      ME     RMSE      MAE       MPE
MAPE
## Training set     0.000000000003641324 59271.32 44457.55 -0.137529
2.814302
## Test set     44255.320284163266478572 96357.83 75372.88  2.795172
5.078317
##                    MASE      ACF1 Theil's U
## Training set 0.3670078 0.7020265        NA
## Test set     0.6222213 0.6245326 0.7888625
```

The results of the third model show a small improvement in the model accuracy with 2.81% on the training set and 5.07% on the testing set.

Model selection

Now it's time to select a model. At this point, it is clear that the second and third models perform better than the first model. As both the second and third model achieved a fairly similar MAPE score, with a small advantage for the third model, we should ask ourselves whether an improvement of less than 0.2% on the error rate of the testing set is worth the cost of using the lag variable (that is, the loss of 365 observations and additional cost of a degree of freedom for the model).

It depends.

There is no straightforward answer to this question. Furthermore, it depends on the goal of the forecast. It is recommended that you consider the following test:

- **The first question you should ask in this case**: Is the lag variable statistically significant? If the variable is not statistically significant, there is no point in continuing the discussion, and it is better to drop the variable. In the case of the third model, we can use the `summary` function to observe the level of significance of the `lag365` variable:

```
summary(md_tslm3)$coefficients %>% tail(1)
```

We get the following output:

```
##         Estimate Std. Error t value Pr(>|t|)
## lag365   -0.251      0.022   -11.4 4.06e-29
```

The *p*-value of the `lag365` variable indicated that the variable is statistically significant at a level of 0.001.

- Similarly, we can apply a single ANOVA test with the `anova` function from the **stats** package, and check if the additional variable is significant:

```
anova(md_tslm3)
```

We get the following output:

```
## Analysis of Variance Table
##
## Response: train_ts
##             Df   Sum Sq  Mean Sq F value Pr(>F)
## season     364 6.70e+13 1.84e+11   43.12 <2e-16 ***
## trend        1 1.53e+13 1.53e+13 3595.90 <2e-16 ***
## wday         6 1.91e+13 3.18e+12  745.71 <2e-16 ***
## month       11 6.42e+10 5.84e+09    1.37   0.18
## lag365       1 5.55e+11 5.55e+11  129.98 <2e-16 ***
## Residuals 1791 7.64e+12 4.27e+09
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The ANOVA test also indicated that the `lag365` variable is statistically significant.

- **Backtesting**: It might be the case that the third model is more accurate just by chance and not because the lag variable contributes to the model accuracy, as the difference is relatively small. Therefore, the backtesting of both models could help to validate whether the contribution of the lag variable is consistent over several testing periods. I will leave it to you to conduct backtesting for both of the models as a fun exercise.

More robust methods can apply for feature selection such as stepwise, ridge, or lasso regression. Although those methods are not in the scope of this book, it is recommended that you read about them. In `Chapter 12`, *Forecasting with Machine Learning Models*, we will explore advanced regression approaches with machine learning models, which include feature selection methods.

For the sake of simplicity, we will go with the accuracy criteria and select the third model to forecast the series. The next step is to retrain the model on all series and forecast the next 365 days:

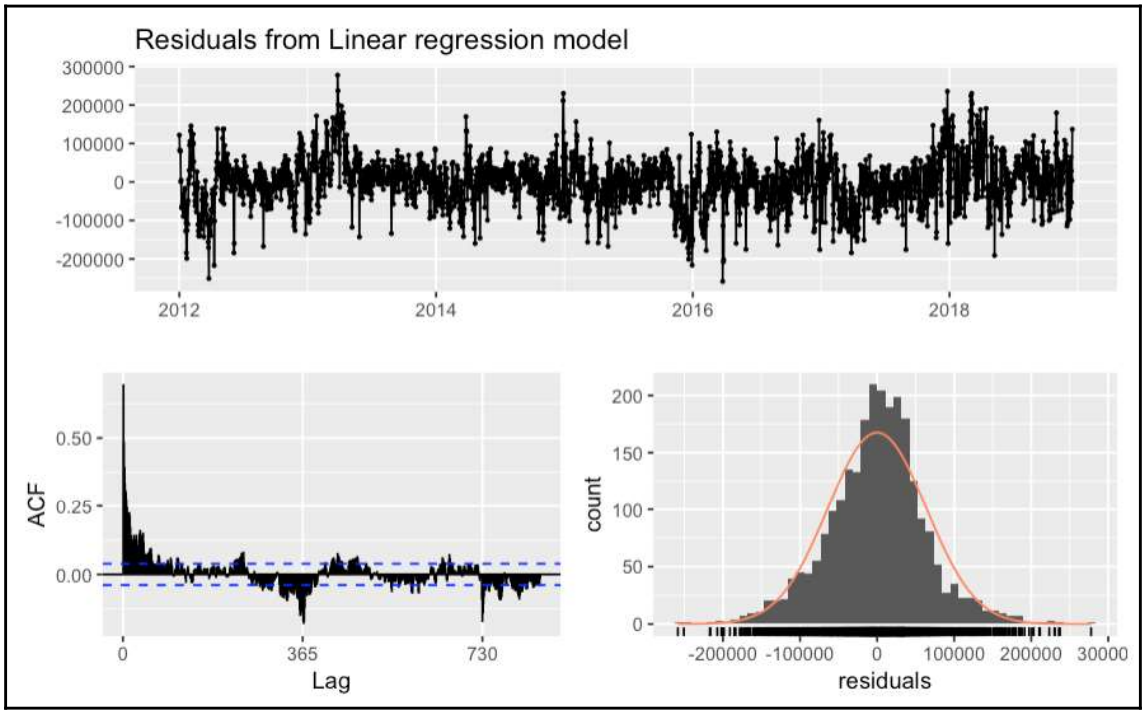
```
final_md <- tslm(UK_ts ~ season + trend + wday + month + lag365,
 data = UKdaily)
```

Residuals analysis

Just before we finalize the forecast, let's do some analysis of the selected model residuals using the `checkresiduals` function:

```
checkresiduals(final_md)
```

We get the following output:



As you can see from the preceding residual summary plot, the residuals are not white noise, as some autocorrelation exists between the residuals series and their lags. This is technically an indication that the model did not capture all the patterns or information that exists in the series. One way to address this issue is to identify additional variables that can explain the variation in the residuals. The main challenge with this approach is that it is hard to identify external variables that can explain the variation of the residuals, and are also feasible to forecast. For example, it is reasonable to assume that weather patterns affect the demand for electricity, yet it is hard to predict those weather patterns a year ahead.

An alternative approach, when patterns *leftover* in the residuals of the model is to treat the model's residuals as a separate time series data and to model it with other time series forecasting model. A common approach is a regression with ARIMA error, which we will introduce in `Chapter 11`, *Forecasting with ARIMA Models*.

Finalizing the forecast

Let's finalize the process and utilize the selected trained model to forecast the future 365 observations. Since we used external variables with the `tslm` function, we will have to generate their future values. This is relatively simple, as we used deterministic variables. Therefore, we will create a `data.frame` object with the values of `wday`, `month`, and `lag365` for the next 365 future observations. A simplistic approach is to first create the corresponding dates of the forecasted observations, and then extract from this object the day of the week and month of the year:

```
UK_fc_df <- data.frame(date = seq.Date(from = max(UKdaily$TIMESTAMP) +
days(1),
                                       by = "day",
                                       length.out = h))
```

Next, we can utilize the `date` variable for creating the `wday` and `month` variables with the **lubridate** package:

```
UK_fc_df$wday <- factor(wday(UK_fc_df$date, label = TRUE), ordered = FALSE)

UK_fc_df$month <- factor(month(UK_fc_df$date, label = TRUE), ordered =
FALSE)

UK_fc_df$lag365 <- tail(UKdaily$ND, h)
```

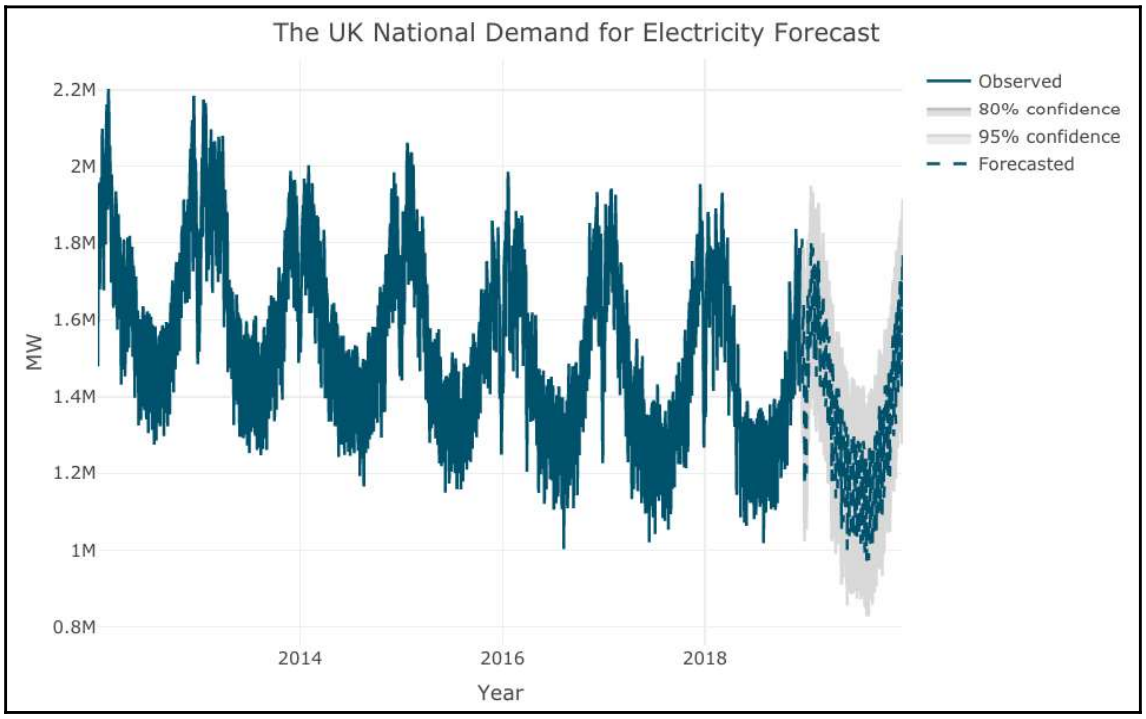
Let's use the `forecast` function to create the forecast:

```
UKgrid_fc <- forecast(final_md, h = h, newdata = UK_fc_df)
```

Last but not least, we will plot the final forecast with the `plot_forecast` function from the **TSstudio** package:

```
plot_forecast(UKgrid_fc,
              title = "The UK National Demand for Electricity Forecast",
              Ytitle = "MW",
              Xtitle = "Year")
```

We get the following output:



Summary

In this chapter, we introduced the forecasting applications of the linear regression model. Although the linear regression model was not designed to handle time series data, with simple feature engineering we can transform a forecasting problem into a linear regression problem. The main advantage of the linear regression model with respect to other traditional time series models is the ability of the model to incorporate external variables and factors. Nevertheless, this model can handle time series with multiseasonality patterns, as we saw with the UK demand for electricity forecast. Last but not least, the forecasting approaches we demonstrated in this chapter will be the base for advanced modeling with machine learning models that we will discuss in `Chapter 12`, *Forecasting with Machine Learning Models*.

In the next chapter, we will introduce the exponential smoothing methods, a family of forecasting models based on a weighted moving average approach.

10
Forecasting with Exponential Smoothing Models

In `Chapter 5`, *Decomposition of Time Series Data*, we looked at the application of smoothing functions for noise reduction in time series data and trend estimation. In this chapter, we will expand on the use of smoothing functions and introduce their forecasting applications. This family of forecasting models can handle a variety of time series types, from series with neither trends nor seasonal components to series with both trends and seasonal components. We will start with the basic moving average model and simple exponential smoothing models, and then add more layers to the model, as well as the model's ability to handle complex time series data.

In this chapter, we will cover the following topics:

- Forecasting with moving average models
- Forecasting approaches with smoothing models
- Tuning parameters for smoothing models

Technical requirement

The following packages will be used in this chapter:

- **forecast**: Version 8.5 and above
- **h2o**: Version 3.22.1.1 and above and Java version 7 and above
- **TSstudio**: Version 0.1.4 and above
- **plotly**: Version 4.8 and above
- **dplyr**: Version 0.8.1 and above
- **tidyr**: Version 0.8.3 and above
- **Quandl**: Version 2.9.1 and above

You can access the codes for this chapter from the following link:

`https://github.com/PacktPublishing/Hands-On-Time-Series-Analysis-with-R/tree/master/Chapter10`

Forecasting with moving average models

In `Chapter 5`, *Decomposition of Time Series Data*, we looked at the application of the moving average functions to smooth time series data. Those functions, with a small tweak, can be used as a forecasting model. In the upcoming section, we will introduce two of the most common moving average forecasting functions—the simple and weighted moving average. These models, as you will see later on in this chapter, are the foundation for the exponential smoothing forecasting models.

The simple moving average

The **simple moving average** (**SMA**) function, which we used in `Chapter 5`, *Decomposition of Time Series Data*, for smooth time series data can be utilized, with some simple steps, as a forecasting model. First, let's recall the structure of the SMA function for smoothing time series data:

$$Y'_t = \frac{Y_{t-\frac{m-1}{2}} + \ldots + Y_t + \ldots + Y_{t+\frac{m-1}{2}}}{m}$$

This occurs when we use a two-sided rolling window. Let's take a look at the following equation:

$$Y'_t = \frac{Y_{t-m+1} + \ldots + Y_{t-1} + Y_t}{m}$$

This occurs when we use a left tail rolling window. Here the following applies:

- Y_t represents the t observation of series Y with T observations, where $1 \leq t \leq T$
- Y'_t represents the smoothed value of Y_t
- m represents the length of the rolling window, where $m \leq T$

As you can see, in both cases (and generally with any other type of rolling window), the rolling window includes the *t* observation of the series. Converting an SMA smoothing function into a forecasting model is based on averaging the past consecutive *m* observations of the series. For example, $\hat{Y}_{t+1|t=T}$, which is the value of the first forecasting observation of the series (that is, observation *T+1*) is equal to the average of the past *m* observations:

$$\hat{Y}_{t+1|t=T} = \frac{Y_{t-m+1}+\ldots+Y_{t-1}+Y_t}{m}$$

Where, the following terms are used in the preceding equation:

- $Y_{t-m+1}+\ldots+Y_{t-1}+Y_t$ are the past last *m* observations of series *Y* with *T* observations, given $t = T$
- $\hat{Y}_{t+1|t=T}$ represents the first forecasted value of a series with *T* observations (that is, given *t* is equal to the last observation of the series *T*)
- Like the previous equation, *m* represents the length of the rolling window

You can observe from the preceding formulas of both the smoothing function (Y'_t) and forecasting functions ($\hat{Y}_{t+1|t=T}$) that the smoothed value of the last observation of the series (observation *T*) is the forecasted value of observation *T+1*. Moreover, only the first forecasted value (observation *T+1*) is constructed by averaging only actual values of the series (for example, $Y_{t-m+1}+\ldots+Y_{t-1}+Y_t$). As we shift the rolling window to forecast the following observations of the series, the actual values are replaced with the previously forecasted values. For instance, the calculation of the second forecasted value on-line, *T+2*, is defined by the following expression:

$$\hat{Y}_{t+2|t=T} = \frac{Y_{t-m+2}+\ldots+Y_{t-1}+Y_t+\hat{Y}_{t+1}}{m}$$

In this case, the function inputs are the last *m-1* observations of the series, along with the first forecasted value, $\hat{Y}_{t+1|t=T}$. From the $T+m+1$ observation and onward, the function inputs are the last *m* forecasted values.

Forecasting with the SMA function is recommended when the input series has no structural patterns, such as trend and seasonal components. In this case, it is reasonable to assume that the forecasted values are relatively close to the last observations of the series.

In the following example, we will create a customized SMA function and use it to forecast the monthly prices of the Robusta coffee prices in the next 12 months. The Robusta coffee prices (USD per kg) are an example of time series data that has no specific trend or seasonal patterns. Rather, this series has a cycle component, where the magnitude and length of the cycle keep changing from cycle to cycle. This series is part of the `Coffee_Prices` dataset and is available on the **TSstudio** package:

```
library(TSstudio)

data(Coffee_Prices)
```

Let's review the properties of the `Coffee_Prices` dataset with the `ts_info` function:

```
ts_info(Coffee_Prices)
```

We get the following output:

```
## The Coffee_Prices series is a mts object with 2 variables and 1402
observations
## Frequency: 12
## Start time: 1960 1
## End time: 2018 5
```

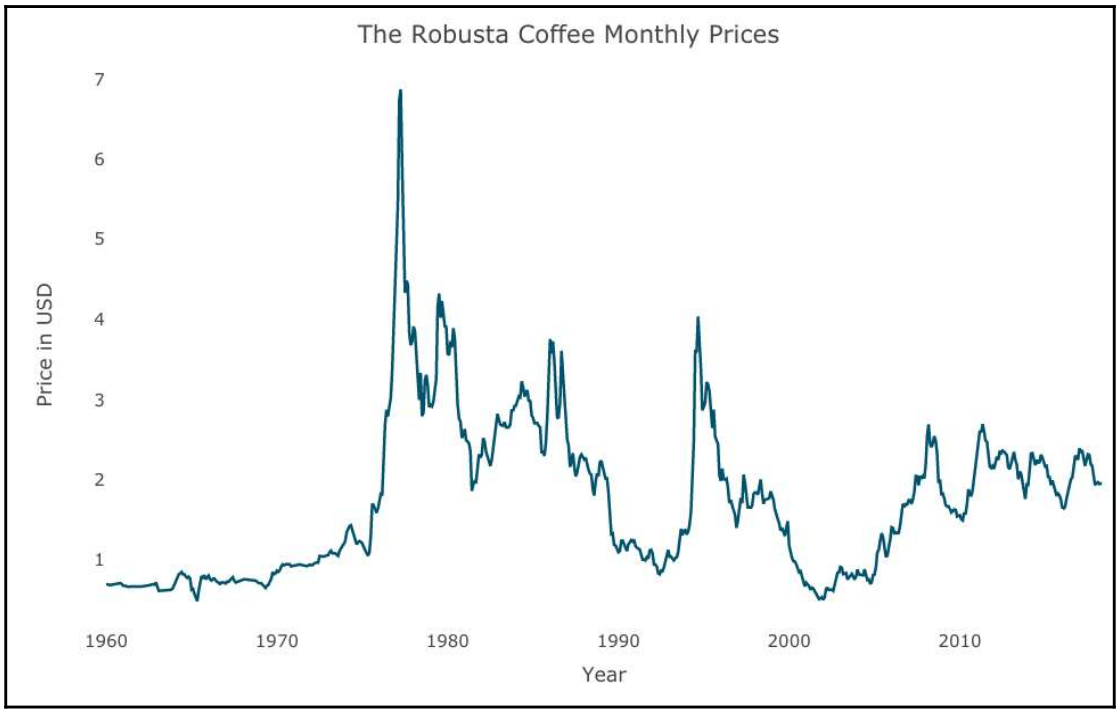
This dataset is an `mts` object and contains both the monthly prices (USD per kg) of the Robusta and Arabica coffee prices between 1960 and 2018. We will extract the monthly prices of the Robusta coffee from the `Coffee_Prices` object:

```
robusta <- Coffee_Prices[,1]
```

Let's review the series:

```
ts_plot(robusta,
        title = "The Robusta Coffee Monthly Prices",
        Ytitle = "Price in USD",
        Xtitle = "Year")
```

The output is as follows:



Next, we will create a basic SMA function using some of the functionality of the **tidyr** package:

```
library(tidyr)

sma_forecast <- function(df, h, m, w = NULL){
  # Error handling
  if(h > nrow(df)){
    stop("The length of the forecast horizon must be shorter than the
length of the series")}
  if(m > nrow(df)){
    stop("The length of the rolling window must be shorter than the length
of the series")}
  if(!is.null(w)){
    if(length(w) != m){
      stop("The weight argument is not aligned with the length of the
rolling window")
    } else if(sum(w) !=1){
      stop("The sum of the average weight is different than 1")
    }}
```

```
  # Setting the average weigths
  if(is.null(w)){
    w <- rep(1/m, m)
  }
  ### Setting the data frame ###
  # Changing the Date object column name
  names(df)[1] <- "date"
  # Setting the training and testing partition
  # according to the forecast horizon
  df$type <- c(rep("train", nrow(df) - h), rep("test", h))
  # Spreading the table by the partition type
  df1 <- df %>% spread(key = type, value = y)
  # Create the target variable
  df1$yhat <- df1$train
  # Simple moving average function
  for(i in (nrow(df1) - h + 1):nrow(df1)){
    r <- (i-m):(i-1)
    df1$yhat[i] <- sum(df1$yhat[r] * w)
  }
  # dropping from the yhat variable the actual values
  # that were used for the rolling window
  df1$yhat <- ifelse(is.na(df1$test), NA, df1$yhat)
  df1$y <- ifelse(is.na(df1$test), df1$train, df1$test)
  return(df1)
}
```

The function arguments are as follows:

- `df`: The input series in a two-column data frame format, where the first column is a `Date` object and the second one is the actual values of the series.
- `h`: The horizon of the forecast. For the purpose of the following example, the function set the last `h` observations as a testing set. This allows us to compare model performance.
- `m`: The length of the rolling window.
- `w`: The weights of the average, by default, using equal weights (or arithmetic average).

The `sma_forecast` function has the following components:

- **Error handling**: Test and verify whether the input arguments of the function are valid. If one of the defined tests isn't true, it will stop the function from running and trigger an error message.
- **Data preparation**: This defines the `data.frame` object based on the window length and the forecast horizon.
- **Data calculation**: Calculates the simple moving average and return the results.

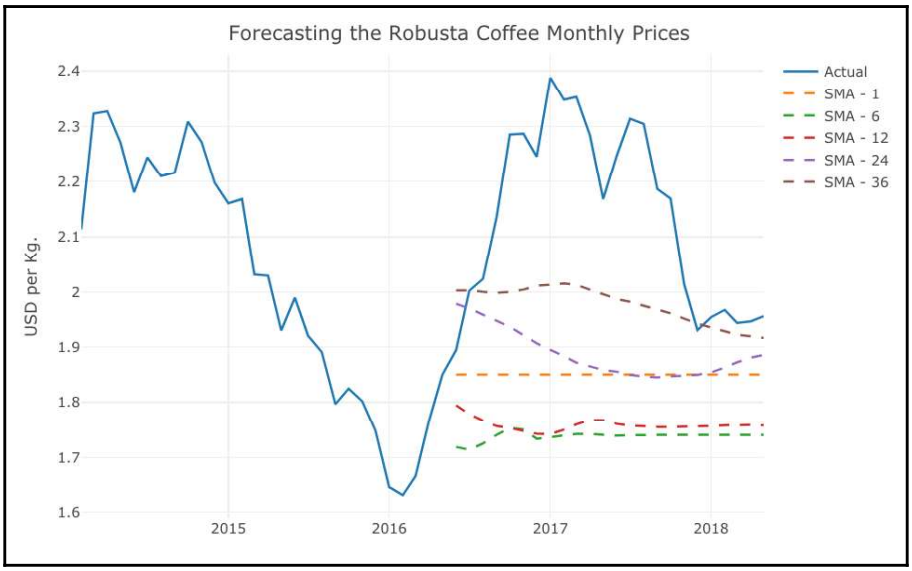
Let's utilize this function to demonstrate the performance of the SMA function. We will forecast the last `24` months of the Robusta series using a rolling window of `3`, `6`, `12`, `24`, and `36` months:

```
robusta_df <- ts_to_prophet(robusta)

robusta_fc_m1 <-  sma_forecast(robusta_df, h = 24, m = 1)
robusta_fc_m6 <-  sma_forecast(robusta_df, h = 24, m = 6)
robusta_fc_m12 <- sma_forecast(robusta_df, h = 24, m = 12)
robusta_fc_m24 <- sma_forecast(robusta_df, h = 24, m = 24)
robusta_fc_m36 <- sma_forecast(robusta_df, h = 24, m = 36)
```

We will use the **plotly** package to plot the results of the different moving average functions:

```
library(plotly)

 plot_ly(data = robusta_df[650:nrow(robusta_df),], x = ~ ds, y = ~ y,
         type = "scatter", mode = "lines",
         name = "Actual") %>%
   add_lines(x = robusta_fc_m1$date, y = robusta_fc_m1$yhat,
             name = "SMA - 1", line = list(dash = "dash")) %>%
   add_lines(x = robusta_fc_m6$date, y = robusta_fc_m6$yhat,
             name = "SMA - 6", line = list(dash = "dash")) %>%
   add_lines(x = robusta_fc_m12$date, y = robusta_fc_m12$yhat,
             name = "SMA - 12", line = list(dash = "dash")) %>%
   add_lines(x = robusta_fc_m24$date, y = robusta_fc_m24$yhat,
             name = "SMA - 24", line = list(dash = "dash")) %>%
    add_lines(x = robusta_fc_m36$date, y = robusta_fc_m36$yhat,
             name = "SMA - 36", line = list(dash = "dash")) %>%
   layout(title = "Forecasting the Robusta Coffee Monthly Prices",
          xaxis = list(title = ""),
          yaxis = list(title = "USD per Kg."))
```

We get the following plot:



The main observations from the preceding plot are as follows:

- If the length of the rolling window is shorter:
    - The range of the forecast is fairly close to the most recent observations of the series
    - The faster the forecast converges to some constant value
- If the window length is longer:
    - The longer it takes until the forecast converges to some constant value
    - It can handle better shocks and outliers
- An SMA forecasting model with a rolling window of a length of 1 is equivalent to the Naïve forecasting model

While the SMA function is fairly simple to use and cheap on compute power, it has some limitations:

- The forecasting power of the SMA function is limited to a short horizon and may have poor performance in the long run.
- This method is limited for time series data, with no trend or seasonal patterns. This mainly effects the arithmetic average that smooths the seasonal pattern and becomes flat in the long run.

Weighted moving average

The **weighted moving average** (**WMA**) is an extended version of the SMA function, and it is based on the use of the weighted average (as opposed to arithmetic average). The main advantage of the WMA function, with respect to the SMA function, is that it allows you to distribute the weight of the lags on the rolling window. This can be useful when the series has a high correlation with some of its lags. The WMA function can be formalized by the following expression:

$$\hat{Y_{T+n}} = w_1 Y_{T+n-m} + \ldots + w_m Y_{T+n-1}$$

Here, $\hat{Y_{T+n}}$ is the n forecasted value of the series at time $T + n$, and w is a scalar of a size m which defines the weight of each observation in the rolling window. The use of the weighted average provides more flexibility as it can handle series with a seasonal pattern. In the following example, we will use the `sma_forecast` function we created previously to forecast the last 24 months of the `USgas` dataset. In this case, we will utilize the w argument to set the average weight and therefore transform the function from SMA to WMA. Like we did previously, we will first transform the series into `data.frame` format with the `ts_to_prophet` function:

```
data(USgas)

USgas_df <- ts_to_prophet(USgas)
```

In the following example, we will use the following two strategies:

- The WMA model for applying all the weight on the seasonal lag (lag `12`):

```
USgas_fc_m12a <- sma_forecast(USgas_df,
                              h = 24,
                              m = 12,
                              w = c(1, rep(0,11)))
```

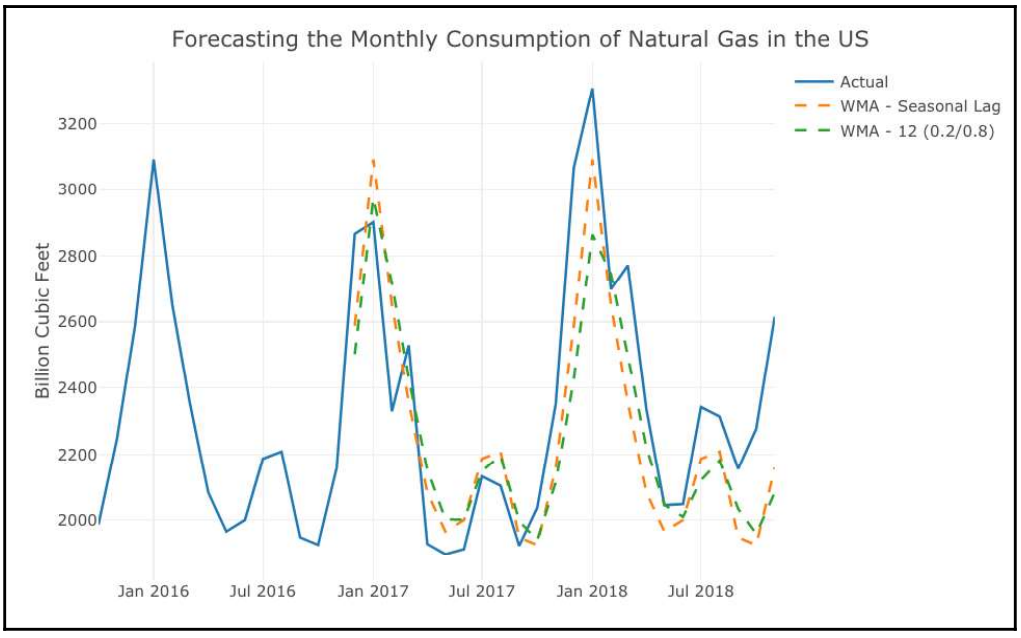
- The WMA model for weighting the first lag with `0.2` and the seasonal lag (lag `12`) with `0.8`:

```
USgas_fc_m12b <- sma_forecast(USgas_df,
                              h = 24,
                              m = 12,
                              w = c(0.8, rep(0,10), 0.2))
```

Let's utilize the **plotly** package to plot the output of both of the WMA models:

```
plot_ly(data = USgas_df[190:nrow(USgas_df),],
        x = ~ ds,
        y = ~ y,
        type = "scatter",
        mode = "lines",
        name = "Actual") %>%
  add_lines(x = USgas_fc_m12a$date,
            y = USgas_fc_m12a$yhat,
            name = "WMA - Seasonal Lag",
            line = list(dash = "dash")) %>%
  add_lines(x = USgas_fc_m12b$date,
            y = USgas_fc_m12b$yhat,
            name = "WMA - 12 (0.2/0.8)",
            line = list(dash = "dash")) %>%
  layout(title = "Forecasting the Monthly Consumption of Natural
Gas in the US",
         xaxis = list(title = ""),
         yaxis = list(title = "Billion Cubic Feet"))
```

We get the following output:



As you can see in the preceding plot, both models captured the seasonal oscillation of the series to some extent. Setting the full weight on the seasonal lag is equivalent to the seasonal Naïve model. This strategy could be useful for a series with a dominant seasonal pattern, such as `USgas`. In the second example, we weighted the average between the most recent lag and the seasonal lag. It would make sense to distribute the weights between the different lags when the series has a high correlation with those lags.

While WMA can capture the seasonal component of a series, it cannot capture the series trend (due to the average effect). Therefore, this method will start to lose its effectiveness once the forecast horizon crosses the length of the series frequency (for example, more than a year for monthly series). Later on in this chapter, we will introduce the Holt-Winters model, which can handle time series with both seasonal and trend components.

Forecasting with exponential smoothing

Among the traditional time series forecasting models, the exponential smoothing functions are one of the most popular forecasting approaches. This approach, conceptually, is close to the moving average approach we introduced previously, as both are based on forecasting the future values of the series by averaging the past observations of the series. The main distinction between the exponential smoothing and the moving average approaches is that the first is averaging all series observations, as opposed to a subset of *m* observations by the latter.

Furthermore, the advance exponential smoothing functions can handle series with a trend and seasonal components. In this section, we will focus on the main exponential smoothing forecasting models:

- Simple exponential smoothing model
- Holt model
- Holt-Winters model

Simple exponential smoothing model

The **Simple exponential smoothing** (**SES**) as its name implies, is the simplest forecasting model among the exponential smoothing family. The main assumption of this model is that the series stays at the same level (that is, the local mean of the series is constant) over time, and therefore, this model is suitable for series with neither trend nor seasonal components. The SES model shares some of the attributes of the WMA model, as both models forecast the future values of the series by a weighted average of the past observations of the series.

On the other hand, the main distinction between the two is that the SES model is utilizing all the previous observations, whereas the WMA model is using only the most recent *m* observations (for a model with a rolling window of a length of *m*). The main attribute of the SES model is the weighted average technique, which is based on the exponential decay of the observation weights according to their chronological distance (that is, series index or timestamp) from the first forecasted values. The decay rate of the observation weights is set by α, the smoothing parameter of the model.

In addition, the SES function is a step function, where the *n* forecasted value of the model becomes the input of the next forecast of the next observations, *n+1*. We can formalize this relationship by using the following equations:

$$\hat{Y_{T+1}} = \alpha Y_T + (1-\alpha)\hat{Y_T}$$

Where, the following terms are used in the preceding equation:

- $\hat{Y_{T+1}}$ is the forecasted value of the observation *T+1* for a series with n observations (for example, $T = 1,...,n$)
- Y_T is the T observation of the series
- α is the smoothing parameter of the model, where $0 < \alpha \leq 1$
- $\hat{Y_T}$ is the forecasted value of observation T at step *T-1*

Since the model assumes that the level of the model doesn't change over time, we can generalize the preceding equation for the forecast of the *T+n* observation of the series (where $n \geq 1$):

$$\hat{Y_{T+n}} = \hat{Y_{T+1}} = \alpha Y_T + (1-\alpha)\hat{Y_T}$$

Hence, the goal of the model is to calculate the series level based on the input data. This results in a flat forecast.

