Module I - Advanced Machine Learning

Dr. Varalatchoumy M
Prof.&Head – Dept. of AIML
Head –CHOSS,
Cambridge Institute of Technology, Bangalore

6.1 | OVERVIEW

Machine learning algorithms are a subset of artificial intelligence (AI) that imitates the human learning process.

Humans learn through multiple experiences how to perform a task.

Similarly, machine learning algorithms develop multiple models (usually using multiple datasets) and each model is analogous to an experience

Mitchell (2006) defined machine learning as follows:

Machine learns with respect to a particular task T, performance metric P following experience E, if the system reliably improves its performance P at task T following experience E.

- Let the task T be a classification problem
- Performance P can be measured through several metrics such as overall accuracy, sensitivity, specificity, and area under the receive operating characteristic curve (AUC)
- Experience E is analogous to different classifiers generated in machine learning algorithms

The major difference between statistical learning and machine learning is that

statistical learning depends heavily on validation of model assumptions and hypothesis testing, objective of **machine learning** is to improve prediction accuracy.

For example, while developing a regression model, we check for assumptions such as normality of residuals, significance of regression parameters and so on. However, in the case of the random forest using classification trees, the most important objective is the accuracy/performance of the model.

Two ML algorithms:

1. **Supervised Learning**: In supervised learning, the datasets have the values of input variables (feature values) and the corresponding outcome variable. The algorithms learn from the training dataset and predict the outcome variable for a new record with values of input variables. Linear regression and logistic regression are examples of supervised learning algorithms.
2. **Unsupervised Learning**: In this case, the datasets will have only input variable values, but not the output. The algorithm learns the structure in the inputs. Clustering and factor analysis are examples of unsupervised learning and will be discussed in Chapter 7.

6.1.1 | How Machines Learn?

In supervised learning, the algorithm learns using a function called **loss function, cost function or error function**, which is a function of predicted output and the desired output. If h(Xi) is the predicted output and yi is the desired output, then the loss function is

$$L = \frac{1}{n}\sum_{i=1}^{n}[h(X_i) - y_i]^2$$

n is the total number of records for which the predictions are made.
The function defined above is a sum of squared error (SSE).
SSE is the loss function for a regression model.
The objective is to learn the values of parameters (aka feature weights) that minimize the cost function.
Machine learning uses optimization algorithms which can be used for minimizing the loss function.
Most widely used optimization technique is called the **Gradient Descent**.

In the next section, we will discuss a regression problem and understand how gradient descent algorithm minimizes the loss function and learn the model parameters.

6.2 | GRADIENT DESCENT ALGORITHM

In this section, we will discuss how gradient descent (GD) algorithm can be used for estimating the values of regression parameters, given a dataset with inputs and outputs

The functional form of a simple linear regression model is given by

$$Y_i = \beta_0 + \beta_1 X_i + \varepsilon_i \tag{6.2}$$

where β_0 is called bias or intercept, β_1 is the feature weight or regression coefficient, ε_i is the error in prediction.

The predicted value of Y_i is written as $\widehat{Y_i}$ and it is given by

$$\widehat{Y_i} = \widehat{\beta_0} + \widehat{\beta_1} X_i \tag{6.3}$$

where $\widehat{\beta_0}$ and $\widehat{\beta_1}$ are the estimated values of β_0 and β_1. The error is given by

The error is given by,

$$\varepsilon_i = Y_i - \hat{Y}_i = (Y_i - \hat{\beta}_0 - \hat{\beta}_1 X_i) \tag{6.4}$$

The **cost function** for the linear regression model is the total error (mean squared error) across all N records and is given by

$$\text{MSE} = \varepsilon_{\text{mse}} = \frac{1}{N}\sum_{i=1}^{N}(Y_i - \hat{\beta}_0 - \hat{\beta}_1 X_i)^2 \tag{6.5}$$

The error is a function of β_0 and β_1. It is pure convex function and has a global minimum as shown in Figure 6.1. The gradient descent algorithm starts at a random starting value (for β_0 and β_1) and moves towards the optimal solution as shown in Figure 6.1.

FIGURE 6.1 Cost function for linear regression model.

Gradient descent finds the optimal values of β_0 and β_1 that minimize the loss function using the following steps:

1. Randomly guess the initial values of β_0 (bias or intercept) and β_1 (feature weight).
2. Calculate the estimated value of the outcome variable $\widehat{Y}_i$ for initialized values of bias and weights.
3. Calculate the mean square error function (MSE).
4. Adjust the β_0 and β_1 values by calculating the gradients of the error function.

$$\beta_0 = \beta_0 - \alpha \times \frac{\partial \varepsilon_{\text{mse}}}{\partial \beta_0} \tag{6.6}$$

$$\beta_1 = \beta_1 - \alpha \times \frac{\partial \varepsilon_{\text{mse}}}{\partial \beta_1} \tag{6.7}$$

where α is the learning rate (a hyperparameter). The value of α is chosen based on the magnitude of update needed to be applied to the bias and weights at each iteration.

The partial derivatives of MSE with respect to β_0 and β_1 are given by

$$\frac{\partial \varepsilon_{\text{mse}}}{\partial \beta_0} = -\frac{2}{N}\sum_{i=1}^{N}(Y_i - \beta_0 - \beta_1 X_i) = -\frac{2}{N}\sum_{i=1}^{N}(Y_i - \widehat{Y}_i) \tag{6.8}$$

$$\frac{\partial \varepsilon_{\text{mse}}}{\partial \beta_1} = -\frac{2}{N}\sum_{i=1}^{N}(Y_i - \beta_0 - \beta_1 X_i) = -\frac{2}{N}\sum_{i=1}^{N}(Y_i - \widehat{Y}_i) \times X_i \tag{6.9}$$

5. Repeat steps 1 to 4 for several iterations until the error stops reducing further or the change in cost is infinitesimally small.

The values of β_0 and β_1 at the minimum cost points are best estimates of the model parameters.

6.2.1 | Developing a Gradient Descent Algorithm for Linear Regression Model

- For better understanding the GD algorithm, we will implement the GD algorithm using the dataset Advertising.csv.
- The dataset contains the examples of advertisement spends across multiple channels such as Radio, TV, and Newspaper, and the corresponding sales revenue generated at different time periods.

The dataset has the following elements:

1. TV – Spend on TV advertisements
2. Radio – Spend on radio advertisements
3. Newspaper – Spend on newspaper advertisements
4. Sales – Sales revenue generated

For predicting future sales using spends on different advertisement channels, we can build a regression model.

kaggle

Search

Create

Home

Competitions

Datasets

Models

Code

Discussions

Learn

More

Your Work

SOUVIK DEY · UPDATED 3 YEARS AGO

42

New Notebook

Download (2 kB)

Advertising.csv

It is data set for linear regrssion

Data Card | Code (59) | Discussion (0)

About Dataset

No description available

Usability

3.53

License

Unknown

6.2.1.1 Loading the Dataset

Untitled8.ipynb

File Edit View Insert Runtime Tools Help All changes saved

Comment Share

Files

..

sample_data

Advertising.csv

Disk 81.46 GB available

+ Code + Text

RAM

Disk

0s

```python
import pandas as pd
import numpy as np
import warnings
warnings.filterwarnings("ignore")
sales_df = pd.read_csv("Advertising.csv")
# Printing first few records
sales_df.head()
```

	Unnamed: 0	TV	Radio	Newspaper	Sales
0	1	230.1	37.8	69.2	22.1
1	2	44.5	39.3	45.1	10.4
2	3	17.2	45.9	69.3	9.3
3	4	151.5	41.3	58.5	18.5
4	5	180.8	10.8	58.4	12.9

6.2.1.2 Set X and Y Variables

For building a regression model, the inputs *TV*, *Radio*, and *Newspaper* are taken as *X* features and *Sales Y* is taken as the outcome variable.

6.2.1.3 Standardize X and Y

It is important to convert all variables into one scale. This can be done by subtracting mean from each value of the variable and dividing by the corresponding standard deviation of the variable.

6.2.1.4 *Implementing the Gradient Descent Algorithm*

To implement the steps explained in Section 6.2, *Gradient Descent*, a set of following utility methods need to be implemented:

1. **Method 1:** Method to randomly initialize the bias and weights.
2. **Method 2:** Method to calculate the predicted value of Y, that is, Y given the bias and weights.
3. **Method 3:** Method to calculate the cost function from predicted and actual values of Y.
4. **Method 4:** Method to calculate the gradients and adjust the bias and weights.

Method 1: Random Initialization of the Bias and Weights

The method randomly initializes the bias and weights. It takes the number of weights that need to be initialized as a parameter.

```
def initialize( dim ):
  np.random.seed(seed=42)
  random.seed(42)
  #Initialize the bias.
  b = random.random()
  #Initialize the weights.
  w = np.random.rand( dim )
  return b, w
```

```
#dim - is the number of weights to be
initialized besides the bias
```

To initialize the bias and 3 weights, as we have three input variables TV, Radio and Newspaper, we can invoke the initialize() method as follows:

```
b, w = initialize( 3 )
print( "Bias: ", b, "Weights: ", w )
```

Untitled8.ipynb

File Edit View Insert Runtime Tools Help All changes saved

Comment Share

Files

..

sample_data

Advertising.csv

+ Code + Text

RAM Disk

```
import pandas as pd
import numpy as np
import random
import warnings
warnings.filterwarnings("ignore")
sales_df = pd.read_csv("Advertising.csv")
# Printing first few records
sales_df.head()
X = sales_df[["TV", "Radio", "Newspaper"]]
Y = sales_df["Sales"]

Y = np.array( (Y - Y.mean() ) / Y.std() )
X = X.apply( lambda rec: ( rec - rec.mean() ) / rec.std(),axis = 0 )

def initialize( dim ):
  np.random.seed(seed=42)
  random.seed(42)
  #Initialize the bias.
  b = random.random()
  #Initialize the weights.
  w = np.random.rand( dim )
  return b, w

b, w = initialize( 3 )
print( "Bias: ", b, "Weights: ", w )
```

```
Bias:  0.6394267984578837 Weights:  [0.37454012 0.95071431 0.73199394]
```

Your device needs to restart to install updates.

Method 2: Predict Y Values from the Bias and Weights

Calculate the *Y* values for all the inputs, given the bias and weights. We will use matrix multiplication of weights with input variable values. *matmul()* method in *numpy* library can be used for matrix multiplication. Each row of *X* can be multiplied with the weights column to produce the predicted outcome variable.

```
# Inputs:
# b - bias
# w - weights
# X - the input matrix
```

```python
def predict_Y( b, w, X ):
  return b + np.matmul( X, w )

b, w = initialize( 3 )
Y_hat = predict_Y( b, w, X)
Y_hat[0:10]
```

```
0    3.231496
1    1.707849
2    2.824761
3    2.753090
4    0.924486
5    3.171365
6    0.622344
7   -0.349354
8   -2.313095
9   -0.768030
dtype: float64
```

Method 3: Calculate the Cost Function — MSE

Compute mean squared error (MSE) by

1. Calculating differences between the estimated $\widehat{Y}$ and actual Y.
2. Calculating the square of the above residuals, and sum over all records.
3. Dividing it with the number of observations.

```
import math

# Inputs
# Y - Actual values of y
# Y_hat - predicted value of y
def get_cost( Y, Y_hat ):
    # Calculating the residuals - difference between actual and
    # predicted values
      Y_resid = Y - Y_hat
    # Matrix multiplication with self will give the square values
    # Then take the sum and divide by number of examples to
    # calculate mean
    return np.sum( np.matmul( Y_resid.T, Y_resid ) ) / len( Y_resid )
```

Invoking *get_cost()* after initializing the bias and weights and calculating predicted values for outcome variable.

```
b, w = initialize( 3 )
Y_hat = predict_Y( b, w, X)
get_cost( Y, Y_hat )
```

```
1.5303100198505895
```

```
import math
def get_cost( Y, Y_hat ):
  Y_resid = Y - Y_hat
  return np.sum( np.matmul( Y_resid.T, Y_resid ) ) / len( Y_resid )

b, w = initialize( 3 )
Y_hat = predict_Y( b, w, X)
get_cost( Y, Y_hat )
```

```
1.53031001985059
```

Method 4: Update the Bias and Weights

This is the most important method, where the bias and weights are adjusted based on the gradient of the cost function. The bias and weights will be updated in a method *update_beta()* using the following gradients [Eqs. (6.6) and (6.7)]:

$$\beta_0 = \beta_0 - \alpha \times \frac{\partial \varepsilon_{\text{mse}}}{\partial \beta_0}$$

$$\beta_1 = \beta_1 - \alpha \times \frac{\partial \varepsilon_{\text{mse}}}{\partial \beta_1}$$

where α is the learning parameter that decides the magnitude of the update to be done to the bias and weights.

