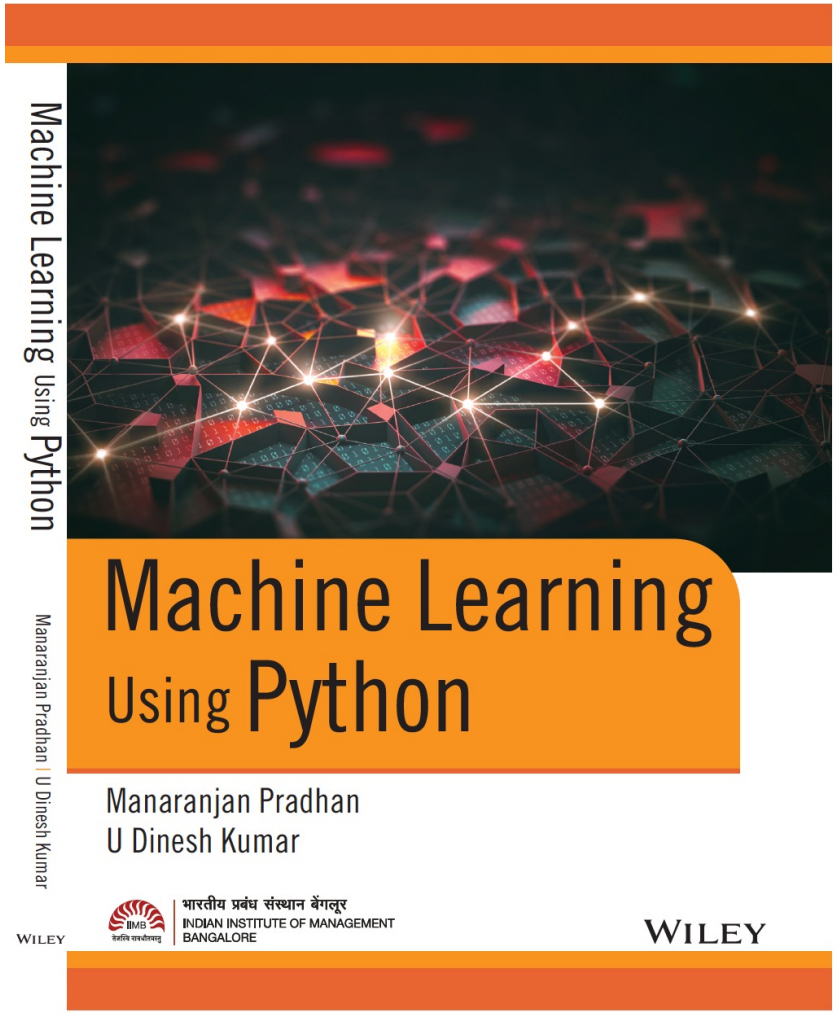

Chapter 06: Advanced Machine Learning

Prepared By: Purvi Tiwari

Learning Objectives

- Understanding the foundations of machine learning algorithms
- Learning the difference between supervised and unsupervised learning algorithms.
- Understanding and developing the gradient descent algorithm.
- Applying machine learning algorithms available in *scikit-learn* to regression and classification problems.
- Understanding the concepts of underfitting – overfitting and use of regularization.
- Understanding ensemble techniques such as Random Forest, Bagging and Boosting.
- Learning feature selection using machine learning models.

Overview

- Machine learning (ML) algorithms are a subset of artificial intelligence (AI) that imitates human learning process.
- ML algorithms develop multiple models and each model is analogues to an experience.
- In ML algorithms, several models are developed which can run into several hundred and each data and model is treated as learning opportunity.
- According to Mitchell (2006)

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

Overview (Cntd.)

- Slearning depends heavily on validation of model assumption and hypothesis testing, whereas the objective of machine learning is to improve prediction accuracy.
- Two types of ML algorithms
  1. Supervised Learning – the datasets have the values of features and the corresponding outcome variable. Example – Linear regression and logistic regression.
  2. Unsupervised learning – the datasets will have only feature values, but not the outcome variables. The algorithm learns the structure in the features. Example – Clustering and factor analysis

How Machines Learn

- In supervised learning, the algorithm learns using a function called loss function, cost function or error function.
- It is a function of predicted output and the desired output.

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

- h(X) is the predicted output and y is the desired output, and n is the total number of recorded for which the predictions are made.
- The objective is to learn the values of parameters that minimizes the cost function.

Gradient Descent Algorithm

- Most widely used optimization technique in ML.
- The functional form of a simple linear regression model

$$Y_i = \beta_0 + \beta_1 X_i + \varepsilon_i$$

- Where β_0 is called bias, β_1 is the feature weight, ε_i is the error in prediction.
- The predicted value of Y_i is written as $\hat{Y}_i$ and is given by

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

- Where $\hat{\beta}_0$ and $\hat{\beta}_1$ are the estimated values of β_0 and β_1

Gradient Descent Algorithm (Cntd.)

- The error is given by

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

- The cost function for the linear regression model is the total error across all N records and given by

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

- Error is a function of β_0 and β_1

Gradient Descent Algorithm (Cntd.)

- Error is a pure convex function and has a global minimum.
- The gradient descent algorithm starts at a random point and moves toward the optimal solution.

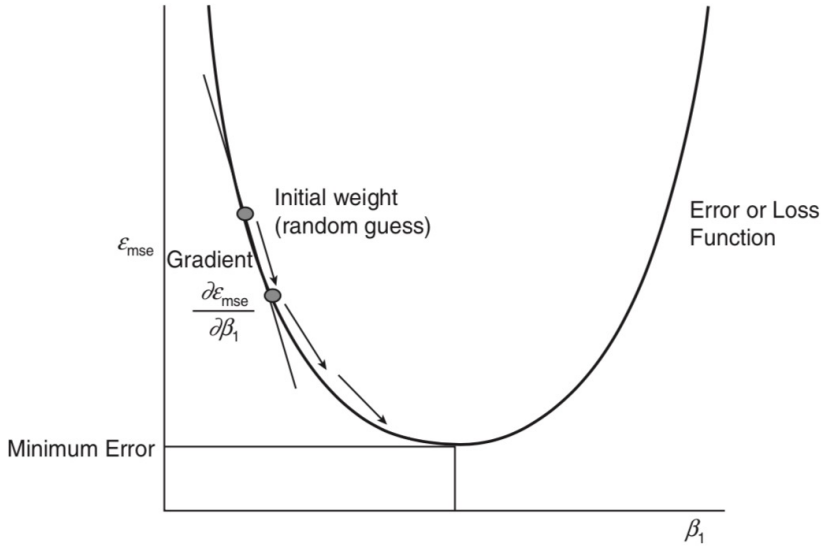

FIGURE 6.1 Cost function for linear regression model.

Gradient Descent Algorithm (Cntd.)

- Steps for finding the optimal values of β_0 and β_1
  1. Randomly guess the initial value of β_0 and β_1.
  2. Calculate the estimated value of the outcome variable $\hat{Y}_i$ for initialized values of bias and weights.
  3. Calculate the mean squared error function (MSE).
  4. Adjust the β_0 and β_1 values by calculating the gradients of the error function

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

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

Where $\propto$ is the learning rate and the magnitude of the update is applied to bias and weights at each iteration.

Gradient Descent Algorithm (Cntd.)

- The partial derivatives of MSE with respect to β_0 and β_1

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

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

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 minimal cost points are best estimates of the model parameters.

Developing Gradient Descent Algorithm

- For Linear Regression Model
- Dataset – *Advertising.csv*
- 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

Developing Gradient Descent Algorithm (Cntd.)

- Loading the dataset

```
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

Developing Gradient Descent Algorithm (Cntd.)

- Setting X (Features) and Y (Outcome) variables

```
X = sales_df[['TV', 'Radio', 'Newspaper']]
Y = sales_df['Sales']
```

- Standardize X an Y

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

Developing Gradient Descent Algorithm (Cntd.)

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

Developing Gradient Descent Algorithm (Cntd.)

Method 1: Random Initialization of the Bias and Weights

- The method randomly initialize the bias and weights.