The process of identifying the model level is relatively simple and can be obtained from the preceding model equation. Since this is a recursive equation, we can expand this equation by assigning the value of $\hat{Y_{T-1}}$ the forecasted value of the previous period, *T-1*:

$$\hat{Y_{T+1}} = \alpha Y_T + (1-\alpha)[\alpha Y_{T-1} + (1-\alpha)\hat{Y_{T-1}}] = \alpha Y_T + \alpha(1-\alpha)Y_{T-1} + (1-\alpha)^2\hat{Y_{T-1}})$$

In the same manner, we can keep expanding this equation by recursively assigning the formulas of all the previous observation's forecasts. This expansion process continues all the way to the first forecasted value of the series (in chronological order) $\hat{Y}_2$:

$$\hat{Y_{T+1}} = \alpha Y_T + \alpha(1-\alpha)Y_{T-1} + \alpha(1-\alpha)^2 Y_{T-1} + \ldots + \alpha(1-\alpha)^{T-2} Y_3 + (1-\alpha)^{T-1}\hat{Y}_2$$

Chronologically, the first observation that can be forecast is Y_2 (since there are no available observations to forecast Y_1, the first observation of the series). Therefore, the preceding equation can be expanded to the forecast value of the second observation, or as follows:

$$\hat{Y}_2 = \alpha Y_1 + (1-\alpha)\hat{Y}_1$$

Since there is no supporting data to forecast Y_1 (since this is the first available observation of the series), we need to initialize the value of $\hat{Y}_1$. This is typically done by assigning the actual value of the first observation (that is, $\hat{Y_1} = Y_1$), or by estimating it. We can simplify this equation further by assigning the equation of $\hat{Y_2}$ and rearranging the right-hand side of this equation:

$$\hat{Y_{T+1}} = \alpha Y_T + \alpha(1-\alpha)Y_{T-1} + \alpha(1-\alpha)^2 Y_{T-1} + \ldots + \alpha(1-\alpha)^{T-1} Y_1 + (1-\alpha)^T \hat{Y}_1$$

This can simplify the preceding equation so that it now looks as follows:

$$\hat{Y_{T+1}} = \alpha[(1-\alpha)^0 Y_T + (1-\alpha)^1 Y_{T-1} + (1-\alpha)^2 Y_{T-1} + \ldots + (1-\alpha)^T Y_1] + (1-\alpha)^T \hat{Y}_1$$

This can be compressed further into the following expression:

$$\hat{Y_{T+1}} = \sum_{i=1}^{T} \alpha(1-\alpha)^{i-1} Y_i + (1-\alpha)^T \hat{Y}_1$$

Another interpretation of the forecast equation of the SES model can be observed by manipulating the forecast equation, as follows:

$$\hat{Y_{T+1}} = \alpha Y_T + (1-\alpha)\hat{Y}_T = \alpha Y_T + \hat{Y}_T - \alpha \hat{Y}_T.$$

As we saw in the previous chapter, $Y_t - \hat{Y}_t$ denote the error term of the fitted value of observation *t*. Therefore, the forecast value of the observation *T+1* by the SES model is nothing but the forecasted value of the previous observation, $\hat{Y}_t$, and the proportion of the forecast error term as a function of α, or as follows:

$$\hat{Y_{T+1}} = \hat{Y}_T + \alpha e_T, \textit{ where}$$

e_T is defined as the forecast error for *T* observation of the series (for example, the last observation of a series with *T* observations), or as follows:

$$e_T = Y_T - \hat{Y}_T$$

A more common form for this equation is the component form, which in the case of the SES model is the estimation of $\hat{Y}_{T+1}$ and $\hat{Y}_T$ to denote the model estimation of the series level at time *T* and *T-1*, respectively. The preceding equation can be rewritten like so:

$$\hat{Y_{T+1}} = l_T = \alpha Y_T + (1 - \alpha) l_{T-1}$$

The smoothing parameter of the model, α, defines the rate of the model weight's decay. Since α is closer to *1*, the weights of the most recent observations are higher. On the other hand, the decay rate, in this case, is faster. A special case is a SES model with $\alpha = 1$, which is equivalent to a naive forecasting model:

$$\hat{Y}_{T+1|\alpha=1} = Y_T$$

This is also equivalent to the *n* forecasted value:

$$\hat{Y}_{T+n|\alpha=1} = Y_T$$

The following example demonstrates the decay of the weights of the observations on the most recent `15` observations, for α values between `0.01` to `1`:

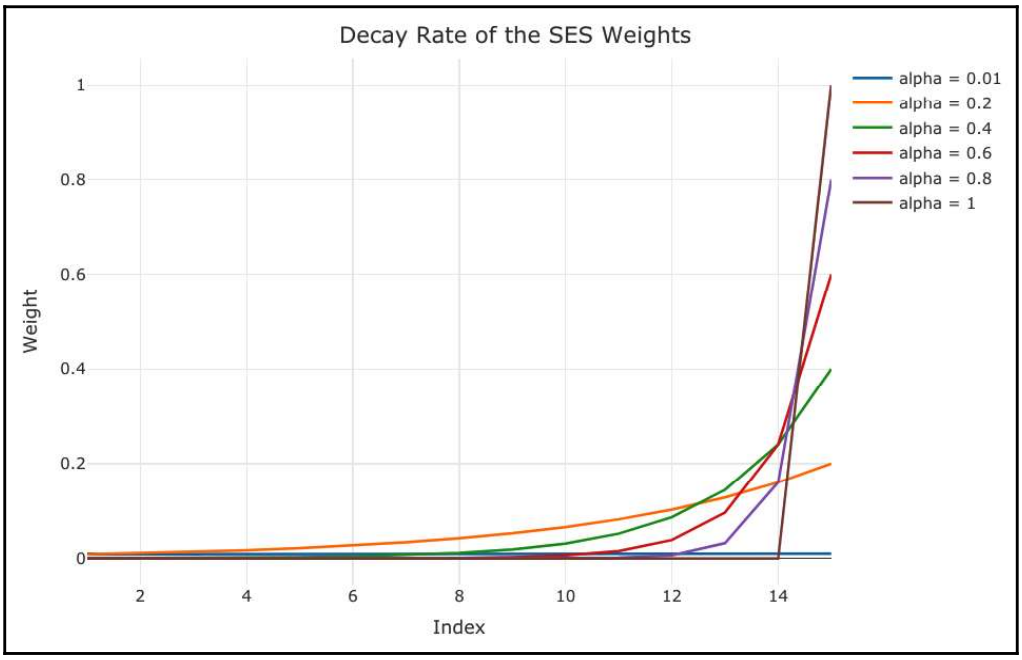
```
alpha_df <- data.frame(index = seq(from = 1, to = 15, by = 1),
                       power = seq(from = 14, to = 0, by = -1))

alpha_df$alpha_0.01 <- 0.01 * (1 - 0.01) ^ alpha_df$power
alpha_df$alpha_0.2 <- 0.2 * (1 - 0.2) ^ alpha_df$power
alpha_df$alpha_0.4 <- 0.4 * (1 - 0.4) ^ alpha_df$power
alpha_df$alpha_0.6 <- 0.6 * (1 - 0.6) ^ alpha_df$power
alpha_df$alpha_0.8 <- 0.8 * (1 - 0.8) ^ alpha_df$power
alpha_df$alpha_1 <- 1 * (1 - 1) ^ alpha_df$power
```

Let's plot the results:

```
plot_ly(data = alpha_df) %>%
  add_lines(x = ~ index, y = ~ alpha_0.01, name = "alpha = 0.01") %>%
  add_lines(x = ~ index, y = ~ alpha_0.2, name = "alpha = 0.2") %>%
  add_lines(x = ~ index, y = ~ alpha_0.4, name = "alpha = 0.4") %>%
  add_lines(x = ~ index, y = ~ alpha_0.6, name = "alpha = 0.6") %>%
  add_lines(x = ~ index, y = ~ alpha_0.8, name = "alpha = 0.8") %>%
```

```
add_lines(x = ~ index, y = ~ alpha_1, name = "alpha = 1") %>%
layout(title = "Decay Rate of the SES Weights",
       xaxis = list(title = "Index"),
       yaxis = list(title = "Weight"))
```

We get the following output:



We can transform the preceding equations into component from, which describes the model by its components. In the case of the SES model, we have a single component—the model level. As we mentioned previously, the model's main assumption is that the series level remains the same over time, and therefore we can rewrite the model equation using the following notations:

$$\hat{Y_{T+1}} = l_T$$

Here, l_T defines the model level at time T, which is a weighted average of Y_T and the level of the previous period, l_{T-1}:

$$\hat{Y_{T+1}} = l_T = \alpha Y_T + (1 - \alpha) l_{T-1}$$

Forecasting with the ses function

The **forecast** package provides a customized SES model with the `ses` function. The main arguments of this function are as follows:

- `initial`: Defines the method for initializing the value of $\hat{Y}_1$, which can be calculated by using the first few observations of the series by setting the argument to simple, or by estimating it with `ets` model (an advanced version of the Holt-Winters model from the **forecast** package) when setting it to optimal.
- `alpha`: Defines the value of the smoothing parameter of the model. If set to `NULL`, the function will estimate it.
- `h`: Sets the forecast horizon.

Let's use the `ses` function to forecast the monthly prices of the Robusta coffee again. We will leave the last `12` months of the series as a testing set for benchmarking the model's performance. We will do this using the `ts_split` function from the **TSstudio** package:

```
robusta_par <- ts_split(robusta, sample.out = 12)
train <- robusta_par$train
test <- robusta_par$test
```

After we set the training and testing partition, we will use the training partition to train a SES model with the `ses` function:

```
library(forecast)
fc_ses <- ses(train, h = 12, initial = "optimal")
```

Let's review the model details:

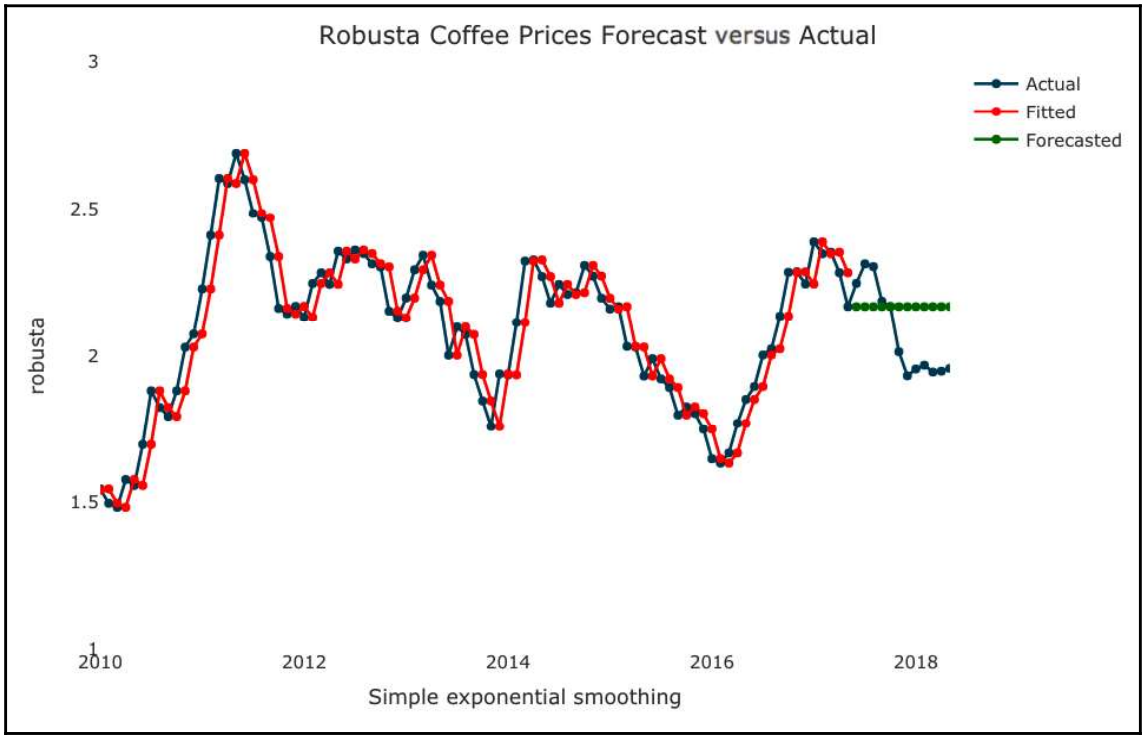
```
fc_ses$model
```

```
## Simple exponential smoothing
##
## Call:
##  ses(y = train, h = 12, initial = "optimal")
##
##   Smoothing parameters:
##     alpha = 0.9999
##
##   Initial states:
##     l = 0.6957
##
##   sigma:  0.161
##
##      AIC     AICc      BIC
## 1989.646 1989.681 2003.252
```

As you can see from the output of the model, the `ses` function set the initial value to 0.69, which is fairly close to the value of the first observations ($Y_1 = 0.696$), and set the `alpha` parameter to `0.9999` (which technically is fairly close to the NAIVE forecast). We can review the model's performance by using the `test_forecast` function:

```
test_forecast(actual = robusta,
              forecast.obj = fc_ses,
              test = test) %>%
  layout(title = "Robusta Coffee Prices Forecast vs. Actual",
         xaxis = list(range = c(2010, max(time(robusta)))),
         yaxis = list(range = c(1, 3)))
```

We get the following output:

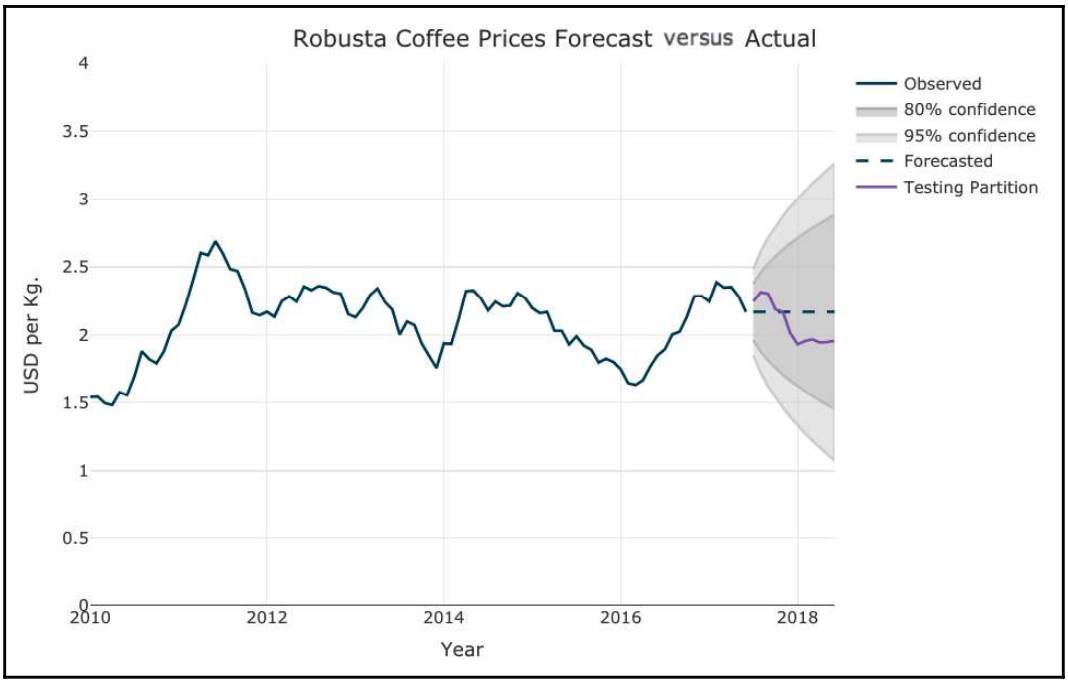


As we can see from the preceding forecast plot, the `ses` function is utilizing the training set to identify the series level by estimating the `alpha` parameter and the initial level of the model (or $\hat{Y}_1$). The forecast value level is fairly close to the value of the last observation of the series since the α value, in this case, is close to *1*. Since the goal of the SES model is to forecast the level of the series, the model won't capture any short-term oscillation.

In the case of a flat forecast, the confidence intervals of the model play a critical role, since the level of uncertainty is higher. Therefore, it will be useful to evaluate whether the forecast values are within the model confidence interval bounds. We will use the `plot_forecast` function from the **TSstudio** package to create an interactive plot for the `fs_ses` model we created and plot the testing set:

```
plot_forecast(fc_ses) %>%
  add_lines(x = time(test) + deltat(test),
            y = as.numeric(test),
            name = "Testing Partition") %>%
  layout(title = "Robusta Coffee Prices Forecast vs. Actual",
         xaxis = list(range = c(2010, max(time(robusta)) +
deltat(robusta))),
         yaxis = list(range = c(0, 4)))
```

The output is as follows:



As you can see from the preceding forecast plot, the testing set is within the range of the **80% confidence** interval.

Model optimization with grid search

The `ses` function optimizes the values of the model parameters (α and l_0) that minimize the **sum of squared errors** (**SSE**) of the model on the training set. An alternative optimization approach is to use a grid search. A grid search is a simplistic but powerful approach that's used to identify the values of the model's parameters that minimize the model error. In the case of the SES model, we will apply a grid search to identify the optimal value of α that minimizes some error metric of the model (for example, MAPE, RMSE, and so on).

In the following example, we will use a grid search to tune the model parameters α and l_0, which minimize the model's MAPE for the Robusta prices series. As we saw in the preceding performance plot of the Robusta forecast, there is a gap between the structure of the fitted values (marked in red) and the forecasted value (marked in green) since the SES model has a flat forecast. Therefore, we will split the model into training, testing, and validation partitions, and for each value of α, we will apply the following steps:

1. Train the SES model using the training partition and forecast the observations of the testing partition
2. Evaluate the performance of the model on both the training and testing partition
3. Append the training and testing partitions (in chronological order) and retrain the model on the new partition (training and testing) before forecasting the values of the validation partition
4. Evaluate the performance of the second model on the validation partition

These steps will allow us to select the alpha value that minimizes the MAPE on the testing partition and use the validation set to validate the performance of the model. Before we set the function, let's set the training, testing, and validation partitions using the `ts_split` function:

```
robusta_par1 <- ts_split(robusta, sample.out = 24)

train1 <- robusta_par1$train
test1 <- ts_split(robusta_par1$test, sample.out = 12)$train

robusta_par2 <- ts_split(robusta, sample.out = 12)

train2 <- robusta_par2$train
valid <- robusta_par2$test
```

We will use the `train1` and `test1` variables for training and testing partition, and `train2` for retraining the model and validating the results on the `valid` partition. The following `alpha` variable defines the search range. We will assign a sequence of values between `0` and `1` with an increment of `0.01` using the `seq` function:

```
alpha <- seq(from = 0, to = 1, by = 0.01)
```

Since the value of `alpha` must be greater than zero, we will replace `0` with a small number that's fairly close to zero:

```
alpha[1] <- 0.001
```

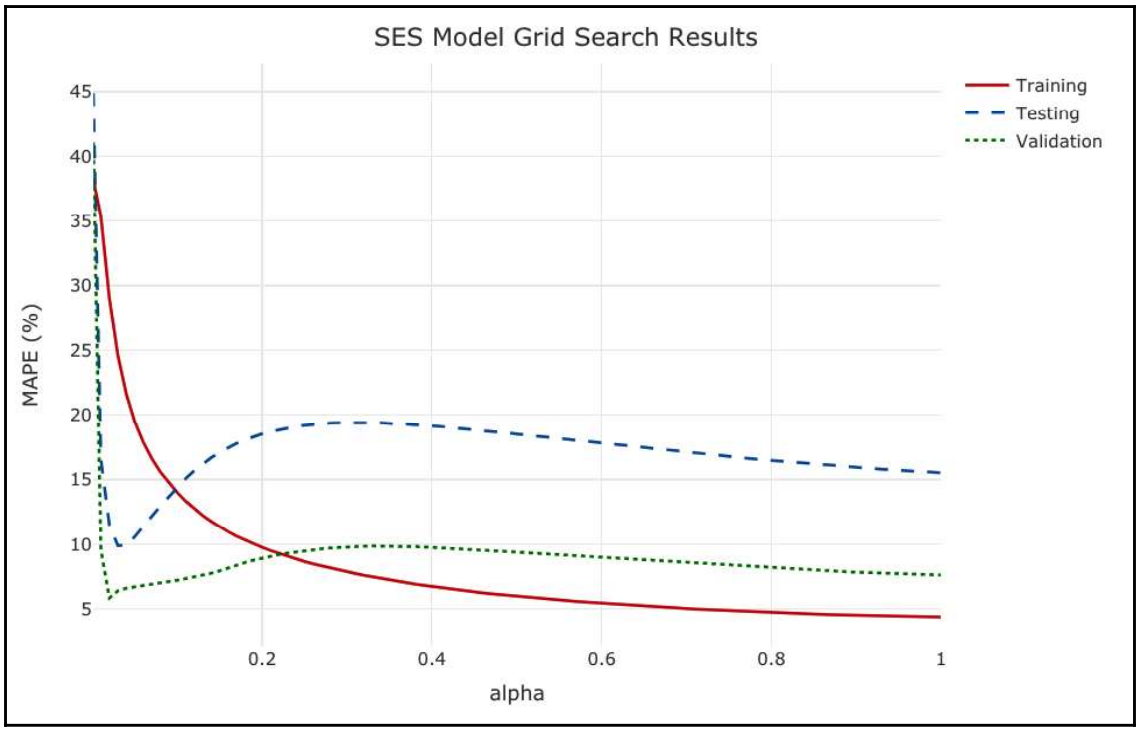
We will use the `lapply` function to iterate on the model using the different values of α:

```
library(dplyr)

ses_grid <- lapply(alpha, function(i){
  md1 <- md_accuracy1 <- md2 <- md_accuracy2 <- results <-  NULL
  md1 <- ses(train1, h = 12, alpha = i, initial = "simple")
  md_accuracy1 <- accuracy(md1, test1)
  md2 <- ses(train2, h = 12, alpha = i, initial = "simple")
  md_accuracy2 <- accuracy(md2, valid)

  resutls <- data.frame(alpha = i,
                        train = md_accuracy1[9],
                        test = md_accuracy1[10],
                        valid = md_accuracy2[10])

 }) %>% bind_rows()
```

As you can see in the following testing results, while $\alpha = 1$ minimizes the MAPE on the training partition, $\alpha = 0.03$ minimizes the error rate on the testing partition. The results on the validation partition are following the same pattern as the testing partition, with an MAPE score of 9.98% on the testing partition and 6.60% on the validation partition:

```
plot_ly(data = ses_grid, x = ~ alpha, y = ~ train,
         line = list(color = 'rgb(205, 12, 24)'),
         type = "scatter",
         mode = "lines",
         name = "Training") %>%
   add_lines(x = ~ alpha, y = ~ test, line = list(color = "rgb(22, 96,
167)", dash = "dash"), name=  "Testing") %>%
   add_lines(x = ~ alpha, y = ~ valid, line = list(color = "green", dash =
"dot"), name = "Validation") %>%
   layout(title = "SES Model Grid Search Results",
          yaxis = list(title = "MAPE (%)"))
```

The output is shown in the following screenshot:



Now, let's take a look at the Holt method.

Holt method

The Holt method, also known as the double exponential smoothing model, is an expanded version of the SES model. It's based on estimating the most recent level and trend with the use of two smoothing parameters, α and β. Once the model estimates the most recent level and trend (L_T, and T_T, respectively), it utilizes them to construct the series forecast using the following equation:

- For a series with additive trend, $\hat{Y_{T+h}} = L_T + hT_T$
- For a series with multiplicative trend, $\hat{Y_{T+1}} = L_T \times hT_T$

Here, the variables of those equations are as follows:

- $\hat{Y_{T+h}}$ is the forecast value of the h forecasted value of a series with T observations
- L_T is the estimate of the series level at time T
- T_T is the estimate of the marginal impact of the series trend at time T (for example, the added value from advance series by one frequency unit)
- h is the number of forecasting steps since time T

Similar to the SES model, the calculation of the series level and trend by the Holt model is based on a weighted average with the use of two smoothing parameters, α and β. For a series with an additive trend, the calculation of the most recent level and trend of the series can be obtained with the following equations:

$$L_T = \alpha Y_T + \alpha(1-\alpha)(L_{T-1} + T_{T-1})$$

$$T_T = \beta(L_T - L_{T-1}) + (1-\beta)T_{T-1}$$

The following equations can be used for a series with a multiplicative trend:

$$L_T = \alpha Y_T + \alpha(1-\alpha)(L_{T-1} \times T_{T-1})$$

$$T_T = \beta(\frac{L_T}{L_{T-1}}) + (1-\beta)T_{T-1}$$

As we saw in the previous chapters, a multiplicative model can transform into an additive model if we apply *log* transformation on both sides of the equation.

The Holt model estimates the values of L_T and T_T, the series level and trend at time T, using a weighted average of all the series observations. In a similar manner to the SES model estimation process, this recursive process starts with the forecast of the second observation of the model:

$$\hat{L}_2 = \alpha Y_1 + \alpha(1-\alpha)(\hat{L}_1 + \hat{T}_1)$$

$$\hat{T}_2 = \beta(\hat{L}_2 - \hat{L}_1) + (1-\beta)\hat{T}_1$$

Where, the following terms are used in the preceding equation:

- $\hat{L}_1$ and $\hat{L}_2$ are the forecast of the level of the first and second observations of the series, respectively
- $\hat{T}_1$ and $\hat{T}_2$ are the trend forecast of the first and second observations of the series, respectively

Since we cannot forecast the level and trend value for the first observation of the series, we will have to approximate $\hat{L}_1$ and $\hat{T}_2$ (similar to the $\hat{L}_1$ approximation process of the SES model). The forecast of the next observations in line (for example, 3 to T) are recursively created using the level and trend estimation, as well as the actual values of the preceding observations.

Forecasting with the holt function

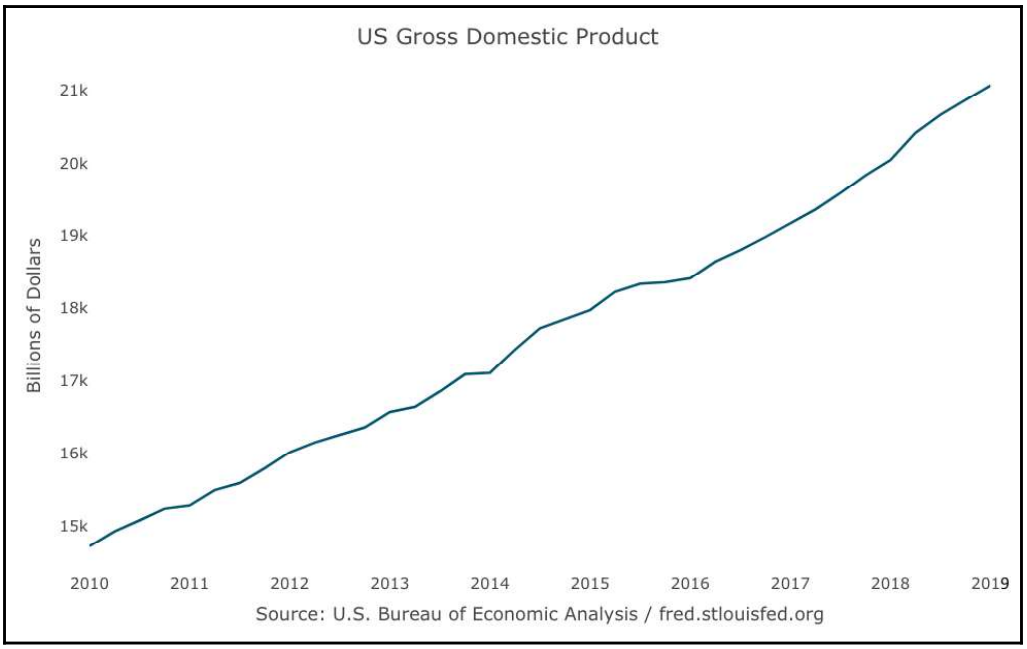
The **forecast** package provides an implementation of the Holt model with the `holt` function. This function automatically initializes the values of $\hat{L}_1$ and $\hat{T}_1$, and identifies the values of α and β that minimize the SSE of the model on the training set. In the following example, we will retrieve the US **Gross Domestic Product** (**GDP**) quarterly data from the **Federal Reserve Economic Data** (**FRED**) API using the **Quandl** package:

```
library(Quandl)

gdp <- Quandl("FRED/GDP", start_date = "2010-01-01", type = "ts")

ts_info(gdp)
##  The . series is a ts object with 1 variable and 37 observations
##  Frequency: 4
##  Start time: 2010 1
##  End time: 2019 1
```

You will notice in the following plot that the GDP series has a strong linear trend and no seasonal component (since the series is seasonally adjusted):

```
ts_plot(gdp,
         title = "US Gross Domestic Product",
         Ytitle = "Billions of Dollars",
         Xtitle = "Source: U.S. Bureau of Economic Analysis /
fred.stlouisfed.org")
```

The output is as follows:



Like we did previously, we will leave the last eight quarters for testing and train the model with the rest of the observations of the series with the `holt` function:

```
gdp_par <- ts_split(gdp, sample.out = 8)

train <- gdp_par$train
test <- gdp_par$test

fc_holt <- holt(train,  h = 8, initial = "optimal")
```

Let's review the model's parameters:

```
fc_holt$model
```

We get the following output:

```
## Holt's method
##
## Call:
##  holt(y = train, h = 8, initial = "optimal")
##
##   Smoothing parameters:
```

```
##     alpha = 0.7418
##     beta  = 0.0001
##
##   Initial states:
##     l = 14583.6552
##     b = 157.8047
##
##   sigma:  80.2774
##
##      AIC     AICc      BIC
## 357.7057 360.3144 364.5422
```

The initialized values of $\hat{L}_1$ and $\hat{T}_1$ of the function are relatively close to the values of the first observation of the series ($Y_1 = 14721.35$) and the average difference between each quarter. In addition, the selected α of 0.74 indicated that the model heavily weighed the last observation of the series, Y_T. On the other hand, the value of β is fairly close to zero, which indicates that updating the trend value from period to period doesn't take into account the change in the level and carries the initial value of the trend, $\hat{T}_1$, forward.

Let's compare the model's performance in the training and testing partitions with the `accuracy` function:

```
accuracy(fc_holt, test)
```

We get the following output:

```
##                       ME      RMSE       MAE           MPE      MAPE
## Training set  -0.2600704  74.53567  58.08445 -0.004075414 0.3403528
## Test set     362.8135830 423.92339 362.81358  1.764266683 1.7642667
##                    MASE       ACF1 Theil's U
## Training set 0.09300079 0.04713728        NA
## Test set     0.58091192 0.67212430  1.776891
```

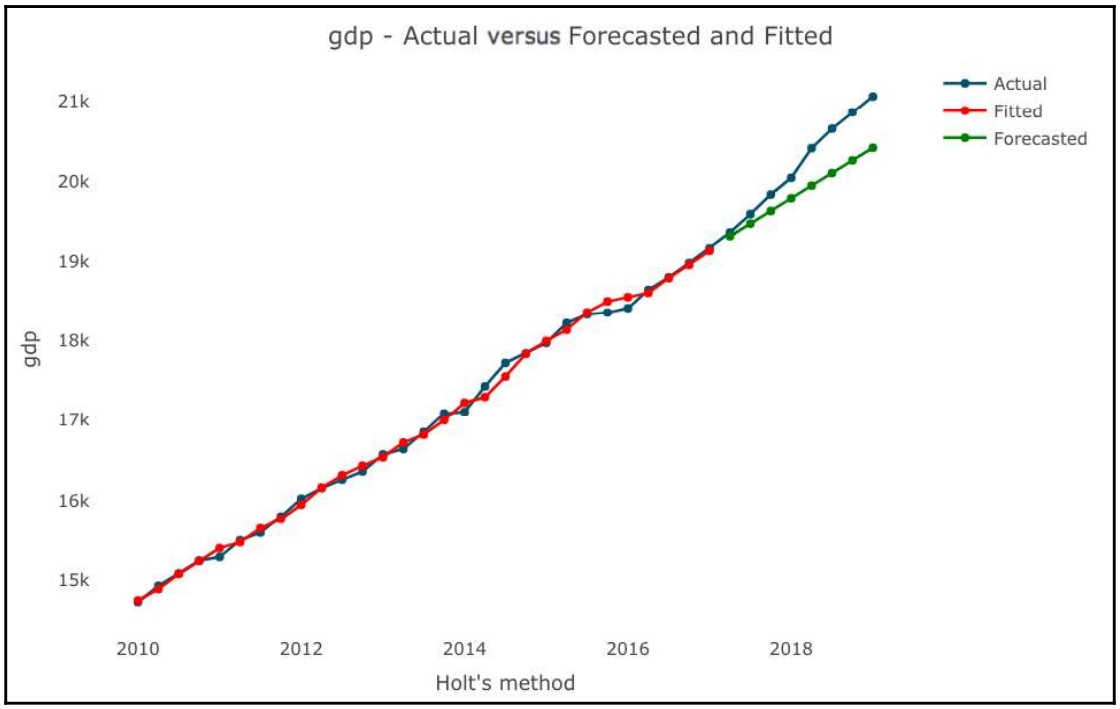
As you can see from the output of the `accuracy` function, the ratio between the error rate on the testing and training set is more than 5 times for the RMSE and 4.5 for the MAPE. This large ratio in the error metrics is mainly derived from the following two reasons:

- The fitted values of the model on the training set are not bound by a linear line (as opposed to the forecast output)
- The growth of the trend in the last few quarters shift from a linear rate of growth to an exponential rate

The changes in the trend growth and the forecast can be observed with the `test_forecast` function:

```
test_forecast(gdp, forecast.obj = fc_holt, test = test)
```

The output is shown in the following screenshot:



While the Holt model was designed to handle time series with the linear trend, the `exponential` argument of the `holt` function provides the option to handle series with exponential or decaying trends when set to `TRUE`. For the preceding example, we can utilize the exponential argument to modify the growth pattern of the trend.

In this case, we would like to have a higher weight for the trend, and we will set β to 0.75 (a more robust approach for identifying the optimal value of β would be to use a grid search):

```
fc_holt_exp <- holt(train,
                    h = 8,
                    beta = 0.75,
                    initial = "optimal",
                    exponential = TRUE)
```

The output of this model is as follows:

```
fc_holt_exp$model
```

We get the following output:

```
## Holt's method with exponential trend
##
## Call:
##  holt(y = train, h = 8, initial = "optimal", exponential = TRUE,
##
##  Call:
##      beta = 0.75)
##
##   Smoothing parameters:
##     alpha = 0.75
##     beta  = 0.75
##
##   Initial states:
##     l = 14586.0855
##     b = 1.0117
##
##   sigma:  0.0064
##
##      AIC     AICc      BIC
## 358.7179 360.4570 364.0467
```

Let's review the model's accuracy on the training and testing set with the `accuracy` function:

```
accuracy(fc_holt_exp, test)
```

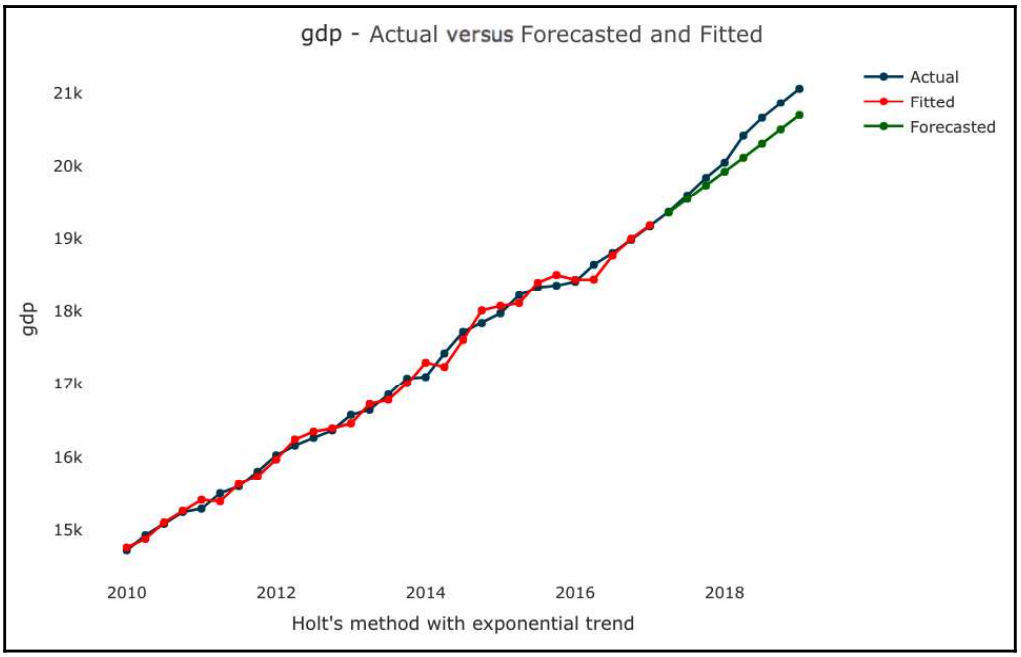
We get the following output:

```
##                       ME     RMSE       MAE         MPE     MAPE
MASE
## Training set  -1.637148 100.0329  83.38575 -0.01160733 0.489303
0.1335115
## Test set     210.935365 255.0026 210.93537  1.02384022 1.023840
0.3377351
##                     ACF1 Theil's U
## Training set -0.2076372        NA
## Test set      0.6778485  1.069639
```

Similarly, we can plot the fitted and forecasted values against the actual data with the `test_forecast` functions:

```
test_forecast(gdp, forecast.obj = fc_holt_exp, test = test)
```

The output is as follows:



As you can see, the error rate of the second Holt model is more balanced, where the ration between the error on the testing and training set is 2.5 and 2.1 for the RMSE and MAPE metrics, respectively.