The parameters passed to the function are:

1. *x, y*: the input and output variables
2. *y_hat*: predicted value with current bias and weights
3. *b_0, w_0*: current bias and weights
4. *learning rate*: learning rate to adjust the update step

```
def update_beta( x, y, y_hat, b_0, w_0, learning_rate ):

    #gradient of bias
    db = (np.sum( y_hat - y ) * 2) / len(y)
    #gradient of weights
    dw = (np.dot( ( y_hat - y ), x ) * 2 ) / len(y)
    #update bias
    b_1 = b_0 - learning_rate * db
    #update beta
    w_1 = w_0 - learning_rate * dw

    #return the new bias and beta values
    return b_1, w_1
```

The following is an example of updating bias and weights once after initializing. The learning parameter used is 0.01.

```
b, w = initialize( 3 )
print( "After Initialization - Bias: ", b, " Weights: ", w )
Y_hat = predict_Y( b, w, X)
b, w = update_beta( X, Y, Y_hat, b, w, 0.01 )
print( "After first update - Bias: ", b, " Weights: ", w )
```

```
After initialization - Bias:  0.6394267984578837 Weights:  [0.37454
012 0.95071431 0.73199394]
After first update - Bias:    0.6266382624887261 Weights:  [0.3807909
3 0.9376953 0.71484883]
```

[33]

```python
def update_beta( x, y, y_hat, b_0, w_0, learning_rate ):
#gradient of bias
  db = (np.sum( y_hat - y ) * 2) / len(y)
  #gradient of weights
  dw = (np.dot( ( y_hat - y ), x ) * 2 ) / len(y)
  #update bias
  b_1 = b_0 - learning_rate * db
  #update beta
  w_1 = w_0 - learning_rate * dw
  #return the new bias and beta values
  return b_1, w_1
```

```python
b, w = initialize( 3 )
print("After Initialization - Bias: ", b, "Weights: ", w )
Y_hat = predict_Y( b, w, X)
b, w = update_beta( X, Y, Y_hat, b, w, 0.01 )
print("After first update - Bias: ", b, "Weights: ", w )
```

```
After Initialization - Bias:  0.6394267984578837 Weights:  [0.37454012 0.95071431 0.73199394]
After first update - Bias:  0.6266382624887261 Weights:  [0.38079093 0.9376953  0.71484883]
```

6.2.1.5 Finding the Optimal Bias and Weights

The updates to the bias and weights need to be done iteratively, until the cost is minimum. It can take several iterations and is time-consuming. There are two approaches to stop the iterations:

1. Run a fixed number of iterations and use the bias and weights as optimal values at the end these iterations.

2. Run iterations until the change in cost is small, that is, less than a predefined value (e.g., 0.001).

We will define a method *run_gradient_descent*(), which takes *alpha* and *num_iterations* as parameters and invokes methods like *initialize(), predict_Y(), get_cost(),* and *update_beta().*

Also, inside the method,

1. variable *gd_iterations_df* keeps track of the cost every 10 iterations.

2. default value of 0.01 for the learning parameter and 100 for number of iterations will be used.

```python
def run_gradient_descent( X,
Y,
alpha = 0.01,
num_iterations = 100):
# Initialize the bias and weights
  b, w = initialize( X.shape[1] )
  iter_num = 0
  # gd_iterations_df keeps track of the cost every 10 iterations
  gd_iterations_df = pd.DataFrame(columns = ["iteration", "cost"])
  result_idx = 0
  # Run the iterations in loop
  for each_iter in range(num_iterations):
  # Calculate predicted value of y
    Y_hat = predict_Y( b, w, X )
  # Calculate the cost
  this_cost = get_cost( Y, Y_hat )
  # Save the previous bias and weights
  prev_b = b
  prev_w = w
  # Update and calculate the new values of bias and weights
  b, w = update_beta( X, Y, Y_hat, prev_b, prev_w, alpha)
  # For every 10 iterations, store the cost i.e. MSE
  if( iter_num % 10 == 0 ):
    gd_iterations_df.loc[result_idx] = [iter_num, this_cost]
    result_idx = result_idx + 1
  iter_num += 1
  print( "Final estimate of b and w: ", b, w )
  #return the final bias, weights and the cost at the end
  return gd_iterations_df, b, w
```

```
[38] gd_iterations_df, b, w = run_gradient_descent( X, Y, alpha = 0.001, num_iterations = 200 )
```

```
Final estimate of b and w:  0.6381479448609679 [0.3751652  0.94941241 0.73027943]
```

```
gd_iterations_df[0:10]
```

	iteration	cost
0	0.0	1.53031

```
gd_iterations_df[0:10]
```

	iteration	cost
0	0.0	1.530310
1	10.0	1.465201
2	20.0	1.403145
3	30.0	1.343996
4	40.0	1.287615
5	50.0	1.233868
6	60.0	1.182630
7	70.0	1.133780
8	80.0	1.087203
9	90.0	1.042793

It can be noted that the cost is reducing at the end of each iteration.

6.2.1.6 Plotting the Cost Function against the Iterations

The value of the cost function at the end of each iteration can be visualized by plotting cost at every 10 iterations using the following commands:

```
import matplotlib.pyplot as plt
import seaborn as sn
%matplotlib inline
```

```
plt.plot( gd_iterations_df['iteration'], gd_iterations_df['cost'] );
plt.xlabel("Number of iterations")
plt.xlabel("Cost or MSE")
```

```
Text(0.5, 0,'Cost or MSE')
```

```
print( "Final estimates of b and w: ", b, w )
```

```
Final estimates of b and w: 0.42844895817391493 [0.48270238 0.75265
969 0.46109174]
```

FIGURE 6.2 Cost or MSE at the end of iterations.

From Figure 6.2, it can be noticed that the cost is still reducing and has not reached the minimum point. We can run more iterations and verify if the cost is reaching a minimum point or not.

```
alpha_df_1, b, w = run_gradient_descent(X, Y, alpha = 0.01,
                                        num_iterations = 2000)
```

```
Final estimate of b and w: 2.7728016698178713e-16
                           [ 0.75306591 0.5 3648155 -0.00433069]
```

What happens if we change the learning parameter and use smaller value (e.g., 0.001)?

```
alpha_df_2, b, w = run_gradient_descent(X, Y, alpha = 0.001,
                                        num_iterations = 2000)
```

```
Final estimate of b and w:  0.011664695556930518 [0.74315125 0.52779
                            959 0.01171703]
```

Now we plot the cost after every iteration for different learning rate parameters (alpha values).

```
plt.plot( alpha_df_1['iteration'], alpha_df_1['cost'], label =
                                                 "alpha = 0.01" );
plt.plot( alpha_df_2['iteration'], alpha_df_2['cost'], label =
                                                 "alpha = 0.001");
```

```
plt.legend()
plt.ylabel('Cost');
plt.xlabel('Number of Iterations');
plt.title('Cost Vs. Iterations for different alpha values');
```

FIGURE 6.3 Cost or MSE at the end of iterations for different alpha values.

The plot in Figure 6.3 shows that the learning is faster for *alpha* value 0.01 compared to 0.001. For smaller values, the learning could be slower whereas higher learning rate could lead to skipping the minima of cost function. It is imperative to search for the optimal learning parameter.

6.3.1 | Steps for Building Machine Learning Models

The steps to be followed for building, validating a machine learning model and measuring its accuracy
are as follows:

1. Identify the features and outcome variable in the dataset.
2. Split the dataset into training and test sets.
3. Build the model using training set.
4. Predict outcome variable using a test set.
5. Compare the predicted and actual values of the outcome variable in the test set and measure accuracy using measures such as mean absolute percentage error (MAPE) or root mean square error (RMSE).

6.3.1.1 Splitting Dataset into Train and Test Datasets

The following commands can be used for splitting the dataset using 70:30 ratio. That is, 70% for training and 30% for the test set. Usually, data scientists use a range of 60% to 80% for training and the rest for testing the model.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
                                   sales_df[["TV", "Radio",
                                   "Newspaper"]],
                                   sales_df.Sales, test_size=0.3,
                                   random_state = 42)
```

The following commands can be used for finding the number of records sampled into training and test sets.

```
len( X_train )
```

```
140
```

```
len( X_test )
```

```
60
```

6.3.1.2 Building Linear Regression Model with Train Dataset

Linear models are included in sklearn.linear_model module. We will use LinearRegression method for building the model and compare with the results we obtained through our own implementation of gradient descent algorithm.

1.1. Linear Models

The following are a set of methods intended for regression in which the target value is expected to be a linear combination of the features. In mathematical notation, if $\hat{y}$ is the predicted value.

$$\hat{y}(w, x) = w_0 + w_1 x_1 + \ldots + w_p x_p$$

Across the module, we designate the vector $w = (w_1, \ldots, w_p)$ as `coef_` and w_0 as `intercept_`.

To perform classification with generalized linear models, see Logistic regression.

1.1.1. Ordinary Least Squares

`LinearRegression` fits a linear model with coefficients $w = (w_1, \ldots, w_p)$ to minimize the residual sum of squares between the observed targets in the dataset, and the targets predicted by the linear approximation. Mathematically it solves a problem of the form:

1.1.1.1. Non-Negative Least Squares

It is possible to constrain all the coefficients to be non-negative, which may be useful when they represent some physical or naturally non-negative quantities (e.g., frequency counts or prices of goods). `LinearRegression` accepts a boolean `positive` parameter: when set to `True` Non-Negative Least Squares are then applied.

Examples:

- Non-negative least squares

1.1.1.2. Ordinary Least Squares Complexity

The least squares solution is computed using the singular value decomposition of X. If X is a matrix of shape `(n_samples, n_features)` this method has a cost of $O(n_{\text{samples}} n_{\text{features}}^2)$, assuming that $n_{\text{samples}} \geq n_{\text{features}}$.

1.1.2. Ridge regression and classification

https://scikit-learn.org/stable/modules/linear_model.html#:~:text=The%20following%20are%20a%20set,if%20is%20the%20predicted%20value.

6.3.1.2 Building Linear Regression Model with Train Dataset

```
from sklearn.linear_model import LinearRegression
```

Steps for building a model in *sklearn* are

1. Initialize the model.
2. Invoke *fit()* method on the model and pass the input (X) and output(Y) values.
3. *fit()* will run the algorithm and return the final estimated model parameters.

```
# Initializing the model
linreg = LinearRegression()
# Fitting training data to the model
linreg.fit( X_train, y_train )
```

After the model is built, the model parameters such as intercept (bias) and coefficients (weights) can be obtained using the following commands:

```
linreg.intercept_
```

```
2.708949092515912
```

```
linreg.coef_
```

```
array([0.04405928, 0.1992875, 0.00688245])
```

To associate the coefficient values with the variable names, we can use *zip()* in Python. *zip()* returns a dictionary with variable names mapped to coefficient values.

```
list( zip( ["TV", "Radio", "Newspaper"], list( linreg.coef_ ) ) )
```

```
[('TV', 0.04405928),
 ('Radio', 0.1992874968989395),
 ('Newspaper', 0.006882452222754)]
```

The estimated model is

$$Sales = 2.708 + 0.044 * TV + 0.199 * Radio + 0.006 * Newspaper$$

The weights are different than what we estimated earlier. This is because we have not standardized the values in this model. The model indicates that for every unit change in TV spending, there is an increase of 0.044 units in sales revenue.

6.3.1.3 Making Prediction on Test Set

sklearn provides *predict()* method for all ML models, which takes the *X* values and predicts the outcome variable *Y* as shown in the following code:

```
# Predicting the y value from the test set
y_pred = linreg.predict( X_test )
```

To compare the actual and predicted values of the outcome variable and the residuals, we will create and store these values in a DataFrame. The residual here refers to the difference between the actual and the predicted values.

```
# Creating DataFrame with 3 columns named: actual, predicted and residuals
# to store the respective values
test_pred_df = pd.DataFrame( { 'actual': y_test,
                               'predicted': np.round( y_pred, 2 ),
                               'residuals': y_test - y_pred } )
# Randomly showing the 10 observations from the DataFrame
test_pred_df.sample(10)
```

	actual	predicted	residuals
126	6.6	11.15	−4.553147
170	8.4	7.35	1.049715
95	16.9	16.57	0.334604
195	7.6	5.22	2.375645
115	12.6	13.36	−0.755569
38	10.1	10.17	−0.070454
56	5.5	8.92	−3.415494
165	11.9	14.30	−2.402060
173	11.7	11.63	0.068431
9	10.6	12.18	−1.576049

6.3.1.4 Measuring Accuracy

Root Mean Square Error (RMSE) and R-squared are two key accuracy measures for *Linear Regression Models*.

sklearn.metrics package provides methods to measure various metrics.

For regression models, *mean_squared_error* and *r2_score* can be used to calculate MSE and R-squared values, respectively.