```
import random

#dim - is the number of weights to be initialized besides the bias
def initialize( dim ):
    # For reproducible results, the seed it set to 42.
    # Reader can comment the following two lines
    # and try other initialiazation values.
    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
```

Developing Gradient Descent Algorithm (Cntd.)

- Initializing the parameters

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

```
Bias:    0.6394267984    Weights:    [0.37454012 0.95071431 0.73199394]
```

Developing Gradient Descent Algorithm (Cntd.)

Method 2: Predict Y values from the Bias and Weights

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

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]
```

```
array([ 3.23149557, 1.70784873, 2.82476076, 2.75309026, 0.92448558,
        3.17136498, 0.62234399, -0.34935444, -2.313095,  -0.76802983])
```

Developing Gradient Descent Algorithm (Cntd.)

Method 3: Calculate the Cost Function - MSE

- Computing mean squared error (MSE) by

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

Developing Gradient Descent Algorithm (Cntd.)

```
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 )
```

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

```
1.530310019850589 5
```

Developing Gradient Descent Algorithm (Cntd.)

Method 4: Update the Bias and Weights

- Most important method, where the bias and weights are adjusted based on the gradient of cost function.
- The bias and weights are updates using the following gradients

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

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

- Where $\propto$ is the learning parameter that decides the magnitude of the update to be done to the bias and weights.

Developing Gradient Descent Algorithm (Cntd.)

- 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
```

Developing Gradient Descent Algorithm (Cntd.)

- Updating bias and weights once after initializing.

```
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]
```

Developing Gradient Descent Algorithm (Cntd.)

- Finding the Optimal Bias and Weights
  - The updates to the bias and weights need to be done iteratively, until the cost is minimum.
  - There are two approaches to stop the iterations:
  1. Run a fixed number of iterations and use the bias and wrights 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.

Developing Gradient Descent Algorithm (Cntd.)

```
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
```

Developing Gradient Descent Algorithm (Cntd.)

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

```
Final estimate of b and w:  0.428448958173914 93 [0.48270238 0.752659
69 0.46109174]
```

```
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

Developing Gradient Descent Algorithm (Cntd.)

- Plotting the cost Function against the Iterations

```
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]
```

Developing Gradient Descent Algorithm (Cntd.)

- The cost is still reducing and has not reached the minimum point. More iterations can be run to verify if the cost is reaching a minimum point or not.

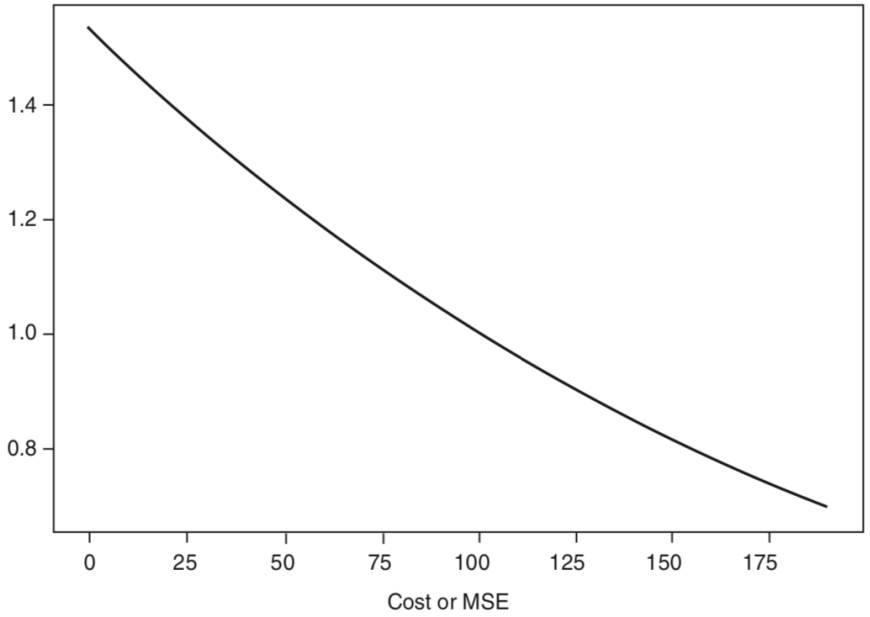

FIGURE 6.2 Cost or MSE at the end of iterations.

Developing Gradient Descent Algorithm (Cntd.)

```
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]
```

```
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");
```

Developing Gradient Descent Algorithm (Cntd.)

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

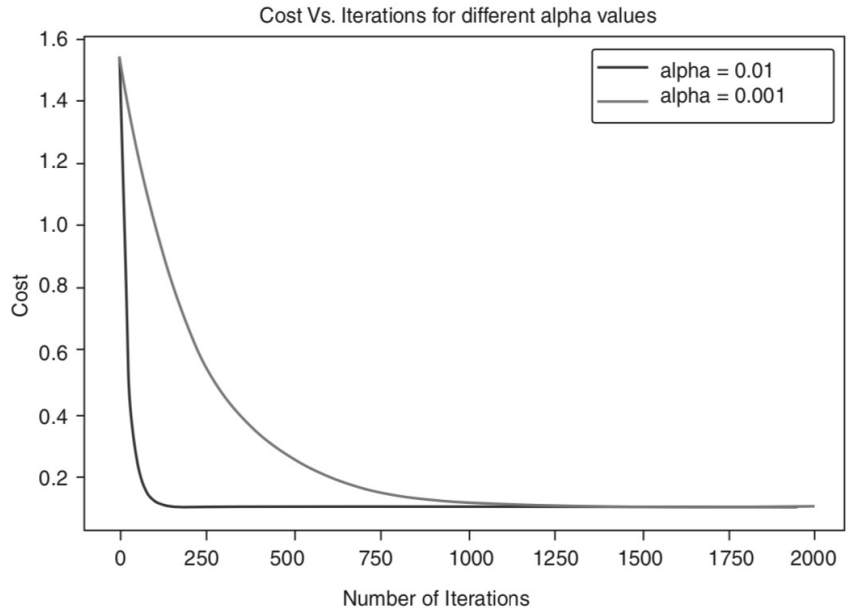

FIGURE 6.3 Cost or MSE at the end of iterations for different alpha values.

Scikit-learn Library for Machine Learning

- Open-source Pyhton library for building machine learning models.
- scikit-learn provides a comprehensive set of algorithms for the following kind of problems:
  1. Regression
  2. Classification
  3. Clustering
- It provides an extensive set of methods for data pre-processing and feature selection.

Steps for Building Machine Learning Models

- 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).

Steps for Building Machine Learning Models

- Splitting Dataset into Train and Test Datasets

```
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) # Seed value
                                   of 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
```

Steps for Building Machine Learning Models

- Building Linear Regression Model with Train Dataset
- 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.

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

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

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

Steps for Building Machine Learning Models

- After the model is built, the model parameters such as intercept and coefficients can be obtained as follows

```
linreg.intercept_
```

```
2.708949092515912
```

```
linreg.coef_
```

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

Steps for Building Machine Learning Models

- Associating the coefficient values with the variable names

```
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 model indicates that for every unit change in TV spending, there is an increase of 0.44 units in sales revenue.
- The weights are different than what we estimated earlier as we have not used standardized values in this model.

Steps for Building Machine Learning Models

- Making Prediction on Test Set

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

```
# 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)
```

Steps for Building Machine Learning Models

- Making Prediction on Test Set

	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

Steps for Building Machine Learning Models

- Measuring Accuracy

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

- 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.

Steps for Building Machine Learning Models

- RMSE Calculation

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

```
# 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
```

Steps for Building Machine Learning Models

- Bias-Variance Trade-Off
  - Models errors can be decomposed into two components: bias and variance.
  - Avoid model overfitting or underfitting.
  - High bias can lead to building underfitting model, whereas high variance can lead to overfitting models.
- Understanding with example - dataset *curve.csv*