The use of the exponential or damped arguments required some prior knowledge or assumption on the future growth rate of the trend.

Holt-Winters model

We will close this chapter with the third and most advanced model among the exponential smoothing family of forecasting models—the Holt-Winters model. The **Holt-Winters** (**HW**) model is an extended version of the Holt model and can handle time series data with both trend and seasonal components. Forecasting the seasonal component required a third smoother parameter and equation, in addition to the ones of the level and trend.

Both of the trend and seasonal components could have either an additive or multiplicity structure, which adds some complexity to the model as there are multiple possible combinations:

- Additive trend and seasonal components
- Additive trend and multiplicative seasonal components
- Multiplicative trend and additive seasonal components
- Multiplicative trend and seasonal components

Therefore, before building an HW model, we need to identify the structure of the trend and the seasonal components. The following equations describe the HW model for a series with additive seasonal component:

$$\hat{Y}_{T+1} = L_T + hT_T + S_{T+h-m}$$

$$L_T = \alpha(Y_T - S_{T-m}) + (1-\alpha)(L_{T-1} + T_{T-1})$$

$$T_T = \beta(L_T - L_{T-1}) + (1-\beta) + (1-\beta)T_{T-1}$$

$$S_T = \gamma(Y_T - L_T) + (1-\gamma)S_{T-m}$$

The following equations describe the HW model for a series with a multiplicative seasonal structure:

$$\hat{Y}_{T+1} = (L_T + kT_T)S_{T+k-m}$$

$$L_T = \frac{\alpha Y_T}{S_{T-m}} + (1-\alpha)(L_{T-1} + T_{T-1})$$

$$T_T = \beta(L_T - LT - 1) + (1-\beta)T_{T-1}$$

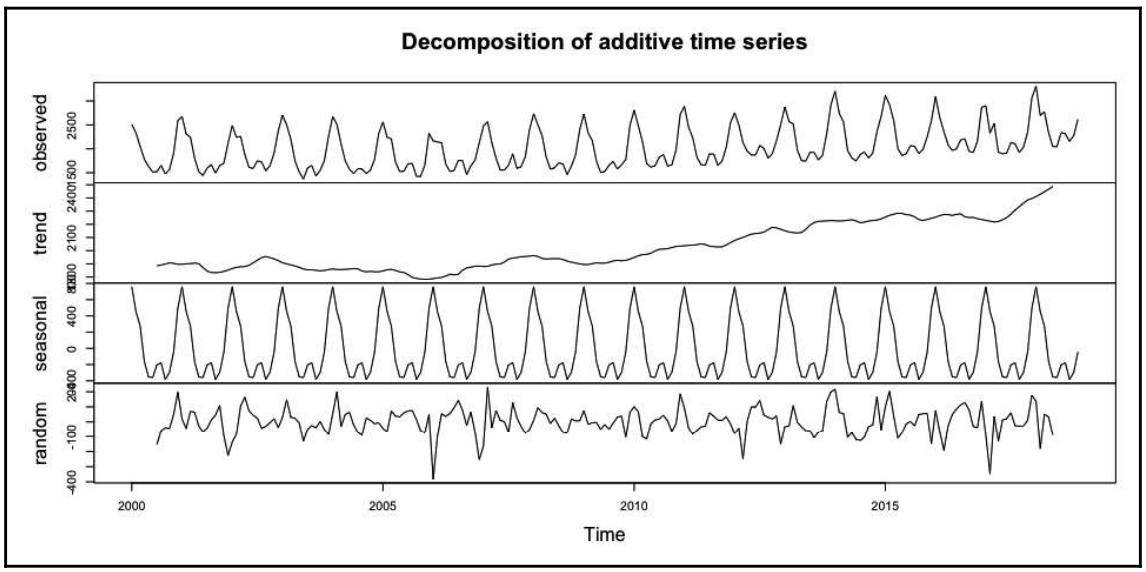
$$S_T = \gamma\frac{Y_T}{L_T} + (1-\gamma)S_{T-m}$$

The most common implementation of the HW model in R is the `HoltWinters` and `hw` functions from the **stats** and **forecast** packages. The main difference between the two functions is that the `hw` function can handle time series with an exponential or damped trend (similar to the Holt model). In the following example, we will use the `HoltWinters` function to forecast the last 12 months of the `USgas` series.

Let's use the decompose function to diagnose the structure of the trend and seasonal components of the series:

```
data(USgas)

decompose(USgas) %>% plot()
```

The output is as follows:



We can observe from the preceding plot that both the trend and seasonal components of the series have an additive structure. Like we did previously, we will create training and testing partitions using the last `12` months of the series to evaluate the performance of the model:

```
USgas_par <- ts_split(USgas, 12)

train <- USgas_par$train
test <- USgas_par$test
```

Next, we will use the `HoltWinters` model to forecast the last 12 months of the series (or the testing set):

```
md_hw <- HoltWinters(train)
```

Let's review the parameters of the trained model:

```
md_hw
```

We get the following output:

```
## Holt-Winters exponential smoothing with trend and additive seasonal
component.
 ##
 ## Call:
 ## HoltWinters(x = train)
 ##
 ## Smoothing parameters:
 ##  alpha: 0.3462899
 ##  beta : 0
 ##  gamma: 0.3766804
 ##
 ## Coefficients:
 ##             [,1]
 ## a   2299.3445761
 ## b     -0.1287005
 ## s1   516.2274996
 ## s2   788.9916937
 ## s3   396.5624290
 ## s4   285.0458175
 ## s5  -200.5519515
 ## s6  -311.8029253
 ## s7  -309.0225553
 ## s8  -133.0658271
 ## s9  -141.1034536
 ## s10 -339.8285611
 ## s11 -270.8429591
 ## s12   23.6032284
```

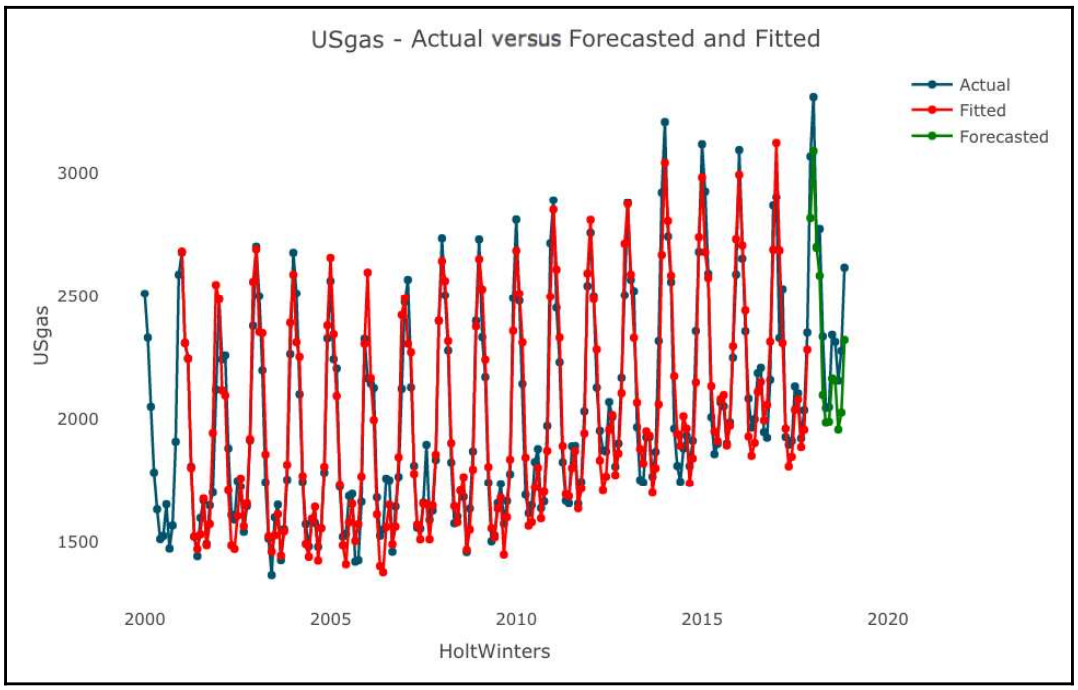
You will notice from the preceding model output that the model is mainly learning from the level and seasonal update (with $\alpha = 0.35$ and $\gamma = 0.37$). On the other hand, there is no learning from the trend initialized value $\beta = 0$. The next step is to forecast the next `12` months (or the values of the testing set) and evaluate the model's performance with the `accuracy` and `test_forecast` functions:

```
fc_hw <- forecast(md_hw, h = 12)

accuracy(fc_hw, test)
##                      ME     RMSE       MAE       MPE     MAPE      MASE
## Training set   6.08587 115.1122  86.92766 0.2617263 4.273029 0.8102333
## Test set     174.39927 194.4311 174.39927 6.9486812 6.948681 1.6255368
##                    ACF1 Theil's U
## Training set 0.2407121        NA
## Test set     0.1170619 0.7198667
```

The accuracy metrics of the model are fairly balanced, with an MAPE of 4.3% in the training set and 7% in the testing set. In the plot of the following model performance, you will notice that most of the forecast errors are related to the seasonal peak and the last observations of the series, which the model was underestimating:

```
test_forecast(actual = USgas,
              forecast.obj = fc_hw,
              test = test)
```

The output is as follows:



Plotting the fitted and forecasted values provides us with a better understanding of the model's performance. As we can see in the preceding plot, the HW model is doing a good job of capturing both the series seasonal patterns. On the other hand, the model is missing the peak of the year during the month of January in most cases.

Alternatively, the model can be trained with a grid search in a similar manner to what we did with the SES model. In this case, there are three parameters to optimize: α, β, and γ. The **TSstudio** package provides a customized grid search function based on the backtesting approach for training a `HoltWinters` function.

A grid search is a generic optimization approach for tuning models with multiple tuning parameters, such as tuning the α, β, and γ parameters of the HW model. This simple algorithm is based on setting (when applicable) a set of values for each parameter of the model and then iterating and testing the model with a different combination of the model's parameter values. Based on the selected error metric, the combination that minimizes the error criteria will be used with the final model. Generally, the main caveat of this approach is that it could be expensive to compute as the number of search combinations increase. In `Chapter 12`, *Forecasting with Machine Learning Models*, we will look at a more robust grid search algorithm with the **h2o** package.

For efficiency reasons, we will start with a shallow search that includes larger increments in the parameters' values. This will help us narrow down the search area and then apply a deeper search on those areas. The shallow search will include backtesting over `6` different periods using a sequence between `0` and `1` with an increment of `0.1`:

```
shallow_grid <- ts_grid(train,
                        model = "HoltWinters",
                        periods = 6,
                        window_space = 6,
                        window_test = 12,
                        hyper_params = list(alpha = seq(0,1,0.1),
                                            beta = seq(0,1,0.1),
                                            gamma = seq(0,1,0.1)),
                        parallel = TRUE,
                        n.cores = 8)
```

The output of the following grid provides any combination of α, β, and γ with the error rate on each testing period and its overall mean. The table sorts the overall mean of the model from the best combination to the worst:

```
shallow_grid$grid_df[1:10,]
```

We get the following output:

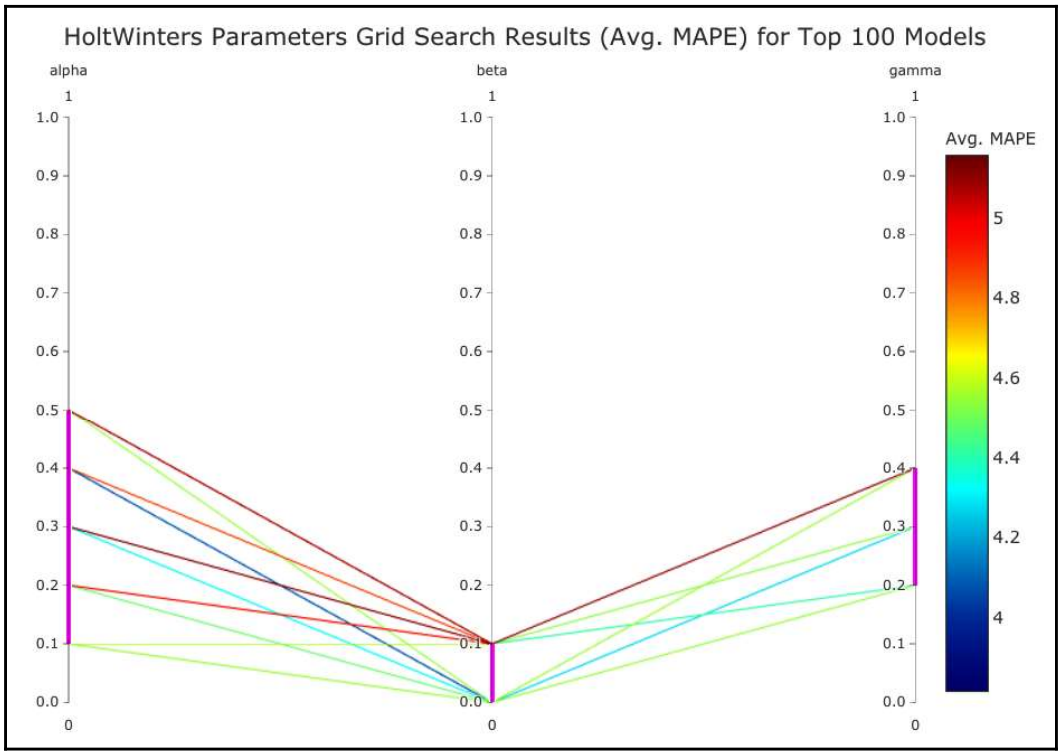
```
##    alpha    beta gamma        1        2        3        4        5
## 1    0.4 0.00001   0.3 4.413769 2.605433 2.257627 3.725283 5.237407
## 2    0.3 0.00001   0.2 4.260377 2.632491 2.377467 3.922444 5.333127
## 3    0.3 0.10000   0.2 5.066178 2.548095 2.453950 3.743206 5.249969
## 4    0.3 0.00001   0.3 4.557990 2.455461 2.351108 4.313726 5.431035
## 5    0.4 0.00001   0.4 4.592116 2.311381 2.480636 4.241562 5.322397
## 6    0.2 0.00001   0.2 4.480924 2.962627 2.408871 4.231048 5.455692
## 7    0.4 0.00001   0.2 4.282419 2.688454 3.023123 3.872748 5.440050
## 8    0.5 0.00001   0.4 5.241874 3.425948 2.410089 3.373926 5.109618
## 9    0.1 0.00001   0.2 4.010023 3.321816 2.802651 4.507483 5.615869
```

```
## 10   0.1 0.10000   0.2 5.777916 2.320443 2.768832 4.249258 5.507453
##            6     mean
## 1   4.662878 3.817066
## 2   4.579267 3.850862
## 3   4.385781 3.907863
## 4   4.435515 3.924139
## 5   4.908380 3.976079
## 6   4.350566 3.981621
## 7   4.958675 4.044245
## 8   4.844823 4.067713
## 9   4.278671 4.089419
## 10  4.145535 4.128239
```

The `plot_grid` provides an intuitive view of the optimal range of values of each parameter by using a paracords plot. By default, the function is highlighting the top 10% models:

```
plot_grid(shallow_grid)
```

The output is as follows:



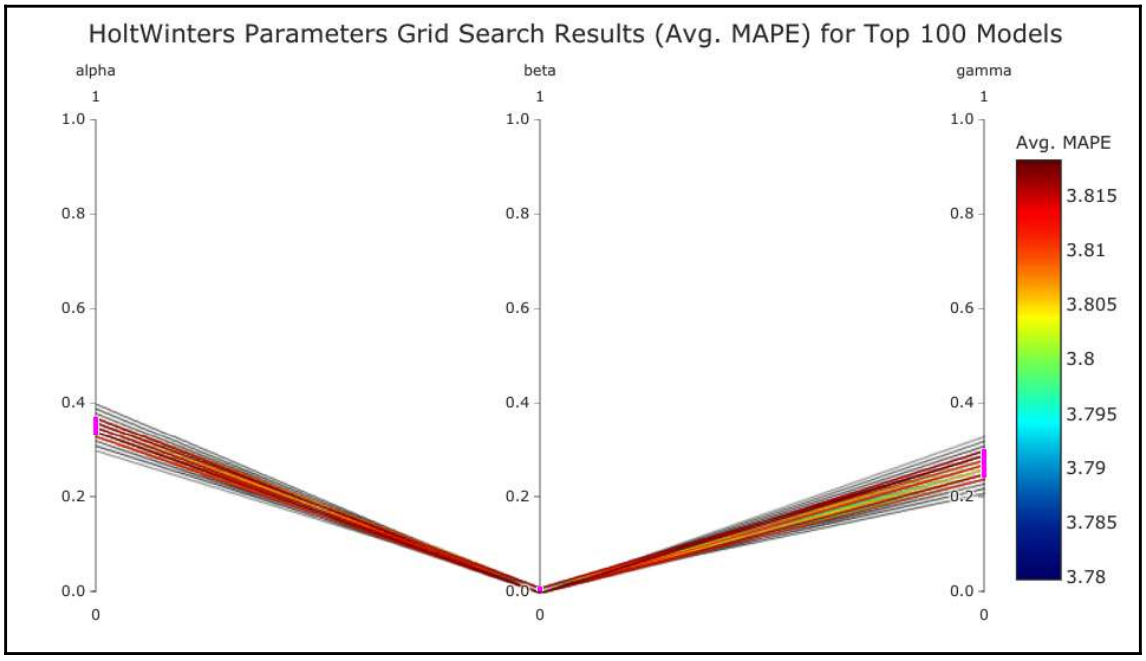
You will notice from the search plot output that the optimal range of α varies between `0.1` and `0.5`, β is between `0` and `0.1`, and γ is between `0.2` and `0.4`. This will help us set the range of the hyperparameter search for the deeper grid search by narrowing down the range of search but using a more granular search:

```
deep_grid <-  ts_grid(train,
                          model = "HoltWinters",
                          periods = 6,
                          window_space = 6,
                          window_test = 12,
                          hyper_params = list(alpha = seq(0.1,0.5,0.01),
                                              beta = seq(0,0.1,0.01),
                                              gamma = seq(0.2,0.4,0.01)),
                          parallel = TRUE,
                          n.cores = 8)
```

As you can see in the following plot, the error range of the top 10% models has dropped with respect to the one of the shallow search:

```
plot_grid(deep_grid)
```

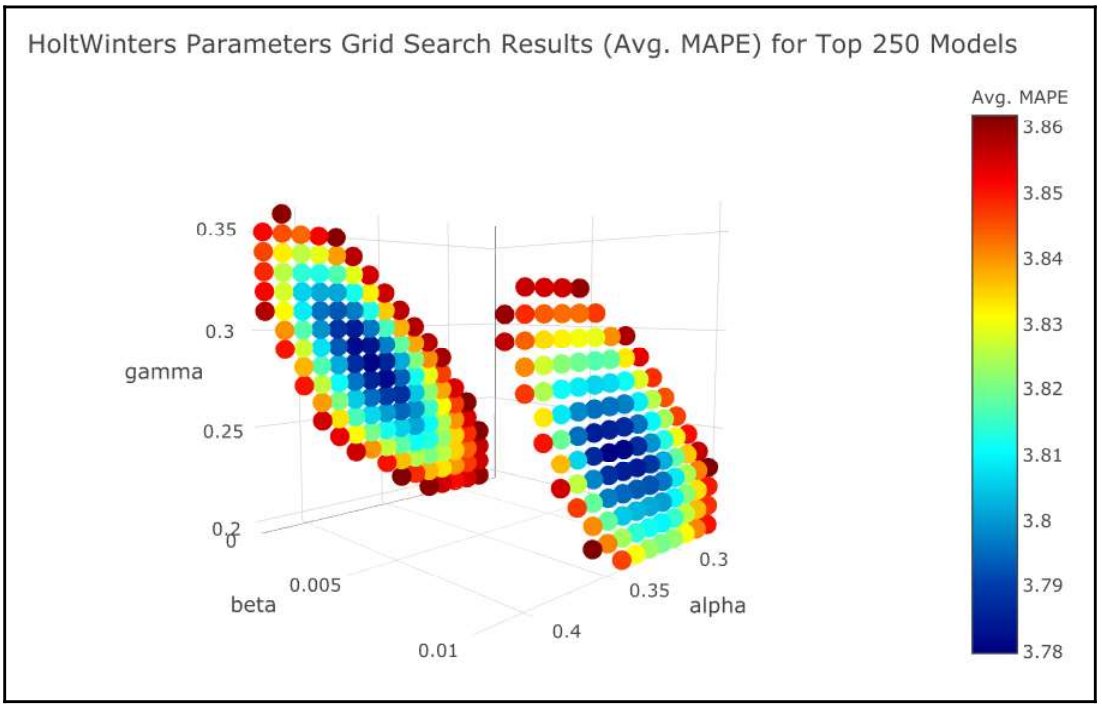
The output is as follows:



This plot can be viewed in a 3D view (when conducting a search for three parameters):

```
plot_grid(deep_grid, type = "3D", top = 250)
```

The output is shown in the following screenshot:



The last step of this process is to pull the values of the optimal smoothing parameters from the grid model based on the search, retrain the HW model, and utilize it to forecast the future values of the series:

```
md_hw_grid <- HoltWinters(train,
                          alpha = deep_grid$alpha,
                          beta = deep_grid$beta,
                          gamma = deep_grid$gamma)

fc_hw_grid <- forecast(md_hw_grid, h = 12)
```

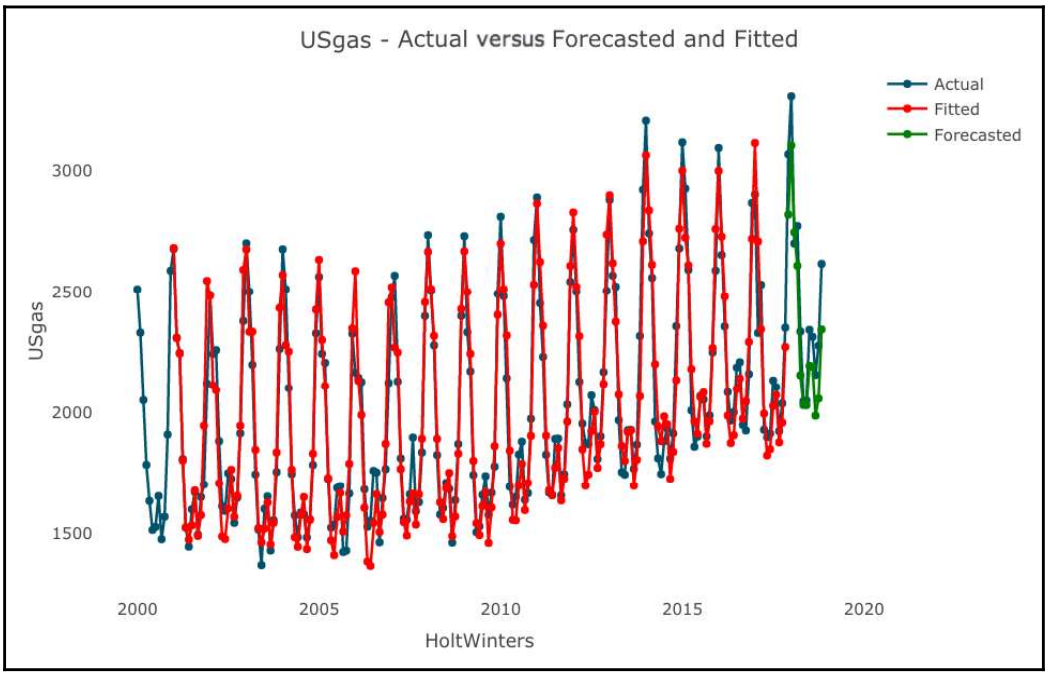
We will use the `accuracy` and `test_forecast` functions in order to review the performance of the HW model that's been optimized by a grid search:

```
accuracy(fc_hw_grid, test)
##                       ME      RMSE       MAE        MPE      MAPE      MASE
## Training set   3.839412 116.7249  89.1783 0.2097637 4.401512 0.831211
## Test set     143.356247 171.8245 151.0337 5.6297541 5.914167 1.407751
##                    ACF1 Theil's U
## Training set 0.2560556        NA
## Test set     0.1230779 0.6220483
```

The plot of the fitted and forecasted values versus the actual values can be seen as follows:

```
test_forecast(actual = USgas,
forecast.obj = fc_hw_grid,
test = test)
```

The output is as follows:



As you can see from the output of the `accuracy` and `test_forecast` functions, optimizing the model with grid search provides, in this case, a lift in the model's performance by reducing the MAPE score from 6.95% to 5.92%.

Summary

In this chapter, we introduced the use of a weighted average of past observations for forecast time series data. We started with a simplistic and naive forecasting approach with the moving average function. Although this function is limited to short-term forecasts and can only handle time series with no seasonal and trend components, it provides context for exponential smoothing functions. The exponential smoothing family of forecasting models is based on the use of different smoothing parameters, that is α, β, and γ, for modeling the main components of time series data—level, trend, and seasonal, respectively. The main advantages of exponential smoothing functions are their simplicity, they're cheap for computing, and their modularity, which allows them to handle different types of time series data, such as linear and exponential trends and seasonal components.

In the next chapter, we will learn about a more advanced forecasting approach with the ARIMA family of forecasting models.

11
Forecasting with ARIMA Models

The **Autoregressive Integrated Moving Average** (**ARIMA**) model is the generic name for a family of forecasting models that are based on the **Autoregressive** (**AR**) and **Moving Average** (**MA**) processes. Among the traditional forecasting models (for example, linear regression, exponential smoothing, and so on), the ARIMA model is considered as the most advanced and robust approach. In this chapter, we will introduce the model components—the AR and MA processes and the differencing component. Furthermore, we will focus on methods and approaches for tuning the model's parameters with the use of differencing, the **autocorrelation function** (**ACF**), and the **partial autocorrelation function** (**PACF**).

In this chapter, we will cover the following topics:

- The stationary state of time series data
- The random walk process
- The AR and MA processes
- The ARMA and ARIMA models
- The seasonal ARIMA model
- Linear regression with the ARIMA errors model

Technical requirement

The following packages will be used in this chapter:

- **forecast**: Version 8.5 and above
- **TSstudio**: Version 0.1.4 and above
- **plotly**: Version 4.8 and above
- **dplyr**: Version 0.8.1 and above
- **lubridate**: Version 1.7.4 and above
- **stats**: Version 3.6.0 and above
- **datasets**: Version 3.6.0 and above
- **base**: Version 3.6.0 and above

You can access the codes for this chapter from the following link:

`https://github.com/PacktPublishing/Hands-On-Time-Series-Analysis-with-R/tree/master/Chapter11`

The stationary process

One of the main assumptions of the ARIMA family of models is that the input series follows the stationary process structure. This assumption is based on the Wold representation theorem, which states that any stationary process can be represented as a linear combination of white noise. Therefore, before we dive into the ARIMA model components, let's pause and talk about the stationary process. The stationary process, in the context of time series data, describes a stochastic state of the series. Time series data is stationary if the following conditions are taking place:

- The mean and variance of the series do not change over time
- The correlation structure of the series, along with its lags, remains the same over time

In the following examples, we will utilize the `arima.sim` function from the **stats** package to simulate a stationary and non-stationary time series data and plot it with the `ts_plot` function from the **TSstudio** package. The `arima.sim` function allows us to simulate time series data based on the ARIMA model's components and main characteristics:

- **An Autoregressive (AR) process**: Establish a relationship between the series and its past *p* lags with the use of a regression model (between the series and its *p* lags)
- **A Moving Average (MA) process**: Similar to the AR process, the MA process establishes the relationship with the error term at time *t* and the past error terms, with the use of regression between the two components (error at time *t* and the past error terms)
- **Integrated (I) process**: The process of differencing the series with its *d* lags to transform the series into a stationary state

Here, the model argument of the function defines *p*, *q*, and *d*, as well as the order of the AR, MA, and I processes of the model. For now, don't worry if you are not familiar with this function—we will discuss it in detail later in this chapter.

The `arima.sim` function has a random component. Therefore, in order to be able to reproduce the examples throughout this chapter, we will use the `set.seed` function. The `set.seed` function allows you to create random numbers in a reproducible manner in R by setting the random generating seed value.

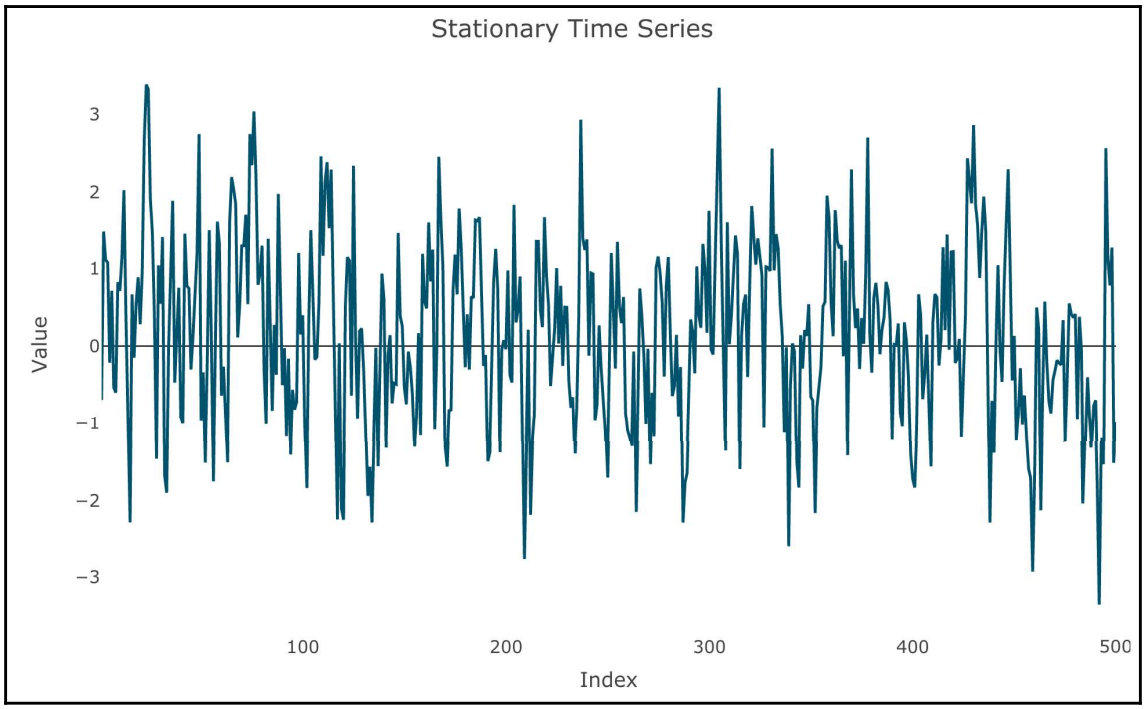
For instance, in the following example, we will simulate an AR process with one lag (that is, *p = 1*) and 500 observations with the `arima.sim` function. Before running the simulation, we will set the `seed` value to `12345`:

```
set.seed(12345)

stationary_ts <- arima.sim(model = list(order = c(1,0,0),
                                        ar = 0.5 ),
                           n = 500)
```

Now, let's plot the simulate time series with the `ts_plot` function:

```
library(TSstudio)

ts_plot(stationary_ts,
        title = "Stationary Time Series",
        Ytitle = "Value",
        Xtitle = "Index")
```

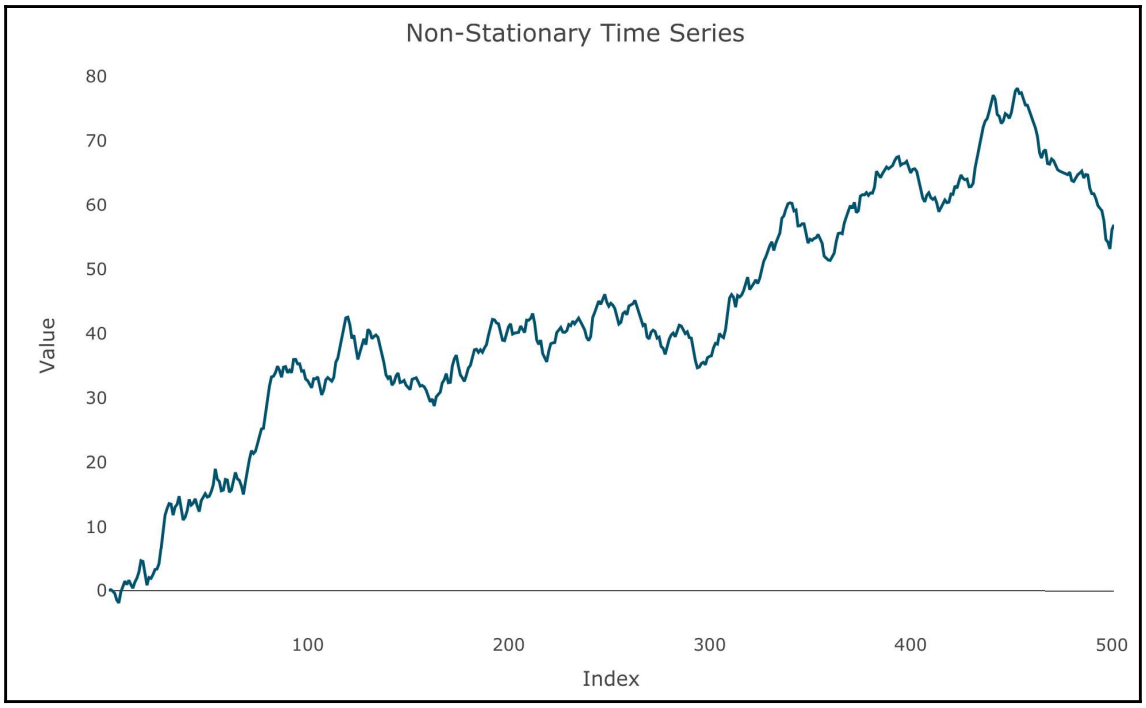
We get the following output:



In this case, you can see that, overall, the mean of the series, over time, remains around the zero line. In addition, the series' variance does not change over time. Let's utilize the `arima.sim` function to create an example for non-stationary series:

```
set.seed(12345)

non_stationary_ts <- arima.sim(model = list(order = c(1,1,0),ar = 0.3),n =
500)

ts_plot(non_stationary_ts,
        title = "Non-Stationary Time Series",
        Ytitle = "Value",
        Xtitle = "Index")
```

We get the following output:



On the other hand, the second example violates the stationary condition as it is trending over time, which means it is changing over time.

