Chapter 6: Advanced Machine Learning

6.2.1 Developing a Gradient Descent Algorithm for Linear Regression Model

6.2.1.1 Loading the dataset

```
import pandas as pd
import numpy as np

import warnings
warnings.filterwarnings('ignore')

sales_df = pd.read_csv( 'Advertising.csv' )
# Pring 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 Variable

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

6.2.1.3 Standardize X & Y

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

6.2.1.4 Implementing the Gradient Descent Algorithm

Method 1: Random Initialization 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
```

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

```
Bias:  0.6394267984578837  Weights:  [0.37454012 0.95071431 0.731993
94]
```

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,  0.924485
58,
        3.17136498,  0.62234399, -0.34935444, -2.313095  , -0.768029
83])
```

Method 3: Calculate the cost function: MSE

```
import math

# Inputs
# Y - Actual values of y
# Y_hat - predicted value of y
def get_cost( Y, Y_hat ):
    # Calculating the residuals from taking difference between actual and predic
ted values
    Y_resid = Y - Y_hat
    # Matrix multiplication with self will give the square values
    # Then takin the sum and dividing 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 )
```

```
1.5303100198505895
```

Method 4: Update the bias and weights

```
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
```

```
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]
```

6.2.1.5 Finding the optimal bias and weights

```
def run_gradient_descent( X,
                          Y,
                          alpha = 0.01,
                          num_iterations = 100):

    # Intialize 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):
        # Calcuated 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
```

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

```
Final estimate of b and w:  0.42844895817391493 [0.48270238 0.752659
69 0.46109174]
```

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

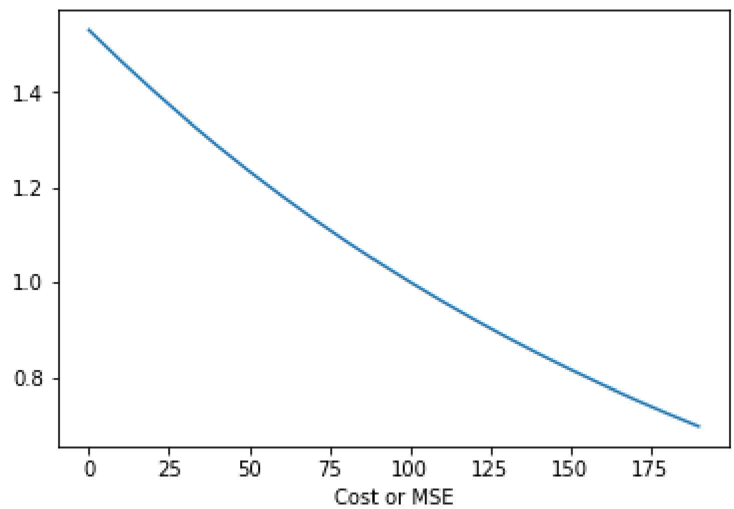
	iteration	cost
0	0.0	1.530310
1	10.0	1.465201
2	20.0	1.403145
3	30.0	1.343996
4	40.0	1.287615
5	50.0	1.233868
6	60.0	1.182630
7	70.0	1.133780
8	80.0	1.087203
9	90.0	1.042793

6.2.1.6 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]
```

```
alpha_df_1, b, w = run_gradient_descent( X, Y, alpha = 0.01, num_iterations = 20
00 )
```