```
# 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

Steps for Building Machine Learning Models

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

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

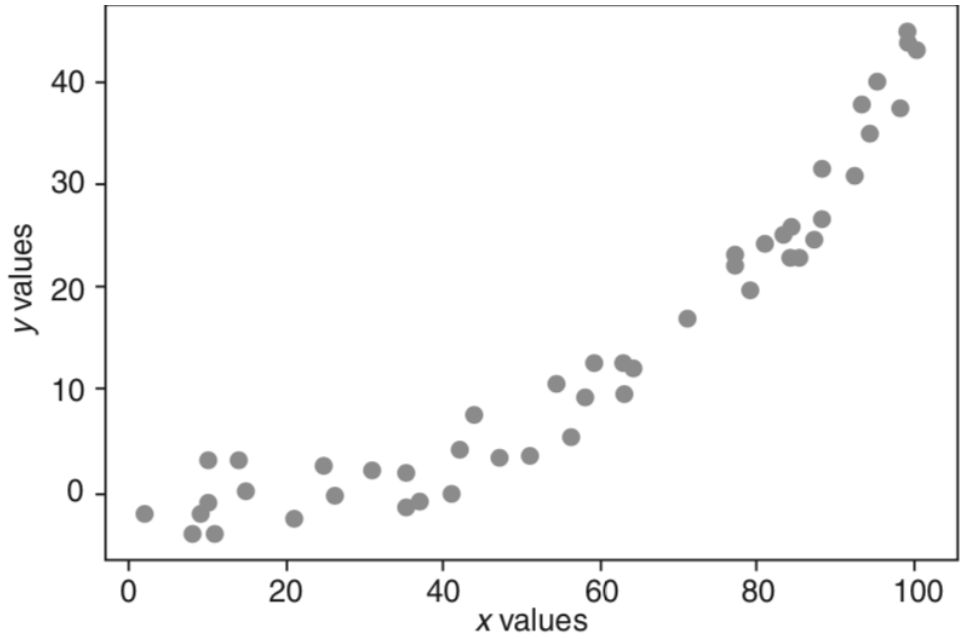

FIGURE 6.4 Scatter plot between *x* and *y*.

Steps for Building Machine Learning Models

- It can be observed that the relation between y and x is not linear.
- Need to try various polynomial forms of x and verify the model.

```
# 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.

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

Steps for Building Machine Learning Models

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

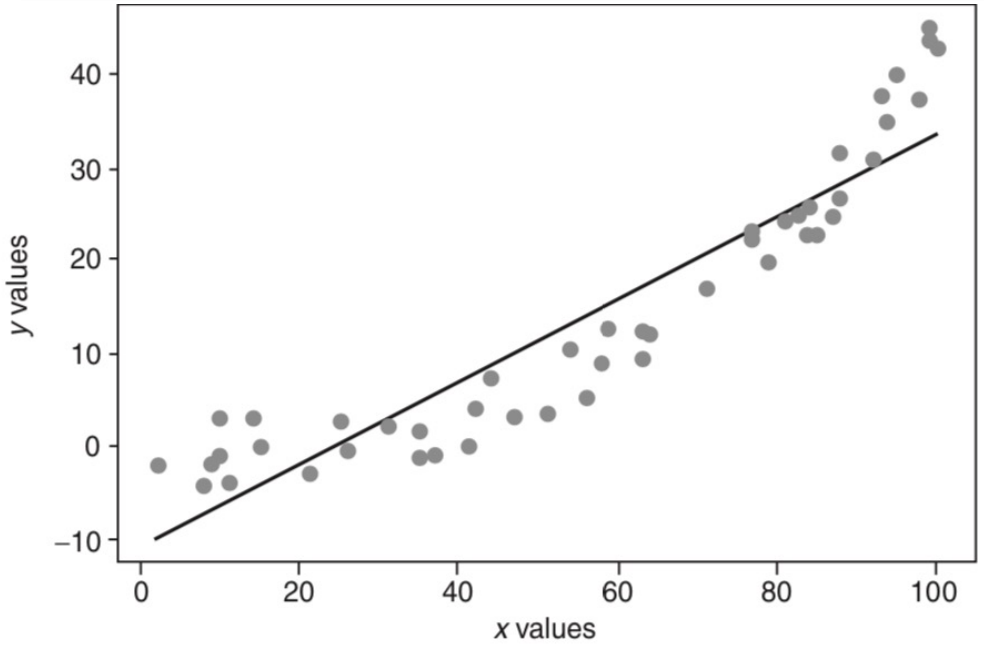

FIGURE 6.5 Linear regression model between *x* and *y*.

Steps for Building Machine Learning Models

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

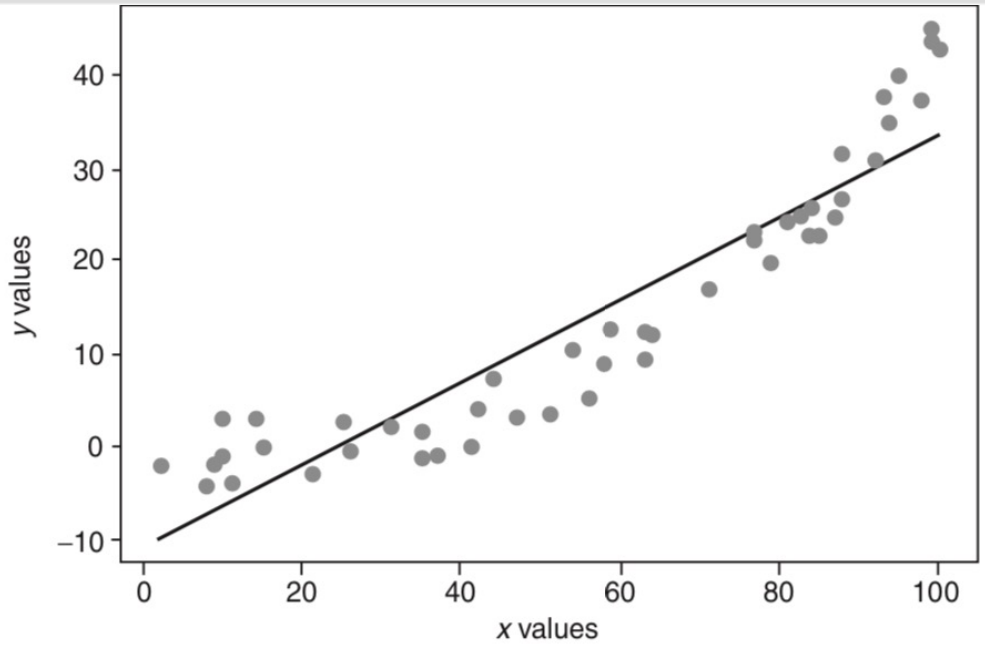

FIGURE 6.5 Linear regression model between *x* and *y*.

Steps for Building Machine Learning Models

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

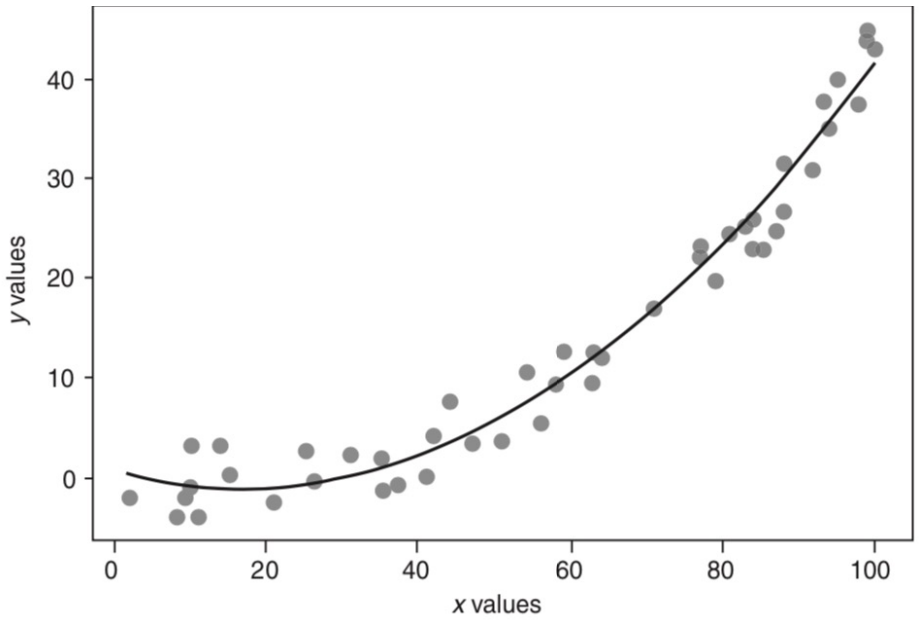

FIGURE 6.6 Linear regression model between *y* and polynomial terms of *x* of degrees 1 and 2.

Steps for Building Machine Learning Models

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

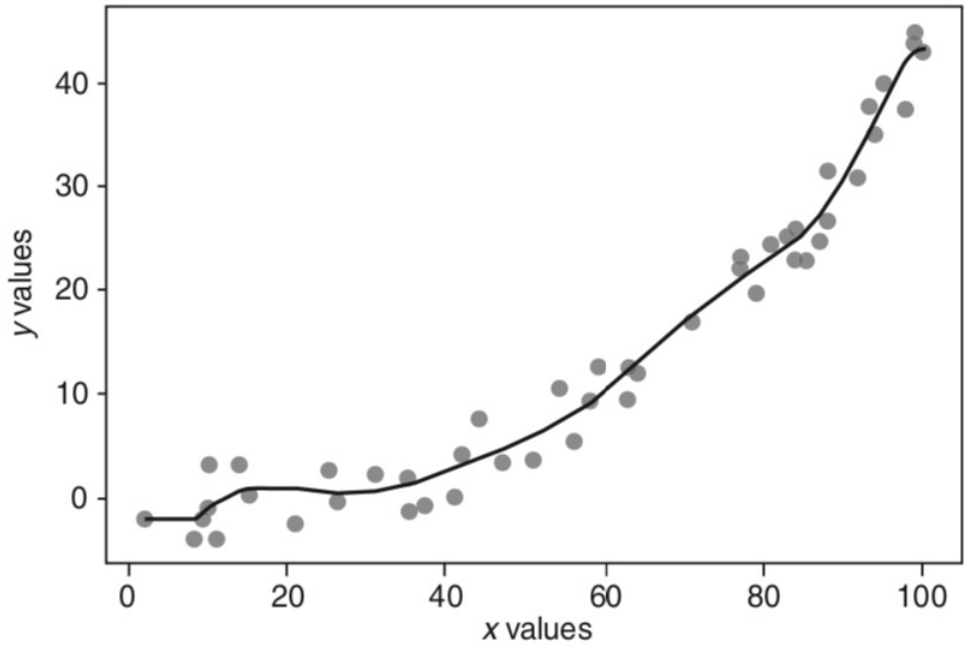

FIGURE 6.7 Linear regression model between *y* and polynomial terms of *x* of degree ranging from 1 to 10.

Steps for Building Machine Learning Models

- Following code is used to build models with degrees ranging from 1 to 15
- Storing the degree and error details in different columns of DataFrame names *rmse_df.*
  1. *degree:* Degree of the model.
  2. *rmse_train*: RMSE error on train set.
  3. *rmse_test:* RMSE error on test set.

Steps for Building Machine Learning Models