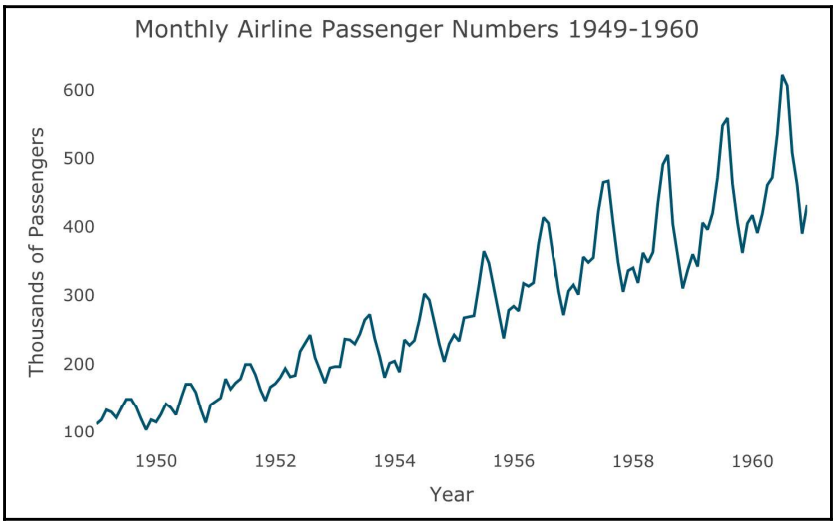
We would consider a time series data as non-stationary whenever those conditions do not hold. Common examples of a series with a non-stationary structure are as follows:

- **A series with a dominant trend**: The series' mean changes over time as a function of the change in the series trend, and therefore the series is non-stationary
- **A series with a multiplicative seasonal component**: In this case, the variance of the series is a function of the seasonal oscillation over time, which either increases or decreases over time

The classic `AirPassenger` series (the monthly airline passenger numbers between 1949 and 1960) from the **datasets** package is a good example of a series that violates the two conditions of the stationary process. Since the series has both a strong linear trend and a multiplicative seasonal component, the mean and variance are both changing over time:

```
data(AirPassengers)

ts_plot(AirPassengers,
        title = "Monthly Airline Passenger Numbers 1949-1960",
        Ytitle = "Thousands of Passengers",
        Xtitle = "Year")
```

We get the following output:



Transforming a non-stationary series into a stationary series

In most cases, unless you are very lucky, your raw data would probably come with a trend or other form of oscillation that violates the assumptions of the stationary process. Therefore, to handle this, you will have to apply some transformation steps in order to bring the series into a stationary state. Common transformations methods are differencing the series (or de-trending) and *log* transformation (or both). Let's review the applications of these methods.

Differencing time series

The most common approach to transforming a non-stationary time series data into a stationary state is by differencing the series with its lags. The main effect of differencing a series is the removal of the series trend (or detrending the series), which help to stabilize the mean of the series. We measured the degree or order of the series differencing by the number of times we difference the series with its lags. For example, the following equation defines the first order difference:

$$Y'_t = Y_t - Y_{t-1}$$

Here, Y'_t represents the first order difference of the series, and Y_t and Y_{t-1} represent the series itself and its first lag. In some cases, the use of the first order difference is not sufficient to bring the series to a stationary state, and you may want to apply second order differencing:

$$Y^*_t = Y'_t - Y'_{t-1} = (Y_t - Y_{t-1}) - (Y_{t-1} - Y_{t-2}) = Y_t - 2Y_{t-1} + Y_{t-2}$$

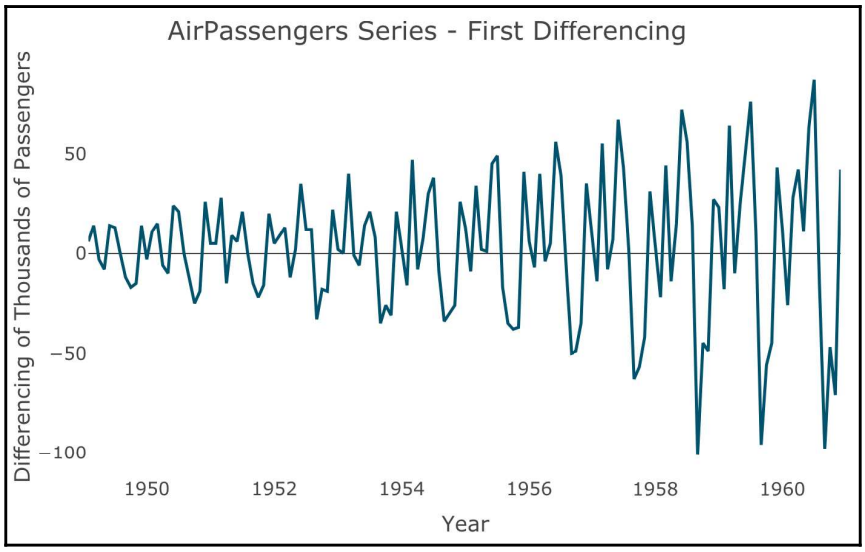
Another form of differencing is seasonal differencing, which is based on differencing the series with the seasonal lag:

$$Y'_t = Y_t - Y_{t-f}$$

Here, *f* represents the frequency of the series and Y_{t-f} represents the seasonal lag of the series. It is common to use seasonal differencing when a series has a seasonal component. The `diff` function from the **base** package differences the input series with a specific lag by setting the `lag` argument of the function to the relevant lag. Let's go back to the `AirPassenger` series and see how the first order and seasonal differencing affect the structure of the series. We will start with the first order difference:

```
ts_plot(diff(AirPassengers, lag = 1),
        title = "AirPassengers Series - First Differencing",
        Xtitle = "Year",
        Ytitle = "Differencing of Thousands of Passengers")
```

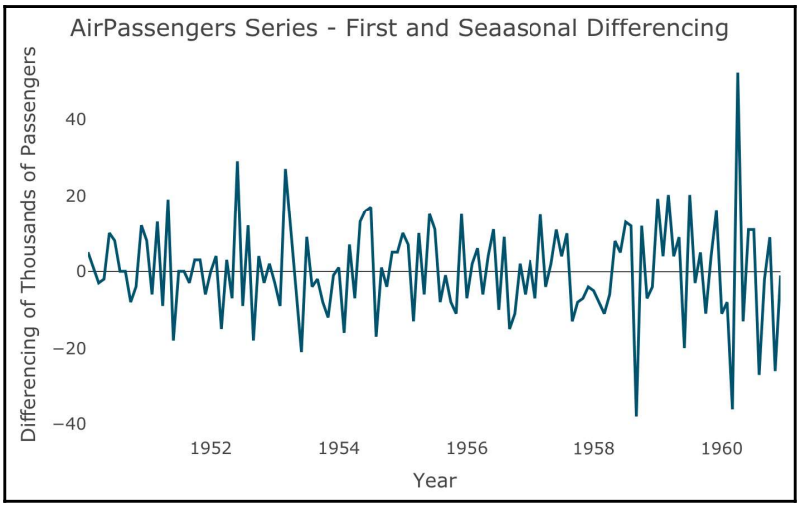
We get the following output:



You can see that the first difference of the `AirPassenger` series removed the series trend and that the mean of the series is, overall, constant over time. On the other hand, there is clear evidence that the variation of the series is increasing over time, and therefore the series is not stationary yet. In addition to the first order difference, taking the seasonal difference of the series could solve this issue. Let's add the seasonal difference to the first order difference and plot it again:

```
ts_plot(diff(diff(AirPassengers, lag = 1), 12),
        title = "AirPassengers Series - First and Seasonal Differencing",
        Xtitle = "Year",
        Ytitle = "Differencing of Thousands of Passengers")
```

We get the following output:



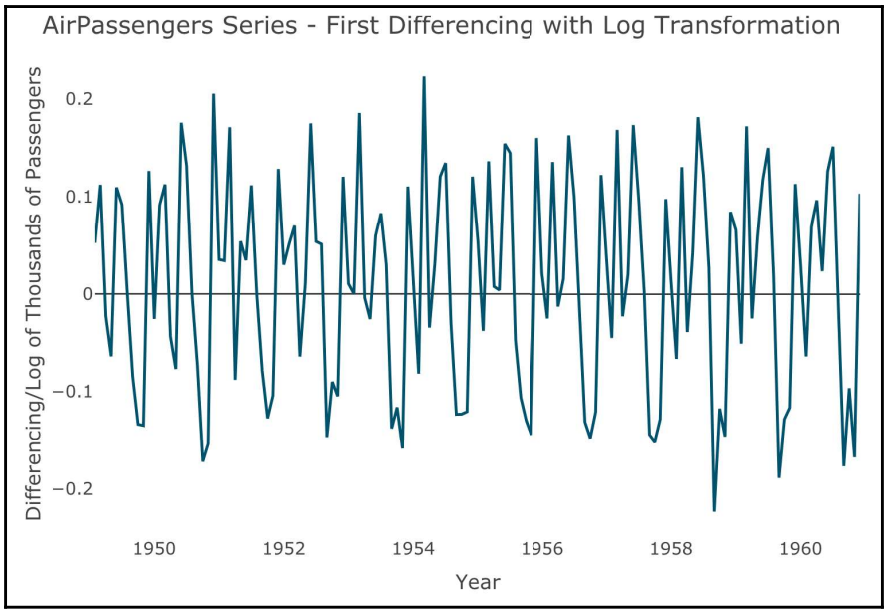
The seasonal difference did a good job of stabilizing the series variation, as the series now seems to be stationary.

Log transformation

In `Chapter 5`, *Decomposition of Time Series Data*, we saw the applications of the *log* transformation for handling multiplicative time series data. Likewise, we can utilize this approach to stabilize a multiplicative seasonal oscillation, if it exists. This approach is not a replacement for differencing, but an addition. For instance, in the example of the `AirPassenger` in the preceding section, we saw that the first differencing is doing a great job in stabilizing the mean of the series, but is not sufficient enough to stabilize the variance of the series. Therefore, we can apply a *log* transformation to transform the seasonal structure from multipliable to additive and then apply the first-order difference to stationarize the series:

```
ts_plot(diff(log(AirPassengers), lag = 1),
        title = "AirPassengers Series - First Differencing with Log
Transformation",
        Xtitle = "Year",
        Ytitle = "Differencing/Log of Thousands of Passengers")
```

We get the following output:



The *log* transformation with the first-order differencing is doing a better job of transforming the series into a stationary state with respect to the double differencing (first-order with seasonal differencing) approach we used prior.

The random walk process

The random walk, in the context of time series, describes a stochastic process of an object over time, where the main characteristics of this process are as follows:

- The starting point of this process at time *0* - Y_0 is known
- The movement (or the walk) of the series with random walk process from time *t-1* to time *t* are defined with the following equation:

$$Y_t = Y_{t-1} + \epsilon_t$$

Here, Y_{t-1} and Y_t represent the value of the series at time *t-1* and *t*, respectively, and ϵ_t represents a random number (or a white noise) with a mean of 0 and a variance of σ_ϵ^2. While the random walk process is not stationary, the first difference of a random walk represents a stationary process like so:

$$Y_t' = Y_t - Y_{t-1} = \epsilon_t$$

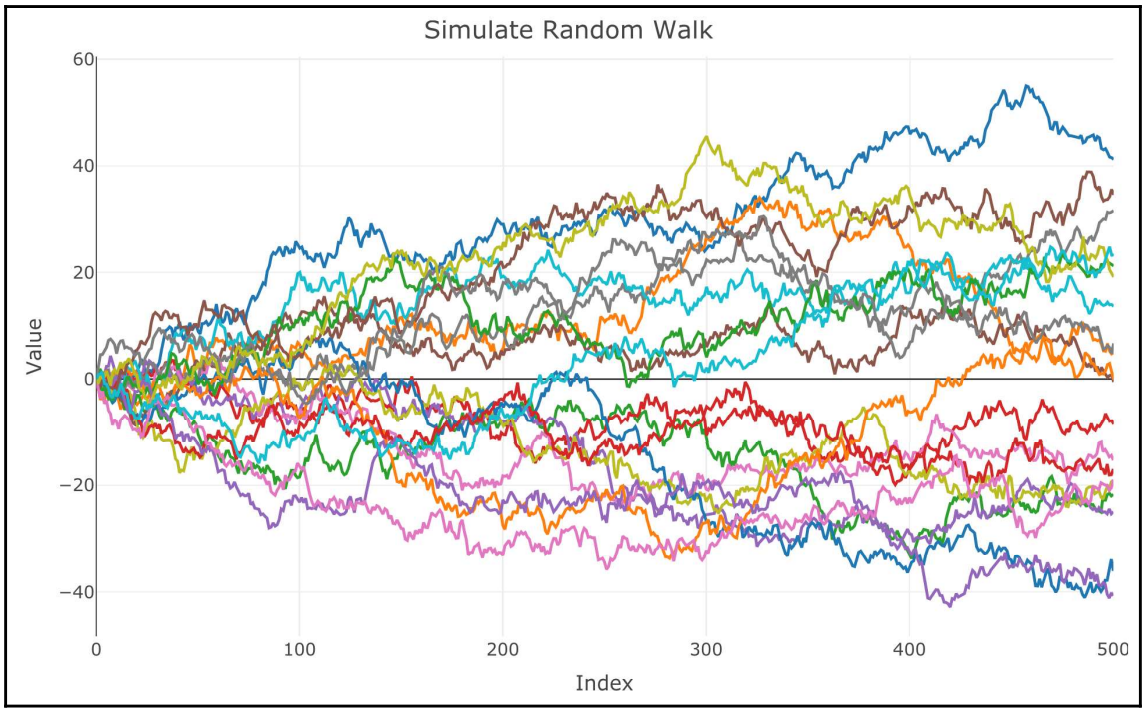
As we mentioned previously, ϵ_t has a constant mean and variance, and therefore Y_t' is a stationary process. A random walk is commonly used to simulate possible future paths of a series. For instance, we can simulate a random walk with the `arima.sim` function by setting the `d` parameter of the `order` argument to `1`. This is equivalent to a non-stationary series with a first difference structure. The following code demonstrates the simulation of `20` different random walk paths of `500` steps, all starting at point `0` at time `0`. We will create two plots: one for the random walk paths and another for their first-order difference. We will use the **plotly** package to do this:

```
library(plotly)

set.seed(12345)

p1 <- plot_ly()
p2 <- plot_ly()

for(i in 1:20){
   rm <- NULL
   rw <- arima.sim(model = list(order = c(0, 1, 0)), n = 500)
   p1 <- p1 %>% add_lines(x = time(rw), y = as.numeric(rw))
   p2 <- p2 %>% add_lines(x = time(diff(rw)), y = as.numeric(diff(rw)))
 }
```

Here, `p1` represents the plot of the random walk simulation:

```
p1 %>% layout(title = "Simulate Random Walk",
              yaxis = list(title = "Value"),
              xaxis = list(title = "Index")) %>%
   hide_legend()
```

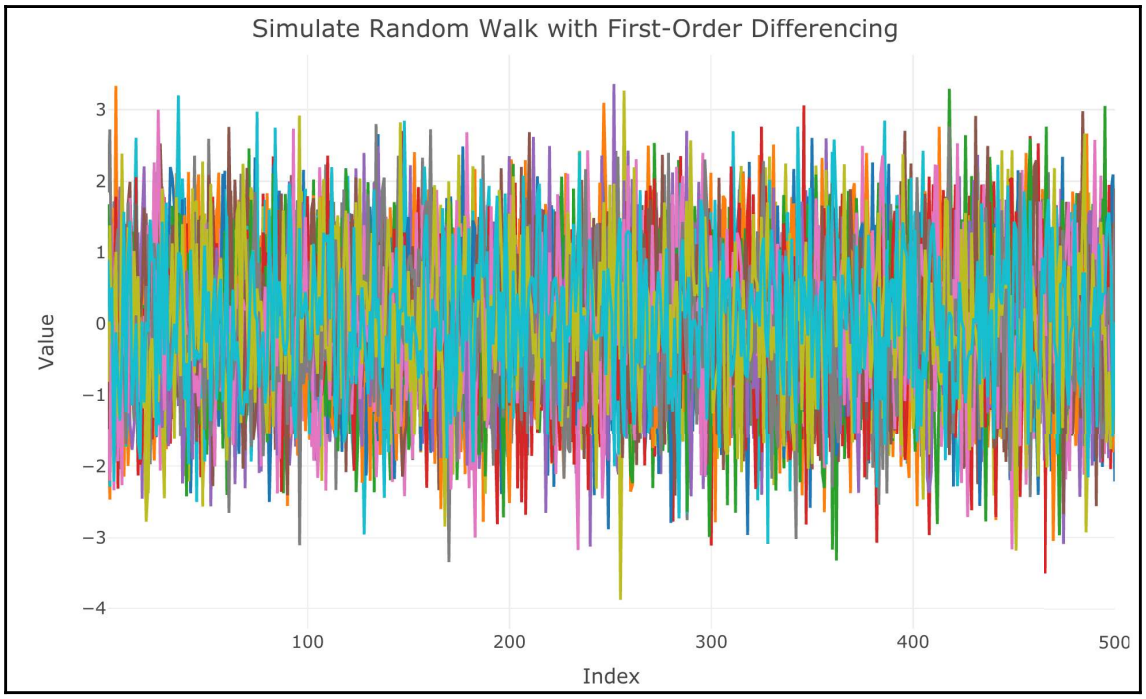
We get the following output:



Here, `p2` represents the corresponding plot of the first-order differencing of the random walk simulation:

```
p2 %>% layout(title = "Simulate Random Walk with First-Order Differencing",
              yaxis = list(title = "Value"),
              xaxis = list(title = "Index")) %>%
   hide_legend()
```

We get the following output:



The AR process

The AR process defines the current value of the series, Y_t, as a linear combination of the previous p lags of the series, and can be formalized with the following equation:

$$AR(p) : Y_t = c + \sum_{i=1}^{p} \phi_i Y_{t-i} + \epsilon_t$$

Following are the terms used in the preceding equation:

- *AR(p)* is the notation for an AR process with p-order
- c represents a constant (or drift)
- p defines the number of lags to regress against Y_t

- ϕ_i is the coefficient of the *i* lag of the series (here, ϕ_1 must be between *-1* and *1*, otherwise, the series would be trending up or down and therefore cannot be stationary over time)
- Y_{t-i} is the *i* lag of the series
- ϵ_t represents the error term, which is white noise

An AR process can be used on time series data if, and only if, the series is stationary. Therefore, before applying an AR process on a series, you will have to verify that the series is stationary. Otherwise, you will have to apply some transformation method (such as differencing, *log* transformation, and so on) to transform the series into a stationary state. Later in this chapter, we will introduce the ARIMA model, which can handle non-stationary time series data.

For instance, *AR(1)*, the first-order AR process, is defined by the following equation:

$$AR(1) : Y_t = c + \phi_1 Y_{t-1} + \epsilon_t$$

Note that in the case of the *AR(1)* process, if the value of $\phi_1 = 1$ or $\phi_1 = -1$, the series is a random walk (assuming ϵ_t is white noise). Whenever $\phi_1 > 1$ or $\phi_1 < -1$ the series is trending and therefore it cannot be stationary.

Similarly, *AR(2)*, the second-order AR process defines by the next equation:

$$AR(2) : Y_t = c + \phi_1 Y_{t-1} + \phi_2 Y_{t-2} + \epsilon_t, \; where \; -1 < \phi_1 + \phi_2 < 1$$

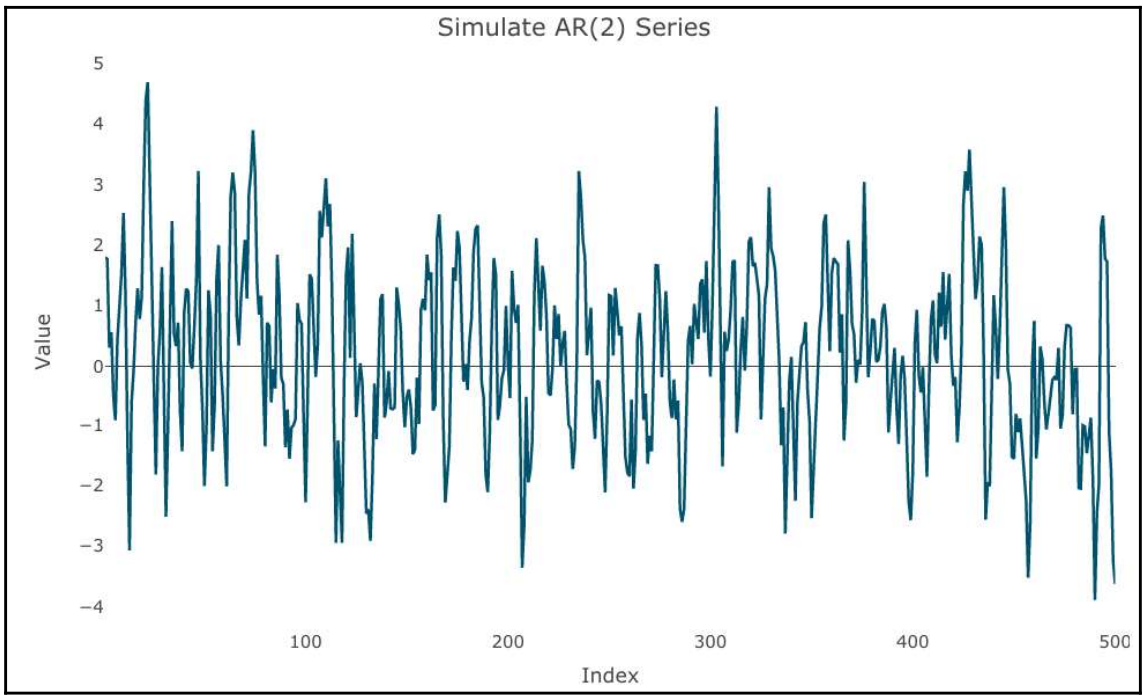
In the following example, we will utilize the `arima.sim` function again to simulate an *AR(2)* process structure time series data with `500` observations, and then use it to fit an AR model. We will use the model argument to set the AR order to `2` and set the lags coefficients $\phi_1 = 0.9$ and $\phi_2 = -0.3$:

```
set.seed(12345)

ar2 <- arima.sim(model = list(order = c(2,0,0),
                              ar = c(0.9, -0.3)),
                 n = 500)
```

Let's review the simulate time series:

```
ts_plot(ar2,
         title = "Simulate AR(2) Series",
         Ytitle = "Value",
         Xtitle = "Index")
```

We get the following output:



The `ar` function from the **stats** package allows us to fit an AR model on time series data and than forecast its future values. This function identifies the AR order automatically based on the **Akaike Information Criterion** (**AIC**). The `method` argument allows you to define the coefficients estimation method, such as the **ordinary least squares** (**OLS**) (which we saw in `Chapter 9`, *Forecasting with Linear Regression*), **maximum likelihood estimation** (**MLE**), and Yule-Walker (default). Let's apply the `ar` function to identify the AR order and estimate its coefficients accordingly:

```
md_ar <- ar(ar2)
```

Let's review the fitted model details:

```
md_ar
```

We get the following output:

```
##
## Call:
## ar(x = ar2)
##
## Coefficients:
## 1 2
## 0.8851 -0.2900
##
## Order selected 2 sigma^2 estimated as 1.049
```

As you can see from the preceding model summary, the `ar` function was able to identify that the input series is a second order AR process, and provided a fairly close estimate for the value of the actual coefficients, $\hat{\phi_1} = 0.88, \hat{\phi_2} = -0.29$ (as opposed to the actual coefficients' values, $\phi_1 = 0.9, \phi_2 = -0.3$).

Later on in this chapter, we will return to the AR model and discuss methods and approaches for identifying whether a series has an AR structure and its order.

Identifying the AR process and its characteristics

In the preceding example, we simulated an *AR(2)* series, and it was clear that we need to apply an AR model on the data. However, when working with real-time series data, you will have to identify the structure of the series before modeling it. In the world of the non-seasonal ARIMA family of models, a series could have one of the following structures:

- AR
- MA
- Random walk
- A combination of the preceding three (for example, AR and MA processes)

In this section, we will introduce the method for identifying the first case, that is, a series with only an AR structure. In the following sections, we will generalize this method to the rest of the cases (for example, MA, AR, MA, and so on).

Identifying the series structure includes the following two steps:

- Categorizing the type of process (for example, AR, MA, and so on)
- Once we have classified the process type, we need to identify the order of the process (for example, *AR(1)*, *AR(2)*, and so on)

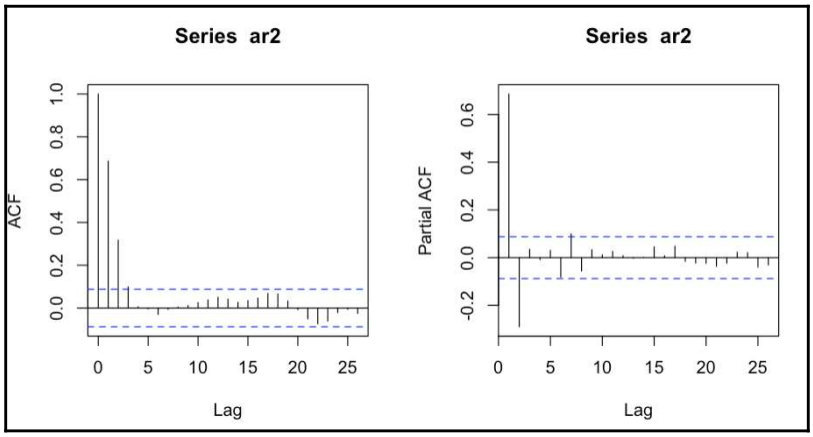
Utilizing the **autocorrelation function** (**ACF**) and **partial autocorrelation function** (**PACF**), which we introduced in `Chapter 7`, *Correlation Analysis*, allows us to classify the process type and identify its order. If the ACF output tails off and the PACF output cuts off at lag *p*, this indicates that the series is an *AR(p)* process. Let's calculate and plot the ACF and PACF for the simulated *AR(2)* series we created previously with the ACF and PACF functions. First, we will use the `par` function to plot the two plots side by side by setting the `mfrow` argument to `c(1,2)` (one row, two columns):

```
par(mfrow=c(1,2))
```

Now, we will generate the plots with the `acf` and `pacf` functions:

```
acf(ar2)
pacf(ar2)
```

We get the following output:



In the case of the `ar2` series, you can see that the ACF plot is tailing off and that the PACF plot is cut off at the second lag. Therefore, we can conclude that the series has a second order AR process.

The moving average process

In some cases, the forecasting model is unable to capture all the series patterns, and therefore some information is left over in model residuals (or forecasting error) ϵ_t. The goal of the moving average process is to capture patterns in the residuals, if they exist, by modeling the relationship between Y_t, the error term, ϵ_t, and the past q error terms of the models (for example, $\epsilon_{t-1}, \epsilon_{t-2}, \cdots, \epsilon_{t-q}$). The structure of the MA process is fairly similar to the ones of the AR. The following equation defines an MA process with a q order:

$$MA(q) : Y_t = \mu + \epsilon_t + \sum_{i=1}^{q} \theta_i \epsilon_{t-i}$$

The following terms are used in the preceding equation:

- *MA(q)* is the notation for an MA process with q-order
- μ represents the mean of the series
- $\epsilon_{t-q}, \cdots, \epsilon_t$ are white noise error terms
- θ_i is the corresponding coefficient of ϵ_{t-i}
- q defines the number of past error terms to be used in the equation

Like the AR process, the MA equation holds only if the series is a stationary process; otherwise, a transformation must be used on the series before applying the MA process.

For example, the second-order MA process is defined by the following equation:

$$MA(2) : Y_t = \mu + \epsilon_t + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2}$$

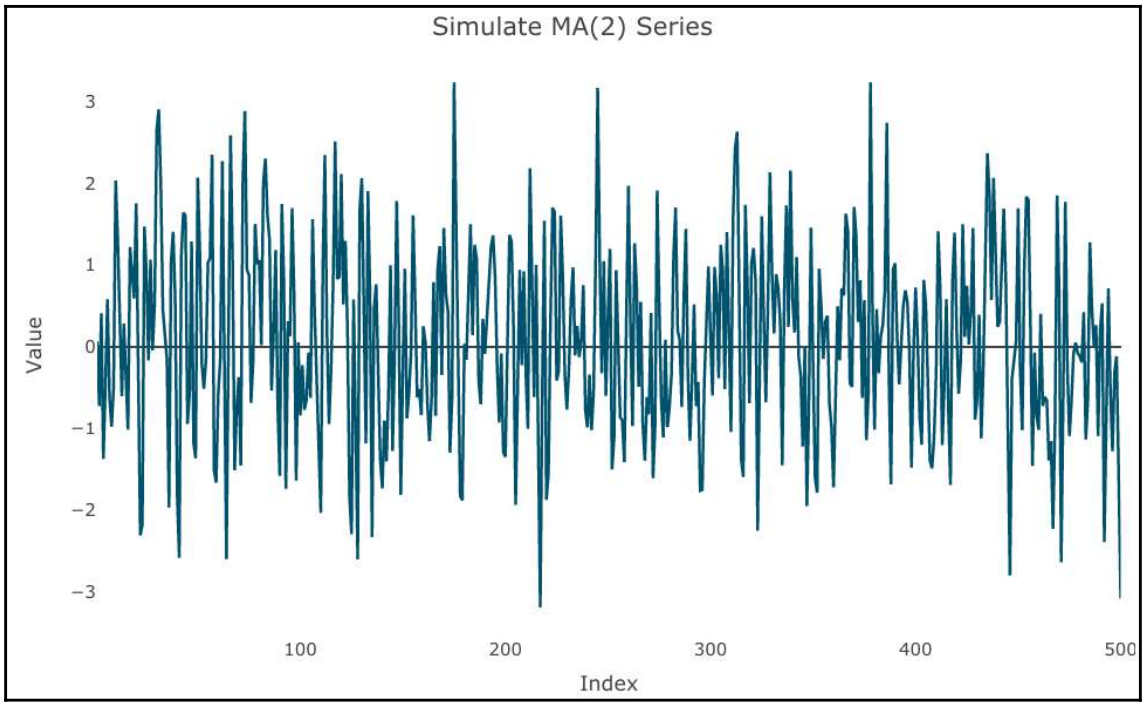
In a similar manner, since we simulated the *AR(2)* series in the previous section, we will utilize the `arima.sim` function to simulate a series with an *MA(2)* structure. In this case, we will set the q parameter in the `order` argument to `2` and set the MA coefficients to $\theta_1 = 0.5$ and $\theta_2 = -0.3$:

```
set.seed(12345)

ma2 <- arima.sim(model = list(order = c(0, 0, 2),
 ma = c(0.5, -0.3)),
 n = 500)
```

We will use the `ts_plot` function to plot the simulated series:

```
ts_plot(ma2,
        title = "Simulate MA(2) Series",
        Ytitle = "Value",
        Xtitle = "Index")
```

We get the following output:



Modeling the MA process can be done with the `arima` function from the **stats** package. This function, when setting the order of the AR and the differencing components of the model to **0** with the order argument (that is, $p = 0$ and $d = 0$), is modeling only on the MA component. For instance, let's apply a second-order MA model with the `arima` function on the simulated *MA(2)* series:

```
md_ma <- arima(ma2, order = c(0,0,2), method = "ML")
```

Similar to the `ar` function, you can select the coefficients estimation approach. In this case, there are three methods: **maximum likelihood** (**ML**), minimize **conditional sum-of-squares** (**CSS**), and the combination of the two, which is known as **CSS-ML**.

The output of the `arima` function is more detailed than the ones of the `ar` function, as it also provides the level of significance of each coefficient (the `s.e.`):

```
md_ma
```

We get the following output:

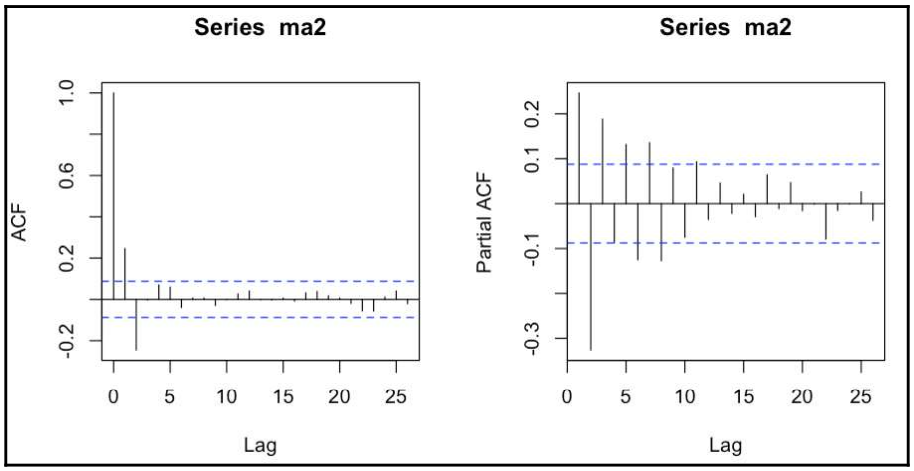
```
##
## Call:
## arima(x = ma2, order = c(0, 0, 2), method = "ML")
##
## Coefficients:
## ma1 ma2 intercept
## 0.530 -0.3454 0.0875
## s.e. 0.041 0.0406 0.0525
##
## sigma^2 estimated as 0.9829: log likelihood = -705.81, aic = 1419.62
```

Identifying the MA process and its characteristics

Similar to the AR process, we can identify an MA process and its order with the ACF and PACF functions. If the ACF is cut off at lag q and the PACF function tails off, we can conclude that the process is an *MA(q)*. Let's repeat the process we applied on the `ar2` series with the `ma2` series:

```
par(mfrow=c(1,2))
acf(ma2)
pacf(ma2)
```

We get the following output:



In the case of the `ma2` series, the ACF plot is cut off on the second lag (note that lag **0** is the correlation of the series with itself, and therefore it is equal to 1 and we can ignore it), and so the PACF tails off. Therefore, we can conclude that the `ma2` series is an MA(2) process.