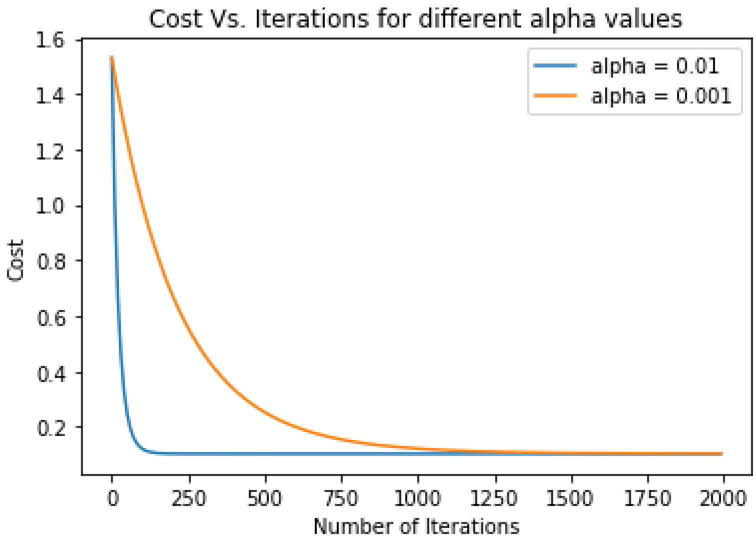
```
Final estimate of b and w:  2.7728016698178713e-16 [ 0.75306591  0.5
3648155 -0.00433069]
```

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 = 2
000 )
```

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

```
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');
```



6.3 scikit-learn Library for Machine Learning

6.3.1 Steps for Building Machine Learning Models

6.3.1.1 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
for reproducibility
```

```
len( X_train )
```

```
140
```

```
len( X_test )
```

```
60
```

6.3.1.2 Building Linear Regression model with train dataset

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

```
## Initiliazing 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=None,
         normalize=False)
```

```
linreg.intercept_
```

```
2.708949092515912
```

```
linreg.coef_
```

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

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

```
[('TV', 0.0440592809574652),
 ('Radio', 0.1992874968989395),
 ('Newspaper', 0.0068824522222754)]
```

6.3.1.3 Making prediction on test set

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

```
# Creating a DataFrame with three 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

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

R-Squared Value

```
## y_train contain 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 Sqaured: ", r2)
```

```
R Sqaured:  0.9055159502227753
```

RMSE

```
# 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
```

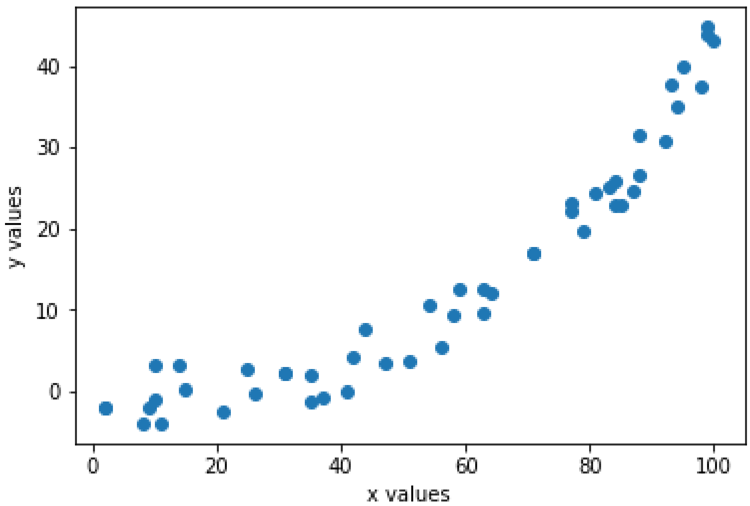
6.3.2 Bias-Variance Trade-off