```
## Importing metrics from sklearn
from sklearn import metrics
```

R-Squared Value

The R-squared value is calculated on the training data, which is the amount of variance in *Y* that can be explained by the model. *metrics.r2_score()* takes the actual and the predicted values of *Y* to compute R-squared value. The following command can be used for calculating R-squared value:

```
# y_train contains the actual value and the predicted value is
# returned from predict() method after passing the X values of the
# training data.
r2 = metrics.r2_score( y_train, linreg.predict(X_train) )
print("R Squared: ", r2)
```

```
R Squared:   0.9055159502227753
```

The model explains 90% of the variance in *Y*.

RMSE Calculation

metrics.mean_squared_error() takes actual and predicted values and returns MSE.

```
# y_pred contains predicted value of test data
mse = metrics.mean_squared_error( y_test, y_pred )
```

RMSE value can be calculated by the square root of *mse* using the following command:

```
# Taking square root of MSE and then round off to two decimal values
rmse = round( np.sqrt(mse), 2 )
print("RMSE: ", rmse)
```

```
RMSE: 1.95
```

The RMSE value indicates the model prediction has a standard deviation of 1.95.

To understand the model error in detail, we need to understand the components of the error term and how to deal with those components for improving model performance. In the next section, we will discuss these components in detail.

6.3.2 | Bias-Variance Trade-off

Model errors can be decomposed into two components: *bias* and *variance*. Understanding these two components is key to diagnosing model accuracies and avoiding model overfitting or underfitting.
High bias can lead to building underfitting model, whereas high variance can lead to overfitting models.

The term "variance" refers to the degree of change that may be expected in the estimation of the target function as a result of using multiple sets of training data. The disparity between the values that were predicted and the values that were actually observed is referred to as bias

Let us take an example to understand bias and variance in detail. The dataset *curve.csv* has records with two variables *x* and *y*, where *y* depends on *x*.

```
# Reading the file curve.csv and printing first few examples
curve = pd.read_csv( "curve.csv" )
curve.head()
```

	x	*y*
0	2	–1.999618
1	2	–1.999618
2	8	–3.978312
3	9	–1.969175
4	10	–0.957770

As there are only two variables, we can observe the relationship by plotting them using scatter plot (Figure 6.4).

```
plt.scatter( curve.x, curve.y );
plt.xlabel("x values")
plt.ylabel("y values")
```

```
Text(0,0.5,'y values')
```

FIGURE 6.4 Scatter plot between x and y.

It can be observed from Figure 6.4 that the relation between y and x is not linear and looks like some polynomial. But we are not sure of the degree of the polynomial form. We need to try various polynomial forms of x and verify which model fits the data best.

To explore various polynomial forms, *polyfit()* from *numpy* library can be used. *polyfit()* takes X and Y values, and the degree of x features to be used to fit a model. Degree 1 means only value of x is used to predict y, whereas degree 2 means x and x^2 are used to predict y.

The following commands can be used for implementing a generic method *fit_poly()*, which takes degree as a parameter and builds a model with all required polynomial terms.

```
# Input
# degree - polynomial terms to be used in the model
def fit_poly( degree ):
    # calling numpy method polyfit
    p = np.polyfit( curve.x, curve.y, deg = degree )
    curve['fit'] = np.polyval( p, curve.x )
    # draw the regression line after fitting the model
    sn.regplot( curve.x, curve.y, fit_reg = False )
    # Plot the actual x and y values
    return plt.plot( curve.x, curve.fit, label='fit' )
```

Fitting a mode with degree = 1.

$$y = \beta_1 x_1 + \varepsilon$$

```
fit_poly( 1 );
## Plotting the model form and the data
plt.xlabel("x values")
plt.ylabel("y values");
```

FIGURE 6.5 Linear regression model between *x* and *y*.

Linear model ($y = \beta_0 + \beta_1 x_1$) is shown in Figure 6.5. It is evident from the figure that the regression model does not seem to fit the data well. It is not able to learn from data as a simplistic form of model is assumed. This is a case of underfitting or bias.

Let us fit a regression model with a polynomial term, that is, square of x.

$$y = \beta_1 x_1 + \beta_2 x_1^2 + \varepsilon_i$$

```
fit_poly( 2 );
plt.xlabel("x values")
plt.ylabel("y values");
```

FIGURE 6.6 Linear regression model between y and polynomial terms of x of degrees 1 and 2.

Figure 6.6 shows the fitted model which includes x_1 and x_1^2. The regression line seems to fit data better than the previous model. But we are not sure if this is the best model. We can try building a model with all polynomial terms up to degrees ranging from 1 to 10 as shown below:

$$y = \beta_1 x_1 + \beta_2 x_1^2 + \beta_2 x_1^3 + \beta_2 x_1^4 + \beta_2 x_1^5 + \beta_2 x_1^6 + \beta_2 x_1^7 + \beta_2 x_1^8 + \beta_2 x_1^9 + \beta_2 x_1^{10} + \varepsilon_i$$

```
fit_poly( 10 );
plt.xlabel("x values")
plt.ylabel("y values");
```

It can be observed from Figure 6.7 that as we build models with more and more polynomial terms, the model starts to fit every data point in the training set. This is the case of overfitting. The model with polynomial terms up to degree 10 will be sensitive to any changes in training examples. Any addition and removal of a single observation from the dataset can alter the model parameters significantly.

An underfitting model has a large error because of high bias, and an overfitting model has a large error because of high variance. An optimal model will be somewhere between an underfitting and an overfitting model, and will have low bias and low variance. This can be observed by comparing RMSE in training and test sets.

We use the following code to build models with degrees ranging from 1 to 15 and storing the degree and error details in different columns of a DataFrame named *rmse_df*.

1. *degree*: Degree of the model (number of polynomial terms).
2. *rmse_train*: RMSE error on train set.
3. *rmse_test*: RMSE error on test set.

```
# Split the dataset into 60:40 into training and test set
train_X, test_X, train_y, test_y = train_test_split( curve.x,
                                                     curve.y,
                                                     test_size=0.40,
                                                     random_
                                                     state=100 )

# Define the dataframe store degree and rmse for training and test set
rmse_df = pd.DataFrame( columns = ["degree", "rmse_train",
                                   "rmse_test"] )

# Define a method to return the rmse given actual and predicted values.

def get_rmse( y, y_fit ):
    return np.sqrt( metrics.mean_squared_error( y, y_fit ) )

# Iterate from degree 1 to 15
for i in range( 1, 15 ):
    # fitting model
    p = np.polyfit( train_X, train_y, deg = i )
    # storing model degree and rmse on train and test set
    rmse_df.loc[i-1] = [ i,
                         get_rmse(train_y, np.polyval(p, train_X)),
                         get_rmse(test_y, np.polyval(p, test_X))]
```

Print the ***rmse_df*** records using the following code:

```
rmse_df
```

	degree	rmse_train	rmse_test
0	1.0	5.226638	5.779652
1	2.0	2.394509	2.755286
2	3.0	2.233547	2.560184
3	4.0	2.231998	2.549205
4	5.0	2.197528	2.428728
5	6.0	2.062201	2.703880
6	7.0	2.039408	2.909237
7	8.0	1.995852	3.270892
8	9.0	1.979322	3.120420
9	10.0	1.976326	3.115875
10	11.0	1.964484	3.218203
11	12.0	1.657948	4.457668
12	13.0	1.656719	4.358014
13	14.0	1.642308	4.659503

Now we can plot the train and test errors against the degree of the models for better understanding.

```
# Plotting the rmse for training set in red color
plt.plot( rmse_df.degree,
          rmse_df.rmse_train,
          label='RMSE on Training Set',
          color = 'r' )

# Plotting the rmse for test set in green color
plt.plot( rmse_df.degree,
          rmse_df.rmse_test,
          label='RMSE on Test Set',
          color = 'g' )
# Mention the legend
plt.legend(bbox_to_anchor=(1.05, 1),
           loc=2,
           borderaxespad=0.);
plt.xlabel("Model Degrees")
plt.ylabel("RMSE");
```

FIGURE 6.8 Model accuracy on training and test sets against the model complexity.

Three key observations from Figure 6.8 are as follows:

1. Error on the test set are high for the model with complexity of degree 1 and degree 15.
2. Error on the test set reduces initially, however increases after a specific level of complexity.
3. Error on the training set decreases continuously.

6.3.3 | K-Fold Cross-Validation

K-fold cross-validation is a robust validation approach that can be adopted to verify if the model is overfitting. The model, which generalizes well and does not overfit, should not be very sensitive to any change in underlying training samples. K-fold cross-validation can do this by building and validating multiple models by resampling multiple training and validation sets from the original dataset.

The following steps are used in K-fold cross-validation:

1. Split the training data set into K subsets of equal size. Each subset will be called a fold. Let the folds be labelled as $f_1, f_2, \ldots, f_K$. Generally, the value of K is taken to be 5 or 10.
2. For $i = 1$ to K
   (a) Fold f_i is used as validation set and all the remaining $K - 1$ folds as training set.
   (b) Train the model using the training set and calculate the accuracy of the model in fold f_i.

Calculate the final accuracy by averaging the accuracies in the test data across all K models. The average accuracy value shows how the model will behave in the real world. The variance of these accuracies is an indication of the robustness of the model.

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5
Iteration 1	train	train	train	train	test
Iteration 2	train	train	train	test	train
Iteration 3	train	train	test	train	train
Iteration 4	train	test	train	train	train
Iteration 5	test	train	train	train	train

FIGURE 6.9 K-fold cross-validation.

Figure 6.9 depicts the process of K-fold cross-validation with 5 folds.

6.4 | ADVANCED REGRESSION MODELS

IPL dataset

First, we will build a linear regression model to understand the shortcomings and then proceed to advanced regression models.

6.4.1.1 Loading IPL Dataset

Load the dataset and display information about the dataset using the following commands:

```
ipl_auction_df = pd.read_csv( 'IPL IMB381IPL2013.csv' )
ipl_auction_df.info()
```

We will use only a subset of features for building the model. The list *X_features* is initialized with the name of the features to be used as described below.

```
X_features = ['AGE', 'COUNTRY', 'PLAYING ROLE','T-RUNS', 'T-WKTS',
              'ODI-RUNS-S', 'ODI-SR-B','ODI-WKTS', 'ODI-SR-BL',
              'CAPTAINCY EXP', 'RUNS-S','HS', 'AVE', 'SR-B', 'SIX-
              ERS', 'RUNS-C', 'WKTS', 'AVE-BL', 'ECON', 'SR-BL']
```

Out of these, there are four categorical features that need to be encoded into dummy features using OHE (One Hot Encoding). The details of encoding categorical features are already discussed in Chapter 4.

```
# Initialize a list with the categorical feature names.
categorical_features = ['AGE', 'COUNTRY', 'PLAYING ROLE',
                                          'CAPTAINCY EXP']
#get_dummies() is invoked to return the dummy features.
ipl_auction_encoded_df = pd.get_dummies( ipl_auction_df[X_features],
                                         columns = categorical_
                                         features,
                                         drop_first = True )
```

To display all feature names along with the new dummy features we use the following code:

```
ipl_auction_encoded_df.columns
```

```
Index(['T-RUNS', 'T-WKTS', 'ODI-RUNS-S', 'ODI-SR-B', 'ODI-WKTS',
       'ODI-SR-BL','RUNS-S', 'HS', 'AVE', 'SR-B', 'SIXERS', 'RUNS-
       C, 'WKTS', 'AVE-BL','ECON', 'SR-BL', 'AGE_2', 'AGE_3',
       'COUNTRY_BAN', 'COUNTRY_    ENG','COUNTRY_IND', 'COUNTRY_NZ',
       'COUNTRY_PAK', 'COUNTRY_SA', 'COUNTRY_SL','COUNTRY_WI',
       'COUNTRY_ZIM', 'PLAYING ROLE_Batsman','PLAYING ROLE_Bowler',
       'PLAYING ROLE_W. Keeper', 'CAPTAINCY EXP_1'],
      dtype='object')
```

We will create two variables *X* and *Y* for building models. So, initialize *X* to values from all the above features and *Y* to *SOLD PRICE*, the outcome variable.

```
X = ipl_auction_encoded_df
Y = ipl_auction_df['SOLD PRICE']
```

6.4.1.2 Standardization of X and Y

Standardization is the process of bringing all features or variables into one single scale (normalized scale). This can be done by subtracting mean from the values and dividing by the standard deviation of the feature or variable. Standardizing helps to manage difference in scales of measurements of different variables. *StandardScaler* available in *sklearn.preprocessing* package provides this functionality.

```
from sklearn.preprocessing import StandardScaler
```

```
## Initializing the StandardScaler
X_scaler = StandardScaler()
## Standardize all the feature columns
X_scaled = X_scaler.fit_transform(X)

## Standardizing Y explictly by subtracting mean and
## dividing by standard deviation
Y = (Y - Y.mean()) / Y.std()
```

6.4.1.3 Split the Dataset into Train and Test

Split the dataset into train and test with 80:20 split. *random_state* (seed value) is set to 42 for reproducibility of exact results.