```
# 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))]
```

Steps for Building Machine Learning Models

```
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

Steps for Building Machine Learning Models

```
# 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");
```

Steps for Building Machine Learning Models

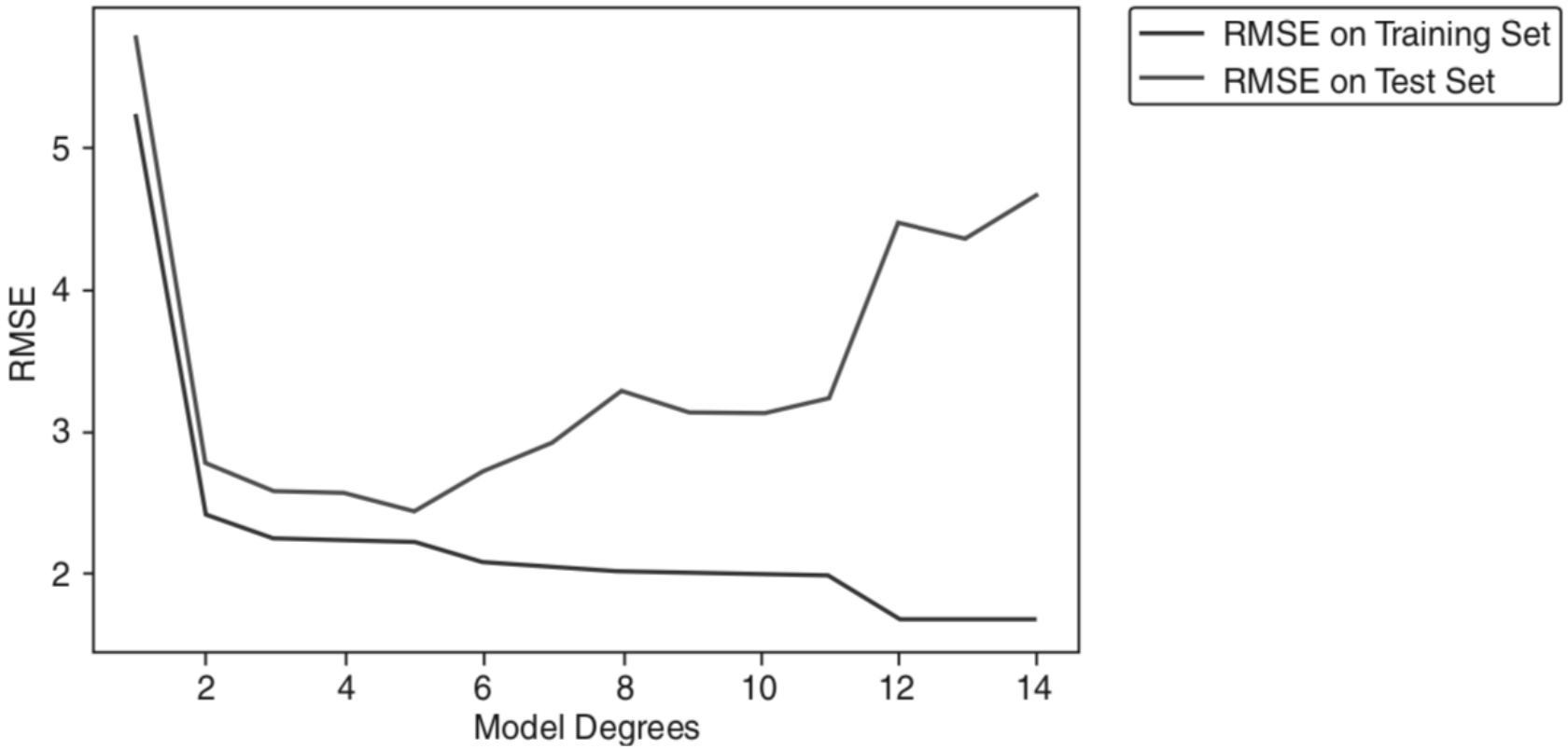

FIGURE 6.8 Model accuracy on training and test sets against the model complexity.

Steps for Building Machine Learning Models

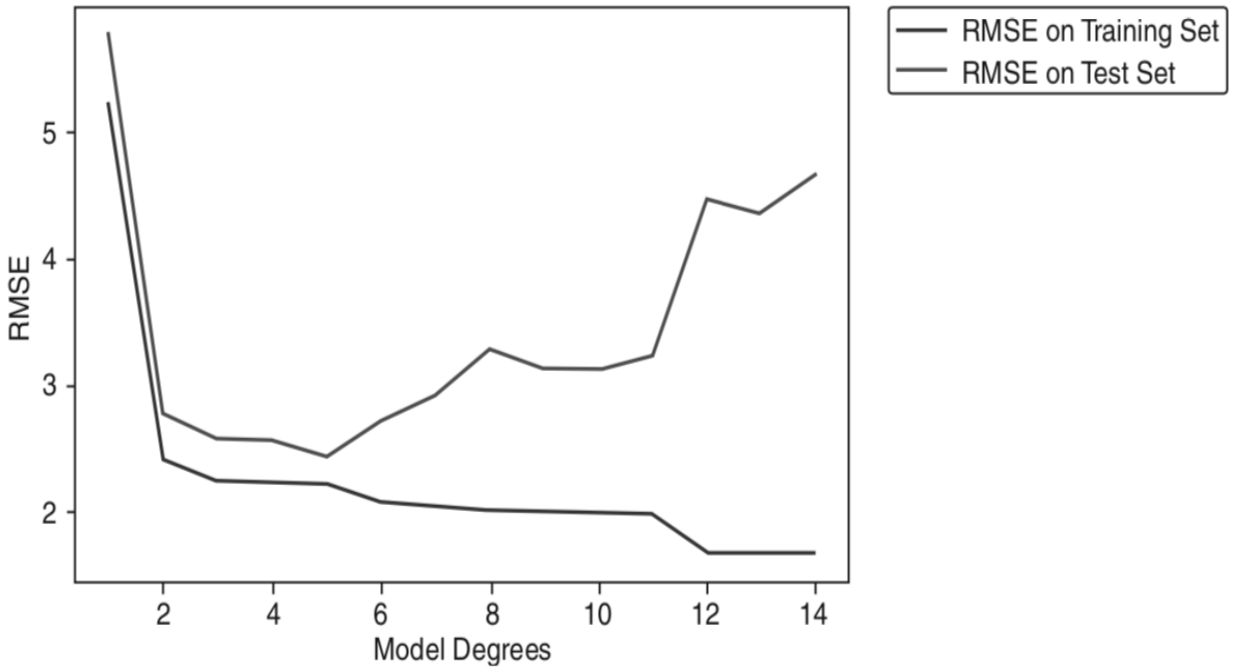

FIGURE 6.8 Model accuracy on training and test sets against the model complexity.