The ARMA model

Up until now, we have seen how the applications of AR and MA are processed separately. However, in some cases, combining the two allows us to handle more complex time series data. The ARMA model is a combination of the AR(p) and *MA(q)* processes and can be written as follows:

$$ARMA(p,q) : Y_t = c + \sum_{i=1}^{p} \phi_i Y_{t-i} + \sum_{i=1}^{q} \theta_i \epsilon_{t-i} + \epsilon_t$$

The following terms are used in the preceding equation:

- *ARMA(p,q)* defines an ARMA process with a *p*-order AR process and *q*-order moving average process
- Y_t represents the series itself
- *c* represents a constant (or drift)
- *p* defines the number of lags to regress against Y_t
- ϕ_i is the coefficient of the *i* lag of the series
- Y_{t-1} is the *i* lag of the series
- *q* defines the number of past error terms to be used in the equation
- θ_i is the corresponding coefficient of ϵ_{t-i}
- $\epsilon_{t-q}, \ldots, \epsilon_t$ are white noise error terms
- ϵ_t represents the error term, which is white noise

For instance, an *ARMA(3,2)* model is defined by the following equation:

$$Y_t = c + \phi_1 Y_{t-1} + \phi_2 Y_{t-2} + \phi_3 Y_{t-3} + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \epsilon_t$$

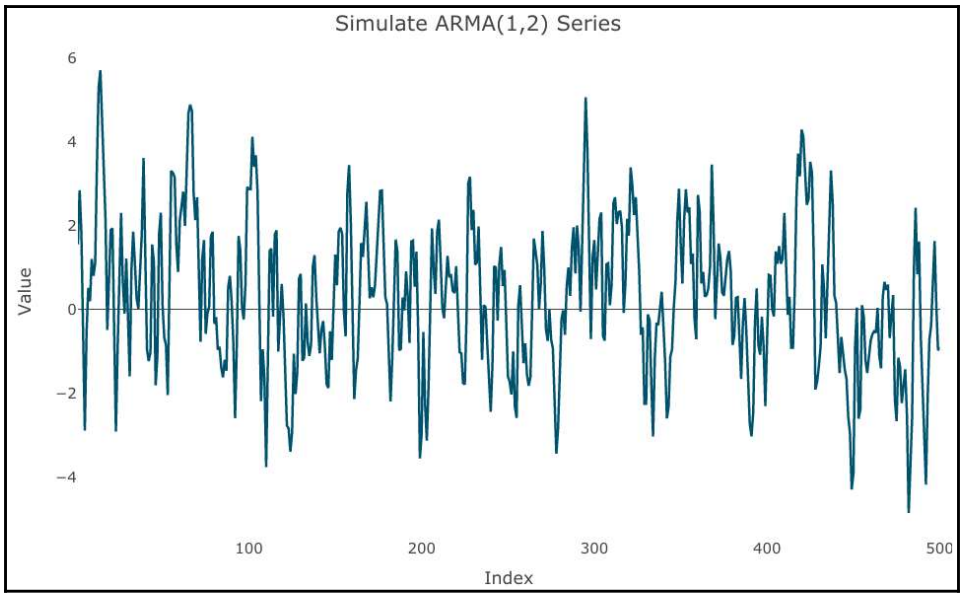
Let's simulate a time series data with an *ARMA(1,2)* structure with the `arima.sim` function and review the characteristics of the ARMA model. We will set the *p* and *q* parameters of the order argument to *1* and *2*, respectively, and set the AR coefficient as $\phi_1 = 0.7$, and the MA coefficients as $\theta_1 = 0.5$ and $\theta_2 = -0.3$:

```
set.seed(12345)

arma <- arima.sim(model = list(order(1,0,2),
                               ar = c(0.7),
                               ma = c(0.5,-0.3)),
                  n = 500)
```

Let's plot and review the series structure with the `ts_plot` function:

```
ts_plot(arma,
        title = "Simulate ARMA(1,2) Series",
        Ytitle = "Value",
        Xtitle = "Index")
```

We get the following output:



Fitting an ARMA model is straightforward with the `arima` function. In this case, we have to set the p and q parameters on the order argument:

```
arma_md <- arima(arma, order = c(1,0,2))
```

You can observe from the following output of the fitted model that the values of the model coefficients are fairly close to the one we simulated:

```
arma_md
```

We get the following output:

```
##
## Call:
## arima(x = arma, order = c(1, 0, 2))
##
## Coefficients:
##          ar1     ma1      ma2  intercept
##       0.7439  0.4785  -0.3954     0.2853
## s.e.  0.0657  0.0878   0.0849     0.1891
##
## sigma^2 estimated as 1.01:  log likelihood = -713.18,  aic = 1436.36
```

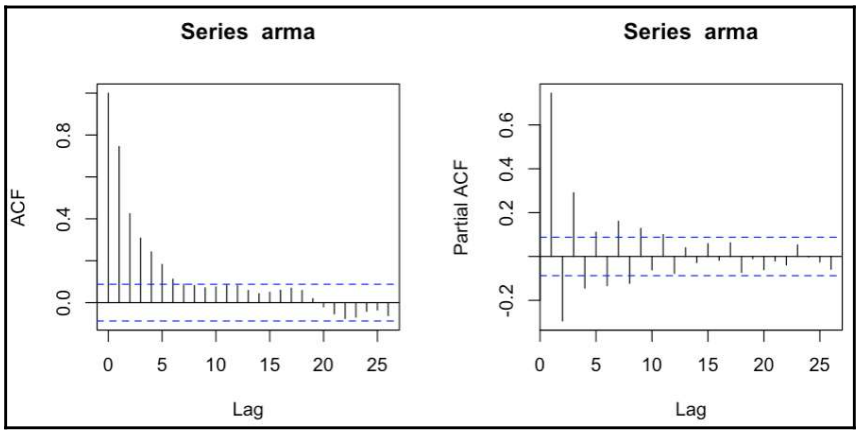
Here, as the coefficient's name implies, `ar1`, `ma1`, and `ma2` represent the estimation of the ϕ_1, θ_1, and θ_2 coefficients, respectively. You can note that the intercept parameter is not statistically significant, which should make sense as we didn't add an intercept to the simulated data.

Identifying an ARMA process

Identifying the ARMA process follows the same approach that we used previously with the AR and MA processes. An ARMA process exists in time series data if both the ACF and PACF plots tail off, as we can see in the following example:

```
par(mfrow=c(1,2))
acf(arma)
pacf(arma)
```

We get the following output:



On the other hand, unlike the AR and MA processes, we cannot conclude the order of the ARMA process. There are several approaches for tuning the ARMA p and q parameters:

- **Manual tuning**: By starting with a combination of p and q and using some error criteria for identifying the model parameters.
- **Grid search**: By trying different combinations of the p and q parameters based on the grid matrix. Likewise, manual tuning and the selection of a specific combination of p and q parameters should be based on error criterion.
- **Algorithm-based search**: By using a function or algorithm for tuning the model parameters.

Later on in this chapter, we will introduce the `auto.arima` function from the **forecast** package, which is one of the most popular algorithms for the automation of the ARIMA family of models, including the ARMA model.

Manual tuning of the ARMA model

Manually tuning the ARMA model is mainly based on experimentation, intuition, common sense, and experience. The tuning process is based on the following steps:

1. Set some initial values for p and q. Typically, it is recommended to start with the minimum values of p and q (for example, $p = 1$ and $q = 1$).
2. Evaluate the fit of the model based on some error criterion. The most common error criteria are the **Akaike Information Criterion** (**AIC**) or **Bayesian information criterion** (**BIC**).
3. Adjust the values of either p and q.
4. Evaluate the change in the error metric.
5. Repeat the last two steps until you cannot achieve additional improvements of the error metric.

The main reasons for starting the tuning with minimum values of p and q is related to the following reasons:

- **Cost**: Since the order of the model is higher, the cost of the model is also higher and the degree of freedom of the model is reduced
- **Complexity**: Increases as the order of the model increases, which may result in overfitting

AIC and BIC are the most appropriate to use in this case since these two methods penalize models with higher order. This can be seen in the following formulas:

$$AIC = 2k - 2ln(\hat{L}),\ and$$

$$BIC = ln(n)k - 2ln(\hat{L})$$

Following are the terms used in the preceding equations:

- k represents the model number of parameters, or p+q
- $\hat{L}$ represents the maximum value of the likelihood function
- n is the number of input observations

The lower the AIC or BIC score, the better the model will fit. You can note that penalizing the BIC metric is higher with respect to AIC whenever $ln(n) > 2, \text{ or } n > e^2$. Let's use the simulated ARMA series we created previously to apply this tuning approach. For simplicity reasons, we will not include an intercept and restrict the maximum value of *k*, that is, the model order (or *p+q*), to four. We will try the ARMA(1,1) model first:

```
arima(arma, order = c(1, 0, 1), include.mean = FALSE)
```

We get the following output:

```
##
## Call:
## arima(x = arma, order = c(1, 0, 1), include.mean = FALSE)
##
## Coefficients:
##          ar1     ma1
##       0.4144  0.8864
## s.e.  0.0432  0.0248
##
## sigma^2 estimated as 1.051:  log likelihood = -723,  aic = 1452
```

The AIC score of the initial model is `1452`.

By default, the model output returns the AIC score of the model. If you wish to retrieve the BIC value, you can apply the `BIC` function, which returns the BIC value. For example, `BIC(arima(arma, order = c(1, 0, 1), include.mean = FALSE))` will return 1441.73, which is the BIC score of the ARMA model we fit to the simulated data we created in the preceding example.

We will now start to increase the value of *p* and *q* while monitoring the change of the AIC score. The next model is the ARMA(2,1) model:

```
arima(arma, order = c(2, 0, 1), include.mean = FALSE)
```

We get the following output:

```
##
## Call:
## arima(x = arma, order = c(2, 0, 1), include.mean = FALSE)
##
## Coefficients:
##          ar1     ar2     ma1
##       0.3136  0.1918  0.9227
## s.e.  0.0486  0.0484  0.0183
##
## sigma^2 estimated as 1.019:  log likelihood = -715.26,  aic = 1438.52
```

Increasing *p* from `1` to `2` improves the AIC score, and therefore the ARMA(2,1) model is superior over the ARMA(1,1) model. Now, we will check ARMA(1,2):

```
arima(arma, order = c(1, 0, 2), include.mean = FALSE)
```

We get the following output:

```
##
## Call:
## arima(x = arma, order = c(1, 0, 2), include.mean = FALSE)
##
## Coefficients:
##          ar1     ma1      ma2
##       0.7602  0.4654  -0.4079
## s.e.  0.0626  0.0858   0.0832
##
## sigma^2 estimated as 1.014:  log likelihood = -714.27,  aic = 1436.54
```

The AIC score of ARMA(1,2) is even lower than the one of ARMA(2,1), and therefore it is now the superior model. Now, we will try the last combination of ARMA(2,2) and see if we can achieve additional improvements:

```
arima(arma, order = c(2, 0, 2), include.mean = FALSE)
```

We get the following output:

```
##
## Call:
## arima(x = arma, order = c(2, 0, 2), include.mean = FALSE)
##
## Coefficients:
##          ar1     ar2     ma1      ma2
##       0.7239  0.0194  0.4997  -0.3783
## s.e.  0.2458  0.1257  0.2427   0.2134
##
## sigma^2 estimated as 1.014:  log likelihood = -714.26,  aic = 1438.51
```

The AIC score of the ARMA(2,2) model is higher than the one of the ARMA(1,2), and so we will select the ARMA(1,2) model. The following table summarizes the AIC score of the four models we tested:

Model	**AIC Score**
ARMA(1,1)	1452.00
ARMA(2,1)	1438.52
ARMA(1,2)	**1436.54**
ARMA(2,2)	1438.51

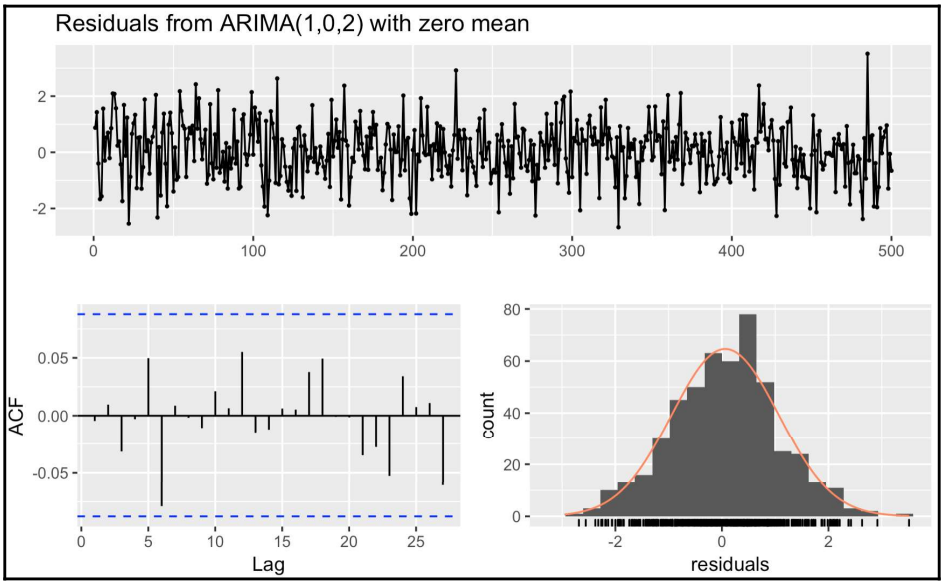
The use of the AIC or BIC score as a model selection criterion does not guarantee that the selected model (by either AIC or BIC score) does not violate the model assumptions. Therefore, the next step is to apply a residual analysis in order to verify that, as we saw in `Chapter 8`, *Forecasting Strategies*, the residuals are as follows:

- Mean and variance are constant
- White noise or not correlated
- Normally distributed

For instance, we can use the `checkresiduals` function from the **forecast** package in order to validate those assumptions on the selected model:

```
library(forecast)

checkresiduals(arima(arma, order = c(1, 0, 2), include.mean = FALSE))
```

You can observe in the following `checkresiduals` output plot that the residuals of the ARMA(1,2) model satisfy the model assumptions that we defined previously:



The `checkresiduals` function also returns the Ljung-Box test results, which suggests that the residuals are white noise:

```
##
##  Ljung-Box test
```

```
##
## data:  Residuals from ARIMA(1,0,2) with zero mean
## Q* = 5.3129, df = 7, p-value = 0.6218
##
## Model df: 3.   Total lags used: 10
```

Forecasting AR, MA, and ARMA models

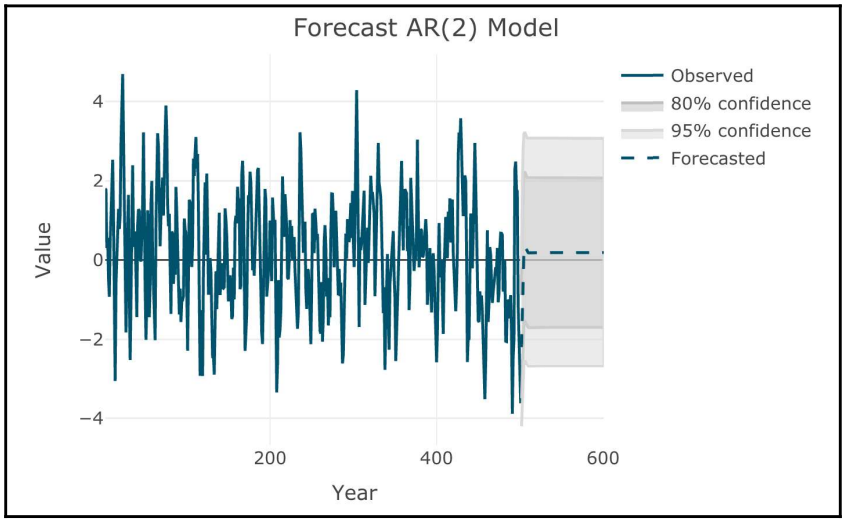
Forecasting any of the models we saw until now was straightforward: we used the `forecast` function from the **forecast** package in a similar manner to how we used it in the previous chapter. For instance, the following code demonstrates the forecast of the next `100` observations of the AR model we trained previously in *The AR process* section with the `ar` function:

```
ar_fc <- forecast(md_ar, h = 100)
```

We can use `plot_forecast` to plot the forecast output:

```
plot_forecast(ar_fc,
 title = "Forecast AR(2) Model",
 Ytitle = "Value",
 Xtitle = "Year")
```

We get the following output:



The ARIMA model

One of the limitations of the AR, MA, and ARMA models is that they cannot handle non-stationary time series data. Therefore, if the input series is non-stationary, a preprocessing step is required to transform the series from a non-stationary state into a stationary state. The ARIMA model provides a solution for this issue by adding the integrated process for the ARMA model. The **Integrated** (**I**) process is simply differencing the series with its lags, where the degree of the differencing is represented by the *d* parameter. The differencing process, as we saw previously, is one of the ways you can transform the methods of a series from non-stationary to stationary. For instance, Y_t - Y_{t-1} represents the first differencing of the series, while (Y_t - Y_{t-1}) - (Y_{t-1} - Y_{t-2}) represents the second differencing. We can generalize the differencing process with the following equation:

$$Y_d = (Y_t - Y_{t-1}) - \ldots - (Y_{t-d+1} - Y_{t-d}),\ where$$

Y_d is the *d* differencing of the series. Let's add the differencing component to the ARMA model and formalize the ARIMA model:

$$ARIMA(p,d,q) : Y_d = c + \sum_{i=1}^{p} \phi_i Y_{d-i} + \sum_{i=1}^{q} \theta_i \epsilon_{t-i} + \epsilon_t$$

Following are the terms used in the preceding equation:

- *ARIMA(p,d,q)* defines an ARIMA process with a *p*-order AR process, *d*-degree of differencing, and *q*-order MA process
- Y_d is the *d* difference of series Y_t
- *c* represents a constant (or drift)
- *p* defines the number of lags to regress against Y_t
- ϕ_i is the coefficient of the *i* lag of the series
- Y_{d-i} is the d difference of the *i* lag of the series
- *q* defines the number of past error terms to be used in the equation
- θ_i is the corresponding coefficient of ϵ_{t-i}
- $\epsilon_{t-q}, \ldots, \epsilon_t$ are white noise error terms
- ϵ_t represents the error term, which is white noise

As you can see, both the AR and MA models can be represented with the ARMA model, you can also represent the AR, MA, or ARMA models with the ARIMA model, for example:

- The ARIMA(0, 0, 0) model is equivalent to white noise
- The ARIMA(0, 1, 0) model is equivalent to a random walk
- The ARIMA(1, 0, 0) model is equivalent to an AR(1) process
- The ARIMA(0, 0, 1) model is equivalent to an MA(1) process
- The ARIMA(1, 0, 1) model is equivalent to an ARMA(1,1) process

Identifying an ARIMA process

In this section, we will apply an ARIMA model, instead of the ARMA model, whenever the input series is not stationary. Differencing is required to transfer it into a stationary state. Identifying and setting the ARIMA model is a two-step process and is based on the following steps:

1. Identify the degree of differencing that is required to transfer the series into a stationary state
2. Identify the ARMA process (or AR and MA processes), as introduced in the previous section

Based on the findings of these steps, we will set the model parameters p, d, and q.

Identifying the model degree of differencing

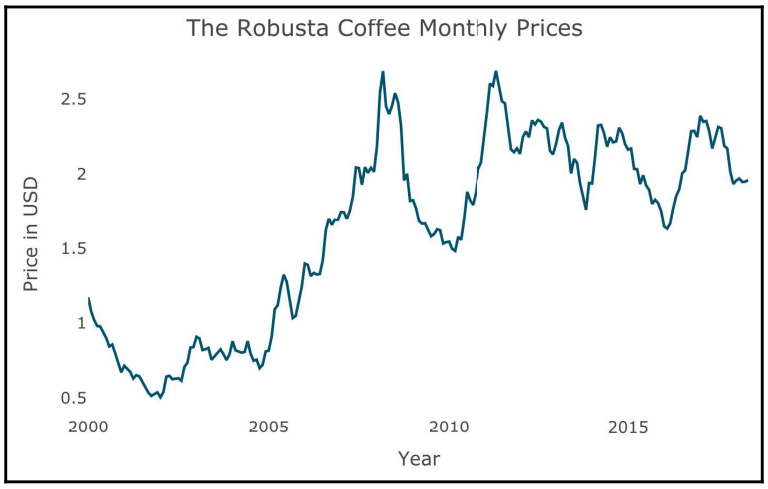
Similar to the p and q parameters, setting the d parameter (the degree of differencing of the series) can be done with the ACF and PACF plots. In the following example, we will use the monthly prices of Robusta coffee since 2000. This series is part of the `Coffee_Prices` multiple time series objects from the **TSstudio** package. We will start by loading the `Coffee_Prices` series and subtracting the Robusta monthly prices since January 2010 using the `window` function:

```
data("Coffee_Prices")

robusta_price <- window(Coffee_Prices[,1], start = c(2000, 1))
```

Let's plot the `robusta_price` series and review its structure with the `ts_plot` function:

```
ts_plot(robusta_price,
        title = "The Robusta Coffee Monthly Prices",
        Ytitle = "Price in USD",
        Xtitle = "Year")
```

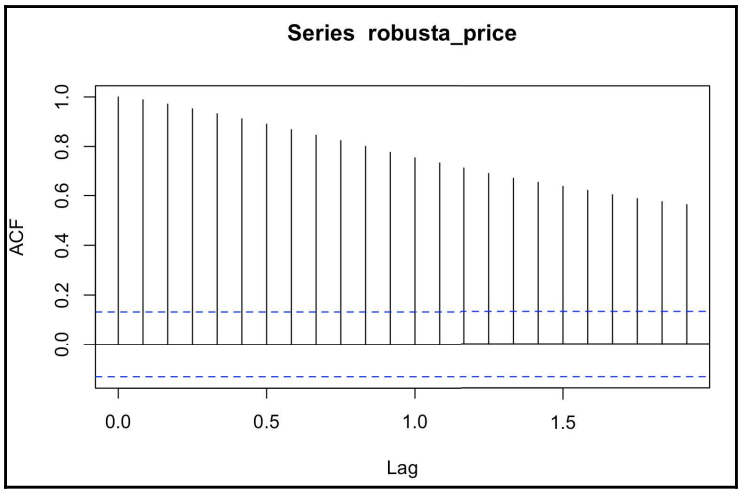
We get the following output:



As you can see, the Robusta coffee prices over time are trending up, and therefore it is not in a stationary state. In addition, since this series represents continues prices, it is likely that the series has a strong correlation relationship with its past lags (as changes in price are typically close to the previous price). We will use the `acf` function again to identify the type of relationship between the series and it lags:

```
acf(robusta_price)
```

We get the following output:



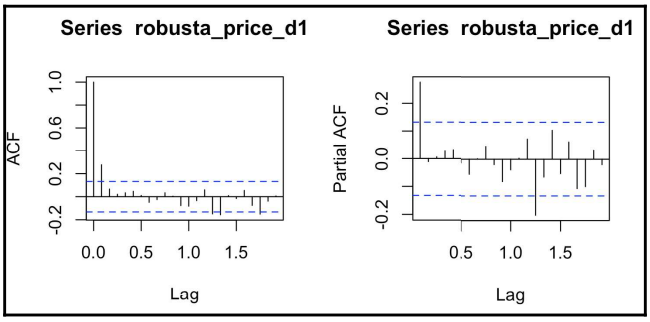
As you can see in the preceding output of the ACF plot, the correlation of the series with its lags is slowly decaying over time in a linear manner. Removing both the series trend and correlation between the series and its lags can be done by differencing the series. We will start with the first differencing using the `diff` function:

```
robusta_price_d1 <- diff(robusta_price)
```

Let's review the first difference of the series with the `acf` and `pacf` functions:

```
par(mfrow=c(1,2))
acf(robusta_price_d1)
pacf(robusta_price_d1)
```

We get the following output:



The ACF and PACF plots of the first difference of the series indicate that an AR(1) process is appropriate to use on the differenced series since the ACF is tailing off and the PACF cuts on the first lag. Therefore, we will apply an ARIMA(1,1,0) model on the `robusta_price` series to includes the first difference:

```
robusta_md <- arima(robusta_price, order = c(1, 1, 0))
```

We will use the `summary` function to review the model details:

```
summary(robusta_md)
```

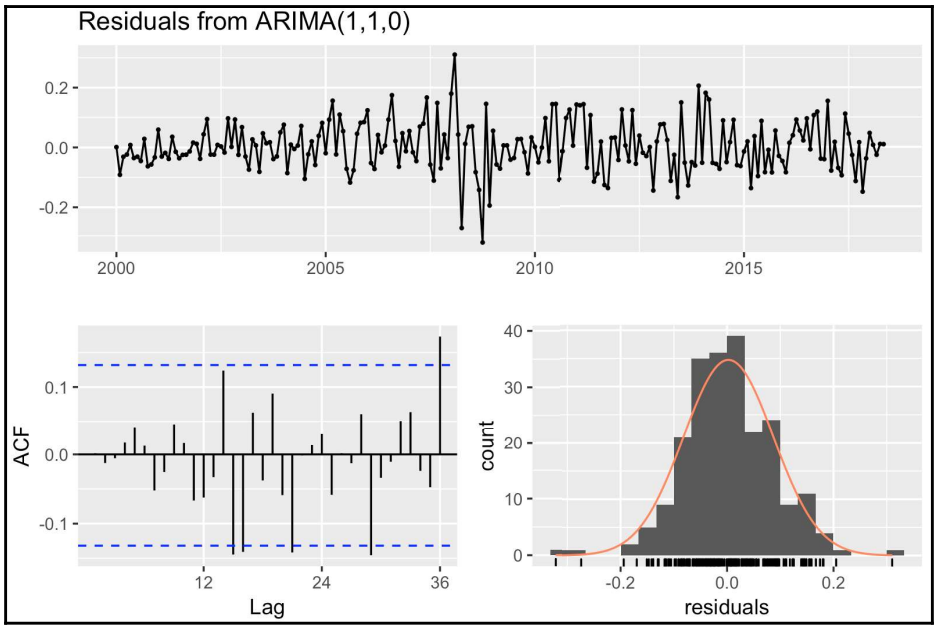
We get the following output:

```
##
## Call:
## arima(x = robusta_price, order = c(1, 1, 0))
##
## Coefficients:
##          ar1
##       0.2780
## s.e.  0.0647
##
## sigma^2 estimated as 0.007142:  log likelihood = 231.38,  aic = -458.76
##
## Training set error measures:
##                       ME       RMSE        MAE        MPE     MAPE
## Training set 0.002595604 0.08432096 0.06494772 0.08104715 4.254984
##                   MASE        ACF1
## Training set 1.001542 0.001526295
```

You can see from the model summary output that the `ar1` coefficient is statistically significant. Last but not least, we will check the model residuals:

```
checkresiduals(robusta_md)
```

We get the following output:



The Ljung-Box test suggested that the residuals are white noise:

```
##
##  Ljung-Box test
##
## data:  Residuals from ARIMA(1,1,0)
## Q* = 26.896, df = 23, p-value = 0.2604
##
## Model df: 1.   Total lags used: 24
```

Overall, the plot of the model's residuals and the Ljung-Box test indicate that the residuals are white noise. The ACF plot indicates that there are some correlated lags, but they are only on the border of being significant and so we can ignore them.

The seasonal ARIMA model

The **Seasonal ARIMA** (**SARIMA**) model, as its name implies, is a designated version of the ARIMA model for time series with a seasonal component. As we saw in `Chapter 6`, *Seasonality Analysis*, and `Chapter 7`, *Correlation Analysis*, a time series with a seasonal component has a strong relationship with its seasonal lags. The SARIMA model is utilizing the seasonal lags in a similar manner to how the ARIMA model is utilizing the non-seasonal lags with the AR and MA processes and differencing. It does this by adding the following three components to the ARIMA model:

- **SAR(P) process**: A seasonal AR process of the series with its past P seasonal lags. For example, a SAR(2) is an AR process of the series with its past two seasonal lags, that is, $Y_t = c + \Phi_1 Y_{t-f} + \Phi_2 Y_{t-2f} + \epsilon_t$, where Φ represents the seasonal coefficient of the SAR process, and f represents the series frequency.
- **SMA(Q) process**: A seasonal MA process of the series with its past Q seasonal error terms. For instance, a SMA(1) is a moving average process of the series with its past seasonal error term, that is, $Y_t = \mu + \epsilon_t + \Theta_1 \epsilon_{t-f}$, where Θ represents the seasonal coefficient of the SMA process, and f represents the series frequency.
- **SI(D) process**: A seasonal differencing of the series with its past D seasonal lags. In a similar manner, we can difference the series with its seasonal lag, that is, $Y_{D=1} = Y_t - Y_{t-f}$.

We use the following notation to denote the *SARIMA* parameters:

$$SARIMA(p,d,q) \times (P,D,Q)_s$$

Like before, the p and q parameters define the order of the AR and MA processes with its non-seasonal lags, respectively, and d defines the degree of differencing of the series with its non-seasonal lags. Likewise, the P and Q parameters represent the corresponding order of the seasonal AR and MA processes of the series with its seasonal lags, and D defines the degree of differencing of the series with its non-seasonal lags. For example, a $SARIMA(1,0,0) \times (1,1,0)_s$ model is a combination of an AR process of one non-seasonal and one seasonal lag, along with seasonal differencing.

Tuning the SARIMA model

The tuning process of the SARIMA model follows the same logic as one of the ARIMA models. However, the complexity of the model increases as there are now six parameters to tune, that is, p, d, q, P, D, and Q, as opposed to three with the ARIMA model. Luckily, the tuning of the P, D, and Q seasonal parameters follows the same logic as the ones of p, d, q, respectively, with the use of the ACF and PACF plots. The main difference between the tuning of these two groups of parameters (non-seasonal and seasonal) is that the non-seasonal parameters are tuned with the non-seasonal lags, as we saw previously with the ARIMA model. On the other hand, the tuning of the seasonal parameters are tuned with the seasonal lags (for example, for monthly series with lags 12, 24, 36, and so on).

Tuning the non-seasonal parameters

Applying the same logic that we used with the ARIMA model, tuning the non-seasonal parameters of the SARIMA model is based on the ACF and PACF plots:

- An *AR(p)* process should be used if the non-seasonal lags of the ACF plot are tailing off, while the corresponding lags of the PACF plots are cutting off on the p lag
- Similarly, an *MA(q)* process should be used if the non-seasonal lags of the ACF plot are cutting off on the q lag and the corresponding lags of the PACF plots are tailing off
- When both the ACF and PACF non-seasonal lags are tailing off, an ARMA model should be used
- Differencing the series with the non-seasonal lags should be applied when the non-seasonal lags of the ACF plot are decaying in a linear manner

Tuning the seasonal parameters

Tuning the seasonal parameters of the SARIMA model with ACF and PACF follows the same guidelines as the ones we used for selecting the ARIMA parameters:

- We will use a seasonal autoregressive process with an order of P, or SAR(P), if the seasonal lags of the ACF plot are tailing off and the seasonal lags of the PACF plot are cutting off by the P seasonal lag
- Similarly, we will apply a seasonal moving average process with an order of Q, or SMA(Q), if the seasonal lags of the ACF plot are cutting off by the Q seasonal lag and the seasonal lags of the PACF plot are tailing off

- An ARMA model should be used whenever the seasonal lags of both the ACF and PACF plots are tailing off
- Seasonal differencing should be applied if the correlation of the seasonal lags are decaying in a linear manner

Forecasting US monthly natural gas consumption with the SARIMA model – a case study

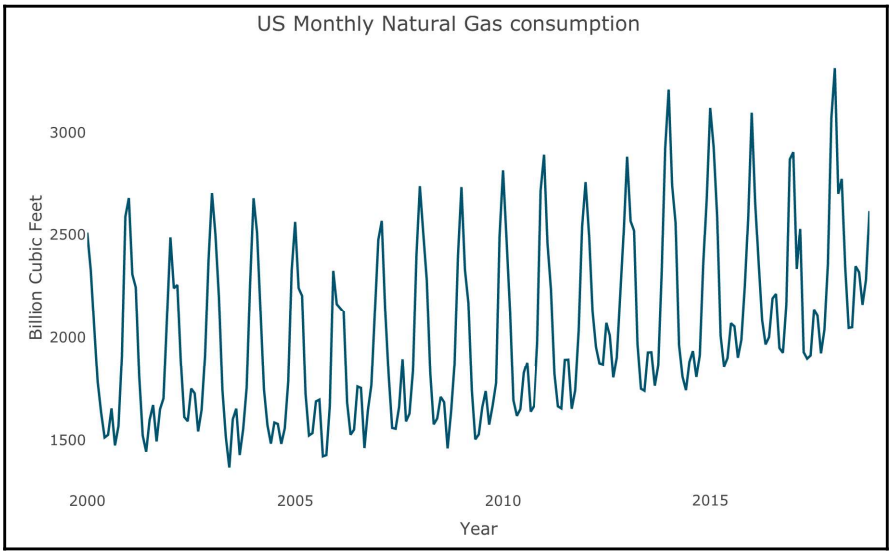
In this section, we will apply what we learned throughout this chapter and forecast the monthly consumption of natural gas in the US using the SARIMA model. Let's load the `USgas` series from the **TSstudio** package:

```
data(USgas)
```

Let's plot the series with the `ts_plot` function and review the main characteristics of the series:

```
ts_plot(USgas,
        title = "US Monthly Natural Gas consumption",
        Ytitle = "Billion Cubic Feet",
        Xtitle = "Year")
```

We get the following output:



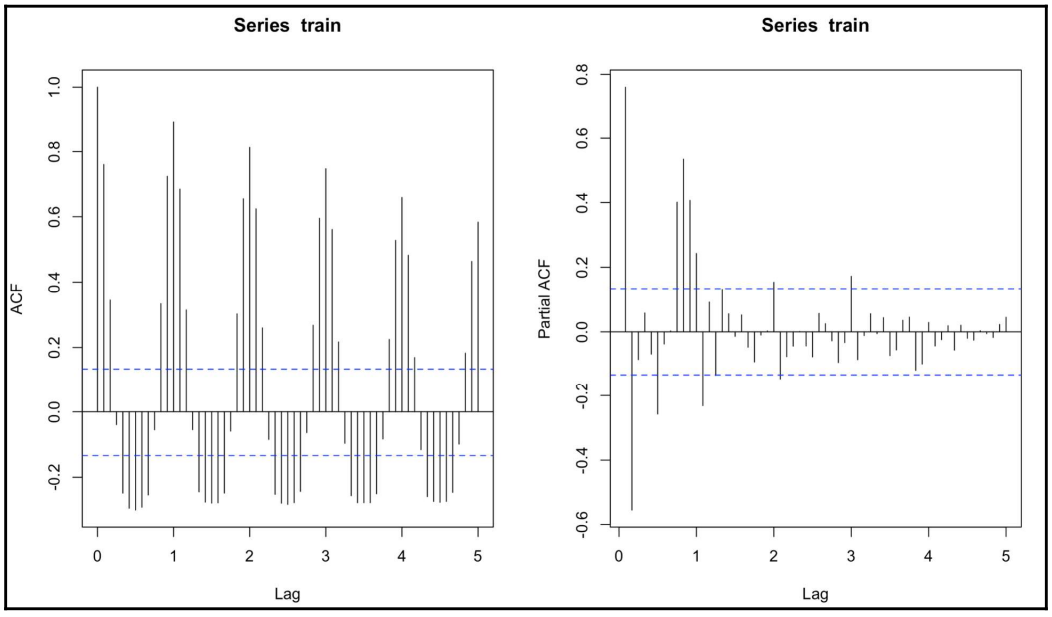
As we saw in the previous chapters, the `USgas` series has a strong seasonal pattern, and therefore among the ARIMA family of models, the SARIMA model is the most appropriate model to use. In addition, since the series is trending up, we can already conclude that the series is not stationary and some differencing of the series is required. We will start by setting the training and testing partitions with the `ts_split` functions, leaving the last `12` months of the series as the testing partition:

```
USgas_split <- ts_split(USgas, sample.out = 12)

train <- USgas_split$train
test <- USgas_split$test
```

Before we start the training process of the SARIMA model, we will conduct diagnostics in regards to the series correlation with the ACF and PACF functions. Since we are interested in viewing the relationship of the series with its seasonal lags, we will increase the number of lags to calculate and display by setting the `lag.max` argument to `60` lags:

```
par(mfrow=c(1,2))

acf(train, lag.max = 60)

pacf(train, lag.max = 60)
```