```
from sklearn.model_selection import train_test_split
```

```
X_train, X_test, y_train, y_test = train_ test_split(
X_scaled,
Y,
test_size=0.2,
random_state = 42)
```

6.4.1.4 Build the Model

SGDRegressor in *sklearn.linear_model* is a variation of gradient descent algorithm for building linear regression model. Gradient descent algorithm uses all the training examples to learn to minimize the cost function, whereas *SGDRegressor* (Stochastic Gradient Descent) uses a subset of examples in each iteration for learning.

sklearn.linear_model also provides *LinearRegression* to build a linear regression model. We can initialize the *LinearRegression* class and then call *fit()* and pass *X_train* and *y_train* to build the model.

```
from sklearn.linear_model import LinearRegression
```

```
linreg = LinearRegression()
linreg.fit(X_train, y_train)
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1,
normalize=False)
```

Let us print the coefficient using the following code:

```
linreg.coef_
```

```
array([-0.43539611, -0.04632556,  0.50840867, -0.03323988,  0.2220377,
       -0.05065703,  0.17282657, -0.49173336,  0.58571405, -0.11654753,
        0.24880095,  0.09546057,  0.16428731,  0.26400753, -0.08253341,
       -0.28643889, -0.26842214, -0.21910913, -0.02622351,  0.24817898,
        0.18760332,  0.10776084,  0.04737488,  0.05191335,  0.01235245,
        0.00547115, -0.03124706,  0.08530192,  0.01790803, -0.05077454,
        0.18745577] )
```

The sign of the coefficients indicates positive or negative effect on a player's *SOLD PRICE*. We will store the beta coefficients and respective column names in a DataFrame and then sort the coefficient values in descending order to observe the effects.

```
## The dataframe has two columns to store feature name and the
## corresponding coefficient values
columns_coef_df = pd.DataFrame( { 'columns': ipl_auction_encoded_
                                            df.columns,
                                  'coef': linreg.coef_ } )
```

```
## Sorting the features by coefficient values in descending order
sorted_coef_vals = columns_coef_df.sort_values( 'coef',ascending=False)
```

6.4.1.5 *Plotting the Coefficient Values*

The feature names and the coefficients are plotted using a bar plot to observe the effect (Figure 6.10). The vertical axis is the feature name and the horizontal axis is the coefficient value. As the features are standardized, the magnitude of the values also indicates the effect on outcome i.e. *SOLD PRICE*. The following commands are used to plot the coefficients of values:

```
plt.figure( figsize = ( 8, 6 )) ## Creating a bar plot
sn.barplot(x="coef", y="columns", data=sorted_coef_vals);
plt.xlabel("Coefficients from Linear Regression")
plt.ylabel("Features")
```

FIGURE 6.10 Bar plot depicting coefficient values of features in the model, sorted in descending order.

Few observations from Figure 6.10 are as follows:

1. AVE, ODI-RUNS-S, SIXERS are top three highly influential features which determine the player's *SOLD PRICE*.

2. Higher ECON, SR-B and AGE have negative effect on *SOLD PRICE*.

3. Interestingly, higher test runs (T-Runs) and highest score (HS) have negative effect on the *SOLD PRICE*. Note that few of these counter-intuitive sign for coefficients could be due to multicollinearity. For example, we expect SR-B (batting strike rate) to have a positive effect on the SOLD PRICE.

6.4.1.6 *Calculate RMSE*

We can calculate the RMSE on training and test sets to understand the model's ability to predict *SOLD PRICE*. We will develop an utility method *get_train_test_rmse()* to calculate and print the RMSE of train and test sets for comparison. It will take the model as a parameter. This will be used in all the models that we will be discussing in this chapter.

```
from sklearn import metrics

# Takes a model as a parameter
# Prints the RMSE on train and test set
def get_train_test_rmse( model ):
    # Predicting on training dataset
    y_train_pred = model.predict( X_train )
    # Compare the actual y with predicted y in the training dataset
    rmse_train = round(np.sqrt(metrics.mean_squared_error( y_train,
                                                            y_train_
                                                            pred)),3)
    # Predicting on test dataset
    y_test_pred = model.predict( X_test )
    # Compare the actual y with predicted y in the test dataset
    rmse_test = round(np.sqrt(metrics.mean_squared_error(y_test,
                                                          y_test_
                                                          pred)),3)
    print( "train: ", rmse_train, " test:", rmse_test )
```

Invoke the method *get_train_test_rmse()* with the model *linreg* as a parameter to get train and test accuracy.

```
get_train_test_rmse( linreg )
```

```
train: 0.679  test: 0.749
```

RMSE on the training set is 0.679, while it is 0.749 on the test set. A good model that generalizes well needs to have a very similar error on training and test sets. Large difference indicates that the model may be overfitting to the training set. Most widely used approach to deal with model overfitting is called *Regularization*, which will be discussed in the next section.

6.4.2 | Applying Regularization

One way to deal with overfitting is regularization. It is observed that overfitting is typically caused by inflation of the coefficients. To avoid overfitting, the coefficients should be regulated by penalizing potential inflation of coefficients. Regularization applies penalties on parameters if they inflate to large values and keeps them from being weighted too heavily.

The coefficients are penalized by adding the coefficient terms to the cost function. If the coefficients become large, the cost increases significantly. So, the optimizer controls the coefficient values to minimize the cost function. Following are the two approaches that can be used for adding a penalty to the cost function:

1. **L1 Norm:** Summation of the absolute value of the coefficients. This is also called Least Absolute Shrinkage and Selection Operator (LASSO Term) (Tibshirani, 1996). The corresponding cost function is given by

$$\varepsilon_{\text{mse}} = \frac{1}{N}\sum_{i=1}^{N}(Y_i - (\beta_0 + \beta_1 X_1 + \cdots s + \beta_n X_n))^2 + \alpha\sum_{i=0}^{n}|\beta_i| \tag{6.10}$$

   where α is the multiplier.

2. **L2 Norm:** Summation of the squared value of the coefficients. This is called *Ridge Term* (Hoerl A E and Kennard Kennard, 1970). The cost function is given by

$$\varepsilon_{\text{mse}} = \frac{1}{N}\sum_{i=1}^{N}(Y_i - (\beta_0 + \beta_1 X_1 + \cdots + \beta_n X_n))^2 + \alpha\sum_{i=1}^{n}\beta_i^2 \tag{6.11}$$

Ridge term distributes (smoothens) the coefficient values across all the features, whereas LASSO seems to reduce some of the coefficients to zero. Features with coefficients value as zero can be treated as features with no contribution to the model. So, LASSO can also be used for feature selection, that is, remove features with zero coefficients, thereby reducing the number of features.

Figure 6.11 depicts the LASSO and Ridge constraint applied to the cost function (James *et al.*, 2013).

FIGURE 6.11 Effect of LASSO and Ridge terms on the cost function.

6.4.2.1 Ridge Regression

sklearn.linear_model provides Ridge regression for building linear models by applying L2 penalty.

Ridge regression takes the following parameters:

1. *alpha* (α) – float – is the regularization strength; regularization strength must be a positive float. Regularization improves the estimation of the parameters and reduces the variance of the estimates. Larger values of alpha imply stronger regularization.
2. *max_iter* – int (integer) – is the maximum number of iterations for the gradient solver.

```
# Importing Ridge Regression
from sklearn.linear_model import Ridge

# Applying alpha = 1 and running the algorithms for maximum of 500
# iterations
ridge = Ridge(alpha = 1, max_iter = 500)
ridge.fit( X_train, y_train )
```

```
Ridge(alpha=1, copy_X=True, fit_intercept=True, max_iter=500,
normalize=False, random_state=None, solver='auto', tol=0.001)
```

```
get_train_test_rmse( ridge )
```

```
train: 0.68  test: 0.724
```

The difference in RMSE on train and test has reduced because of penalty effect. The difference can be reduced by applying a stronger penalty. For example, apply α value as 2.0.

```
ridge = Ridge(alpha = 2.0, max_iter = 1000)
ridge.fit( X_train, y_train )
get_train_test_rmse( ridge )
```

```
train: 0.682       test: 0.706
```

The difference in model accuracy on training and test has reduced. We need to calculate the optimal value for α. This can be achieved in many ways. Multiple values of α can be tested before arriving at the optimal value. The parameters which can be tuned are called hyperparameters in machine learning. Here α is a hyperparameter.

sklearn.model_selection.GridSearchCV can help search for the optimal value and will be discussed later in the chapter. For now, let us assume the optimal value for α is 2.0.

6.4.2.2 LASSO Regression

sklearn.linear_model provides LASSO regression for building linear models by applying L1 penalty. Two key parameters for LASSO regression are:

1. *alpha* – float – multiplies the L1 term. Default value is set to 1.0.
2. *max_iter* – int – Maximum number of iterations for gradient solver.

```
# Importing LASSO Regression
from sklearn.linear_model import Lasso

# Applying alpha = 1 and running the algorithms for maximum of 500
# iterations
lasso = Lasso(alpha = 0.01, max_iter = 500)
lasso.fit( X_train, y_train )
```

```
Lasso(alpha=0.01, copy_X=True, fit_intercept=True, max_iter=500,
      normalize=False, positive=False, precompute=False, random_
      state=None, selection='cyclic', tol=0.0001, warm_start=False)
```

```
get_train_test_rmse( lasso )
```

```
train: 0.688        test: 0.698
```

It can be noticed that the model is not overfitting and the difference between train RMSE and test RMSE is very small. LASSO reduces some of the coefficient values to 0, which indicates that these features are not necessary for explaining the variance in the outcome variable.

We will store the feature names, coefficient values in a DataFrame and then filter the features with zero coefficients.

```
## Storing the feature names and coefficient values in the DataFrame
lasso_coef_df = pd.DataFrame( { 'columns': ipl_auction_encoded_
                                df.columns,
                                'coef': lasso.coef_ } )
```

```
## Filtering out coefficients with zeros
lasso_coef_df[lasso_coef_df.coef == 0]
```

	coef	columns
1	0.0	T-WKTS
3	0.0	ODI-SR-B
13	0.0	AVE-BL
28	0.0	PLAYING ROLE_Bowler

The LASSO regression indicates that the features listed under "columns" are not influencing factors for predicting the *SOLD PRICE* as the respective coefficients are 0.0.

6.4.2.3 Elastic Net Regression

ElasticNet regression combines both L1 and L2 regularizations to build a regression model. The corresponding cost function is given by

$$\varepsilon_{\text{mse}} = \frac{1}{N}\sum_{i=1}^{N}(Y_i - (\beta_0 + \beta_1 X_1 + \cdots + \beta_n X_n))^2 + \gamma\sum_{i=0}^{n}|\beta_i| + \sigma\sum_{i=1}^{n}\beta_i^2 \tag{6.12}$$

While building *ElasticNet* regression model, both hyperparameters σ (L2) and γ (L1) need to be set. *ElasticNet* takes the following two parameters:

1. *alpha* – Constant that multiplies the penalty terms. Default value is set to 1.0. (alpha = $\sigma + \gamma$)
2. *l1_ratio* – The ElasticNet mixing parameter, with 0 <= *l*1_ratio <= 1.

$$l1_ratio = \frac{\gamma}{\sigma + \gamma}$$

where
*l*1_ratio = 0 implies that the penalty is an L2 penalty.
*l*1_ratio = 1 implies that it is an L1 penalty.
0 < *l*1_ratio < 1 implies that the penalty is a combination of L1 and L2.

In the example below, penalties applied are $\gamma = 0.01$ and $\sigma = 1.0$. So

$$alpha = \sigma + \gamma = 1.01 \text{ and } l1_\text{ratio} = \frac{\gamma}{\sigma + \gamma} = 0.0099$$

```
from sklearn.linear_model import ElasticNet

enet = ElasticNet(alpha = 1.01, l1_ratio = 0.001, max_iter = 500)
enet.fit( X_train, y_train )
get_train_test_rmse( enet )
```

```
train: 0.789        test: 0.665
```

As we can see, applying both the regularizations did not improve the model performance. It has become worse. In this case, we can choose to apply only L1 (LASSO) regularization, which seems to deal with the overfitting problem efficiently.

In this section, we will take a binary classification problem and explore it through machine learning algorithms such as K-Nearest Neighbors (KNN), Random Forest, and Boosting.

Bank marketing dataset available at the University of California, Irvine machine learning repository is used in this section for the demonstration of various techniques.

The dataset is based on a telemarketing campaign carried out by a Portuguese bank for subscription of a term deposit.

The data has several features related to the potential customers and whether they subscribed the term deposit or not (outcome).

The objective, in this case, is to predict which customers may respond to their marketing campaign to open a term deposit with the bank.

The response variable $Y = 1$ implies that the customer subscribed a term deposit after the campaign and 0 otherwise. The marketing campaign is based on phone calls.

We can use the following commands for reading the dataset and printing a few records.