- Key Observations

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.

Steps for Building Machine Learning Models

- K-Fold Cross-Validation

1. A robust validation approach that can be adopted to verify if the model is overfitting.
2. The model, which generalizes well and does not overfit, should not be very sensitive to any change in underlying training samples.
3. It builds and validate multiple models by resampling multiple training and validation sets from the original dataset.

Steps for Building Machine Learning Models

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

1. Split the training dataset into k subsets of equal size. Each subset will be called a fold. Let the folds be labelled as f, f, …, f. Generally, the value of k is taken to be 5 or 10.
2. For i=1 to k
   a) Fold f 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.
3. Calculate the final accuracy by averaging the accuracies in the test data across all k models.

Steps for Building Machine Learning 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.

Advanced Regression Models

- Dataset – IPL dataset
- Linear Regression model – the model will predict SOLD PRICE of a player based on past performance measures of the players.

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

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 130 entries, 0 to 129
Data columns (total 26 columns):
Sl.NO.          130    non-null   int64
PLAYER NAME     130    non-null   object
AGE             130    non-null   int64
COUNTRY         130    non-null   object
TEAM            130    non-null   object
PLAYING ROLE    130    non-null   object
T-RUNS          130    non-null   int64
T-WKTS          130    non-null   int64
ODI-RUNS-S      130    non-null   int64
ODI-SR-B        130    non-null   float64
ODI-WKTS        130    non-null   int64
ODI-SR-BL       130    non-null   float64
CAPTAINCY EXP   130    non-null   int64
RUNS-S          130    non-null   int64
HS              130    non-null   int64
AVE             130    non-null   float64
SR-B            130    non-null   float64
SIXERS          130    non-null   int64
RUNS-C          130    non-null   int64
WKTS            130    non-null   int64
AVE-BL          130    non-null   float64
ECON            130    non-null   float64
SR-BL           130    non-null   float64
AUCTION YEAR    130    non-null   int64
BASE PRICE      130    non-null   int64
SOLD PRICE      130    non-null   int64
dtypes: float64(7), int64(15), object(4)
memory usage: 26.5+ KB
```

Advanced Regression Models (Cntd.)

- Getting the features

```
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']
```

- Encoding the categorical variables

```
# 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 )
```

Advanced Regression Models (Cntd.)

- Displaying all the feature names along with the new dummy features.

```
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')
```

Advanced Regression Models (Cntd.)

- Creating variables X and Y

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

- Standardization of X and Y

```
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()
```

Advanced Regression Models (Cntd.)

- Splitting the dataset into Train and Test

```
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)
```

Advanced Regression Models (Cntd.)

- 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)
```

```
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] )
```

Advanced Regression Models (Cntd.)

- Storing the beta coefficients and respective columns in a DataFrame

```
## 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)
```

Advanced Regression Models (Cntd.)

- Plotting the Coefficient 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")
```

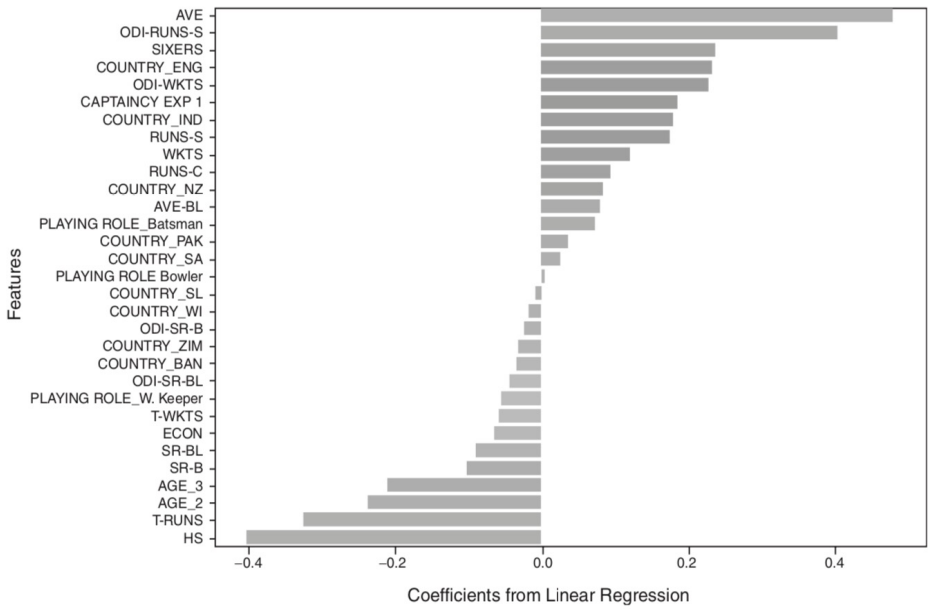

FIGURE 6.10 Bar plot depicting coefficient values of features in the model, sorted in descending order.

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.
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 multi-collinearity. For example – SR-B is expected to have a positive effect on the SOLD PRICE.

Advanced Regression Models (Cntd.)

- Calculate RMSE

```
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 )
```

```
get_train_test_rmse( linreg )
```

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

Advanced Regression Models (Cntd.)

- Applying Regularization
  - Regularization deals with overfitting
  - Overfitting is typically caused by inflation of the 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.
  - Optimizer controls the coefficient values to minimize the cost function.
  - Following are the twoo approaches that can be used for adding a penalty to the cost function:

1. L1 Norm
2. L2 Norm

Advanced Regression Models (Cntd.)

1. L1 Norm – Summation of the absolute value of the coefficients, called Least Absolute Shrinkage and Selection Operator (LASSO Term).

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

Where $\propto$ is the multiplier term

2. L2 Norm – Summation of the squared value of the coefficients, called Ridge Term.

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

Advanced Regression Models (Cntd.)

- Effect of LASSO and Ridge constraint applied to the cost function

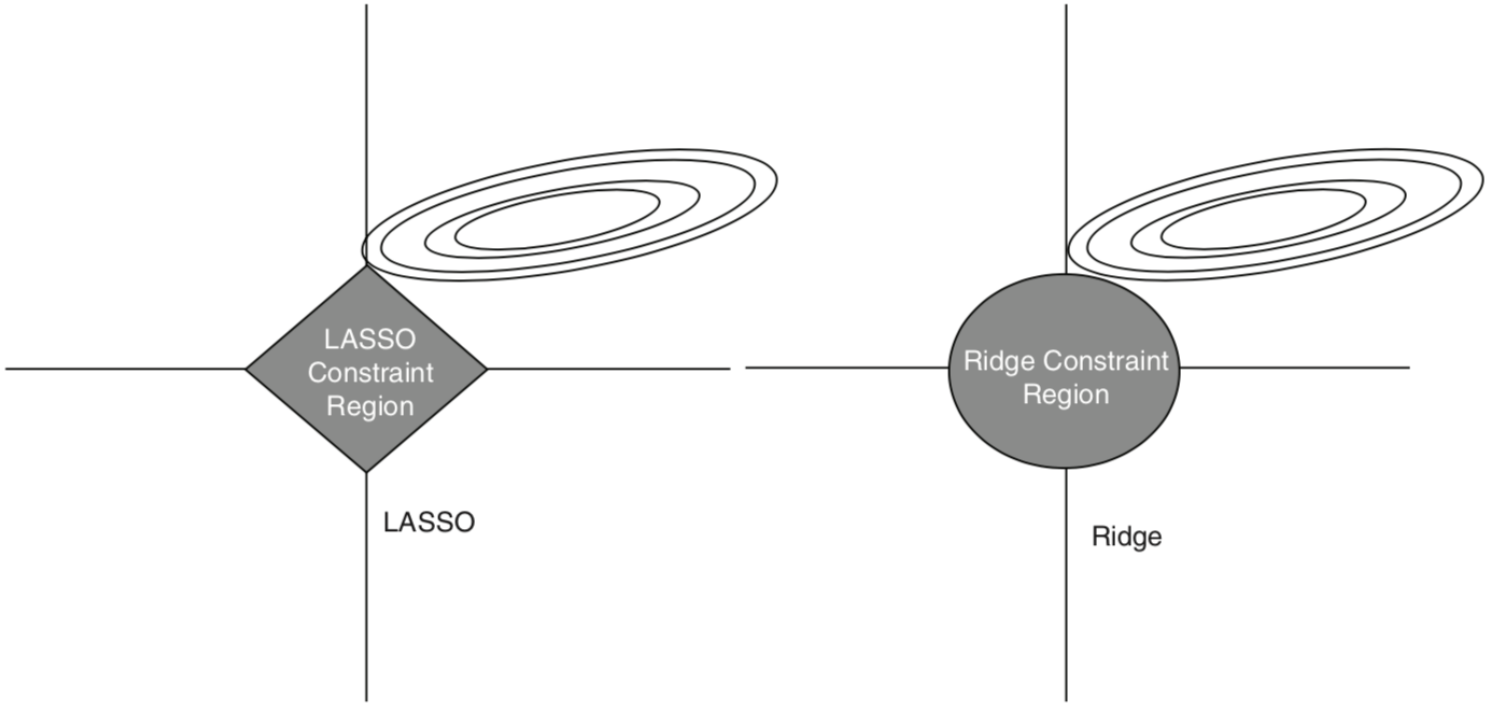

FIGURE 6.11 Effect of LASSO and Ridge terms on the cost function.

Advanced Regression Models (Cntd.)

- Ridge Regression
  1. *sklearn.linear_model* provides Ridge regression for building linear models by applying L2 penalty.
  2. Ridge regression takes the following parameters:

  a) *alpha* – float – is the regularization strength and must be positive.
  Larger values of alpha imply stronger regularization

  b) *max_iter* – int – is the maximum umber 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)
```