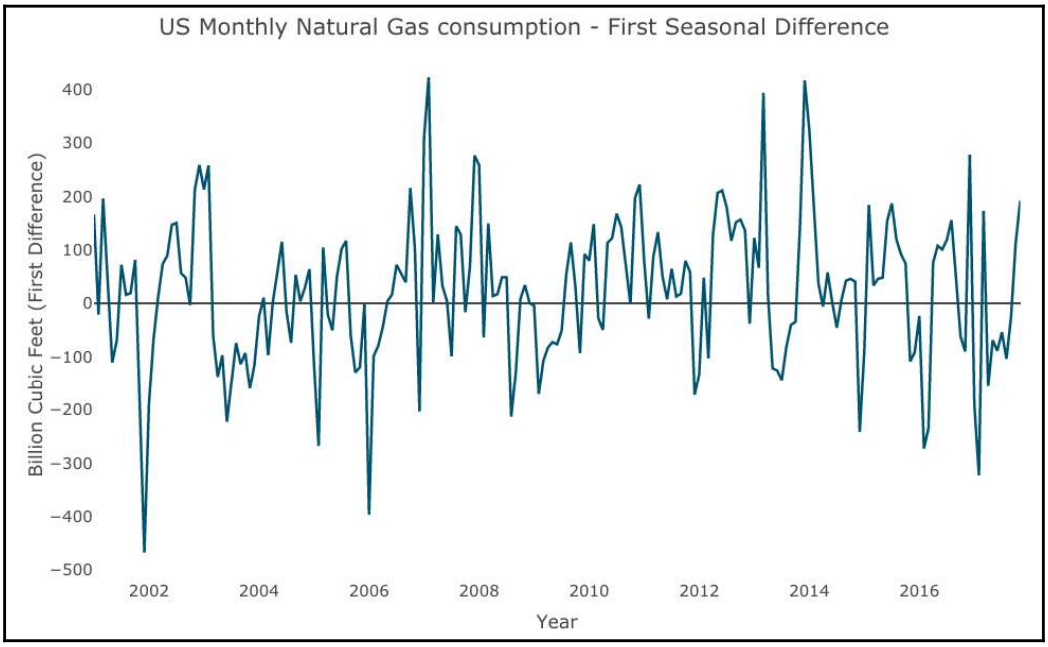
We get the following output:



The preceding ACF plot indicates that the series has a strong correlation with both the seasonal and non-seasonal lags. Furthermore, the linear decay of the seasonal lags indicates that the series is not stationary and that seasonal differencing is required. We will start with a seasonal differencing of the series and plot the output to identify whether the series is in a stationary state:

```
USgas_d12 <- diff(train, 12)

ts_plot(USgas_d12,
        title = "US Monthly Natural Gas consumption - First Seasonal
Difference",
        Ytitle = "Billion Cubic Feet (First Difference)",
        Xtitle = "Year")
```

We get the following output:

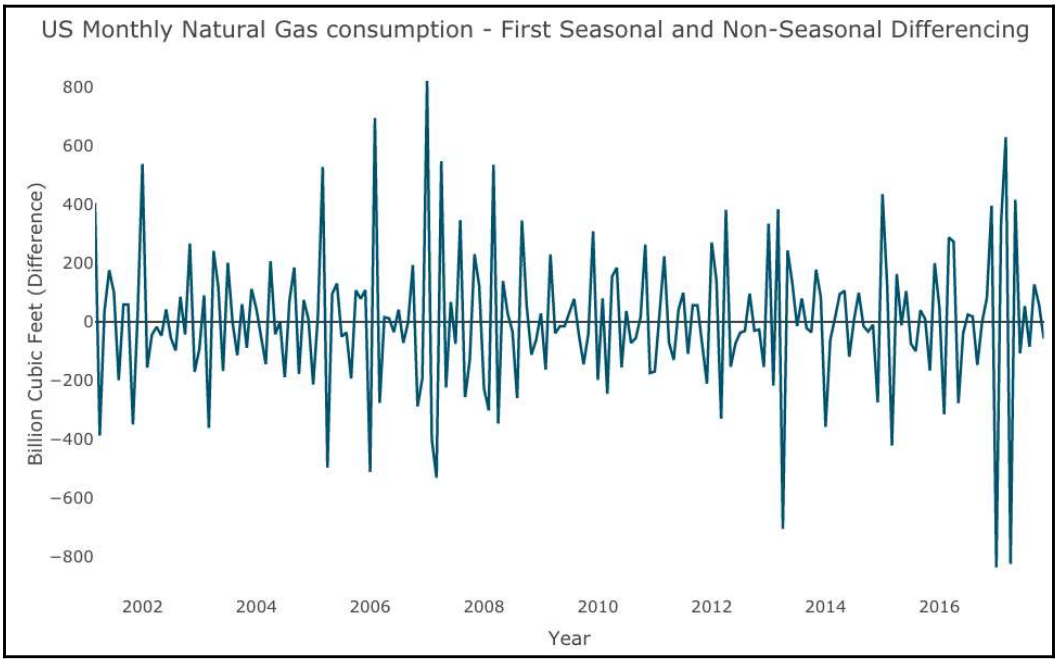


While we removed the series trend, the variation of the series is not stable yet. Therefore, we will also try to take the first difference of the series:

```
USgas_d12_1 <- diff(diff(USgas_d12, 1))

ts_plot(USgas_d12_1,
        title = "US Monthly Natural Gas consumption - First Seasonal and
Non-Seasonal Differencing",
```

```
        Ytitle = "Billion Cubic Feet (Difference)",
        Xtitle = "Year")
```

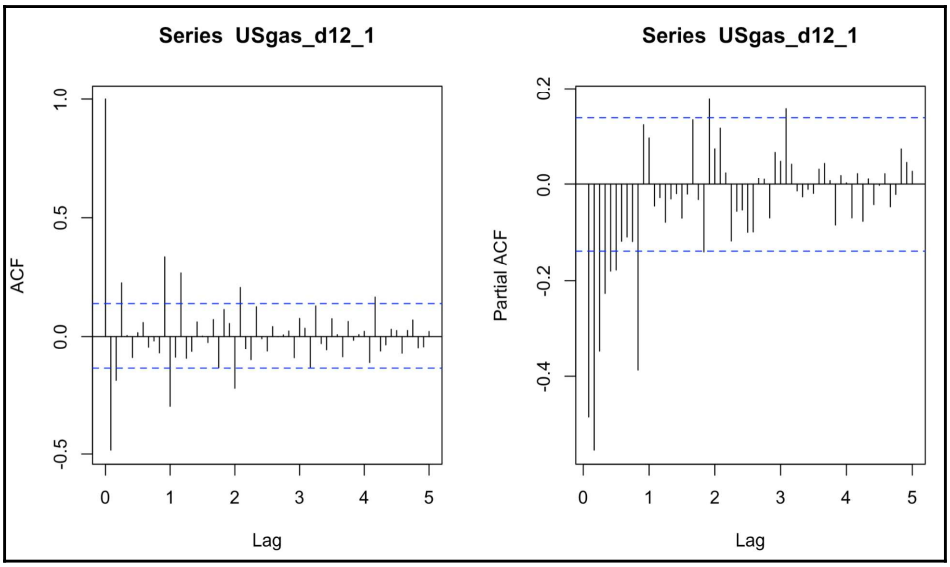
We get the following output:



After taking the first order differencing, along with the first order seasonal differencing, the series seems to stabilize around the zero *x* axis line (or fairly close to being stable). After transforming the series into a stationary state, we can review the ACF and PACF functions again to identify the required process:

```
par(mfrow=c(1,2))

acf(USgas_d12_1, lag.max = 60)

pacf(USgas_d12_1, lag.max = 60)
```

We get the following output:



The main observation from the preceding ACF and PACF plots is that both the non-seasonal and seasonal lags (in both plots) are tailing off. Hence, we can conclude that after we difference the series and transform them into a stationary state, we should apply an ARMA process for both the seasonal and non-seasonal components of the SARIMA model.

The tuning process of the SARIMA model parameters follow the same steps that we applied previously with the ARMA model:

- We set the model maximum order (that is, the sum of the six parameters of the model)
- We set a range of a possible combination of the parameters' values under the model's maximum order constraint
- We test and score each model, that is, a typical score methodology with the AIC (which we used previously) or BIC
- We select a set of parameter combinations that give the best results

Now, we will start the tuning process for the `USgas` series by setting the model order to seven and setting the values of the model parameters to be in the range of `0` and `2`. Given that we already identified the values of *d* and *D* (for example, *d = 1* and *D = 1*), which are the differencing parameters of the SARIMA model, we now can focus on tuning the remaining four parameters of the model, that is, `p`, `q`, `P`, and `Q`. Let's define those parameters and assign the search values:

```
p <- q <- P <- Q <- 0:2
```

Under the model's order constraint and the possible range of values of the model parameters, there are 66 possible combinations. Therefore, it will make sense, in this case, to automate the search process and build a grid search function to identify the values of the parameters that minimize the AIC score. We will utilize the `expand.grid` function in order to create a `data.frame` with all the possible search combinations:

```
arima_grid <- expand.grid(p,q,P,Q)

names(arima_grid) <- c("p", "q", "P", "Q")

arima_grid$d <- 1

arima_grid$D <- 1
```

Next, we will trim the grid search table by using combinations that exceed the order constraint of the model (for example, $k \leq 7$). We will calculate and assign this to the `k` variable with the `rowSums` function:

```
arima_grid$k <- rowSums(arima_grid)
```

We will utilize the `filter` function from the **dplyr** package to remove combinations where the `k` value is greater than `7`:

```
library(dplyr)

arima_grid <- arima_grid %>% filter(k <= 7)
```

Now that the grid search table is ready, we can start the search process. We will use the `lapply` function to iterate over the grid search table. This function will train the SARIMA model and score its AIC for each set of parameters in the grid search table. The `arima` function can train the SARIMA model by setting the `seasonal` argument of the model with the values of `P`, `D`, and `Q`:

```
arima_search <- lapply(1:nrow(arima_grid), function(i){
   md <- NULL
   md <- arima(train, order = c(arima_grid$p[i], 1, arima_grid$q[i]),
seasonal = list(order = c(arima_grid$P[i], 1, arima_grid$Q[i])))
   results <- data.frame(p = arima_grid$p[i], d = 1, q = arima_grid$q[i],
                         P = arima_grid$P[i], D = 1, Q = arima_grid$Q[i],
                         AIC = md$aic)
 }) %>% bind_rows() %>% arrange(AIC)
```

We used the `bind_rows` and `arrange` functions to append the search results and arranged the table for the **dplyr** functions. Let's review the top results of the search table:

```
head(arima_search)
```

We get the following output:

```
##   p d q P D Q      AIC
## 1 1 1 1 2 1 1 2459.807
## 2 0 1 2 2 1 1 2461.229
## 3 1 1 1 0 1 1 2463.866
## 4 1 1 1 1 1 2 2464.550
## 5 1 1 1 0 1 2 2464.873
## 6 1 1 1 1 1 1 2465.310
```

The leading model based on the preceding search table is the $SARIMA(1,1,1)(2,1,1)_{12}$ model. Before we finalize the forecast, let's evaluate the selected model's performance on the testing set. We will retrain the model using the settings of the selected model:

```
USgas_best_md <- arima(train, order = c(1,1,1), seasonal = list(order =
c(2,1,1)))
```

The model coefficients, as we can see in the following model summary, are all statistically significant at a level of 0.1:

```
USgas_best_md
```

We get the following output:

```
##
## Call:
## arima(x = train, order = c(1, 1, 1), seasonal = list(order = c(2, 1,
1)))
##
## Coefficients:
##          ar1      ma1     sar1    sar2     sma1
##       0.4143  -0.9244  -0.0216  -0.256  -0.7565
## s.e.  0.0733   0.0291   0.0956   0.088   0.0803
##
## sigma^2 estimated as 9837:  log likelihood = -1223.9,  aic = 2459.81
```

Let's use the `USgas_best_md` trained model to forecast the corresponding observations of the testing set:

```
USgas_test_fc <- forecast(USgas_best_md, h = 12)
```

Like we did previously, we will assess the model's performance with the accuracy function:

```
accuracy(USgas_test_fc, test)
```

We get the following output:

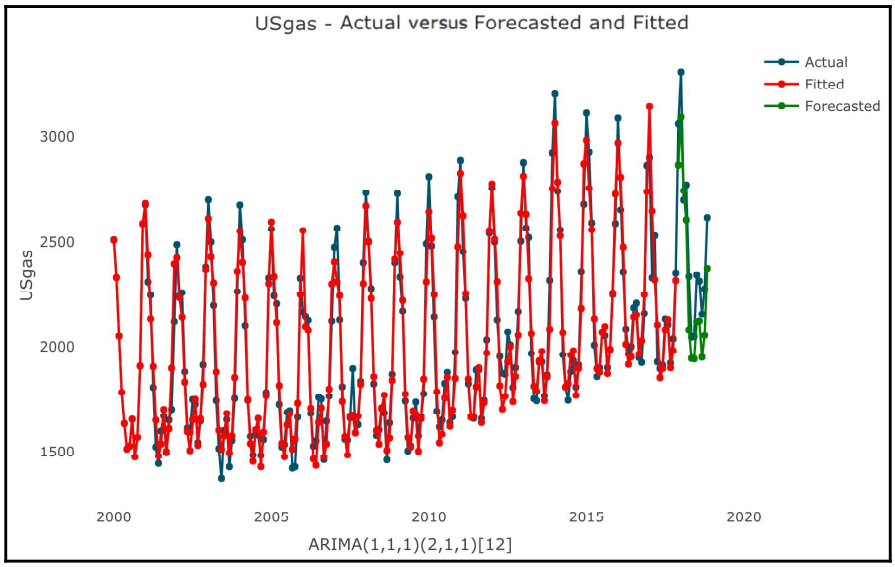
```
## ME RMSE MAE MPE MAPE MASE
## Training set 3.486041 96.13846 70.81773 0.03448147 3.445378 0.6600763
## Test set 173.187631 190.79318 180.62400 7.05220971 7.327692 1.6835560
## ACF1 Theil's U
## Training set 0.008556121 NA
## Test set -0.056179946 0.730488
```

We can use the performance of the seasonal naive model we used in `Chapter 8`, *Forecasting Strategies* (using the same training and testing set), as a benchmark for the SARIMA model's performance. Recall that the seasonal naive model's MAPE score was 5.2% on the training set and 9.7% on the testing set. Therefore, the SARIMA provides us with a lift in accuracy with a MAPE score of 3.4% and 7.3% on the training and testing partitions, respectively.

Now, we will use the `test_forecast` function to get a more intuitive view of the model's performance on the training and testing partitions:

```
test_forecast(USgas,
              forecast.obj = USgas_test_fc,
              test = test)
```

We get the following output:



As you can see, the SARIMA model successfully captures the seasonal and trend pattern of the series. On the other hand, the model finds it challenging to capture the seasonal peaks (month of January) on the training partition and has 6.7% absolute error for the month of January (yearly peak) in the testing partition. This is the result of high fluctuation during the winter time. We can handle this uncertainty of the model during peak times with the model confidence intervals or path simulations the we looked at in `Chapter 8`, *Forecasting Strategies*.

At this point, you should pause and evaluate the general performance of the model thus far on the major metrics:

- AIC score on the training set
- The model coefficients significant
- MAPE score on both the training and testing set
- Benchmark the performance against other models (for example, a naive model)
- Fitted and forecasted plot

If you are satisfied with the results of the model on the different performance metrics, you should move forward and retrain the model with all the series and generate the final forecast. Otherwise, you should return the model parameters and repeat the evaluation process.

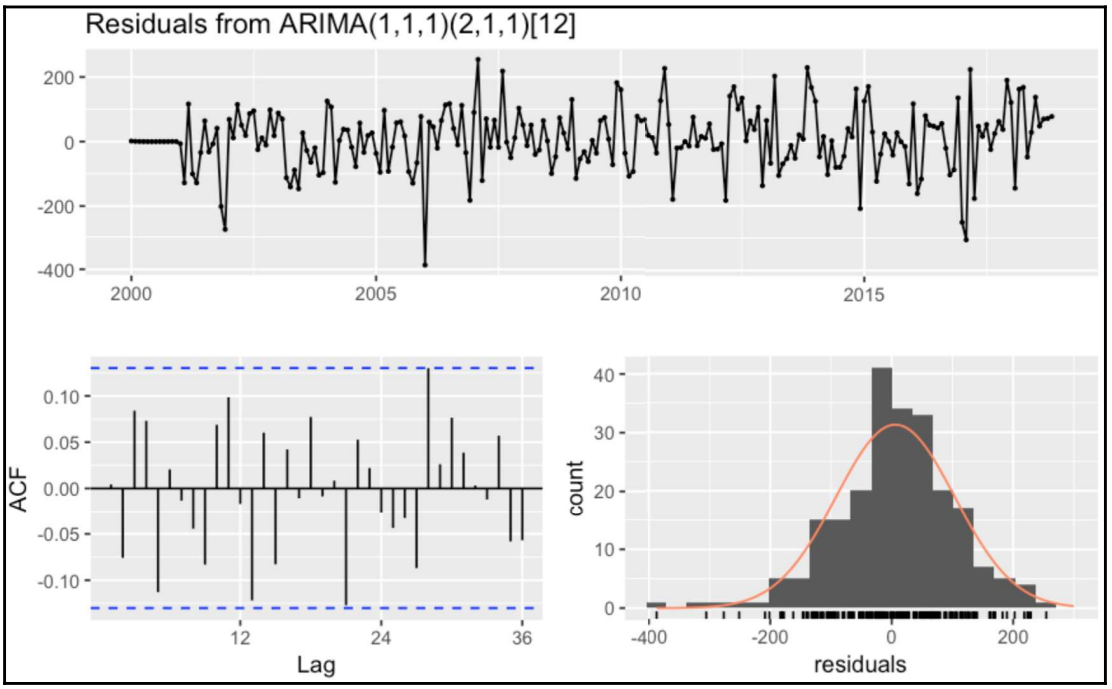
Now that we've satisfied the preceding conditions, we can move on to the last step of the forecasting process and generate the final forecast with the selected model. We will start by retraining the selected model on all the series:

```
final_md <- arima(USgas, order = c(1,1,1), seasonal = list(order =
c(2,1,1)))
```

Before we forecast the next 12 months, let's verify that the residuals of the model satisfy the model condition:

```
checkresiduals(final_md)
```

We get the following output:



The output of the Ljung-Box test suggested that the residuals of the model are white noise:

```
## Ljung-Box test
##
## data: Residuals from ARIMA(1,1,1)(2,1,1)[12]
## Q* = 26.254, df = 19, p-value = 0.1233
##
## Model df: 5. Total lags used: 24
```

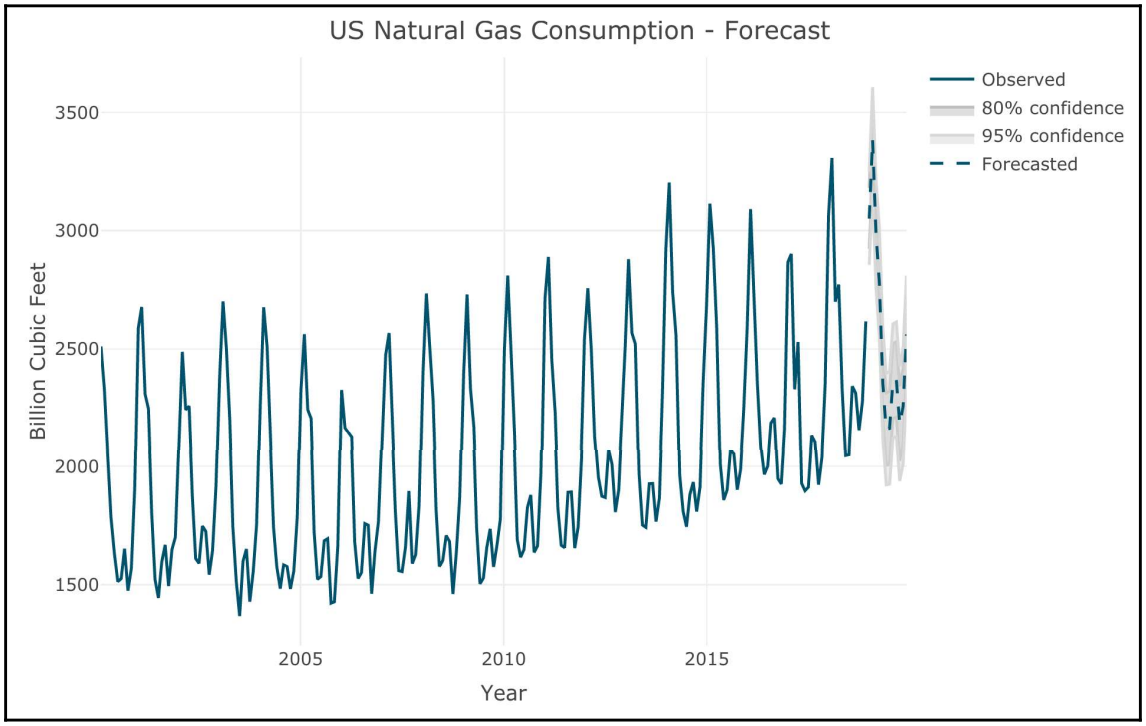
By looking at the preceding residuals plot, you can see that the residuals are white noise and normally distributed. Furthermore, the Ljung-Box test confirms that there is no auto-correlation left on the residuals—with a *p*-value of 0.12, we cannot reject the null hypothesis that the residuals are white noise. We are good to go! Let's use the `forecast` function to forecast the next `12` months of the `USgas` series:

```
USgas_fc <- forecast(final_md, h = 12)
```

We can plot the forecast with the `plot_forecast` function:

```
plot_forecast(USgas_fc,
              title = "US Natural Gas Consumption - Forecast",
              Ytitle = "Billion Cubic Feet",
              Xtitle = "Year")
```

We get the following output:



The auto.arima function

One of the main challenges of forecasting with the ARIMA family of models is the cumbersome tuning process of the models. As we saw in this chapter, this process includes many manual steps that are required for verifying the structure of the series (stationary or non-stationary), data transformations, descriptive analysis with the ACF and PACF plots to identify the type of process, and eventually tune the model parameters. While it might take a few minutes to train an ARIMA model for a single series, it may not scale up if you have dozens of series to forecast.

The `auto.arima` function from the **forecast** package provides a solution to this issue. This algorithm automates the tuning process of the ARIMA model with the use of statistical methods to identify both the structure of the series (stationary or not) and type (seasonal or not), and sets the model's parameters accordingly. For example, we can utilize the function to forecast `USgas`:

```
USgas_auto_md1 <- auto.arima(train)
```

Using the default arguments of the `auto.arima` function returns the ARIMA model that minimizes the AIC score. In this case, a $SARIMA(1,0,4)(1,1,2)_{12}$ model was selected with an AIC score of `2480.57`:

```
USgas_auto_md1
```

We get the following output:

```
## Series: train
## ARIMA(1,0,4)(1,1,2)[12] with drift
##
## Coefficients:
##          ar1     ma1     ma2     ma3     ma4     sar1     sma1     sma2
##       0.3405  0.1773  0.0473  0.1731  0.1617  -0.3743  -0.3040  -0.4527
## s.e.  0.2970  0.2984  0.1429  0.0854  0.1015   0.2831   0.2672   0.2022
##        drift
##       2.3219
## s.e.  0.3678
##
## sigma^2 estimated as 10537:  log likelihood=-1230.28
## AIC=2480.57   AICc=2481.71   BIC=2513.7
```

By default, the `auto.arima` function applies a shorter model search by using a step-wise approach for reducing the search time. The trade-off of this approach is that the model may miss some models that may achieve better results. This can be seen in the results of `USgas_auto_md1`, which achieved a higher AIC score than the model we tuned with the grid search we used previously (`2480.57` versus `2459.81` in the previous model).

We can improvise with the `auto.arima` results by setting the search argument of the model. The step-wise argument, when set to `FALSE`, allows you to set a more robust and thorough search, with the cost of higher search time. This trade-off between performance and compute time can be balanced whenever you have prior knowledge about the series' structure and characteristics. For example, let's retrain the training set of the `USgas` series again, this time using the following settings:

- Set the differencing parameters `d` and `D` to `1`.
- Limit the order of the model to seven by using the `max.order` argument. The `max.order` argument defines the maximum values of *p+q+P+Q*, hence we should set it to five (given that `d` and `D` are set to `1`).
- Under these constraints, search all possible combinations by setting the `stepwise` argument to `FALSE`.
- Set the `approximation` argument to `FALSE` for more accurate calculations of the information criteria:

```
USgas_auto_md2 <- auto.arima(train,
                             max.order = 5,
                             D = 1,
                             d = 1,
                             stepwise = FALSE,
                             approximation = FALSE)
```

You can see from the following results that with the robust search, the `auto.arima` algorithm returns the same model that was identified with the grid search we used in the *Seasonal ARIMA model* section:

```
USgas_auto_md2
```

We get the following output:

```
## Series: train
## ARIMA(1,1,1)(2,1,1)[12]
##
## Coefficients:
## ar1 ma1 sar1 sar2 sma1
## 0.4143 -0.9244 -0.0216 -0.256 -0.7565
## s.e. 0.0733 0.0291 0.0956 0.088 0.0803
##
## sigma^2 estimated as 10087: log likelihood=-1223.9
## AIC=2459.81 AICc=2460.24 BIC=2479.66
```

Linear regression with ARIMA errors

In `Chapter 9`, *Forecasting with Linear Regression*, we saw that with some simple steps, we can utilize a linear regression model as a time series forecasting model. Recall that a general form of the linear regression model can be represented by the following equation:

$$Y_t = \beta_0 + \beta X_t + \epsilon_t$$

One of the main assumptions of the linear regression model is that the error term of the series, ϵ, is the white noise series (for example, there is no correlation between the residuals and their lags). However, when working with time series data, this assumption is eased as, typically, the model predictors do not explain all the variations of the series, and some patterns are left on the model residuals. An example of the failure of this assumption can be seen while fitting a linear regression model to forecast the `AirPassenger` series.

Violation of white noise assumption

To illustrate this point, we will utilize the `tslm` function to regress the `AirPassenger` series with its trend, seasonal component, and seasonal lag (lag 12), and then evaluate the model residuals with the `checkresiduals` function. Before we start with the training process, let's prepare the data and create new features to represent the series trend and seasonal (using the `month` function from the **lubridate** package) components, as well as the seasonal lag:

```
df <- ts_to_prophet(AirPassengers)

names(df) <- c("date", "y")
```

```
df$lag12 <- dplyr::lag(df$y, n = 12)

library(lubridate)

df$month <- factor(month(df$date, label = TRUE), ordered = FALSE)

df$trend <- 1:nrow(df)
```

Here, the three variables represent the following:

- `lag12`: A numeric variable that represents the seasonal lag of the series (that is, lag 12)
- `month`: A categorical variable (12 categories, one for each month of the year) that represents the seasonal component of the series
- `trend`: A numeric variable that represents the marginal effect of moving in time by one index unit, which in this case is the marginal change of the series by moving in time by one month

Now, we will split the series into training and testing partitions, leaving the last `12` months for testing using the `ts_split` function:

```
par <- ts_split(ts.obj = AirPassengers, sample.out = 12)

train <- par$train

test <- par$test
```

For the regression of the time series object with an external `data.frame` object (`df`), we will apply the same split for training and testing partitions on the predictor's `data.frame` object:

```
train_df <- df[1:(nrow(df) - 12), ]
test_df <- df[(nrow(df) - 12 + 1):nrow(df), ]
```

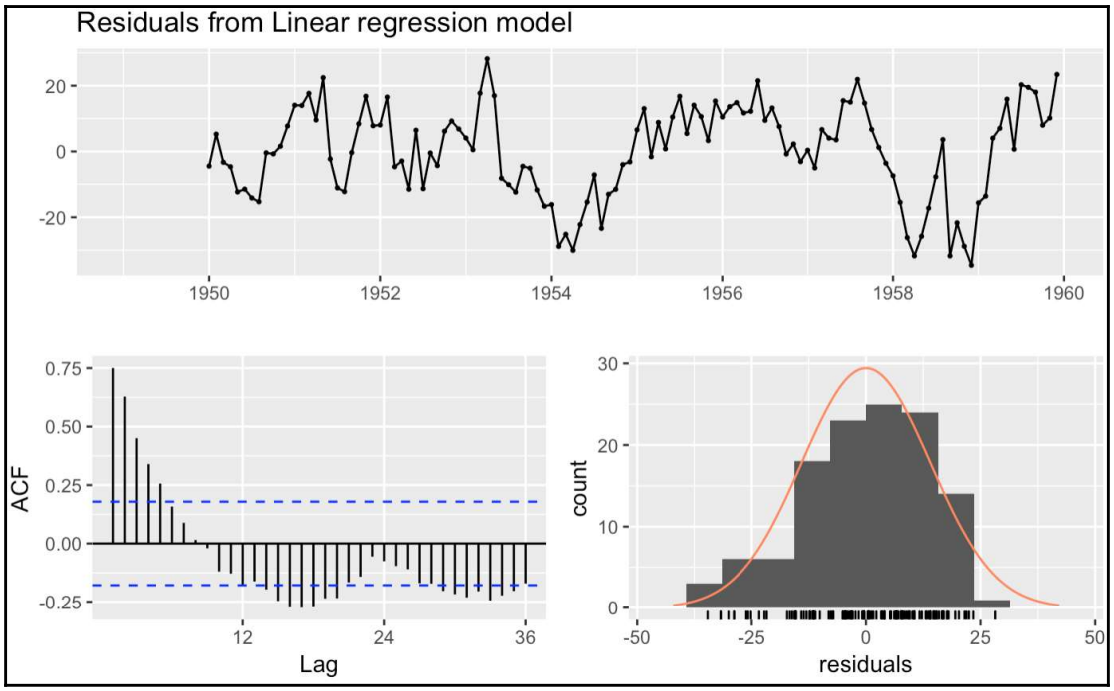
Now, we can start to train the model with the `tslm` function:

```
md1 <- tslm(train ~ season + trend + lag12, data = train_df)
```

Let's use the `checkresiduals` function to review the model's residuals:

```
checkresiduals(md1)
```

We get the following output:



In the preceding ACF plot, the residuals series has a strong correlation with its past lags, and therefore the series is not white noise. We can conclude from the residuals plot that the regression model could not capture all the series patterns. Generally, this should not come as a surprise since the variation of the series could be affected by other variables, such as the ticket and oil prices, the unemployment rate, and other external factors.

A simple solution to this problem is to model the error terms with the ARIMA model and add it to the regression.

Modeling the residuals with the ARIMA model

As we saw in the preceding ACF plot, the `AirPassengers` residuals indicate that the model was unable to capture all the patterns of the series. Another way to think about this is that the modeling of the series is not complete yet, and additional modeling can be applied on the model residuals to reveal the true error term of the model. For instance, the following expression describes the linear relationship between the `AirPassengers` series and the *trend*, *month*, and *lag12* variables:

$$Y_t = \beta_0 + \beta_1 trend_t + \beta_2 month_t + \beta_3 lag12_t + \epsilon_t$$

Then, the correlated error term, ϵ_t, could be described by the following expression:

$$\epsilon_t = \beta_4 X_4 + \ldots + \beta_n X_n + \epsilon'_t$$

The following terms are used in the preceding equation:

- $X_4, \ldots, X_n$ are the additional variables that are required to explain the variation of the series that is not explained by the *trend*, *month*, and *lag12* variables
- $\beta_4, \ldots, \beta_n$ are the corresponding coefficients of the $X_4, \ldots, X_n$ variables
- ϵ'_t is white noise and the true error term of the model

In reality, finding additional predictors that can explain the remaining variation of the series is costly and time-consuming. Moreover, if variables such as ticket prices, economical indicators, and other factors are available, it is not easy to predict them (as they will be needed as inputs for the actual forecast). A simple solution is to utilize the auto-correlation relationship of the model residuals and model them with the ARIMA model. In other words, we can modify the `AirPassengers` linear regression and add *AR(p)* and *MA(p)* processes to the model:

$$Y_t = \beta_0 + \beta_1 trend_t + \beta_2 month_t + \beta_3 lag12_t + \sum_{i=1}^{p} \phi_i Y_{t-i} + \sum_{i=1}^{q} \theta_i \epsilon_{t-i} + \epsilon_t$$

The following terms are used in the preceding equation:

- $\sum_{i=1}^{p} \phi_i Y_{t-i}$ expresses an AR process with p-order
- $\sum_{i=1}^{q} \theta_i \epsilon_{t-i}$ expresses an MA process with q-order

The modified equation of the preceding `AirPassengers` model is a simplistic representation of the linear regression model with ARIMA errors (which in this case are ARMA errors). However, any of the ARIMA models we have looked at in this chapter, including the SARIMA model, can be used to model the residuals of the model.

Let's remodel the `AirPassengers` series, but this time with a linear regression model with ARIMA errors. While modeling the model's error term with an ARIMA model, we should treat the residuals as a series by itself and set the model components p, d, and q (and P, D, and Q if the residuals series has seasonal component) using any of the approaches we introduced in this chapter (for example, manual tuning, grid search, or automating the search process with the `auto.arima` process). For simplicity, we will utilize the `auto.arima` function.

The `auto.arima` function can be used to model a linear regression model with ARIMA errors via the use of the `xreg` argument:

```
md2 <- auto.arima(train,
                  xreg = cbind(model.matrix(~ month,train_df)[,-1],
                               train_df$trend,
                               train_df$lag12),
                  seasonal = TRUE,
                  stepwise = FALSE,
                  approximation = FALSE)
```

Note that the `auto.arima` and `arima` functions do not support categorical variables (that is, the `factor` class) with the `xreg` argument. Capturing the seasonal effect will require one-hot encoding (transforming each category into multiple binary variables) of the categorical variable. Therefore, we used the `model.matrix` function from the **stats** package on the `xreg` argument of the `auto.arima` function to transfer the `month` variable from a categorical variable into a binary variable. In addition, we dropped the first column, which represents the first category (in this case, the month of January), to avoid the dummy variable trap.