```
bank_df = pd.read_csv('bank.csv')
bank_df.head(5)
```

	age	job	marital	education	default	balance	housing-loan	personal-loan	current-campaign	previous-campaign	subscribed
0	30	unemployed	married	primary	no	1787	no	no	1	0	no
1	33	services	married	secondary	no	4789	yes	yes	1	4	no
2	35	management	single	tertiary	no	1350	yes	no	1	1	no
3	30	management	married	tertiary	no	1476	yes	yes	4	0	no
4	59	blue-collar	married	secondary	no	0	yes	no	1	0	no

The following commands can be used for displaying information about the dataset.

```
bank_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4521 entries, 0 to 4520
Data columns (total 11 columns):
 Age                 4521  non-null  int64
 Job                 4521  non-null  object
 Marital             4521  non-null  object
 Education           4521  non-null  object
 Default             4521  non-null  object
 Balance             4521  non-null  int64
 housing-loan        4521  non-null  object
 personal-loan       4521  non-null  object
 current-campaign    4521  non-null  int64
 previous-campaign   4521  non-null  int64
 Subscribed          4521  non-null  object
dtypes: int64(4), object(7)
memory usage: 388.6+ KB
```

The dataset has 4521 observations and 10 features excluding the *subscribed* column (outcome variable). The details of the columns are given in Chapter 5, *Classification*.

6.5.1 | Dealing with Imbalanced Datasets

One of the major problems in machine learning is imbalanced (or unbalanced) dataset. A dataset is imbalanced when there is no equal representation of all classes in the data. For example, in the bank marketing dataset the proportion of customers who responded to the telemarketing is approximately 11.5% and the remaining 88.5% did not respond. Thus, the representation of two classes (responded, did not respond) in the dataset is not equal. In the data, the *subscribed* column indicates whether or not the customer has opened (subscribed) a term deposit account with the bank after the marketing campaign.

1. *yes* – the customer has opened the account
2. *no* – the customer has not opened the account

We have to check the number of records in each class to understand the imbalance. A simple *value_counts()* on the column values will provide the answer. The following command can be used for counting the number of records in each class:

```
bank_df.subscribed.value_counts()
```

```
no      4000
yes      521
Name: subscribed, dtype: int64
```

The dataset is quite imbalanced. Both the classes are not equally represented. There are only 521 (11.5%) observations in which customers have subscribed as opposed to 4000 observations where customers have not subscribed. In such cases, the model may not be able to learn and may be biased towards the class that is over-represented.

Even if the model predicts that no customer will subscribe (all negatives), it will have an accuracy of more than 88%. This is called *Accuracy Paradox*. But the objective of building a model here is to identify the customers who will subscribe to the term deposit (i.e., increase the number of *True Positives*).

One approach to deal with imbalanced dataset is bootstrapping. It involves resampling techniques such as *upsampling* and *downsampling*.

1. *Upsampling:* Increase the instances of under-represented minority class by replicating the existing observations in the dataset. Sampling with replacement is used for this purpose and is also called *Oversampling*.
2. *Downsampling:* Reduce the instances of over-represented majority class by removing the existing observations from the dataset and is also called *Undersampling*.

sklearn.utils has *resample* method to help with upsampling. It takes three parameters:

1. The original sample set
2. *replace*: Implements resampling with replacement. If false, all resampled examples will be unique.
3. *n_samples*: Number of samples to generate.

In this case, the number of examples of *yes* cases will be increased to 2000.

```
## Importing resample from *sklearn.utils* package.
from sklearn.utils import resample

# Separate the case of yes-subscribes and no-subscribes
bank_subscribed_no = bank_df[bank_df.subscribed == 'no']
bank_subscribed_yes = bank_df[bank_df.subscribed == 'yes']

##Upsample the yes-subscribed cases.
df_minority_upsampled = resample(bank_subscribed_yes,
                                 replace=True,
                                 n_samples=2000)
```

```
# Combine majority class with upsampled minority class
new_bank_df = pd.concat([bank_subscribed_no, df_minority_upsampled])
```

After upsampling, the *new_bank_df* contains 4000 cases of *subscribed* = *no* and 2000 cases of *subscribed* = *yes* in the ratio of 67:33. Before using the dataset, the examples can be shuffled to make sure they are not in a particular order. *sklearn.utils* has a method *shuffle()*, which does the shuffling.

```
from sklearn.utils import
shuffle new_bank_df = shuffle(new_bank_df)
```

We now assign all the features column names to *X_features* variable.

```
# Assigning list of all column names in the DataFrame
X_features = list( new_bank_df.columns )
# Remove the response variable from the list
X_features.remove( 'subscribed' )
X_features
```

```
['age',
 'job',
 'marital',
 'education',
 'default',
 'balance',
 'housing-loan',
 'personal-loan',
 'current-campaign',
 'previous-campaign']
```

The following command can be used to encode all the categorical features into dummy features and assign to *X*.

```
## get_dummies() will convert all the columns with data type as
## objects
encoded_bank_df = pd.get_dummies( new_bank_df[X_features],
                                  drop_first = True )
X = encoded_bank_df
```

The **subscribed** column values are string literals and need to be encoded as follows:

1. **yes** to 1
2. **no** to 0

```
# Encoding the subscribed column and assigning to Y
Y = new_bank_df.subscribed.map( lambda x: int( x == 'yes') )
```

Now split the dataset into train and test sets in 70:30 ratio, respectively.

```
from sklearn.model_selection import train_test_split
train_X, test_X, train_y, test_y = train_test_split( X,Y,test_size
= 0.3,random_state = 42)
```

6.5.2 | Logistic Regression Model

6.5.2.1 Building the Model

Logistic regression is discussed in detail in Chapter 5, *Classification*. Cost function for logistic regression is called log loss (log likelihood) or binary cross-entropy function and is given by

$$-\frac{1}{N}\sum_{i=1}^{N}(y_i \ln(p_i)+(1-y_i)\ln(1-p_i)) \tag{6.13}$$

where p_i is the probability that Y belongs to class 1. It is given by

$$p_i = P(y_i = 1) = \frac{e^Z}{1+e^Z}$$

$$Z = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_m X_m$$

where $X_1, X_2, \ldots, X_m$ are features.

LogisticRegression is imported from *sklearn.linear_model* package and is fitted with the training data.

```
from sklearn.linear_model import LogisticRegression

## Initializing the model
logit = LogisticRegression()
## Fitting the model with X and Y values of the dataset
logit.fit( train_X, train_y)
```

Now we use the model to predict on the test set. *pred_y* will be assigned to the predicted classes. Note that we are not predicting class directly, which in turn predicts probabilities and determines the class by using 0.5 as a cut-off probability (or an optimal cut-off).

```
pred_y = logit.predict(test_X)
```

6.5.2.2 Confusion Matrix

We develop a custom method *draw_cm()* to draw the confusion matrix. This method will be used to draw the confusion matrix from the models we discuss in the subsequent sections of the chapter. It takes the actual and predicted class labels to draw the confusion matrix. Usage and interpretation of a confusion matrix is already explained in detail in Chapter 5.

```
## Importing the metrics
from sklearn import metrics

## Defining the matrix to draw the confusion matrix from actual and
## predicted class labels
def draw_cm( actual, predicted ):
    # Invoking confusion_matrix from metric package. The matrix
    # will be oriented as [1,0] i.e. the classes with label 1 will be
    # represented by the first row and 0 as second row
    cm = metrics.confusion_matrix( actual, predicted, [1,0] )
    # Confusion will be plotted as heatmap for better visualization
    # The labels are configured to better interpretation from the plot
    sn.heatmap(cm, annot=True,     fmt='.2f',
               xticklabels = ["Subscribed", "Not Subscribed"],
               yticklabels = ["Subscribed", "Not Subscribed"] )
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.show()
```

Plot the confusion matrix using the following command:

```
cm = draw_cm( test_y, pred_y )
```

FIGURE 6.12 Confusion matrix of logistic regression model.

Figure 6.12 shows that the model has very few true positives and a large number of false negatives. In this business context, the objective is to build a model that will have a high percentage of true positives.

In the confusion matrix in Figure 5.2, the columns represent the predicted label (class), while the rows represent the actual label (class).

1.Left-top quadrant represents actual bad credit and is correctly classified as bad credit. This is called True Positives (TP).

2. Left-down quadrant represents actual good credit and is incorrectly classified as bad credit. This is called False Positives (FP).

3. Right-top quadrant represents actual bad credit and is incorrectly classified as good credit. This is called False Negatives (FN).

4. Right-down quadrant represents actual good credit and is correctly classified as good credit. This is called True Negatives (TN).

5.3.8 | Measuring Accuracies

In classification, the model performance is often measured using concepts such as sensitivity, specificity, precision, and F-score. The ability of the model to correctly classify positives and negatives is called **sensitivity** (also known as recall or true positive rate) and **specificity** (also known as true negative rate), respectively. The terminologies sensitivity and specificity originated in medical diagnostics.

Sensitivity or Recall (True Positive Rate)

Sensitivity is the conditional probability that the predicted class is positive given that the actual class is positive. Mathematically, sensitivity is given by

$$\text{Sensitivity} = \frac{TP}{TP + FN} \tag{5.3}$$

Specificity (True Negative Rate)

Specificity is the conditional probability that the predicted class is negative given that the actual class is negative. Mathematically, specificity is given by

$$\text{Specificity} = \frac{TN}{TN + FP} \tag{5.4}$$

Precision

Precision is the conditional probability that the actual value is positive given that the prediction by the model is positive. Mathematically, precision is given by

$$\text{Precision} = \frac{TP}{TP + FP} \tag{5.5}$$

F-Score

F-Score is a measure that combines precision and recall (harmonic mean between precision and recall). Mathematically, F-Score is given by

$$\text{F-Score} = \frac{2 \times \text{Recall} \times \text{Precision}}{\text{Recall} + \text{Precision}} \tag{5.6}$$

classification_report() method in *skearn.metrics* gives a detailed report of precision, recall, and F-score for each class.

```
print( metrics.classification_report( y_pred_df.actual,
                                      y_pred_df.predicted ) )
```

6.5.2.3 Classification Report

The *classification_report()* function in *skearn.metrics* gives a detailed report of precision, recall and F1-score for each class. The report is shown in Table 6.1.

```
print( metrics.classification_report( test_y, pred_y ) )
```

TABLE 6.1 Model performance metrics

	precision	recall	f1-score	Support
0	0.72	0.93	0.81	1225
1	0.62	0.25	0.35	575
avg/total	0.69	0.71	0.67	1800

From Table 6.1, we learn that recall for positive cases is only 0.25. Most of the cases have been predicted as negative.

5.3.9 | Receiver Operating Characteristic (ROC) and Area Under the Curve (AUC)

Receiver operating characteristic (ROC) curve can be used to understand the overall performance (worth) of a logistic regression model (and, in general, of classification models) and used for model selection.

The term has its origin in electrical engineering when electrical signals were used for predicting enemy objects (such as submarines and aircraft) during World War II.

Given a random pair of positive and negative class records, ROC gives the proportions of such pairs that will be correctly classified.

ROC curve is a plot between sensitivity (true positive rate) on the vertical axis and 1 – specificity (false positive rate) on the horizontal axis.

We will write a method *draw_roc()* which takes the actual classes and predicted probability values and then draws the ROC curve (Figure 5.4).

metrics.roc_curve() returns different threshold (cut-off) values and their corresponding false positive and true positive rates.

Then these values can be taken and plotted to create the ROC curve. *metrics. roc_auc_score()* returns the area under the curve (AUC).

FIGURE 5.4 ROC curve.

The diagonal line in Figure 5.4 represents the case of not using a model (no discrimination between positive and negative); the area below the diagonal line is equal to 0.5 (it is a right-angle triangle, area of right-angle triangle is $0.5ab$, where a and b are the lengths of the sides which is equal to 1 in this case). Sensitivity and/or specificity are likely to change when the cut-off probability is changed. The line above the diagonal line in Figure 5.4 captures how sensitivity and (1 – specificity) change when the cut-off probability is changed. Model with higher AUC is preferred and AUC is frequently used for model selection.

As a thumb rule, AUC of at least 0.7 is required for practical application of the model. AUC greater than 0.9 implies an outstanding model. Caution should be exercised while selecting models based on AUC, especially when the data is imbalanced (i.e., dataset which has less than 10% positives). In case of imbalanced datasets, the AUC may be very high (greater than 0.9); however, either sensitivity or specificity values may be poor.

For this example, the AUC is 0.78, which implies the model is fairly good. The AUC can also be obtained by calling *roc_auc_score* from *sklearn.metrics*.

```
auc_score = metrics.roc_auc_score( y_pred_df.actual,
                                   y_pred_df.predicted_prob )
round( float( auc_score ), 2 )
```

```
0.78
```

6.5.2.4 Receiver Operating Characteristic Curve (ROC) and Area under ROC (AUC) Score

ROC and AUC are two important measures of model performance for classification problems. The model's *predict_proba()* method gives the predicted probabilities for the test examples and can be passed to *roc_auc_score()* along with actual class labels to obtain AUC score. AUC score is explained in detail in Chapter 5.