Advanced Regression Models (Cntd.)

- Ridge Regression

```
get_train_test_rmse( ridge )
```

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

- The difference in RMSE on train and test has reduced because of the penalty effect. The difference can be further reduced by applying stronger penalty (for example – apply large alpha 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
```

Advanced Regression Models (Cntd.)

- LASSO Regression
  1. *Sklearn.linear_model* provides LASSO regression for building linear models by applying L1 penalty.
  a) *alpha* – float – multiplies the L1 term. Default value is set to 1.0
  b) *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
```

Advanced Regression Models (Cntd.)

- It can be noticed that the model in not overfitting and the difference between train 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.

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

Advanced Regression Models (Cntd.)

```
## 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

Advanced Regression Models (Cntd.)

- Elastic Net Regression – it combines both L1 and L2 regularizations to build a regression model.
- The corresponding function is given by

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

- *ElasticNet* takes following two parameters:
  1. *Alpha:* constant that multiplies the penalty terms. Default is set to 1.0. (alpha = $\sigma + \gamma$), where σ (L2) and γ(L1) are two hyperparameters.

Advanced Regression Models (Cntd.)

2. *l1_ratio:* The ElasticNet mixing parameter, with 0 <= l1_ratio <= 1

Where

l1_ratio = 0 implies that the penalty is an L2 penalty.

l1_ratio = 1 implies that the penalty is an L1 penalty.

l1_ratio < 0 implies that the penalty is a combination of L1 and L2.

```
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
```

Advanced Machine Learning Algorithms

- We will take a binary classification problem and demonstrate it through ML algorithms such as
  1. K-Nearest Neighbors (KNN),
  2. Random Forest, and
  3. Boosting
- Dataset : bank marketing dataset, available at the University of California, Irvine machine learning repository *(*http://archive.ics.uci.edu/ml/datasets/Bank+Marketing*)*

Advanced Machine Learning Algorithms (Cntd.)

```
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

Advanced Machine Learning Algorithms (Cntd.)

```
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
```

Advanced Machine Learning Algorithms (Cntd.)

- Dealing with Imbalanced Datasets
  - A dataset is imbalanced when there is no equal representation of all classes in data.
  - In our dataset the proportion of customers who responded to the telemarketing is approximately 11.5% and the remaining 88.5% did not respond.
- Number of records of customers who has or has not opened the account

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

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

Advanced Machine Learning Algorithms (Cntd.)

- Dealing with Imbalanced Datasets
  - A dataset is imbalanced when there is no equal representation of all classes in data.
  - In our dataset the proportion of customers who responded to the telemarketing is approximately 11.5% and the remaining 88.5% did not respond.
- Number of records of customers who has or has not opened the account

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

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

Advanced Machine Learning Algorithms (Cntd.)

- Resampling techniques to to deal with imbalanced datasets
    1. Upsampling – Increase the instances of under-represented minority class by replicating the existing observations in the dataset. It 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 replacements. If false, all resampled examples will be unique.
    3. *n_samples:* number of samples to generate.

Advanced Machine Learning Algorithms (Cntd.)

```
## 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])
```

Advanced Machine Learning Algorithms (Cntd.)

- After upsampling, the case of subscribed and unsubscribes customers is 67:33
- Before using the dataset, the examples can be shuffles to make sure they are not in particular order.

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

Advanced Machine Learning Algorithms (Cntd.)

```
# 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']
```

Advanced Machine Learning Algorithms (Cntd.)

- encoding 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') )
```

Advanced Machine Learning Algorithms (Cntd.)

- Splitting into train and test data

```
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)
```

Advanced Machine Learning Algorithms (Cntd.)

- Building Logistic Regression Model

1. Logistics regression is a classification model.
2. The cost function is called log loss (log likelihood) or binary cross-entropy function and is given by

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

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.

Advanced Machine Learning Algorithms (Cntd.)

- Building Logistic Regression Model

```
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)
```

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

Advanced Machine Learning Algorithms (Cntd.)

- Confusion matrix

```
## 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()
```

Advanced Machine Learning Algorithms (Cntd.)

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

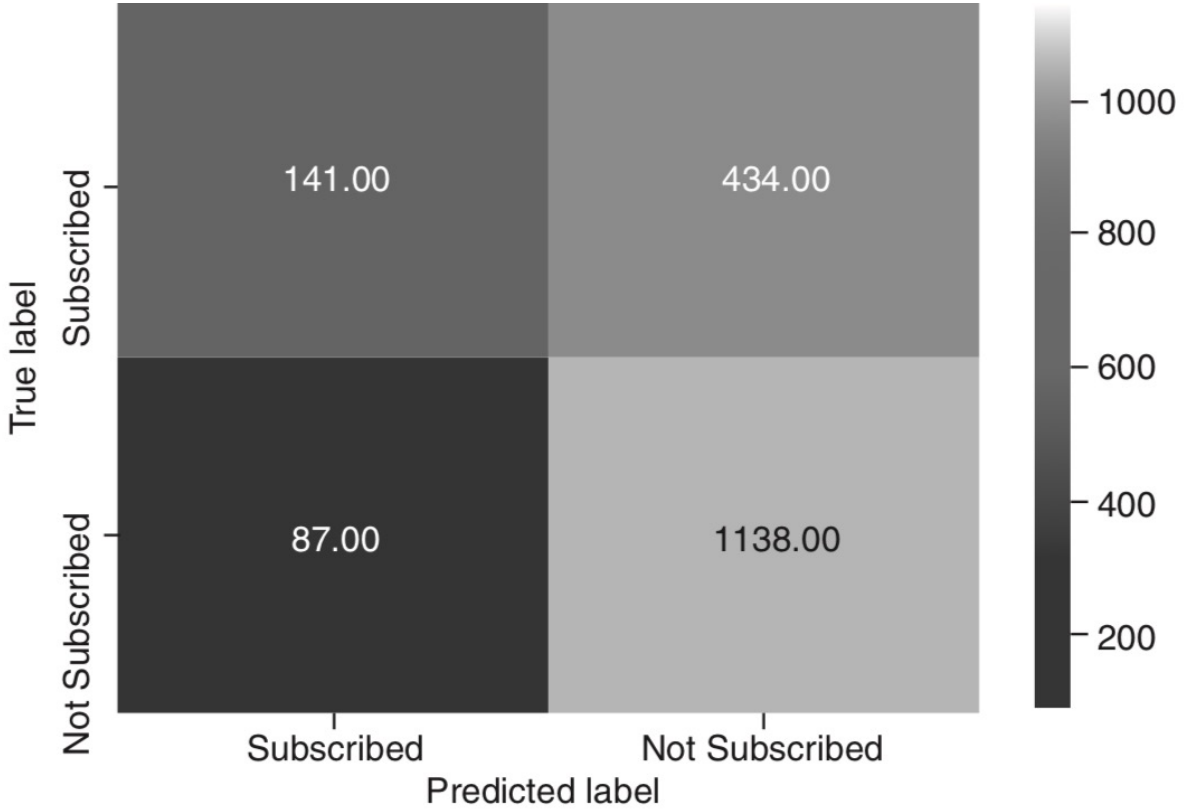

FIGURE 6.12 Confusion matrix of logistic regression model.

Advanced Machine Learning Algorithms (Cntd.)

- Classification Report – the *classification_report* function in *sklearn.metrics* gives a detailed report of precision, recall and F1-score for each class.