```
# 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

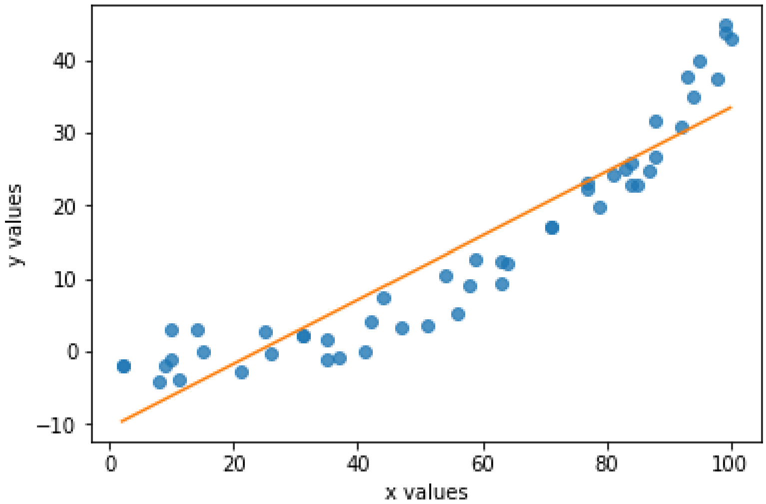
```
plt.scatter( curve.x, curve.y );
plt.xlabel("x values")
plt.ylabel("y values")
```

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

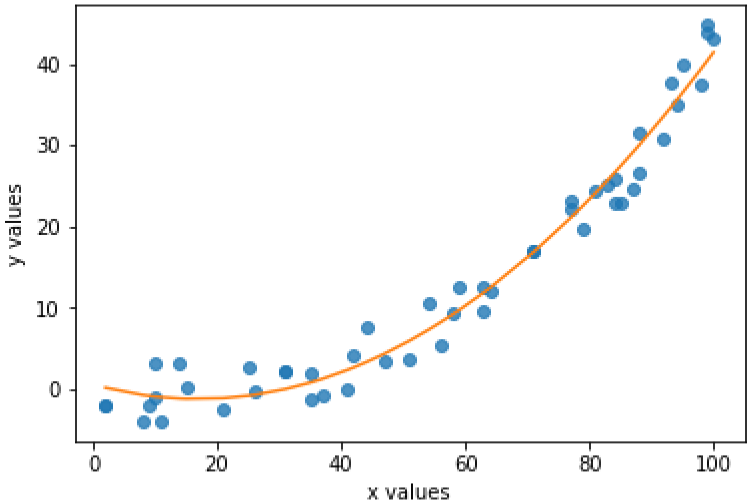



```
# Input
# degree - polynomial terms to be used in the model
def fit_poly( degree ):
    # calling numpy method polyfit
    p = np.polyfit( curve.x, curve.y, deg = degree )
    curve['fit'] = np.polyval( p, curve.x )
    # draw the regression line after fitting the model
    sn.regplot( curve.x, curve.y,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

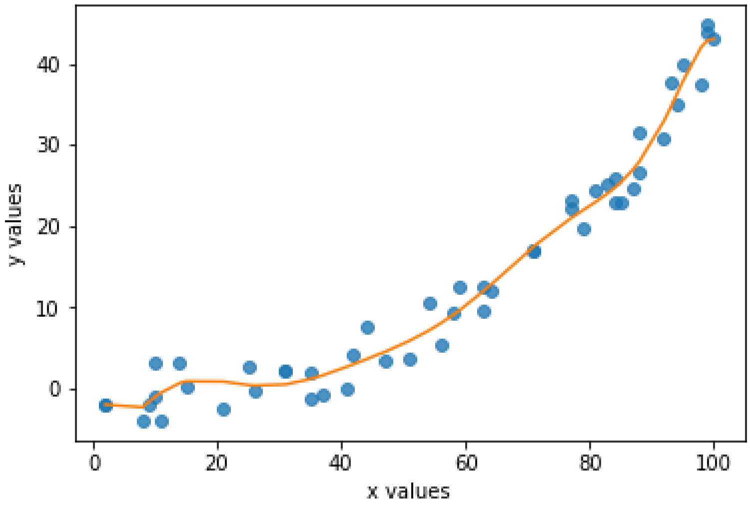
```
fit_poly( 1 );
## Plotting the model form and the data
plt.xlabel("x values")
plt.ylabel("y values");
```




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



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

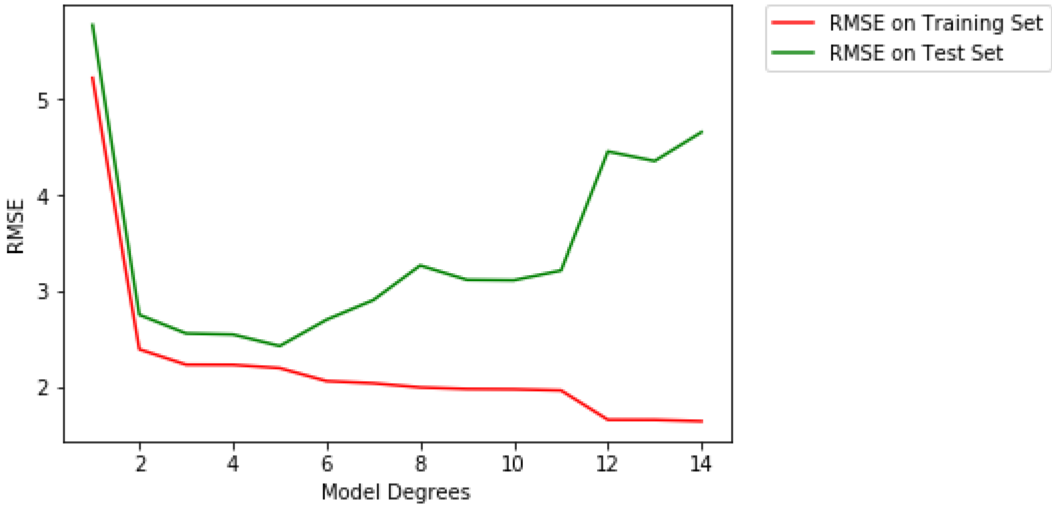