We set the `seasonal` argument to `TRUE` to include a search of both non-seasonal and seasonal models. The `auto.arima` function will conduct a full search when the step-wise and approximation arguments are set to `FALSE`. Let's use the `summary` function to review the model summary:

```
summary(md2)
```

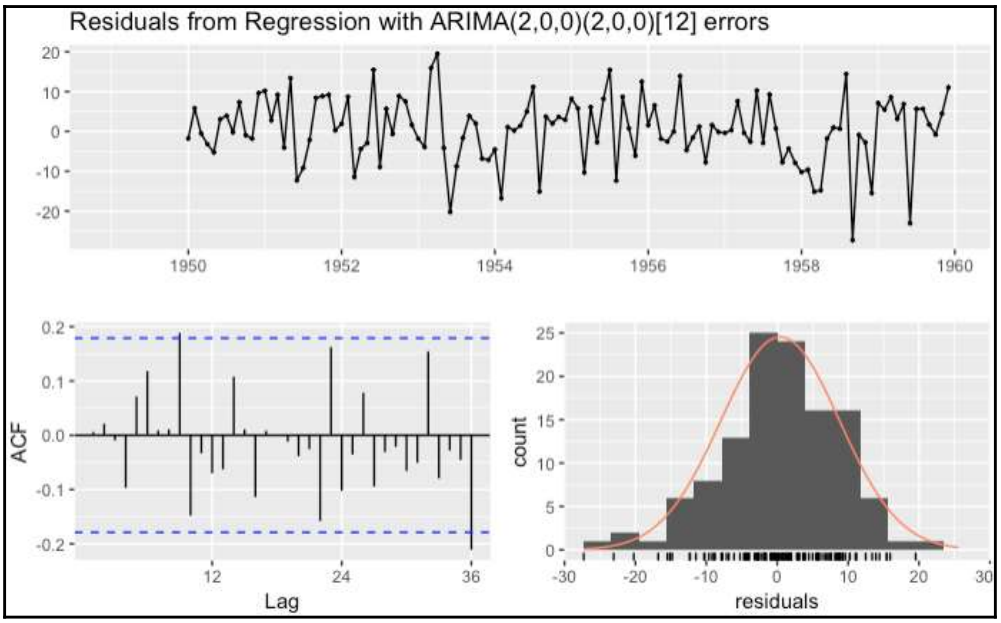
We get the following output:

```
## Series: train
## Regression with ARIMA(2,0,0)(2,0,0)[12] errors
##
## Coefficients:
##          ar1     ar2     sar1     sar2  monthFeb  monthMar  monthApr
##       0.5849  0.3056  -0.4421  -0.2063   -2.7523    0.8231    0.2066
## s.e.  0.0897  0.0903   0.1050   0.1060    1.8417    2.3248    2.4162
##       monthMay  monthJun  monthJul  monthAug  monthSep  monthOct
monthNov
##         2.8279    6.6057   11.2337   12.1909    3.8269    0.6350
-2.2723
## s.e.    2.5560    3.2916    4.2324    4.1198    2.9243    2.3405
2.4211
##       monthDec
##        -0.9918  0.2726  1.0244
## s.e.    1.9172  0.1367  0.0426
##
## sigma^2 estimated as 72.82:  log likelihood=-426.93
## AIC=889.86   AICc=896.63   BIC=940.04
##
## Training set error measures:
##                      ME     RMSE      MAE       MPE     MAPE      MASE
## Training set 0.4117147 8.353629 6.397538 0.2571543 2.549682 0.2100998
##                      ACF1
## Training set 0.005966714
```

The `auto.arima` function regresses the series with the `trend`, `month`, and `lag12` variables. In addition, the model used the AR(2) and SAR(2) processes for modeling the error term. We can now review the residuals of the modified `md2` model with the `checkresiduals` functions:

```
checkresiduals(md2)
```

We get the following output:



You can see the change in the ACF plot after applying the linear regression model with ARIMA errors as the majority of the lags are not statistically significant, as opposed to the preceding linear regression model (`md1`). Let's evaluate the performance of both models (`md1` and `md2`) on the testing set:

```
fc1 <- forecast(md1, newdata = test_df)

fc2 <- forecast(md2, xreg = cbind(model.matrix(~ month,test_df)[,-1],
                                  test_df$trend,
                                  test_df$lag12))
```

When using the `xreg` argument with one of the **forecast** package functions, the corresponding inputs for the forecasting model should be fed to the `xreg` argument of the `forecast` function. Note that you have to execute the input variables in the same order that was used in the training model with the `xreg` argument of the **forecast** package!

Now, let's use the accuracy function to review the two model's performance on both the testing and training partitions:

```
accuracy(fc1, test)
```

We get the following output:

```
##                                       ME     RMSE      MAE         MPE
## Training set -0.000000000000001658222 13.99206 11.47136 -0.1200023
## Test set     10.796107235775016519597 20.82782 18.57391  1.9530865
##                   MAPE      MASE      ACF1 Theil's U
## Training set 4.472154 0.3541758 0.7502578        NA
## Test set     3.828115 0.5734654 0.1653119 0.4052175
```

Let's test the accuracy of the second model:

```
accuracy(fc2, test)
```

We get the following output:

```
##                      ME      RMSE       MAE         MPE     MAPE
MASE
## Training set   0.4117147  8.353629  6.397538  0.2571543 2.549682
0.2100998
## Test set     -11.2123707 17.928195 13.353910 -2.4898843 2.924174
0.4385521
##                      ACF1 Theil's U
## Training set  0.005966714        NA
## Test set     -0.325044930 0.3923795
```

You can see from the output of the `accuracy` function that the linear regression with the ARIMA model provides a significant improvement in the model's accuracy with a lift of 43% on the MAPE score on the training set (from 4.5% to 2.5%) and 31% on the testing set (from 3.8% to 2.9%).

Summary

In this chapter, we introduced the ARIMA family of models, one of the core approaches for forecasting time series data. The main advantages of the ARIMA family of models is their flexibility and modularity, as they can handle both seasonal and non-seasonal time series data by adding or modifying the model components. In addition, we saw the applications of the ACF and PACF plots for identifying the type of process (for example, AR, MA, ARMA, and so on) and its order.

While it is essential to be familiar with the tuning process of ARIMA models, in practice, as the number series to be forecast increase, you may want to automate this process. The `auto.arima` function is one of the most common approaches in R to forecast with ARIMA models as it can scale up when dozens of series need to be forecast.

Last but not least, we saw applications of linear regression with the ARIMA errors model in order to extract lost information from the model residuals.

In the next chapter, we will look at an advanced application of the regression models with machine learning models for forecasting time series data.

12
Forecasting with Machine Learning Models

In `Chapter 9`, *Forecasting with Linear Regression*, we saw how a basic regression model could utilize some simple steps to create a robust time series forecast. The use of a linear regression model for time series forecasting can be easily generalized to other regression approaches, in particular, machine learning-based regressions. In this chapter, we will focus on the use of machine learning models for time series forecasting using the **h2o** package. This chapter requires some basic knowledge of the training and tuning process of machine learning models.

In this chapter, we will cover the following topics:

- Introduction to the **h2o** package and its functionality
- Feature engineering of time series data
- Forecasting with the Random Forest model
- Forecasting with the gradient boosting machine learning model
- Forecasting with the automate machine learning model

Technical requirement

The following packages will be used in this chapter:

- **h2o**: Version 3.22.1.1 and above and Java version 7 and above
- **TSstudio**: Version 0.1.4 and above
- **plotly**: Version 4.8 and above
- **lubridate**: Version 1.7.4 and above

You can access the codes for this chapter from the following link:

`https://github.com/PacktPublishing/Hands-On-Time-Series-Analysis-with-R/tree/master/Chapter12`

Why and when should we use machine learning?

In recent years, the use of **machine learning** (**ML**) models has become popular and accessible due to significant improvement in standard computation power. This led to a new world of methods and approaches for regression and classifications models. The process of creating time series forecasting with ML models follows the same process we used in `Chapter 9`, *Forecasting with Linear Regression*, with the linear regression model.

Before we start diving into the details, it is important to caveat the use of ML models in the context of time series forecasting:

- **Cost**: The use of ML models is typically more expensive than typical regression models, both in computing power and time.
- **Accuracy**: The ML model's performance is highly dependent on the quality (that is, strong casualty relationship with the dependent variable) of the predictors. It is likely that the ML models will over-perform, with respect to traditional methods such as linear regression when strong predictors are available.
- **Tuning**: Processing typical ML models is more complex than with common regression models, as those models have more tuning parameters and therefore, require some expertise.
- **Black-box**: Most of the ML models are considered black boxes, as it is hard to interrupt their output.
- **Uncertainty**: Generally, there is no straightforward method to quantify the forecasting uncertainty with confidence intervals like the traditional time series model does.

On the other hand, the main advantage of the ML models is their predictive power (when quality inputs are available), which, in many cases, is worth the effort and time involved in the process.

In the context of time series forecasting, it will be beneficial to forecast with ML models in the following cases:

- **Structural patterns**: Exits in the series, as those can produce new, predictive features
- **Multiple seasonality**: As a special case for structural patterns since, typically, the traditional time series models are struggling to capture those patterns when they exist

In this chapter, we will focus on the forecast of the US monthly vehicle sales using the following models:

- Random Forest
- Gradient boosting machine
- Auto ML model

Note that the goals of this chapter are not to teach you the principles of ML (as it is a topic by itself and outside the scope of this book), but the principles of building forecasting models with the use of ML methods. Some basic background in training and tuning ML models is recommended for this chapter.

Why h2o?

In this chapter, we will use the **h2o** package to build and train forecasting models with the use of ML models. H2O is an open source, distributed, and Java-based library for machine learning applications. It has APIs for both R (the **h2o** package) and Python, and includes applications for both supervised and unsupervised learning models. This includes algorithms such as **deep learning** (**DL**), **gradient boosting machine** (**GBM**), XGBoost, **Distributed Random Forest** (**RF**), and the **Generalized Linear Model** (**GLM**).

The main advantage of the **h2o** package is that it is based on distributed processing and, therefore, it can be either used in memory or scaled up with the use of external computing power. Furthermore, the **h2o** package algorithms provide several methods so that we can train and tune machine learning models, such as the cross-validation method and the built-in grid search function.

Since the **h2o** package is using the Java language on the backhand, it is required that you install Java on your machine before loading the package. Additional information about the installation process of **h2o** is available in the platform documentation (`http://docs.h2o.ai/h2o/latest-stable/h2o-docs/downloading.html`).

Forecasting monthly vehicle sales in the US – a case study

In this chapter, we will focus on forecasting the total monthly vehicle sales in the US in the next 12 months with ML methods. The total monthly vehicle sales in the US series is available in the **TSstudio** package:

```
library(TSstudio)

data(USVSales)
```

Before we start preparing the series and create new features, we will conduct a short exploratory analysis of the series in order to identify the main characteristics of the series.

Exploratory analysis of the USVSales series

In this section, we will focus on exploring and learning about the main characteristics of the series. These insights will be used to build new features as inputs for the ML model. The exploratory analysis of the USVSales series will focus on the following topics:

- View the time series structure (frequency, start and end of the series, and so on)
- Explore the series components (seasonal, cycle, trend, and random components)
- Seasonality analysis
- Correlation analysis

The series structure

Let's start with `ts_info` and review the structure of the `USVSales` series:

```
ts_info(USVSales)
```

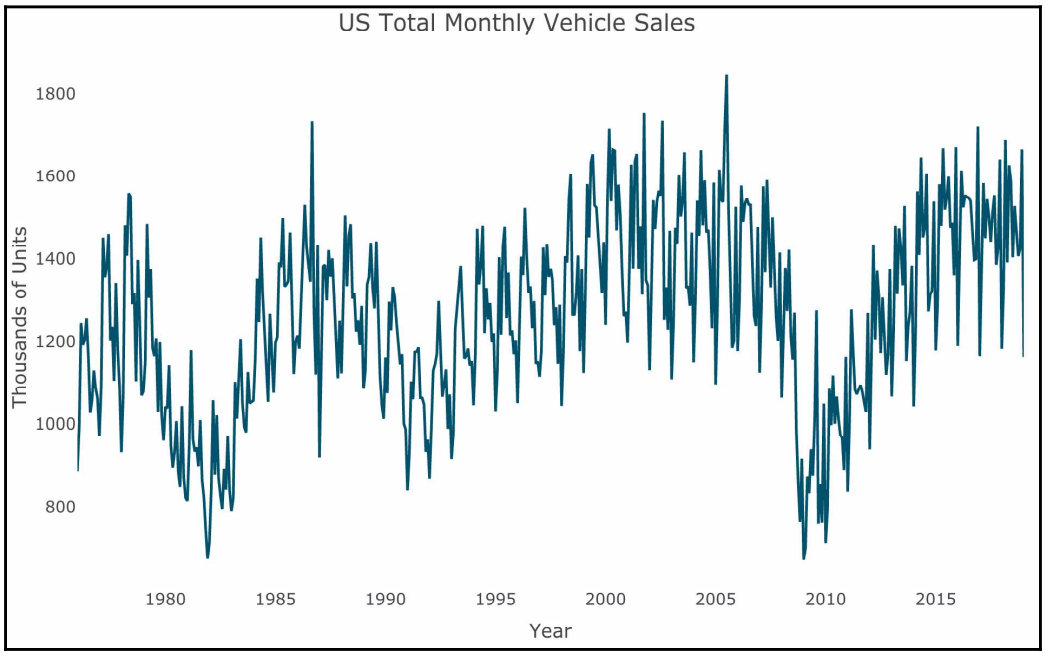
We get the following output:

```
##  The USVSales series is a ts object with 1 variable and 517 observations
##  Frequency: 12
##  Start time: 1976 1
##  End time: 2019 1
```

The `USVSales` series is a monthly `ts` object which represents the total vehicle sales in the US between 1976 and 2018 in thousands of units. Let's plot the series and review its structure with the `ts_plot` function:

```
ts_plot(USVSales,
        title = "US Total Monthly Vehicle Sales",
        Ytitle = "Thousands of Units",
        Xtitle = "Year")
```

We get the following output:



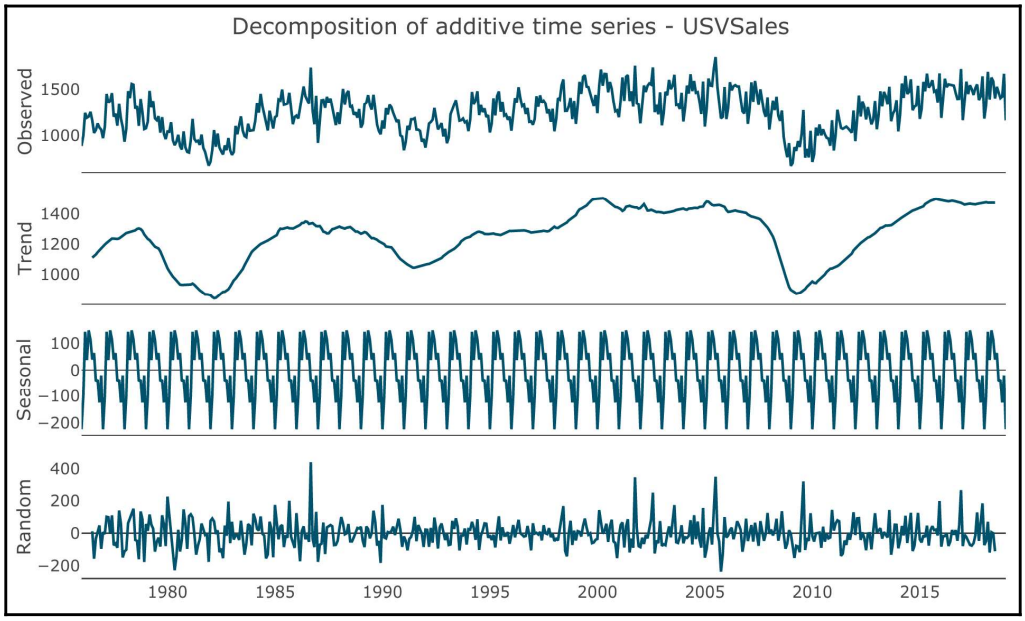
As you can see in the preceding plot, the series has cycle patterns, which is common for a macro economy indicator. In this case, it is a macro indicator of the US economy.

The series components

We can get a deeper view of the series components by decomposing the series into its components and plotting them with the `ts_decompose` function:

```
ts_decompose(USVSales)
```

We get the following output:



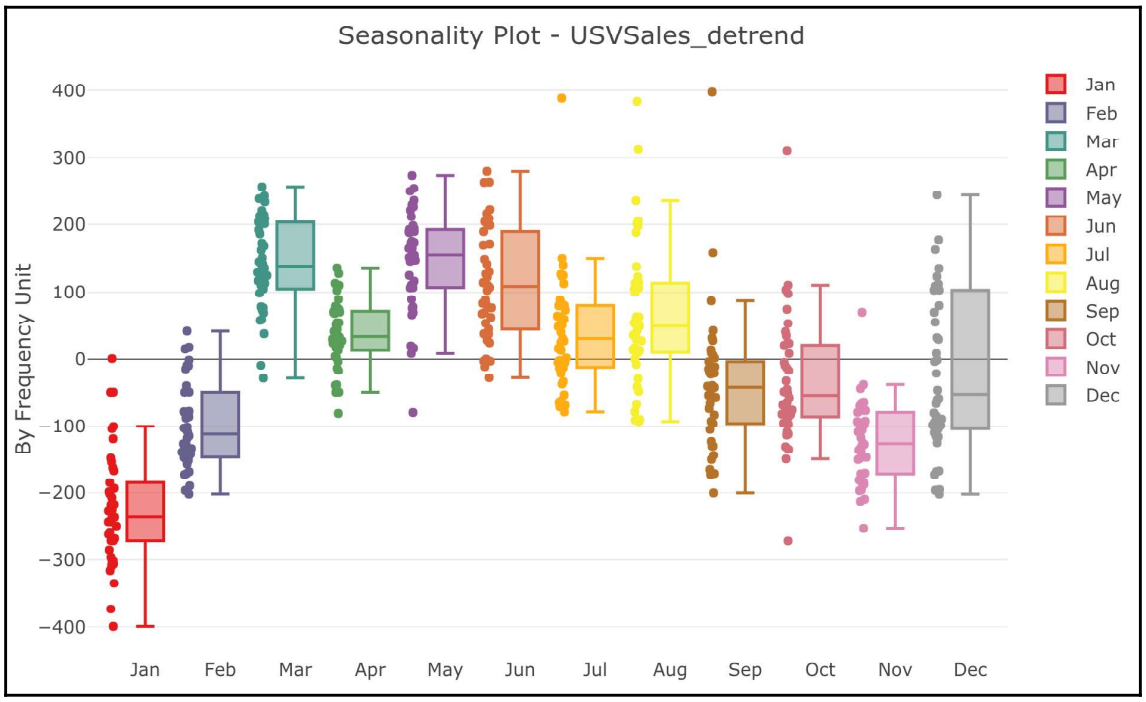
Beside the cycle-trend component, we can observe that the plot has a strong seasonal pattern, which we will explore next.

Seasonal analysis

To get a closer look at the seasonal component of the series, we will subtract from the series, decompose the trend we discussed previously, and use the `ts_seasonal` function to plot the box plot of the seasonal component of the detrend series:

```
USVSales_detrend <- USVSales - decompose(USVSales)$trend

ts_seasonal(USVSales_detrend, type = "box")
```

We get the following output:



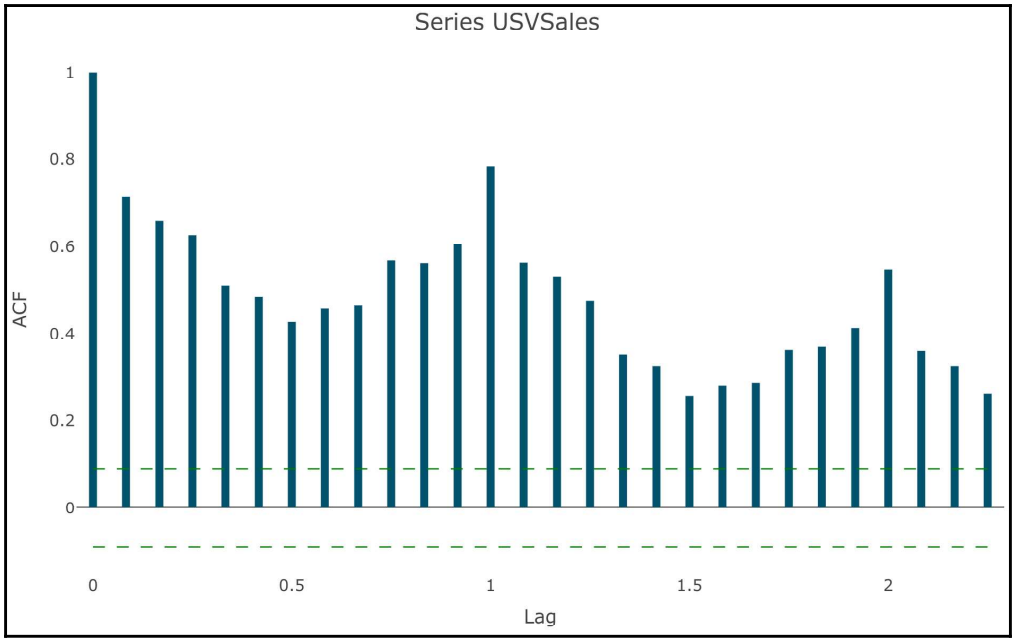
We can see from the preceding seasonal plot that, typically, the peak of the year occurred during the months of March, May, and June. In addition, you can see that the sales decay from the summer months and peak again in December during the holiday seasons. On the other hand, the month of January is typically the lowest month of the year in terms of sales.

Correlation analysis

As we saw in `Chapter 7`, *Correlation Analysis,* the `USVSales` series has a high correlation with its first seasonal lag. We can review this assessment with the use of the `ts_acf` function from the **TSstudio** package for reviewing the autocorrelation of the series:

```
ts_acf(USVSales)
```

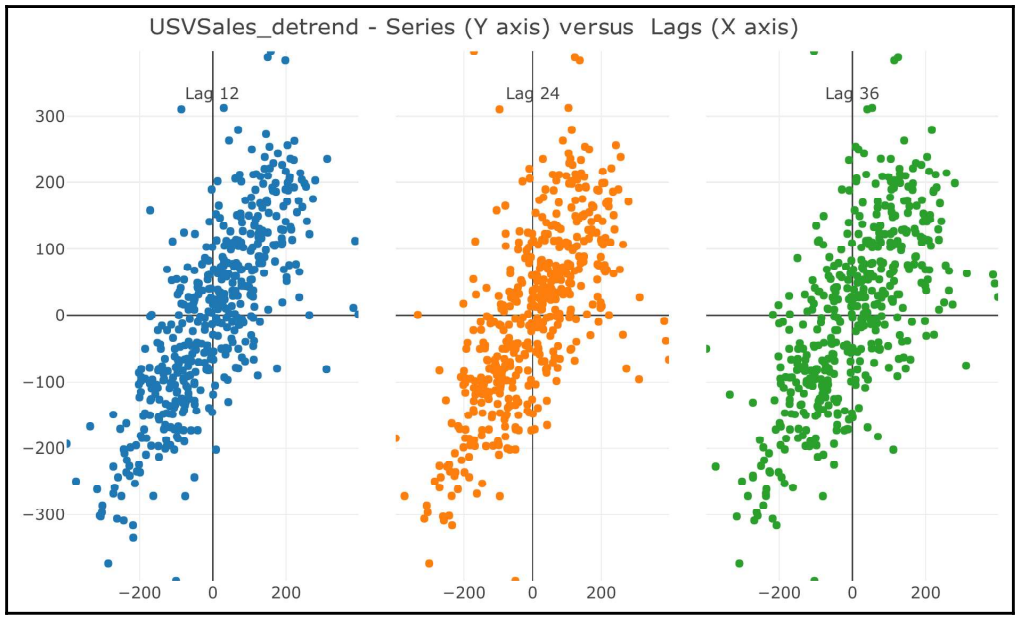
We get the following output:



We can zoom in on the relationship of the series with the last three seasonal lags using the `ts_lags` function:

```
ts_lags(USVSales, lags = c(12, 24, 36))
```

We get the following output:



The relationship of the series with the first and also second seasonal lags has a strong linear relationship, as shown in the preceding lags plot.

Exploratory analysis – key findings

We can conclude our short exploratory analysis of the `USVSales` series with the following observations:

- The `USVSales` series is a monthly series with a clear monthly seasonality
- The series trend has a cyclic shape, and so the series has a cycle component embedded in the trend
- The series' most recent cycle starts right after the end of the 2008 economic crisis, between 2009 and 2010
- It seems that the current cycle reached its peak as the trend starts to flatten out
- The series has a strong correlation with its first seasonal lag

Moreover, as we intend to have a short-term forecast (of 12 months), there is no point in using the full series, as it may enter some noise into the model due to the change of the trend direction every couple of years. (If you were trying to create a long-term forecast, then it may be a good idea to use all or most of the series.) Therefore, we will use the model training observations from 2010 and onward. We will use the `ts_to_prophet` function from the **TSstudio** package to transform the series from a `ts` object into a `data.frame`, and the `window` function to subset the series observations since January 2010:

```
df <- ts_to_prophet(window(USVSales, start = c(2010,1)))

names(df) <-  c("date", "y")

head(df)
```

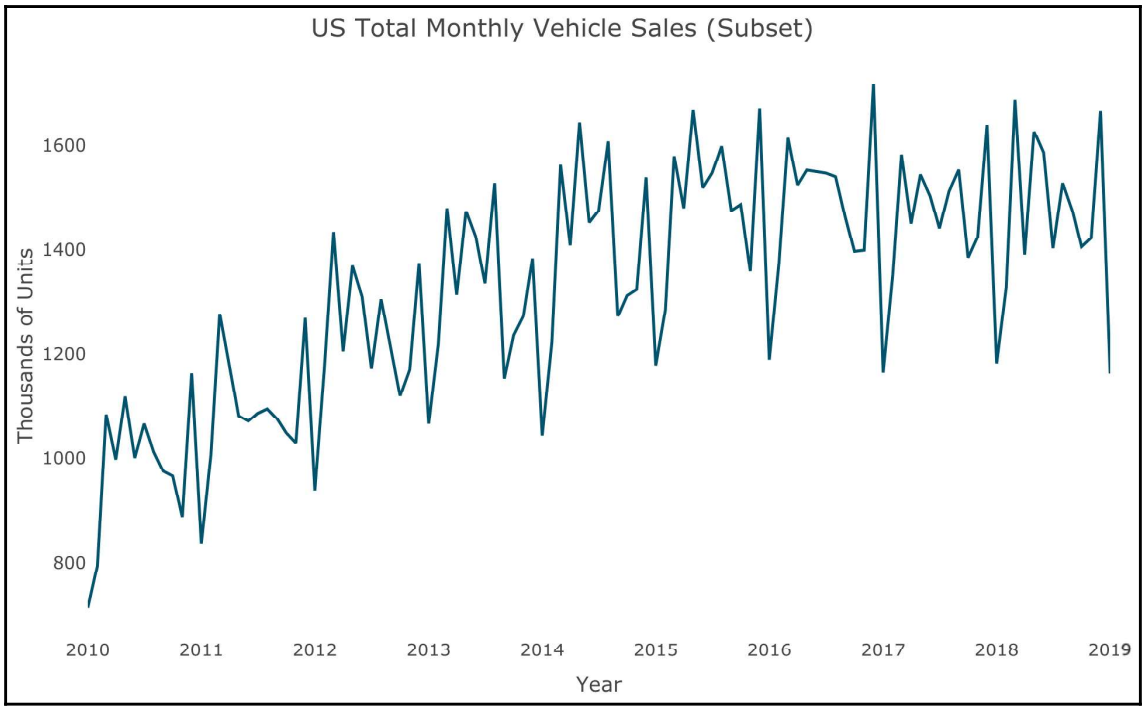
We get the following output:

```
##         date        y
## 1 2010-01-01  712.469
## 2 2010-02-01  793.362
## 3 2010-03-01 1083.953
## 4 2010-04-01  997.334
## 5 2010-05-01 1117.570
## 6 2010-06-01 1000.455
```

Before we move forward and start with the feature engineering stage, let's plot and review the subset series of `USVSales` with the `ts_plot` function:

```
ts_plot(df,
        title = "US Total Monthly Vehicle Sales (Subset)",
        Ytitle = "Thousands of Units",
        Xtitle = "Year")
```

We get the following output:



Feature engineering

Feature engineering plays a pivotal role when modeling with ML algorithms. Our next step, based on the preceding observations, is to create new features that can be used as informative input for the model. In the context of time series forecasting, here are some examples of possible new features that can be created from the series itself:

- **The series trend**: This uses a numeric index. In addition, as the series trend isn't linear, we will use a second polynomial of the index to capture the overall curvature of the series trend.
- **Seasonal component**: This creates a categorical variable for the month of the year to capture the series' seasonality.
- **Series correlation**: This utilizes the strong correlation of the series with its seasonal lag and uses the seasonal lag (`lag12`) as an input to the model.

We will use the **dplyr** and **lubridate** packages to create those features, as we can see in the following code:

```
library(dplyr)

library(lubridate)

df <- df %>% mutate(month = factor(month(date, label = TRUE), ordered =
FALSE),
                    lag12 = lag(y, n = 12)) %>%
             filter(!is.na(lag12))
```

We will then add the `trend` component and its second polynomial (trend squared):

```
df$trend <- 1:nrow(df)

df$trend_sqr <- df$trend ^ 2
```

Let's view the structure of the `df` object after adding the new features:

```
str(df)
```

We get the following output:

```
## 'data.frame':    97 obs. of  6 variables:
##  $ date     : Date, format: "2011-01-01" "2011-02-01" ...
##  $ y        : num  836 1007 1277 1174 1081 ...
##  $ month    : Factor w/ 12 levels "Jan","Feb","Mar",..: 1 2 3 4 5 6 7 8
9 10 ...
##  $ lag12    : num  712 793 1084 997 1118 ...
##  $ trend    : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ trend_sqr: num  1 4 9 16 25 36 49 64 81 100 ...
```

There are additional feature engineering steps, which in the case of the `USVSales` series are not required, but may be required in other cases:

- **Scaling**: This may be required when the numeric inputs of the regression (either the series itself or at least one of the regression numeric predictors) are on a different numeric scale, for instance, regress revenue in billions of USD with a binary flag (for example, 0 or 1). Some models may fail to find coefficients or won't identify the true relationship between the variable. Some common scaling methods are as follows:
    - Log transformation
    - Normalize the series with Z-score
    - Scale the input between 0 and 1
    - Subtract the mean from the input

- **Hot encoding**: Categorical variables may be required when the algorithm does not support the use of factors (categorical variables) as input. In this case, for a variable with *N* categories, we will create new *N+1* variables—one variable for capturing the slope and *N* binary variables and one for each category.

Since the values of the input series are ranging between 800 to 1,700, there is no need to scale the series and its inputs. In addition, since the **h2o** package supports the inputs as R factors, there is no need to apply hot encoding.

Training, testing, and model evaluation

In order to compare the different models that we will be testing in this chapter, we will use the same inputs that we used previously. This includes executing same training and testing partitions throughout this chapter.

Since our forecast horizon is for 12 months, we will leave the last 12 months of the series as testing partitions and use the rest of the series as a training partition:

```
h <- 12

train_df <- df[1:(nrow(df) - h), ]

test_df <- df[(nrow(df) - h + 1):nrow(df), ]
```

Previously, the `h` variable represented the forecast horizon, which, in this case, is also equal to the length of the testing partition. We will evaluate the model's performance based on the MAPE score on the testing partition.

One of the main characteristics of ML models is the tendency to overfit on the training set. Therefore, you should expect that the ratio between the error score on the testing and training partition will be relatively larger than the corresponding results of traditional time series models, such as ARIMA, Holt-Winters, and time series linear regression.

In addition to the training and testing partitions, we need to create the inputs for the forecast itself. We will create a `data.frame` with the dates of the following 12 months and build the rest of the features:

```
forecast_df <- data.frame(date = seq.Date(from = max(df$date) + month(1),
length.out = h, by = "month"),
                          trend = seq(from = max(df$trend) + 1, length.out
= h, by = 1))

forecast_df$trend_sqr <- forecast_df$trend ^ 2
```

```
forecast_df$month <- factor(month(forecast_df$date, label = TRUE), ordered
= FALSE)
```

Last but not least, we will extract the last `12` observations of the series from the `df` object and assign them as the future lags of the series:

```
forecast_df$lag12 <- tail(df$y, 12)
```

Model benchmark

Introduced in `Chapter 8`, *Forecasting Strategies*, the performance of a forecasting model should be measured by the error rate, mainly on the testing partition, but also on the training partition. You should evaluate the performance of the model with respect to some baseline model. In the previous chapters, we benchmarked the forecast of the `USgas` series with the use of the seasonal naive model.

In this chapter, since we are using a family of ML regression models, it makes more sense to use a regression model as a benchmark for the ML models. Therefore, we will train a time series linear regression model, which we introduced in `Chapter 9`, F*orecasting with Linear Regression*. Using the training and testing partitions we created previously, let's train the linear regression model and evaluate its performance with the testing partitions:

```
lr <- lm(y ~ month + lag12 + trend + trend_sqr, data = train_df)
```

We will use the `summary` function to review the model details:

```
summary(lr)
```

We get the following output:

```
##
## Call:
## lm(formula = y ~ month + lag12 + trend + trend_sqr, data = train_df)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -152.238  -38.412    1.175   34.471  119.947
##
## Coefficients:
##              Estimate Std. Error t value      Pr(>|t|)
## (Intercept) 533.54515   80.45593   6.632 0.00000000583 ***
## monthFeb    115.10262   36.87217   3.122       0.002614 **
## monthMar    287.20432   60.75992   4.727 0.00001142840 ***
## monthApr    189.81240   47.11880   4.028       0.000141 ***
## monthMay    258.85449   57.40489   4.509 0.00002550309 ***
```

```
## monthJun    212.14346   48.68294   4.358 0.00004414785 ***
## monthJul    181.74136   46.72620   3.889      0.000226 ***
## monthAug    239.95562   52.11112   4.605 0.00001797442 ***
## monthSep    148.28026   38.56246   3.845      0.000263 ***
## monthOct    118.08867   37.55868   3.144      0.002444 **
## monthNov    120.73354   36.01454   3.352      0.001295 **
## monthDec    267.91309   55.47217   4.830 0.00000777550 ***
## lag12         0.34120    0.11972   2.850      0.005740 **
## trend         8.68109    1.88781   4.598 0.00001838988 ***
## trend_sqr    -0.06869    0.01450  -4.738 0.00001096631 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 60.15 on 70 degrees of freedom
## Multiple R-squared:  0.9213, Adjusted R-squared:  0.9056
## F-statistic: 58.54 on 14 and 70 DF,  p-value: < 0.00000000000000022
```

Next, we will predict the corresponding values of the series on the testing partition with the `predict` function by using `test_df` as input:

```
test_df$yhat <- predict(lr, newdata = test_df)
```

Now, we can evaluate the model's performance on the testing partition:

```
mape_lr <- mean(abs(test_df$y - test_df$yhat) / test_df$y)

mape_lr
```

We get the following output:

```
## [1] 0.04041179
```

Hence, the MAPE score of the linear regression forecasting model is 4.04%. We will use this to benchmark the performance of the ML models.

Starting a h2o cluster

The **h2o** package is based on the use of distributed and parallel computing in order to speed up the compute time and be able to scale up for big data. All of this is done on either in-memory (based on the computer's internal RAM) or parallel distributed processing (for example, AWS, Google Cloud, and so on) clusters. Therefore, we will load the package and then set the in-memory cluster with the `h2o.init` function:

```
library(h2o)

h2o.init(max_mem_size = "16G")
```

We get the following output:

```
##
##
## Starting H2O JVM and connecting: ..... Connection successful!
##
## R is connected to the H2O cluster:
##     H2O cluster uptime:         5 seconds 370 milliseconds
##     H2O cluster timezone:       America/Los_Angeles
##     H2O data parsing timezone:  UTC
##     H2O cluster version:        3.22.1.1
##     H2O cluster version age:    4 months and 7 days !!!
##     H2O cluster name:           H2O_started_from_R_rami_izi146
##     H2O cluster total nodes:    1
##     H2O cluster total memory:   14.22 GB
##     H2O cluster total cores:    8
##     H2O cluster allowed cores:  8
##     H2O cluster healthy:        TRUE
##     H2O Connection ip:          localhost
##     H2O Connection port:        54321
##     H2O Connection proxy:       NA
##     H2O Internal Security:      FALSE
##     H2O API Extensions:         XGBoost, Algos, AutoML, Core V3, Core V4
##     R Version:                  R version 3.4.4 Patched (2018-03-19
r74624)
```

`h2o.init` allows you to set the memory size of the cluster with the `max_mem_size` argument. The output of the function, as shown in the preceding code, provides information about the cluster's setup.

Any data that is used throughout the training and testing process of the models by the **h2o** package must load to the cluster itself. The `as.h2o` function allows us to transform any `data.frame` object into a `h2o` cluster:

```
train_h <- as.h2o(train_df)

test_h <- as.h2o(test_df)
```

In addition, we will transform the `forecast_df` object (the future values of the series inputs) into an `h2o` object, which will be used to generate, later on in this chapter, the final forecast:

```
forecast_h <- as.h2o(forecast_df)
```

For our convenience, we will label the names of the dependent and independent variables:

```
x <- c("month", "lag12", "trend", "trend_sqr")

y <- "y"
```

Now that the data has been loaded into the working cluster, we can start the training process.