```
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

Advanced Machine Learning Algorithms (Cntd.)

- ROC and AUC Score

```
## 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

Advanced Machine Learning Algorithms (Cntd.)

- Creating 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]
```

```
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

Advanced Machine Learning Algorithms (Cntd.)

- ROC AUC score can be obtained using *metrice.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
```

Advanced Machine Learning Algorithms (Cntd.)

- Plotting 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 )
```

Advanced Machine Learning Algorithms (Cntd.)

- Plotting ROC Curve (Cntd.)

```
## 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]')
```

Advanced Machine Learning Algorithms (Cntd.)

- Plotting ROC Curve (Cntd.)

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

    return auc_score, fpr, tpr, thresholds
```

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

Advanced Machine Learning Algorithms (Cntd.)

- Plotting ROC Curve (Cntd.)

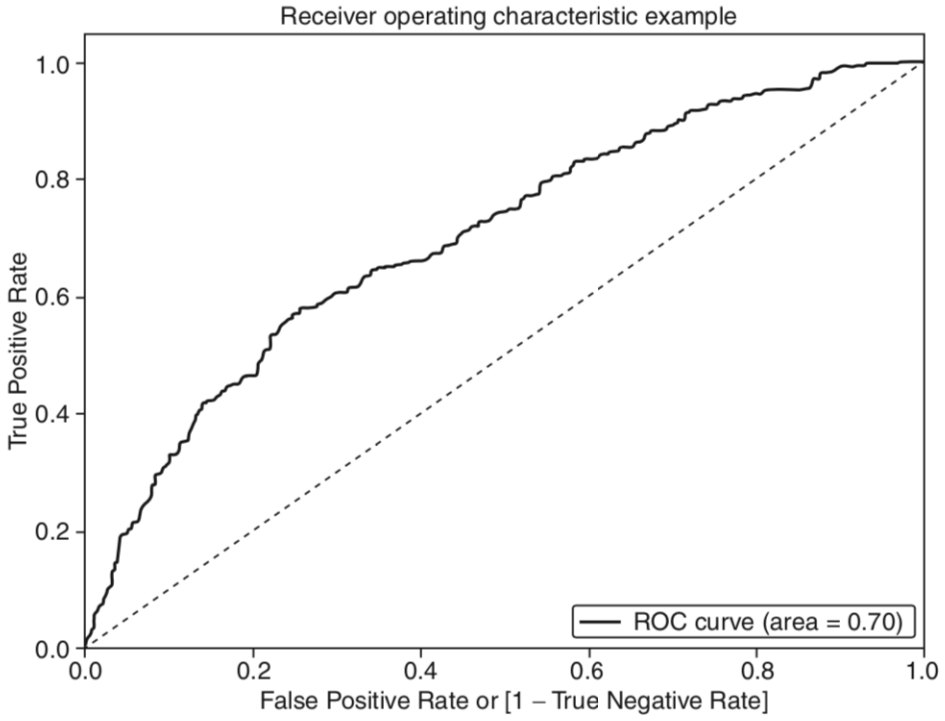

FIGURE 6.13 ROC AUC curve of logistic regression model.

Advanced Machine Learning Algorithms (Cntd.)

- K-nearest Neighbors (KNN) Algorithm

1. A non-parametric, lazy learning algorithm used for regression and classification problems.
2. ML algorithms are of two types: parametric and non-parametric
   - 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.
   - Non-parametric models do not make any assumptions on the underlying data distribution (such as normal distribution). KNN memorizes the data classifies new observations by comparing the training data.

Advanced Machine Learning Algorithms (Cntd.)

- KNN algorithm finds observations in the training set, which are similar to the new observation. These observations are called neighbors.
- Fore 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.

Advanced Machine Learning Algorithms (Cntd.)

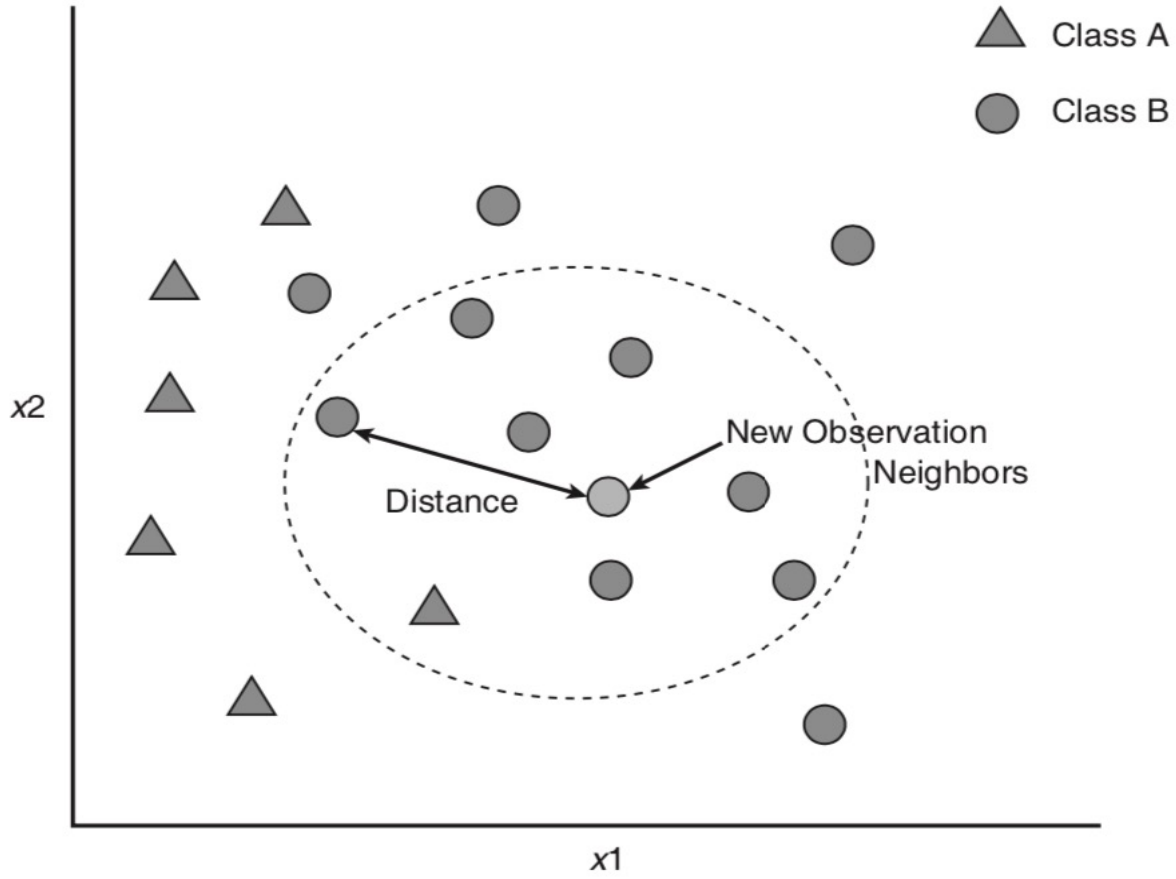

FIGURE 6.14 KNN algorithm.

Advanced Machine Learning Algorithms (Cntd.)