```
## Predicting the probability values for test cases
predict_proba_df = pd.DataFrame( logit.predict_proba( test_X ) )
predict_proba_df.head()
```

	0	1
0	0.505497	0.494503
1	0.799272	0.200728
2	0.646329	0.353671
3	0.882212	0.117788
4	0.458005	0.541995

As shown in the result printed above, the second column in the DataFrame *predict_proba_df* has the probability for class label 1.

Now we create a DataFrame *test_results_df* to store the actual labels and predicted probabilities for class label 1.

```
## Initializing the DataFrame with actual class labels
test_results_df = pd.DataFrame( { 'actual': test_y } )
test_results_df = test_results_df.reset_index()
## Assigning the probability values for class label 1
test_results_df['chd_1'] = predict_proba_df.iloc[:,1:2]
```

Let us print the first 5 records.

```
test_results_df.head(5)
```

	index	actual	chd_1
0	2022	0	0.494503
1	4428	0	0.200728
2	251	0	0.353671
3	2414	0	0.117788
4	4300	1	0.541995

The DataFrame *test_results_df* contains the test example index, the actual class label and predicted probabilities for class 1 in columns *index, actual* and *chd_1*, respectively.

The ROC AUC score can be obtained using *metrics.roc_auc_score()*.

```
# Passing actual class labels and predicted probability values
# to compute ROC AUC score.

auc_score = metrics.roc_auc_score(test_results_df.actual,
                                  test_results_df.chd_1)
round( float( auc_score ), 2 )
```

```
0.7
```

That is, AUC score is 0.7.

Plotting ROC Curve

To visualize the ROC curve, an utility method *draw_roc_curve()* is implemented, which takes the mode, test set and actual labels of test set to draw the ROC curve. It returns the *auc_score*, false positive rate (FPR), true positive rate (TPR) values for different threshold (cut-off probabilities) ranging from 0.0 to 1.0. This method will be used for all future ML models that we will be discussing in subsequent sections. ROC AUC curve is discussed in detail in Section 5.2.10 (ROC and AUC), Chapter 5.

The following custom method is created for plotting ROC curve and calculating the area under the ROC curve.

```
## The method takes the following three parameters
## model: the classification model
  ## test_X: X features of the test set
  ## test_y: actual labels of the test set
  ## Returns
  ##  - ROC Auc Score
  ##  - FPR and TPRs for different threshold values
  def draw_roc_curve( model, test_X, test_y ):
      ## Creating and initializing a results DataFrame with actual
      ## labels
      test_results_df = pd.DataFrame( { 'actual': test_y } )
      test_results_df = test_results_df.reset_index()

    # predict the probabilities on the test set
    predict_proba_df = pd.DataFrame( model.predict_proba( test_X ) )

    ## selecting the probabilities that the test example belongs
    ## to class 1
    test_results_df['chd_1'] = predict_proba_df.iloc[:,1:2]

    ## Invoke roc_curve() to return fpr, tpr and threshold values.
    ## Threshold values contain values from 0.0 to 1.0
    fpr, tpr, thresholds = metrics.roc_curve( test_results_
                                              df.actual,
                                              test_results_
                                              df.chd_1,
                                              drop_intermediate =
                                              False )

    ## Getting roc auc score by invoking metrics.roc_auc_score method
    auc_score = metrics.roc_auc_score( test_results_df.actual,
                                       test_results_df.c hd_1 )

    ## Setting the size of the plot
    plt.figure(figsize=(8, 6))
    ## Plotting the actual fpr and tpr values
    plt.plot(fpr, tpr, label = 'ROC curve (area = %0.2f)' % auc_score)
    ## Plotting th diagnoal line from (0,1)
    plt.plot([0, 1], [0, 1], 'k--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    ## Setting labels and titles
    plt.xlabel('False Positive Rate or [1 - True Negative Rate]')
```

```
    plt.ylabel('True Positive Rate')
    plt.title('Receiver operating characteristic example')
    plt.legend(loc="lower right")
    plt.show()

    return auc_score, fpr, tpr, thresholds
```

The plot of ROC curve is shown in Figure 6.13. The corresponding AUC value is 0.70.

```
## Invoking draw_roc_curve with the logistic regression model
_, _, _, _ = draw_roc_curve( logit, test_X, test_y )
```

FIGURE 6.13 ROC AUC curve of logistic regression model.

6.5.3 | K-Nearest Neighbors (KNN) Algorithm

K-Nearest Neighbors (KNN) algorithm is a non-parametric, lazy learning algorithm used for regression and classification problems. Machine learning algorithms are of two types: parametric and non-parametric.

1. Parametric models estimate a fixed number of parameters from the data and strong assumptions of the data. The data is assumed to be following a specific probability distribution. Logistic regression is an example of a parametric model.
2. Non-parametric models do not make any assumptions on the underlying data distribution (such as normal distribution). KNN memorizes the data and classifies new observations by comparing the training data.

KNN algorithm finds observations in the training set, which are similar to the new observation. These observations are called neighbors. For better accuracy, a set of neighbors (K) can be considered for classifying a new observation. The class for the new observation can be predicted to be same class that majority of the neighbors belong to.

In Figure 6.14, observations belong to two classes represented by triangle and circle shapes. To find the class for a new observation, a set of neighbors, marked by the circle, are examined. As a majority of the neighbors belong to class B, the new observation is classified as *class B*.

FIGURE 6.14 KNN algorithm.

The neighbors are found by computing distance between observations. Euclidean distance is one of the most widely used distance metrics. It is given by

$$D(O_1, O_2) = \sqrt{(X_{11} - X_{21})^2 + (X_{12} - X_{22})^2} \tag{6.14}$$

where O_1 and O_2 are two observations in the data. X_{11}, X_{21} are the values of feature X_1 for records 1 and 2, respectively, and X_{12} and X_{22} are the values of feature X_2 for records 1 and 2, respectively. Few more distance metrics such as Minkowski distance, Jaccard Coefficient and Gower's distance are used. For better understanding of these distances, refer to Chapter 14: Clustering of *Business Analytics: The Science of Data-Driven Decision Making* by U Dinesh Kumar (2017).

sklearn.neighbors provides *KNeighborsClassifier* algorithm for classification problems. *KNeighborsClassifier* takes the following parameters:

1. n_neighbors: int – Number of neighbors to use by default. Default is 5.
2. metric: string – The distance metrics. Default 'Minkowski'. Available distance metrics in *sklearn* are discussed at the *Scikit-learn* site (see Reference section).
3. weights : str – Default is uniform where all points in each neighborhood are weighted equally. Else the distance which weighs points by the inverse of their distance.

Illustration of applying KNN algorithms to the bank marketing dataset is described below. The following default values are used to run the algorithm.

1. n_neighbors = 5
2. metric = 'minkowski'

```
## Importing the KNN classifier algorithm
from sklearn.neighbors import KNeighborsClassifier

## Initializing the classifier
knn_clf = KNeighborsClassifier()
## Fitting the model with the training set
knn_clf.fit( train_X, train_y )
```

```
KNeighborsClassifier(algorithm='auto', leaf_size=30,
metric='minkowski',metric_params=None, n_jobs=1, n_neighbors=5, p=2,
weights='uniform')
```

6.5.3.1 KNN Accuracy

We find ROC AUC score and draw the ROC curve (Figure 6.15).

```
## Invoking draw_roc_curve with the KNN model
_, _, _, _ = draw_roc_curve( knn_clf, test_X, test_y )
```

FIGURE 6.15 ROC AUC curve of KNN model.

As shown in Figure 6.15, KNN has AUC score of 0.84 and is better than the Logistic Regression Model. The confusion matrix can be created using the following commands:

```
## Predicting on test set
pred_y = knn_clf.predict(test_X)
## Drawing the confusion matrix for KNN model
draw_cm( test_y, pred_y )
```

FIGURE 6.16 Confusion matrix of KNN model.

Let us print the classification report for the KNN model.

```
print( metrics.classification_report( test_y, pred_y ) )
```

TABLE 6.2 Classification report for the KNN model

	precision	recall	f1-score	support
0	0.87	0.76	0.81	1225
1	0.59	0.75	0.66	575
avg/total	0.78	0.76	0.76	1800

As shown in Table 6.2, the **recall** of positive cases has improved from 0.25 (logistic regression model) to 0.75 in the KNN model. The above model accuracy is obtained by considering the default number of neighbors (i.e., $k = 5$). Can the accuracy of the model be improved by increasing or reducing the number of neighbors? In other words, what is the most optimal number of neighbors (K) to be considered for classification in this case? *K in KNN is called hyperparameter and the process of finding optimal value for a hyperparameter is called hyperparameter tuning.*

sklearn has a *GridSearch mechanism* in which one or multiple hyperparameters can be searched through for the most optimal values, where the model gives the highest accuracy.

The search mechanism is a brute force approach, that is, evaluate all the possible values and find the most optimal ones.

6.5.3.2 GridSearch for Optimal Parameters

One of the problems in machine learning is the selection of optimal hyperparameters. *sklearn.model_selection* provides a feature called *GridSearchCV*, which searches through a set of given hyperparameter values and reports the most optimal one. *GridSearchCV* does *k*-fold cross-validation for each value of hyperparameter to measure accuracy and avoid overfitting.

GridSearchCV can be used for any machine learning algorithm to search for optimal values for its hyperparameters. *GridSearchCV* searches among a list of possible hyperparameter values and reports the best value based on accuracy measures. *GridSearchCV* takes the following parameters:

1. *estimator* – scikit-learn model, which implements estimator interface. This is the ML algorithm.
2. *param_grid* – A dictionary with parameter names (string) as keys and lists of parameter values to search for.
3. *scoring* – string – the accuracy measure. For example, '*r2*' for regression models and '*f1*', '*precision*', '*recall*' or *roc_auc*' for classification models.
4. *cv* – integer – the number of folds in K-fold.

In our example, the hyperparameters and corresponding set of values to search for are as follows:

1. *n_neighbhors* – All values from 5 to 10.
2. metric (for distance calculation) – '*canberra*', '*euclidean*', '*minkowski*'.

All possible combination of hyperparameters will be evaluated by *GridSearchCV*. Also, to measure the robustness of the model, each set of value will be evaluated by K-fold cross-validation. The accuracy measure reported will be the average accuracy across all the folds from K-fold cross-validation.

```
## Importing GridSearchCV
from sklearn.model_selection import GridSearchCV

## Creating a dictionary with hyperparameters and possible values
## for searching
tuned_parameters = [{'n_neighbors': range(5,10),
                     'metric': ['canberra', 'euclidean',
                     'minkowski']}]

## Configuring grid search
clf = GridSearchCV(KNeighborsClassifier(),
                   tuned_parameters,
                   cv=10,
                   scoring='roc_auc')
## fit the search with training set
clf.fit(train_X, train_y)
```

Module I - Forecasting

Dr. Varalatchoumy M
Prof.&Head – Dept. of AIML
Head –CHOSS,
Cambridge Institute of Technology, Bangalore

8.1 | FORECASTING OVERVIEW

Forecasting is by far the most important and frequently used application of **predictive analytics** because it has significant impact on both the top line and the bottom line of an organization.

Every organization prepares **long-range and short-range planning** and forecasting demand for product and service is an important input for both long-range and short-range planning.

Different **capacity planning problems** such as manpower planning, machine capacity, warehouse capacity, materials requirements planning (MRP) will depend on the forecasted demand for the product/service.

Budget allocation for marketing promotions and advertisements are usually made based on forecasted demand for the product.

8.2 | COMPONENTS OF TIME-SERIES DATA

The time-series data *Yt* is a random variable, usually collected at regular time intervals and in chronological order.

If the time-series data contains observations of just a single variable (such as demand of a product at time *t*), then it is termed as *univariate time-series data*.

If the data consists of more than one variable, for example, demand for a product at time *t*, price at time *t*, amount of money spent by the company on promotion at time *t*, competitors' price at time *t*, etc., then it is called *multivariate timeseries data*.

Table 1: Example of Time Series Data

Date	Week	Sales
1/4/2010	1	96,200
1/11/2010	2	98,491
1/18/2010	3	96,595
1/25/2010	4	93,511
...	...	...
12/5/2011	101	80,407
12/12/2011	102	81,479
12/19/2011	103	83,093
12/26/2011	104	72,752

A time-series data can be broken into the four following components:

1. **Trend Component (T_t):** Trend is the consistent long-term upward or downward movement of the data.
2. **Seasonal Component (S_t):** Seasonal component (measured using seasonality index) is the repetitive upward or downward movement (or fluctuations) from the trend that occurs within a calendar year at fixed intervals (i.e., time between seasons is fixed) such as seasons, quarters, months, days of the week, etc. The upward or downward fluctuation may be caused due to festivals, customs within a society, school holidays, business practices within the market such as "end of season sale", and so on.
3. **Cyclical Component (C_t):** Cyclical component is fluctuation around the trend line at random interval (i.e., the time between cycles is random) that happens due to macro-economic changes such as recession, unemployment, etc. Cyclical fluctuations have repetitive patterns with time between repetitions of more than a year. Whereas in the case of seasonality, the fluctuations are observed within a calendar year and are driven by factors such as festivals and customs that exist in a society. A major difference between seasonal fluctuation and cyclical fluctuation is that seasonal fluctuation occurs at fixed period within a calendar year, whereas cyclical fluctuations have random time between fluctuations. That is, the periodicity of seasonal fluctuations is constant, whereas the periodicity of cyclical fluctuations is not constant.
4. **Irregular Component (I_t):** Irregular component is the white noise or random uncorrelated changes that follow a normal distribution with mean value of 0 and constant variance.

Components of Time Series

(a) Trend

(b) Seasonality

(c) Cyclicality

(d) Irregular

There are several forecasting techniques such as moving average, exponential smoothing, and Auto-Regressive Integrated Moving Average (ARIMA) that are used across various industries.

Moving average and exponential smoothing predict the future value of a time-series data as a function of past observations.

8.3 | MOVING AVERAGE

To illustrate the moving average technique, we will use the demand dataset for *Kesh*, a shampoo brand which is sold in 100 ml bottles by We Sell Beauty (WSB), a manufacturer and distributor of health and beauty products. The dataset is provided in file *wsb.csv*.

8.3.1 | Loading and Visualizing the Time-Series Dataset

We load the data from file wsb.csv onto the DataFrame using *pd.read_csv()*.