Training an ML model

The **h2o** package provides a set of tools for training and testing ML models. The most common model training approaches are as follows:

- **Training/testing**: This is based on splitting the input data into training and testing partitions by allocating most of the input data to the training partition and leaving the rest to the testing partition. As the names of the partitions imply, the training set is used to train the model, while the testing partition is used to test its performance on new data. Typical allocations are 70/30 (that is, 70% of the data to the training partition and 30% to the testing partition), or roughly close to that ratio, where the data allocation between the two partitions must be random.
- **Training/testing/validation**: This is relatively similar to the previous approach, except with added validation partitions. The validation partition is used during the training process to evaluate the tuning of the model parameters. The tuned models are then tested on the testing partition.
- **Cross-validation**: This is one of the most popular training methods for ML models as it reduces the chance of overfitting of the model. This method is based on the following steps:
    1. Splitting the training set, randomly, into *K* folders
    2. Training the model *K* times, each time leaving a different folder out as a testing partition, and training the model with the remaining *K-1* folders
    3. Throughout the training process, the model tunes the model's parameters
    4. The final model is tested on the testing partition

Throughout this chapter, we will use the **cross-validation** (**CV**) approach to train these models.

Forecasting with the Random Forest model

Now that we have prepared the data, created new features, and launched a `h2o` cluster, we are ready to build our first forecasting model with the **Random Forest** (**RF**) algorithm. The RF algorithm is one of the most popular ML models, and it can be used for both classification and regression problems. In a nutshell, the RF algorithm is based on an ensemble of multiple tree models.

As its name implies, it has two main components:

- **Random**: The input for each tree model is based on a random sample, along with the replacement of both the columns and rows of the input data. This method is also known as bagging.
- **Forest**: The collection of tree-based models, which eventually creates the *forest*.

After the forest is built, the algorithm ensembles the prediction of all the trees in the forest into one output. This combination of randomizing the input for each tree model and then averaging their results reduces the likelihood of overfitting the model.

RF has several tuning parameters that allow you to control the level of randomization of the sampling process and how deep the forest is. The `h2o.randomForest` function from the **h2o** package provides the framework for training and tuning the RF model. The following are the main parameters of the `h2o.randomForest` function:

- `x`: A vector of characters with independent variable names
- `y`: A string, which is the name of the dependent variable
- `training_frame`: The input data must be a `data.frame` of **h20** package
- `validation_frame`: Optional; should be used when applying a training approach with the validation set
- `nfolds`: The number of *K*-folds for the CV process
- `ntrees`: Number of trees to be created
- `sample_rate`: Defines the number of rows to sample per tree
- `max_depth`: Sets the maximum tree depth (number of nodes); as the trees grow deeper, the complexity of the mode increases, as well as the risk for overfitting
- `seed`: Sets the `seed` value of the random numbers generator

In addition, this function has several control arguments, which allows you to control the running time of the model. Furthermore, they allow you to set a stop criteria for the model if adding additional trees doesn't improve the model's performance. These arguments are as follows:

- `stopping_metric`: Defines the type of error metric to be used for evaluating whether the model should stop and build new trees as there is no additional improvement
- `stopping_tolerance`: Sets the minimal improvement that is required to continue the training process
- `stopping_rounds`: Sets the number of rounds that should be used before stopping the training

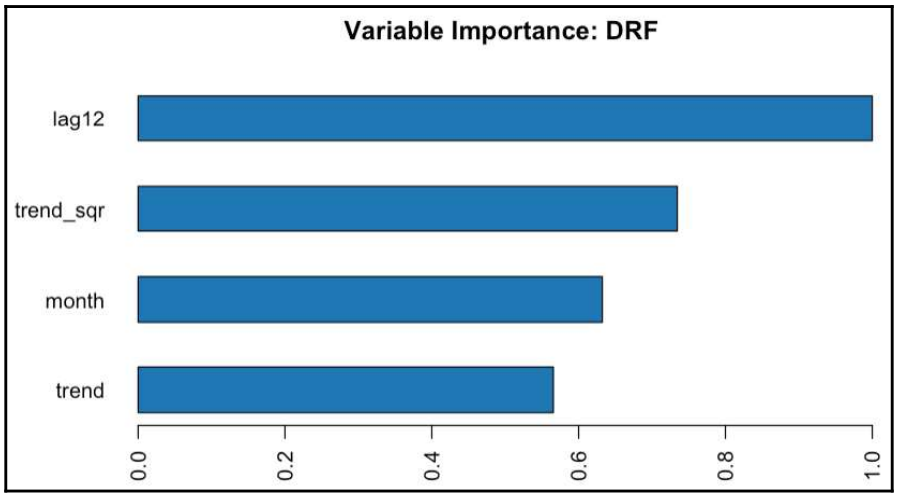
We will start with a simplistic RF model by using `500` trees and `5` folder CV training. In addition, we will add a stop criteria to prevent the model from fitting the model while there is no significant change in the model's performance. In this case, we will set the stopping metric as RMSE, the stopping tolerance as 0.0001, and the stopping rounds to `10`:

```
rf_md <- h2o.randomForest(training_frame = train_h,
                          nfolds = 5,
                          x = x,
                          y = y,
                          ntrees = 500,
                          stopping_rounds = 10,
                          stopping_metric = "RMSE",
                          score_each_iteration = TRUE,
                          stopping_tolerance = 0.0001,
                          seed = 1234)
```

The `h2o.randomForest` function returns an object with information about the parameter settings of the model and its performance on the training set (and validation, if used). We can view the contribution of the model inputs with the `h2o.varimp_plot` function. This function returns a plot with the ranking of the input variables' contribution to the model performance using a scale between 0 and 1, as shown in the following code:

```
h2o.varimp_plot(rf_md)
```

We get the following output:



As we can see from the preceding variable importance plot, the lag variable, `lag12`, is the most important to the model. This shouldn't be a surprise as we saw the strong relationship between the series and its seasonal lag in the correlation analysis. Right after this, the most important variables are `trend_sqr`, `month`, and `trend`.

The output of the model contains (besides the model itself) information about the model's performance and parameters. Let's review the model summary:

```
rf_md@model$model_summary
```

We get the following output:

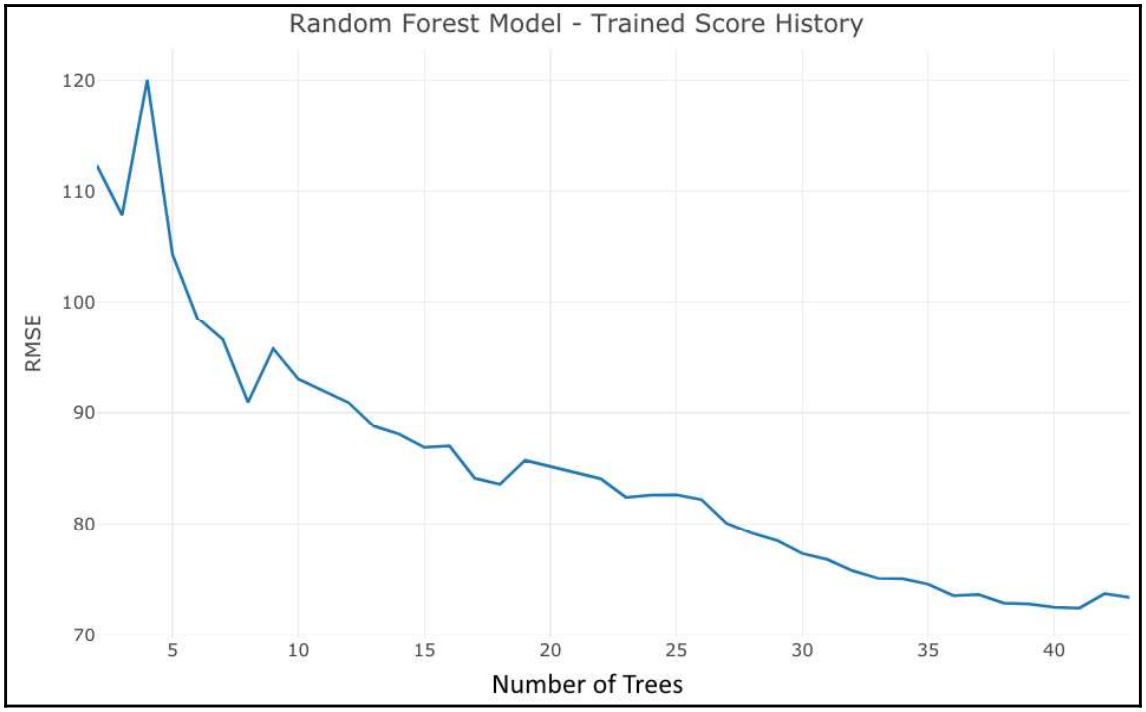
```
## Model Summary:
##   number_of_trees number_of_internal_trees model_size_in_bytes min_depth
## 1              42                        42               29398         7
##   max_depth mean_depth min_leaves max_leaves mean_leaves
## 1        13    9.80952         43         61    51.11905
```

We can see that we utilized only 42 trees out of the 500 that were set by the `ntrees` argument. This is as a result of the stopping parameters that were used on the model. The following plot demonstrates the learning process of the model as a function of the number of trees:

```
library(plotly)

tree_score <- rf_md@model$scoring_history$training_rmse
```

```
plot_ly(x = seq_along(tree_score), y = tree_score,
        type = "scatter", mode = "line") %>%
   layout(title = "The Trained Model Score History",
          yaxis = list(title = "RMSE"),
          xaxis = list(title = "Num. of Trees"))
```

We get the following output:



Last but not least, let's measure the model's performance on the testing partition. We will use the `h2o.predict` function to predict the corresponding values of the series on the testing partition:

```
test_h$pred_rf <- h2o.predict(rf_md, test_h)
```

Next, we will transfer the `h2o` data frame to a `data.frame` object with the `as.data.frame` function:

```
test_1 <- as.data.frame(test_h)
```

Now, we can calculate the MAPE score of the RF model on the test partition:

```
mape_rf <- mean(abs(test_1$y - test_1$pred_rf) / test_1$y)

mape_rf
```

We get the following output:

```
## [1] 0.03749611
```

As you can see from the model error score, the RF model with its default settings was able to achieve a lower error rate than our benchmark model, that is, the linear regression model, with a MAPE score of 3.7% as opposed to 4%.

Generally, when using the default option for the model's parameters, the model may perform well, but there might be some room left for additional optimization and improvement in regards to the performance of the model. There are a variety of techniques for model optimization and tuning parameters, such as manual tuning, grid search, and algorithm-based tuning. In the previous chapters, we looked at examples of these three approaches—we manually tuned the ARMA model in `Chapter 11`, *Forecasting with ARIMA Models*, we used the `ts_grid` function in `Chapter 10`, *Forecasting with Exponential Smoothing Models*, for tuning the Holt-Winters parameters with grid search, and automated the tuning of ARIMA model (again in `Chapter 11`, *Forecasting with ARIMA Models*) with the `auto.arima` algorithm.

We will demonstrate a grid search approach for tuning an ML model with the `h2o.grid` function. Later on in this chapter, we will look at an algorithm-based approach to tuning aN ML model with the `h2o.automl` function.

The `h2o.grid` function allows you to set a set of values for some selected parameters and test their performance on the model in order to identify the tuning parameters that optimize the model's performance. The main arguments of this function are as follows:

- `algorithm`: Defines the type of algorithm to use in the grid search
- `hyper_params`: Sets the search parameter's values
- `search_criteria`: Sets roles for the search, such as the stopping metric of the model, number of models, and runtime

Any of the training approaches of the **h2o** package (such as CV, training with validation) can be applied with the `h2o.grid` function.

We will start by setting the search parameters:

```
hyper_params_rf <-  list(mtries = c(2, 3, 4),
                         sample_rate = c(0.632, 0.8, 0.95),
                         col_sample_rate_per_tree = c(0.5, 0.9, 1.0),
                         max_depth = c(seq(1, 30, 3)),
                         min_rows = c(1, 2, 5, 10))
```

Here, the parameters we selected are as follows:

- `mtries`: Defines the columns to randomly select on each node of the tree
- `sample_rate`: Sets the row sampling for each tree
- `col_sample_rate_per_tree`: Defines the column sample rate per tree
- `max_depth`: Specifies the maximum tree depth
- `min_rows`: Sets the minimum number of observations for a leaf

The more parameters you add or define for a wide range of values, the larger the possible search combination. This could easily get to hundreds of combinations, which may run for a couple of hours (based on the available compute power). Therefore, setting the `search_criteria` argument is very important for efficiency reasons. The strategy argument of the `search_criteria` argument allows you to set either Cartesian (`Cartesian`) search or random (`RandomDiscrete`) search. The Cartesian search iterates and computes the models for all the possible search options in chronological order. On the other hand, the random search randomly selects a search combination from the grid search table.

It would make sense to use the Cartesian method when the number of search combinations is fairly small, or when you aren't limited in search time. On the other hand, for a large amount of search combinations, or when we have a time constraint, it is recommended to use random search. For efficiency reasons, we will set a random search and restrict the search time to 20 minutes with `max_runtime_sec`. We will use the same stopping metric that we used previously:

```
search_criteria_rf <- list(strategy = "RandomDiscrete",
                           stopping_metric = "rmse",
                           stopping_tolerance = 0.0001,
                           stopping_rounds = 10,
                           max_runtime_secs = 60 * 20)
```

After we set the search arguments for the `h2o.grid` function, we can start the search:

```
rf2 <- h2o.grid(algorithm = "randomForest",
                search_criteria = search_criteria_rf,
                hyper_params = hyper_params_rf,
```

```
                x = x,
                y = y,
                training_frame = train_h,
                ntrees = 5000,
                nfolds = 5,
                grid_id = "rf_grid",
                seed = 1234)
```

You will notice that we kept the same training approach, that is, `5` folders CV and the number of trees as `5000`.

Setting a large number of trees with a tree-based model such as RF or GBM, along with a stopping metric, will ensure that the model will keep building additional trees until it meets the stopping criteria. Therefore, setting the stopping criteria plays a critical roll in both the efficiency of the model and its results.

We will now extract the grid results, sort the models by their RMSE score, and pull the lead model:

```
rf2_grid_search <- h2o.getGrid(grid_id = "rf_grid",
                               sort_by = "rmse",
                               decreasing = FALSE)

rf_grid_model <- h2o.getModel(rf2_grid_search@model_ids[[1]])
```

Let's test the model on the testing partition and evaluate its performance:

```
test_h$rf_grid  <- h2o.predict(rf_grid_model, test_h)

mape_rf2 <- mean(abs(test_1$y - test_1$rf_grid) / test_1$y)

mape_rf2
```

We get the following output:

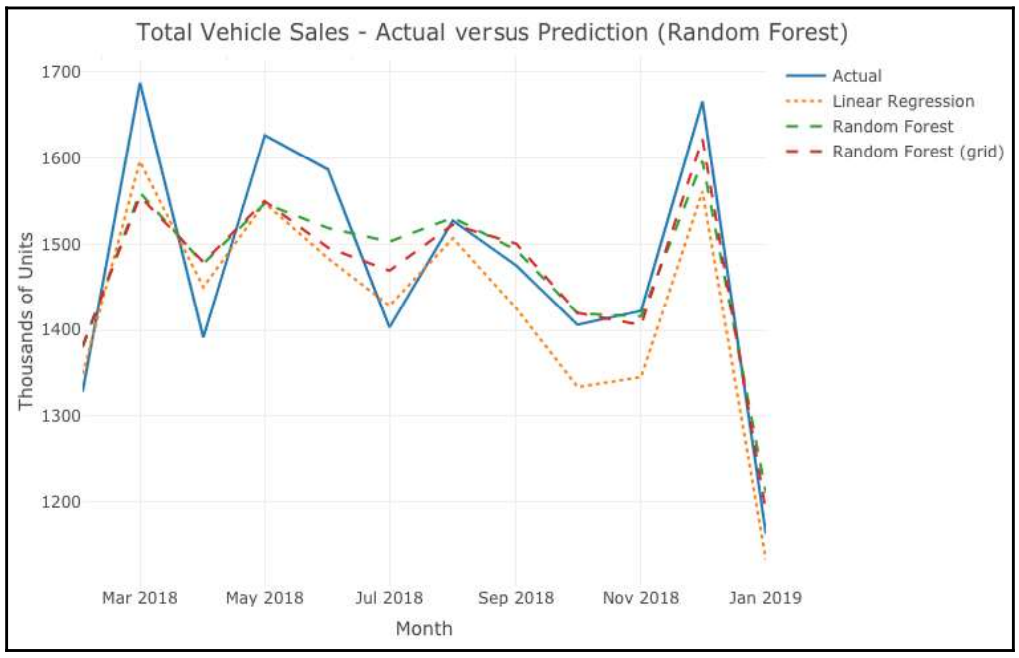
```
## [1] 0.03414109
```

The additional optimization step contributed to the lift in the model's accuracy, with a MAPE score of 3.33% compared to 3.7% and 4% for the first RF model we trained, and the linear regression model, respectively. The following plot provides an additional view of the model's performance:

```
plot_ly(data = test_1) %>%
  add_lines(x = ~ date, y = ~y, name = "Actual") %>%
  add_lines(x = ~ date, y = ~ yhat, name = "Linear Regression", line =
list(dash = "dot")) %>%
```

```
  add_lines(x = ~ date, y = ~ pred_rf, name = "Random Forest", line =
list(dash = "dash")) %>%
  add_lines(x = ~ date, y = ~ rf_grid, name = "Random Forest (grid)", line
= list(dash = "dash")) %>%
  layout(title = "Total Vehicle Sales - Actual vs. Prediction (Random
Forest)",
         yaxis = list(title = "Thousands of Units"),
         xaxis = list(title = "Month"))
```

We get the following output:



Now, let's look at the GBM model.

Forecasting with the GBM model

The GBM algorithm is another ensemble and tree-based model. It uses the *boosting* approach in order to train different subsets of the data, and repeats the training of subsets that the model had with a high error rate. This allows the model to learn from past mistakes and improve the predictive power of the model.

The main arguments of the GBM model are as follows

- `x`: A vector of characters with the independent variables names.
- `y`: A string, which is the name of the dependent variable.
- `training_frame`: The input data must be a `data.frame` of **h20** package.
- `validation_frame`: Optional; this should be used when we're applying a training approach with the validation set.
- `nfolds`: The number of *K*-fold for the CV process.
- `ntrees`: Number of trees to be created.
- `sample_rate`: Defines the number of rows to sample per tree.
- `max_depth`: Sets the maximum tree depth (number of nodes). As the trees grow deeper, the complexity of the mode increases, as well as the risk of overfitting.
- `learn_rate`: Defines the learning rate of the model with values between 0 and 1. The default rate is 0.1. As the learning rate comes closer to 0, the better the learning of the model will be, with the cost of longer compute time, along with a higher number of trees.
- `seed`: Sets the `seed` value of the random numbers generator.

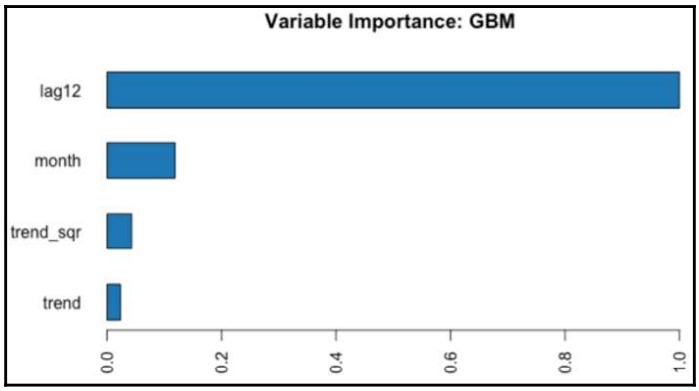
The following example demonstrates the use of the `h2o.gbm` function for training the GBM model with the same input data we used previously:

```
gbm_md <- h2o.gbm(
   training_frame = train_h,
   nfolds = 5,
   x = x,
   y = y,
   max_depth = 20,
   distribution = "gaussian",
   ntrees = 500,
   learn_rate = 0.1,
   score_each_iteration = TRUE
 )
```

Similar to the RF model, we can review the rank of the importance of the model's variables with the `h2o.varimp_plot` function:

```
h2o.varimp_plot(gbm_md)
```

We get the following output:



For RF, the GBM model ranks the **lag12** variable as the most important to the model. Let's test the model's performance on the testing set:

```
test_h$pred_gbm  <- h2o.predict(gbm_md, test_h)

test_1 <- as.data.frame(test_h)

mape_gbm <- mean(abs(test_1$y - test_1$pred_gbm) / test_1$y)

mape_gbm
```

We get the following output:

```
## [1] 0.02735553
```

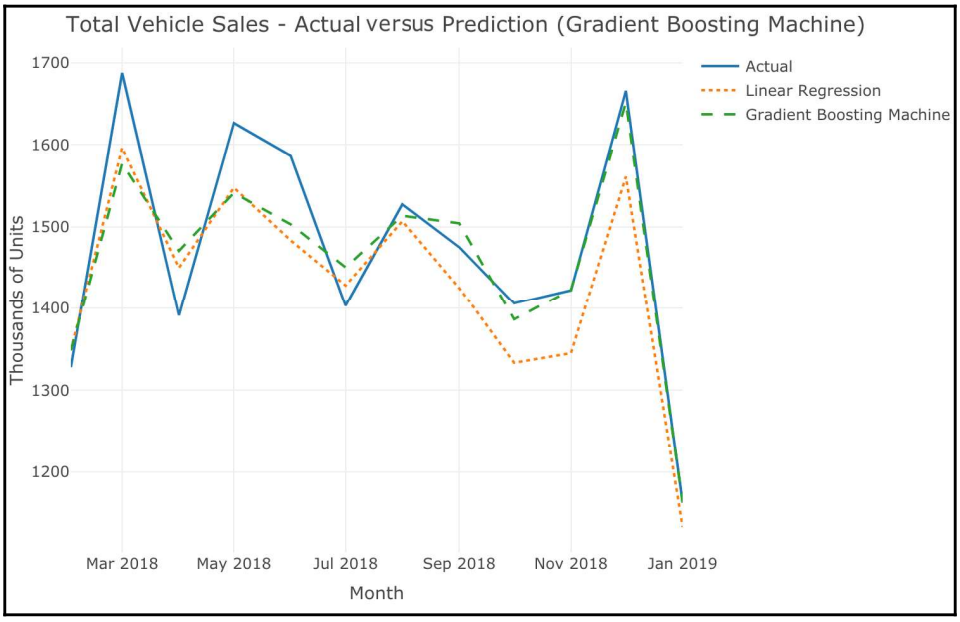
The GBM model scored the lowest MAPE among the models we've tested so far with a 2.7% error rate, compared to 3.3% and 4% with the RF model (with grid search) and linear regression model, respectively. It would be a great exercise to apply a grid search on the GBM model and test whether any additional improvements can be achieved.

Let's visualize the results and compare the prediction with the actual and baseline prediction:

```
plot_ly(data = test_1) %>%
  add_lines(x = ~ date, y = ~y, name = "Actual") %>%
  add_lines(x = ~ date, y = ~ yhat, name = "Linear Regression", line =
list(dash = "dot")) %>%
  add_lines(x = ~ date, y = ~ pred_gbm, name = "Gradient Boosting Machine",
line = list(dash = "dash")) %>%
  layout(title = "Total Vehicle Sales - Actual vs. Prediction (Gradient
Boosting Machine)",
```

```
        yaxis = list(title = "Thousands of Units"),
        xaxis = list(title = "Month"))
```

We get the following output:



Forecasting with the AutoML model

So far, in this chapter, we have looked at two modeling approaches—the first using the algorithm's default setting with the RF and GBM models, and the second by applying a grid search with the RF model. Let's take a look at a third approach to tuning ML models. Here, we will use the `h2o.automl` function. The `h2o.automl` function provides an automated approach to training, tuning, and testing multiple ML algorithms before selecting the model that performed best based on the model's evaluation. It utilizes algorithms such as RF, GBM, DL, and others using different tuning approaches.

Similarly, the `h2o.grid` function can apply any of the training approaches (CV, training with validation, and so on) during the training process of the models. Let's use the same input as before, and train the forecasting model:

```
autoML1 <- h2o.automl(training_frame = train_h,
                      x = x,
                      y = y,
```

```
                    nfolds = 5,
                    max_runtime_secs = 60*20,
                    seed = 1234)
```

We can set the runtime of the function. A longer running time could potentially yield better results.

In the preceding example, the function's running time was set to `20` minutes. The function returns a list with the `leaderboard` of all the tested models:

```
autoML1@leaderboard
```

We get the following output:

```
##                                             model_id
## 1  DeepLearning_grid_1_AutoML_20190507_162701_model_1
## 2  DeepLearning_grid_1_AutoML_20190507_162701_model_12
## 3  DeepLearning_grid_1_AutoML_20190507_162701_model_9
## 4  DeepLearning_grid_1_AutoML_20190507_162701_model_2
## 5  DeepLearning_grid_1_AutoML_20190507_162701_model_5
## 6  DeepLearning_grid_1_AutoML_20190507_162701_model_4
##   mean_residual_deviance     rmse      mse      mae      rmsle
## 1               3977.890 63.07052 3977.890 49.20507 0.04897362
## 2               3986.516 63.13887 3986.516 48.95475 0.04908595
## 3               4121.476 64.19872 4121.476 50.80116 0.04983452
## 4               4289.938 65.49762 4289.938 51.64339 0.05128537
## 5               4440.572 66.63762 4440.572 50.27883 0.05153639
## 6               4441.784 66.64671 4441.784 52.11865 0.05229755
##
## [139 rows x 6 columns]
```

You can see that in this case, the top models are DL models with different tuning settings. We will select the leader model and test its performance on the testing set:

```
test_h$pred_autoML  <- h2o.predict(autoML1@leader, test_h)

test_1 <- as.data.frame(test_h)

mape_autoML <- mean(abs(test_1$y - test_1$pred_autoML) / test_1$y)

mape_autoML
```

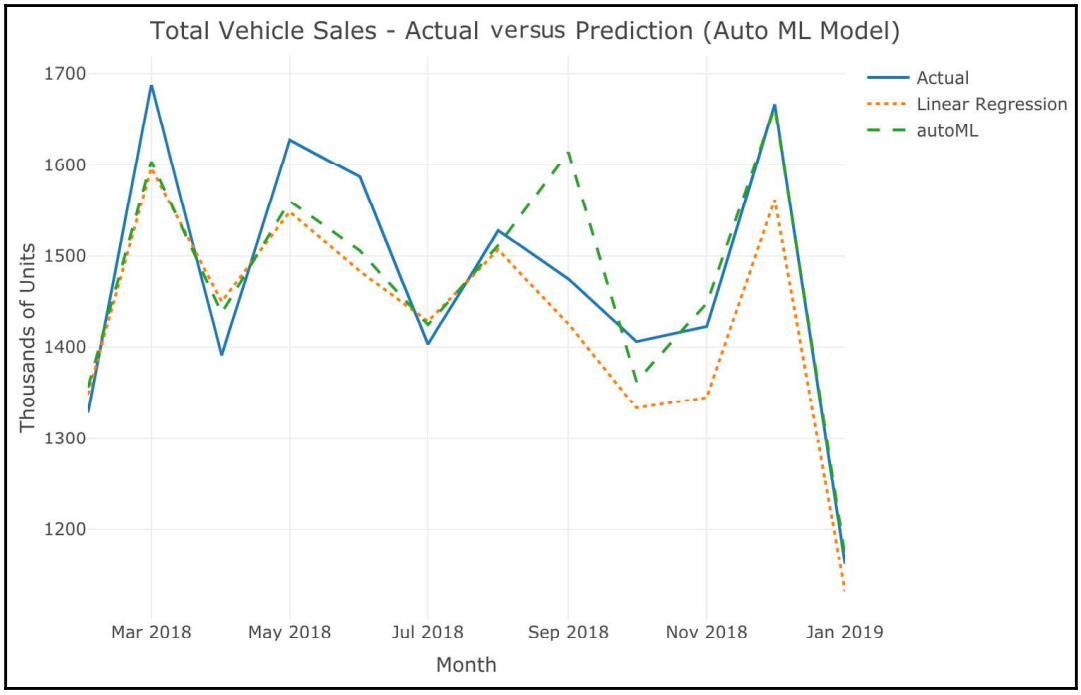
We get the following output:

```
## [1] 0.03481595
```

The leader model of the output of `h2o.automl` achieved a MAPE score of 3.48%. Although the previous models achieved a higher score, it still provides a significant lift with respect to the baseline model. Let's visualize the prediction of the model on the testing partitions with respect to the actual and baseline prediction:

```
plot_ly(data = test_1) %>%
  add_lines(x = ~ date, y = ~y, name = "Actual") %>%
  add_lines(x = ~ date, y = ~ yhat, name = "Linear Regression", line =
list(dash = "dot")) %>%
  add_lines(x = ~ date, y = ~ pred_autoML, name = "autoML", line =
list(dash = "dash")) %>%
  layout(title = "Total Vehicle Sales - Actual vs. Prediction (Auto ML
Model)",
         yaxis = list(title = "Thousands of Units"),
         xaxis = list(title = "Month"))
```

We get the following output:



The main advantage of the `h2o.automl` function is that it can scale up while having multiple series to forecast with minimal intervention from the user. This, of course, comes with additional compute power and time cost.

Selecting the final model

Now that we've finished the training and testing process of the models, it's time to finalize the process and choose the model to forecast with the series. We trained the following models:

- **Baseline model**: Linear regression model with 4% MAPE score on the testing partition
- **RF**: Using default tuning parameters with 3.74% MAPE score on the testing partition
- **RF**: Using grid search for tuning the model parameters with 3.33% MAPE score on the testing partition
- **GBM**: Using the default tuning parameters with 2.75% MAPE score on the testing partition
- **AutoML**: Selecting a deep learning model with 3.48% MAPE score on the testing partition

Since all of these models achieved better results than the baseline, we can drop the baseline model. Also, since the second RF model (with grid search) achieved better results than the first, there is no point in keeping the first model. This leaves us with three forecasting models, that is, RF (with grid search), GBM, and AutoML. Generally, since the GBM model achieved the best MAPE results, we will select it. However, it is always nice to plot the actual forecast and check what the actual forecast looks like. Before we plot the results, let's use these three models to forecast the next 12 months using the `data.frame` forecast we created in the *Forecasting with the Random Forest model* and *Forecasting with the GBM model* sections:

```
forecast_h$pred_gbm <- h2o.predict(gbm_md, forecast_h)
forecast_h$pred_rf <- h2o.predict(rf_grid_model, forecast_h)
forecast_h$pred_automl <- h2o.predict(autoML1@leader, forecast_h)
```

We will transform the object back into a `data.frame` object with the `as.data.frame` function:

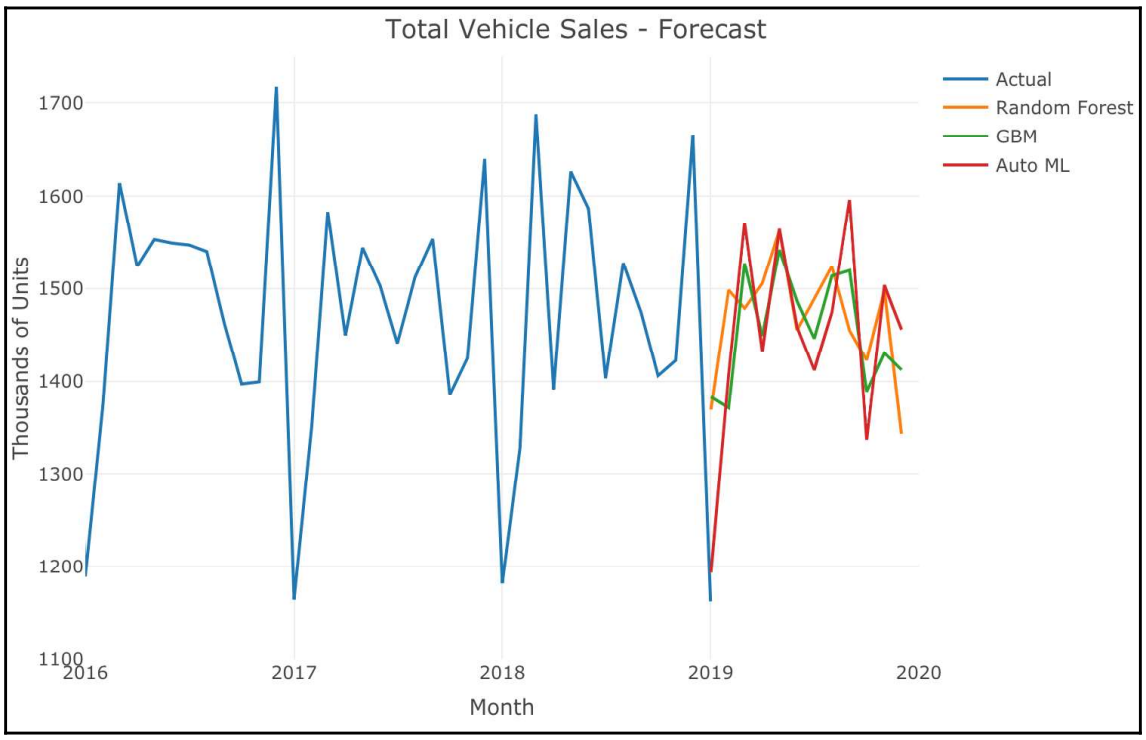
```
final_forecast <- as.data.frame(forecast_h)
```

Now, we can plot the final forecast with the **plotly** package:

```
plot_ly(x = df$date, y = df$y,
        type = "scatter",
        mode = "line",
        name = "Actual") %>%
  add_lines(x = final_forecast$date, y = final_forecast$pred_rf, name =
"Random Forest") %>%
```

```
  add_lines(x = final_forecast$date, y = final_forecast$pred_gbm, name =
"GBM") %>%
  add_lines(x = final_forecast$date, y = final_forecast$pred_automl, name =
"Auto ML") %>%
  layout(title = "Total Vehicle Sales - Final Forecast",
         yaxis = list(title = "Thousands of Units", range = c(1100, 1750)),
         xaxis = list(title = "Month", range = c(as.Date("2016-01-01"),
as.Date("2020-01-01"))))
```

We get the following output:



It seems that all three models capture the seasonality component of the vehicle sales series. However, it seems that the oscillation of AutoML is higher with respect to one of the RF and GBM models. Therefore, it would make sense to select either the GBM or RF models as the final forecast. A more conservative approach would be to create and ensemble the three forecasts by either weighted on regular average. For instance, you can use a simple function for testing the different average of different models and select the combination that minimizes the forecast error rate on the testing set.

Summary

In this chapter, we introduced the applications of ML models for forecasting time series data. Before we jumped into the modeling part, we looked at the usage of the major concepts we've learned about throughout this book. We started with an exploratory analysis of the US vehicle sales series using seasonality and correlation analysis. The insights from this process allowed us to build new features, which we then used as inputs for the ML models. Furthermore, we looked at the advantages of the grid search for tuning and optimizing ML models. Last but not least, we introduced the AutoML model from the **h2o** package in order to complete the automation, tuning, and optimization processes for ML models.

With that, I hope you have enjoyed the learning journey that we have been on throughout this book!