- The neighbors are founded by computing distance between observations. Euclidean distance is one of the widely used metrics.

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

- Where O and O are two observations in the data. X, X are the values of feature for records 1 and 2, respectively, X and X are the values of feature X for records 1 and 2, respectively.
- Other distance measures are Minkowski distance, Jaccard Coefficient and Gower's distance.

Advanced Machine Learning Algorithms (Cntd.)

- *sklearn.neighbors* provides KNeighborsClassifier algorithm for classification problems. It 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'.
  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.

```
## 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')
```

Advanced Machine Learning Algorithms (Cntd.)

- KNN Accuracy

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

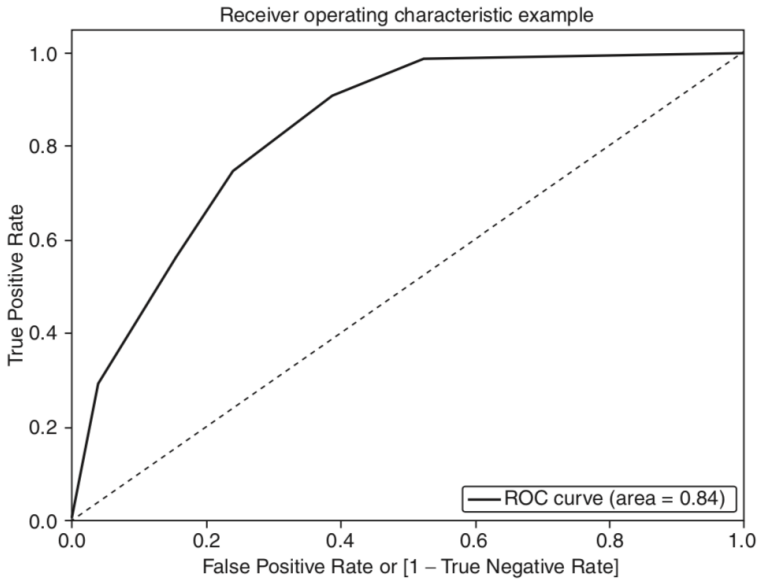

FIGURE 6.15 ROC AUC curve of KNN model.

Advanced Machine Learning Algorithms (Cntd.)

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

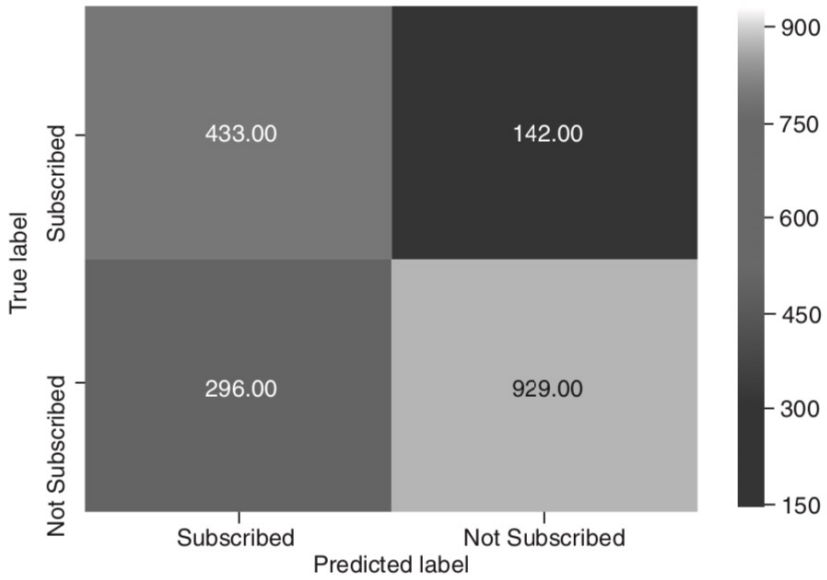

FIGURE 6.16 Confusion matrix of KNN model.

Advanced Machine Learning Algorithms (Cntd.)

```
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

- The recall of positive cases has improved from 0.25 to 0.75 in the KNN model.
- K in KNN Is called hyperparameters and the process of finding optimal value for a hyperparameter is called hyperparameter tuning.

Advanced Machine Learning Algorithms (Cntd.)

- GridSearch for Most Optimal Parameters
  - *sklearn.model_selection* provides a feature called GridSearch_CV, which searches through a set of given hyperparameter values and reports the most optimal one.
  - It does k-fold cross-validation for each value of hyperparameter to measure accuracy and avoid overfitting
  - Can be used for any machine learning algorithm to search for. Optimal values for its hyperparameters.
- GridSearchCV takes the following parameters:
  - Estimator – scikit-learn model, which implements estimator interface.
  - Param_grid – a distionary with parameter names as keys and lists of parameter values to search for
  - Scoring – the accuracy measure.
  - Cv – integer – the number of folds in K-fold.

Advanced Machine Learning Algorithms (Cntd.)

```
## 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)
```

Advanced Machine Learning Algorithms (Cntd.)

```
GridSearchCV(cv=10, error_score='raise',
       estimator=KNeighborsClassifier(algorithm='auto', leaf_size=30,
           metric='minkowski',
           metric_params=None, n_jobs=1, n_neighbors=5, p=2,
           weights='uniform'),
       fit_params=None, iid=True, n_jobs=1,
       param_grid=[{'n_neighbors': range(5, 10), 'metric':
          ['Canberra', 'euclidean', 'minkowski']}],
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring='roc_auc', verbose=0)
```

```
clf.best_score_
```

```
0.8293
```

```
clf.best_params_
```

```
{'metric': 'canberra', 'n_neighbors': 5}
```

Advanced Machine Learning Algorithms (Cntd.)

```
clf.grid_scores_
```

```
[mean : 0.82933, std: 0.01422, params: {'metric' : 'Canberra',
                                        'n_neighbors' : 5},
 mean : 0.81182, std: 0.01771, params: {'metric' : 'Canberra',
                                        'n_neighbors' : 6},
 mean : 0.80038, std: 0.01953, params: {'metric' : 'Canberra',
                                        'n_neighbors' : 7},
 mean : 0.79129, std: 0.01894, params: {'metric' : 'Canberra',
                                        'n_neighbors' : 8},
 mean : 0.78268, std: 0.01838, params: {'metric' : 'Canberra',
                                        'n_neighbors' : 9},
 mean : 0.79774, std: 0.01029, params: {'metric' : 'euclidean',
                                        'n_neighbors' : 5},
 mean : 0.77504, std: 0.01127, params: {'metric' : 'euclidean',
                                        'n_neighbors' : 6},
 mean : 0.75184, std: 0.01258, params: {'metric' : 'euclidean',
                                        'n_neighbors' : 7},
 mean : 0.73387, std: 0.01330, params: {'metric' : 'euclidean',
                                        'n_neighbors' : 8},
 mean : 0.71950, std: 0.01310, params: {'metric' : 'euclidean',
                                        'n_neighbors' : 9},
 mean : 0.79774, std: 0.01029, params: {'metric' : 'minkowski',
                                        'n_neighbors' : 5},
 mean : 0.77504, std: 0.01127, params: {'metric' : 'minkowski',
                                        'n_neighbors' : 6},
 mean : 0.75184, std: 0.01258, params: {'metric' : 'minkowski',
                                        'n_neighbors' : 7},
 mean : 0.73387, std: 0.01330, params: {'metric' : 'minkowski',
                                        'n_neighbors' : 8},
 mean : 0.71950, std: 0.01310, params: {'metric' : 'minkowski',
                                        'n_neighbors' : 9} ]
```

Ensemble Methods

- Learning algorithms that take a set of estimators or classifiers and classify new data points using strategy such as majority vote.
- Also used for regression problems, where the prediction of new data is simple average or weighted average of all the predictions from the set of regression models.
- Multiple datasets are needed for building multiple classifiers.
- In practice, strategy such as bootstrapped samples are drawn from the initial training set and given to each classifier.

Ensemble Methods

- Sometimes bootstrapping involves sampling features along with sampling observations.
- Each resampled set contains a subset of features available in the original set.
- Sampling features help to find important features.
- Records that are not part of specific sample are used for testing the model accuracy. Such records are called Out-of-Bag records
- Process of bootstrapping samples from original set to build multiple models and aggregating the results for final prediction is called Bagging.
- The most widely used bagging technique is Random Forest.