```
import pandas as pd
wsb_df = pd.read_csv('wsb.csv')
wsb_df.head(10)
```

Now we print the first 10 records.

	Month	Sale Quantity	Promotion Expenses	Competition Promotion
0	1	3002666	105	1
1	2	4401553	145	0
2	3	3205279	118	1
3	4	4245349	130	0
4	5	3001940	98	1
5	6	4377766	156	0
6	7	2798343	98	1
7	8	4303668	144	0
8	9	2958185	112	1
9	10	3623386	120	0

To visualize *Sale Quantity* over *Month using plot() method* from Matplotlib use the following code:

```
import matplotlib.pyplot as plt
import seaborn as sn
%matplotlib inline
```

```
plt.figure(figsize=(10,4))
plt.xlabel("Months")
plt.ylabel("Quantity")
plt.plot(wsb_df['Sale Quantity']);
```

FIGURE 8.1 Sales quantity over 48 months.

In Figure 8.1, the sales quantity (vertical axis) shows a lot of fluctuations over the months. But there seems to be an increasing trend as well.

We now print the summary of the dataset using the *info()* method of the DataFrame.

```
wsb_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 48 entries, 0 to 47
Data columns (total 4 columns):
Month                   48 non-null int64
Sale Quantity           48 non-null int64
Promotion Expenses      48 non-null int64
Competition Promotion   48 non-null int64
dtypes: int64(4)
memory usage: 1.6 KB
```

The dataset contains quantity of 100 ml bottles sold during each month for a period of 48 months, promotion expenses incurred by the company, and whether the competition was on promotion (value of 1 implies that the competition was on promotion and 0 otherwise).

8.3.2 | Forecasting Using Moving Average

Moving average is one of the simplest forecasting techniques which forecasts the future value of a time-series data using average (or weighted average) of the past N observations. Forecasted value for time $t+1$ using the simple moving average is given by

$$F_{t+1} = \frac{1}{N} \sum_{k=t+1-N}^{t} Y_k \tag{8.1}$$

Pandas has a function *rolling()* which can be used with an aggregate function like *mean()* for calculating moving average for a time window. For example, to calculate 12 month's moving average using last 12 months' data starting from last month (previous period), *rolling()* will take a parameter *window*, which is set to 12 to indicate moving average of 12-months data, and then use Pandas' *shift()* function, which takes parameter 1 to specify that the 12-months data should start from last month. *shift(1)* means calculating moving average for the specified window period starting from previous observation (in this case last month).

```
wsb_df['mavg_12'] = wsb_df['Sale Quantity'].rolling(window = 12).
mean().shift(1)
```

To display values up to 2 decimal points, we can use *pd.set_option* and floating format %.2f.

```
pd.set_option('display.float_format', lambda x: '%.2f' % x)
wsb_df[['Sale Quantity', 'mavg_12']][36:]
```

Now print the forecasted value from the month 37 onwards.

	Sale Quantity	mavg_12
36	3216483	3928410.33
37	3453239	3810280.00
38	5431651	3783643.33
39	4241851	3970688.42
40	3909887	4066369.08
41	3216438	4012412.75
42	4222005	3962369.58
43	3621034	3946629.42
44	5162201	3940489.50
45	4627177	4052117.17
46	4623945	4130274.75
47	4599368	4204882.33

We use the following code to plot actual versus the predicted values from moving average forecasting:

```
plt.figure(figsize=(10,4))
plt.xlabel("Months")
plt.ylabel("Quantity")
plt.plot(wsb_df['Sale Quantity'][12:]);
plt.plot(wsb_df['mavg_12'][12:], '.');
plt.legend();
```

FIGURE 8.2 Actual versus forecasted sales quantity using moving average.

As observed in Figure 8.2, there is an increasing trend in sales quantity over the months.

8.3.3 | Calculating Forecast Accuracy

Root mean square error (RMSE) and mean absolute percentage error (MAPE) are the two most popular accuracy measures of forecasting. We will be discussing these measures in this section.

8.3.3.1 Mean Absolute Percentage Error

Mean absolute percentage error (MAPE) is the average of absolute percentage error. Assume that the validation data has n observations and forecasting is carried out on these n observations. The mean absolute percentage error is given by

$$\text{MAPE} = \frac{1}{n}\sum_{t=1}^{n}\frac{|Y_t - F_t|}{Y_t} \times 100\% \tag{8.2}$$

MAPE is one of the popular forecasting accuracy measures used by practitioners since it expresses the average error in percentage terms and is easy to interpret. Since MAPE is dimensionless, it can be used for comparing different models with varying scales.

The following custom method *get_mape()* takes the series of actual values and foretasted values, and returns the MAPE.

```
import numpy as np

def get_mape(actual, predicted):
    y_true, y_pred = np.array(actual), np.array(predicted)
    return np.round( np.mean(np.abs((actual - predicted) / actual))
    * 100, 2 )
```

We invoke the above method using column values of *wbd_df* DataFrame. "Sale Quantity" is passed as actual and *mavg_12* is passed as predicted values. Records from the 37th month are used to calculate MAPE.

```
get_mape( wsb_df['Sale Quantity'][36:].values,
          wsb_df['mavg_12'][36:].values)
```

The MAPE in this case is 14.04. So, forecasting using moving average gives us a MAPE of 14.04%.

8.3.3.2 Root Mean Square Error

Root mean square error (RMSE) is the square root of average of squared error calculated over the validation dataset, and is the standard deviation of the errors for unbiased estimator. RMSE is given by

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{t=1}^{n}(Y_t - F_t)^2} \tag{8.3}$$

Lower RMSE implies better prediction. However, it depends on the scale of the time-series data. MSE *(Mean Squared Error)* can be calculated using *mean_squared_error()* method in *sklearn.metrics.* We can pass MSE value to *np.sqrt()* to calculate RMSE.

```
from sklearn.metrics import mean_squared_error

np.sqrt(mean_squared_error(wsb_df['Sale Quantity'][36:].values,
                           wsb_df['mavg_12'][36:].values))
```

The RMSE in this case is 734725.83. So, the RMSE of the moving average model indicates that the prediction by the models has a standard deviation of 734725.83.

8.3.4 | Exponential Smoothing

One of the drawbacks of the simple moving average technique is that it gives equal weight to all the previous observations used in forecasting the future value. Exponential smoothing technique (also known as simple exponential smoothing; SES) assigns differential weights to past observations.

$$F_{t+1} = \alpha Y_t + (1-\alpha)F_t \tag{8.4}$$

where α is called the smoothing constant, and its value lies between 0 and 1. F_{t+1} is the forecasted value at time $t + 1$ using actual value Y_t at time t and forecasted values F_t of time t. But the model applies differential weights to both the inputs using smoothing constant α.

The *ewm()* method in Pandas provides the features for computing the exponential moving average taking *alpha* as a parameter.

```
wsb_df['ewm'] = wsb_df['Sale Quantity'].ewm( alpha = 0.2 ).mean()
```

Set the floating point formatting up to 2 decimal points.

```
pd.options.display.float_format = '{:.2f}'.format
```

Use the following code to display the records from the 37th month.

```
wsb_df[36:]
```

	Month	Sale Quantity	Promotion Expenses	Competition Promotion	mavg_12	Ewm
36	37	3216483	121	1	3928410.33	3828234.64
37	38	3453239	128	0	3810280.00	3753219.93
38	39	5431651	170	0	3783643.33	4088961.93
39	40	4241851	160	0	3970688.42	4119543.81
40	41	3909887	151	1	4066369.08	4077607.99
41	42	3216438	120	1	4012412.75	3905359.34
42	43	4222005	152	0	3962369.58	3968692.78
43	44	3621034	125	0	3946629.42	3899157.24
44	45	5162201	170	0	3940489.50	4151776.99
45	46	4627177	160	0	4052117.17	4246860.31
46	47	4623945	168	0	4130274.75	4322279.35
47	48	4599368	166	0	4204882.33	4377698.31

Now calculate MAPE of the model using records from 37th month.

```
get_mape(wsb_df[['Sale Quantity']][36:].values,
         wsb_df[['ewm']][36:].values)
```

```
The MAPE of the model is 11.15.
```

So, forecasting using exponential smoothing has about 11.15% error (MAPE) from the actual values. It is an improvement compared to the simple moving average model. Let us plot the output to view the difference between the forecasted and actual sales quantity using exponential moving average.

```
plt.figure( figsize=(10,4))
plt.xlabel( "Months" )
plt.ylabel( "Quantity" )
plt.plot( wsb_df['Sale Quantity'][12:] );
plt.plot( wsb_df['ewm'][12:], '.' );
plt.legend();
```

FIGURE 8.3 Forecasted versus actual sales quantity using exponential moving average.

In Figure 8.3, prediction by exponential moving average is shown by the dotted line and shows an increasing trend over the months. Moving average and simple exponential smoothing (SES) assume a fairly steady time-series data with no significant trend, seasonal or cyclical components, that is, the data is stationary. However, many dataset will have trend and seasonality.

8.4 | DECOMPOSING TIME SERIES

The time-series data can be modelled as addition or product of trend, seasonality, cyclical, and irregular components. The additive time-series model is given by

$$Y_t = T_t + S_t + C_t + I_t$$

The additive models assume that the seasonal and cyclical components are independent of the trend component. Additive models are not very common, since in many cases the seasonal component may not be independent of the trend.

The multiplicative time-series model is given by

$$Y_t = T_t \times S_t \times C_t \times I_t$$

Multiplicative models are more common and are a better fit for many datasets. For example, the seasonality effect on sales during festival times like Diwali does not result in constant increase in sales over the years. For example, the increase in sales of cars during festival season is not just 100 units every year. The seasonality effect has a multiplicative effect on sales based on the trend over the years like 10% additional units based on the trend in the current year. So, in many cases the seasonality effect is multiplied with the trend and not just added as in additive model.

In many cases, while building a forecasting model, only trend and seasonality components are used. To estimate the cyclical component, we will need a large dataset. For example, typical period of business cycles is about 58 months as per Investopedia (Anon, 2018). So, to understand the effect of cyclic component we will need observations spanning more than 10 years. Most of the times, data for such a long duration is not available.

For decomposing a time-series data, we can leverage the following libraries:

1. *statsmodel.tsa* provides various features for time-series analysis.
2. *seasonal_decompose()* in *statsmodel.tsa.seasonal* decomposes a time series into trend, seasonal, and residuals. It takes *frequency* parameters; for example, the frequency is 12 for monthly data.

The *plot()* function will render the trend, seasonal, and residuals, as shown in the following code:

```
from statsmodels.tsa.seasonal import seasonal_decompose

ts_decompse = seasonal_decompose(np.array(wsb_df['Sale Quantity']),
                                 model='multiplicative',
                                 freq = 12)

## Plotting the deocomposed time series components
ts_plot = ts_decompose.plot()
```

To capture the seasonal and trend components after time-series decomposition we use the following code. The information can be read from two variables *seasonal* and *trend* in *ts_decompose.*

```
wsb_df['seasonal'] = ts_decompse.seasonal
wsb_df['trend'] = ts_decompse.trend
```

FIGURE 8.4 Plots of decomposed time-series components

8.5 | AUTO-REGRESSIVE INTEGRATED MOVING AVERAGE MODELS

Auto-regressive (AR) and moving average (MA) models are popular models that are frequently used for forecasting.

AR and MA models are combined to create models such as auto-regressive moving average (ARMA) and auto-regressive integrated moving average (ARIMA) models.

ARMA models are basically regression models; auto-regression simply means regression of a variable on itself measured at different time periods.

Time Series and Forecasting

In mathematics, a time series is a series of data points indexed in time order.

Year	Profit
2001	6000
2002	5000
2003	7000
2004	7000
2005	6500
2006	8000

Time	Temperature
10:15 AM	87
10:20 AM	91
10:25 AM	98
10:30 AM	96
10:35 AM	95
10:40 AM	97

City	Latitude	Temperature
New York	41	76
Boston	43	72
Miami	26	83
Los Angeles	34	75
Dallas	33	85
Chicago	42	78

Forecasting is a technique that uses historical data as inputs to make predictions on future trends.

Time Series and Forecasting

Notations used in Time-Series

- **t** — An index denoting time period
- **y_t** — A series of n values measured over a time period
- **φ** — Coefficient Term for each value
- **c** — Constant Term in Equation
- **ε** — Forecast error for time period, t : $y_t - F_t$

What is AutoRegression (AR)?

Auto | Regression

Self

Numeric Value Prediction

What is Auto-Regression (AR)?

Autoregression is **a time series model** that uses observations from previous time steps as input to a regression equation to predict the value at the next time step. It is a very simple idea that can result in accurate forecasts on a range of time series problems.

$$y_t = c + \varphi_1 y_{t-1} + \varphi_2 y_{t-2} + \varphi_p y_{t-p} + \varepsilon$$

What is Stationarity?

The Statistical Properties of a process generating a time series do not change over time

- Mean and Variance remain Constant
- Time series with seasonality, is not stationary

Types of Autoregressive Models

YT -> TARGET
YT-I -> LAGGED TARGET
ε -> ERROR TERM
φ -> COEFFICIENT
C -> CONSTANT

How to Pick the Auto-Regressive Model?

PACF Plot

- PACF is the partial autocorrelation function that explains the partial correlation between the series and lags of itself.
- In simple terms, PACF can be explained using a linear regression where we predict y(t) from y(t-1), y(t-2), ...

How to Pick the Auto-Regressive Model?

YEAR	PROFIT
2010	76
2011	80
2012	90
2013	88
2014	81
2015	82
2016	94
2017	95
2018	93
2019	84
2020	82

$t = 2$

$y_t - y_{t-2}$

YEAR	LAG
2010	0
2012	14
2014	-9
2016	13
2018	-1
2020	-11

How to Pick the Auto-Regressive Model?

YEAR	LAG
2010	0
2012	14
2014	-9
2016	13
2018	-1
2020	-11

$$F_{2021} = c + \varphi y_{2012} + \varphi y_{2016} + \varphi y_{2020} + \varepsilon t$$

8.5.1 | Auto-Regressive (AR) Models

Auto-regression is a regression of a variable on itself measured at different time points. Auto-regressive model with lag 1, AR (1), is given by

$$Y_{t+1} = \mu + \beta Y_t + \varepsilon_{t+1} \tag{8.5}$$

Equation (8.5) can be generalized to include p lags on the right-hand side and is called a AR (p) model. Equation (8.5) can be re-written as

$$Y_{t+1} - \mu = \beta(Y_t - \mu) + \varepsilon_{t+1} \tag{8.6}$$

where ε_{t+1} is a sequence of uncorrelated residuals assumed to follow the normal distribution with zero mean and constant standard deviation. $(Y_t - \mu)$ can be interpreted as a deviation from mean value μ; it is known as *mean centered series*.

One of the important tasks in using the AR model in forecasting is model identification, which is, identifying the value of p (the number of lags). One of the standard approaches used for model identification is using auto-correlation function (ACF) and partial auto-correlation function (PACF).

8.5.1.1 ACF

Auto-correlation of lag k is the correlation between Y_t and Y_{t-k} measured at different k values (e.g., Y_t and Y_{t-1} or Y_t and Y_{t-2}). A plot of auto-correlation for different values of k is called an auto-correlation function (ACF) or *correlogram*.

statsmodels.graphics.tsaplots.plot_acf plots the autocorrelation plot.

T	Y	ACF
1	5	0
2	8	3
3	7	-1
4	4	-3
5	3	-1
6	9	6
7	8	-1
8	6	-2
9	6	0
10	5	-2

8.5.1.2 PACF

Partial auto-correlation of lag k is the correlation between Y_t and Y_{t-k} when the influence of all intermediate values (Y_{t-1}, Y_{t-2}, …, Y_{t-k+1}) is removed (partial out) from both Y_t and Y_{t-k}. A plot of partial auto-correlation for different values of k is called partial auto-correlation function (PACF).

statsmodels.graphics.tsaplots.plot_pacf plots the partial auto-correlation plot.

For applying the AR model, we will use another dataset described in the following example.

T	Y	ACF	PACF
1	5	0	0
2	8	3	
3	7	-1	
4	4	-3	-1
5	3	-1	
6	9	6	
7	8	-1	4
8	6	-2	
9	6	0	
10	5	-2	-3

8.1 EXAMPLE

Aircraft Spare Parts

Monthly demand for avionic system spares used in Vimana 007 aircraft is provided in *vimana.csv*. Build an ARMA model based on the first 30 months of data and forecast the demand for spares for months 31 to 37. Comment on the accuracy of the forecast.

Solution:

We first read the dataset from *viman.csv* onto a DataFrame and print the first 5 records.

```
vimana_df = pd.read_csv('vimana.csv')

vimana_df.head(5)
```

	Month	Demand
0	1	457
1	2	439
2	3	404
3	4	392
4	5	403

Now print the metadata from *viman.csv*.

```
vimana_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 37 entries, 0 to 36
Data columns (total 2 columns):
Month      37 non-null int64
demand     37 non-null int64
dtypes: int64(2)
memory usage: 672.0 bytes
```

Draw the ACF plot to show auto-correlation upto lag of 20 using the following code:

```
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

# Show autocorrelation upto lag 20
acf_plot = plot_acf( vimana_df.demand,
                     lags=20)
```

FIGURE 8.5 ACF plot for aircraft spare parts.

In Figure 8.5, the shaded area represents the upper and lower bounds for critical values, where the null hypothesis cannot be rejected (auto-correlation value is 0). So, as can be seen from Figure 8.5, null hypothesis is rejected only for lag = 1 (i e., auto-correlation is statistically significant for lag 1).

We draw the PACF plot with lag up to 20.

```
pacf_plot = plot_pacf( vimana_df.demand,
                       lags=20 )
```

FIGURE 8.6 PACF plot for aircraft spare parts.

In Figure 8.6, the shaded area represents the upper and lower bounds for critical values, where the null hypothesis cannot be rejected. So, for lag = 1 the partial auto-correlation is significant.

To select the appropriate p in the AR model, the following thumb rule can be used:

1. The partial auto-correlation is significant for first p-values (first p lags) and cuts off to zero.
2. The ACF decreases exponentially. (Signs of stationarity)

8.5.1.3 ***Building AR Model***

The *statsmodels.tsa.arima_model.ARIMA* can be used to build AR model. It takes the following two parameters:

1. endog: list of values – It is the endogenous variable of the time series.
2. order: The (p, d, q) – ARIMA model parameters. Order of AR is given by the value *p*, the order of integration is *d*, and the order of MA is given by *q*.

We will set *d* and *q* to 0 and use $p = 1$ for AR(1) model, and we will use only 30 months of data for building the model and forecast the next six months for measuring accuracy.

Building the model and printing the accuracy.

Building the model and printing the accuracy.

```
from statsmodels.tsa.arima_model import ARIMA
```

```
arima = ARIMA(vimana_df.demand[0:30].astype(np.float64).as_matrix(),
              order = (1,0,0))
ar_model = arima.fit()
```

Printing the model summary.

```
ar_model.summary2()
```

	Coef.	Std.Err.	t	$P > \|t\|$	[0.025	0.975]
const	513.4433	35.9147	14.2962	0.0000	443.0519	583.8348
ar.L1 .y	0.4726	0.1576	2.9995	0.0056	0.1638	0.7814

The model summary indicates the AR with lag 1 is significant variables in the model. The corresponding *p-value* is less than 0.05 (0.0056).

8.5.1.4 Forecast and Measure Accuracy

Forecast the demand for the months 31 to 37. The index will be (month index – 1).

```
forecast_31_37 = ar_model.predict(30, 36)
```

```
forecast_31_37
```

```
array([480.15343682, 497.71129378, 506.00873185, 509.92990963,
       511.78296777, 512.65868028, 513.07252181 ])
```

```
get_mape( vimana_df.demand[30:],
          forecast_31_37 )
```

The MAPE of the AR model with lag 1 is 19.12.

8.5.2 | Moving Average (MA) Processes

MA processes are regression models in which the past residuals are used for forecasting future values of the time-series data.

A moving average process of lag 1 can be written as

$$Y_{t+1} = \alpha_1 \varepsilon_t + \varepsilon_{t+1} \tag{8.7}$$

The model in Eq. (8.7) can be generalized to q lags. The value of q (number of lags) in a moving average process can be identified using the following rules (Yaffee and McGee, 2000):

1. Auto-correlation value is significant for first q lags and cuts off to zero.
2. The PACF decreases exponentially.

We can build an MA model with q value as 1 and print the model summary.

```
arima = ARIMA(vimana_df.demand[0:30].astype(np.float64).as_matrix(),
              order = (0,0,1))
ma_model = arima.fit()
ma_model.summary2()
```

	Coef.	Std.Err.	t	$P > \|t\|$	[0.025	0.975]
const	516.5440	26.8307	19.2520	0.0000	463.9569	569.1312
ma.L1.y	0.3173	0.1421	2.2327	0.0337	0.0388	0.5958

	Real	Imaginary	Modulus	Frequency
MA.1	–3.1518	0.0000	3.1518	0.5000

As per the model summary, moving average with lag 1 is statistically significant as the corresponding *p-value* is less than 0.05. Use the following code to measure the accuracy with the forecasted values of six periods.

```
forecast_31_37 = ma_model.predict(30, 36)
get_mape(vimana_df.demand[30:],
         forecast_31_37 )
```

```
17.8
```

The MAPE of the MA model with lag 1 is 17.8.

8.5.3 | ARMA Model

Auto-regressive moving average (ARMA) is a combination auto-regressive and moving average process. ARMA(p, q) process combines AR(p) and MA(q) processes.

The values of p and q in an ARMA process can be identified using the following thumb rules:

1. Auto-correlation values are significant for first q values (first q lags) and cuts off to zero.
2. Partial auto-correlation values are significant for first p values and cuts off to zero.

Based on the ACF and PACF plots in the previous section, we will develop ARMA(1, 1) model using the following codes:

```
arima = ARIMA( vimana_df.demand[0:30].astype(np.float64).as_
               matrix(), order = (1,0,1))
arma_model = arima.fit()
arma_model.summary2()
```

	Coef.	Std.Err.	t	$P > \|t\|$	[0.025	0.975]
const	508.3995	45.3279	11.2160	0.0000	419.5585	597.2405
ar.L1.y	0.7421	0.1681	4.4158	0.0001	0.4127	1.0715
MA.L1.y	–0.3394	0.2070	–1.6401	0.1126	–0.7451	0.0662

	Real	Imaginary	Modulus	Frequency
AR.1	1.3475	0.0000	1.3475	0.0000
MA.1	2.9461	0.0000	2.9461	0.0000

The model summary indicates that moving average with lag 1 is not significant (p-value is more than 0.05), when both auto regressive with lag1 is also used in the model.

```
forecast_31_37 = arma_model.predict(30, 36)
get_mape(vimana_df.demand[30:],
         forecast_31_37 )
```

```
20.27
```

The MAPE of the MA model with lag 1 is 20.27. Since MA lag is not significant, we will use AR(1) model [or MA(1) model].

8.5.4 | ARIMA Model

ARMA models can be used only when the time-series data is stationary. ARIMA models are used when the time-series data is non-stationary. Time-series data is called stationary if the mean, variance, and covariance are constant over time. ARIMA model was proposed by Box and Jenkins (1970) and thus is also known as Box–Jenkins methodology. ARIMA has the following three components and is represented as ARIMA (p, d, q):

1. AR component with p lags AR(p).
2. Integration component (d).
3. MA with q lags, MA(q).

The main objective of the integration component is to convert a non-stationary time-series process to a stationary process so that the AR and MA processes can be used for forecasting.

8.5.4.1 What is Stationary Data?

Time-series data should satisfy the following conditions to be stationary:

1. The mean values of Y_t at different values of t are constant.

2. The variances of Y_t at different time periods are constant (Homoscedasticity).

3. The covariance of Y_t and Y_{t-k} for different lags depend only on k and not on time t.