```
# Split the dataset into 60:40 split into training and test set
train_X, test_X, train_y, test_y = train_test_split( curve.x,
                                                     curve.y,
                                                     test_size = 0.40,
                                     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

```
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

```
# 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");
```



6.4 Advanced Regression Models

6.4.1.1 Loading IPL Dataset

```
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
```

```
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', 'SIXERS', 'RUNS-C', 'WKTS',
              'AVE-BL', 'ECON', 'SR-BL']
```

```
# categorical_features is initialized with the categorical variable 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 )
```

```
ipl_auction_encoded_df.columns
```

```
Index(['T-RUNS', 'T-WKTS', 'ODI-RUNS-S', 'ODI-SR-B', 'ODI-WKTS', 'OD
I-SR-BL',
       'RUNS-S', 'HS', 'AVE', 'SR-B', 'SIXERS', 'RUNS-C', 'WKTS', 'A
VE-BL',
       'ECON', 'SR-BL', 'AGE_2', 'AGE_3', 'COUNTRY_BAN', 'COUNTRY_EN
G',
       'COUNTRY_IND', 'COUNTRY_NZ', 'COUNTRY_PAK', 'COUNTRY_SA', 'CO
UNTRY_SL',
       'COUNTRY_WI', 'COUNTRY_ZIM', 'PLAYING ROLE_Batsman',
       'PLAYING ROLE_Bowler', 'PLAYING ROLE_W. Keeper', 'CAPTAINCY E
XP_1'],
      dtype='object')
```

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

6.4.1.2 Standardize X & 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()
```

```
/Users/manaranjan/anaconda/lib/python3.5/site-packages/sklearn/prepr
ocessing/data.py:617: DataConversionWarning: Data with input dtype u
int8, int64, float64 were all converted to float64 by StandardScale
r.
  return self.partial_fit(X, y)
/Users/manaranjan/anaconda/lib/python3.5/site-packages/sklearn/base.
py:462: DataConversionWarning: Data with input dtype uint8, int64, f
loat64 were all converted to float64 by StandardScaler.
  return self.fit(X, **fit_params).transform(X)
```

6.4.1.3 Split 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)
```

6.4.1.4 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=None,
         normalize=False)
```

```
linreg.coef_
```

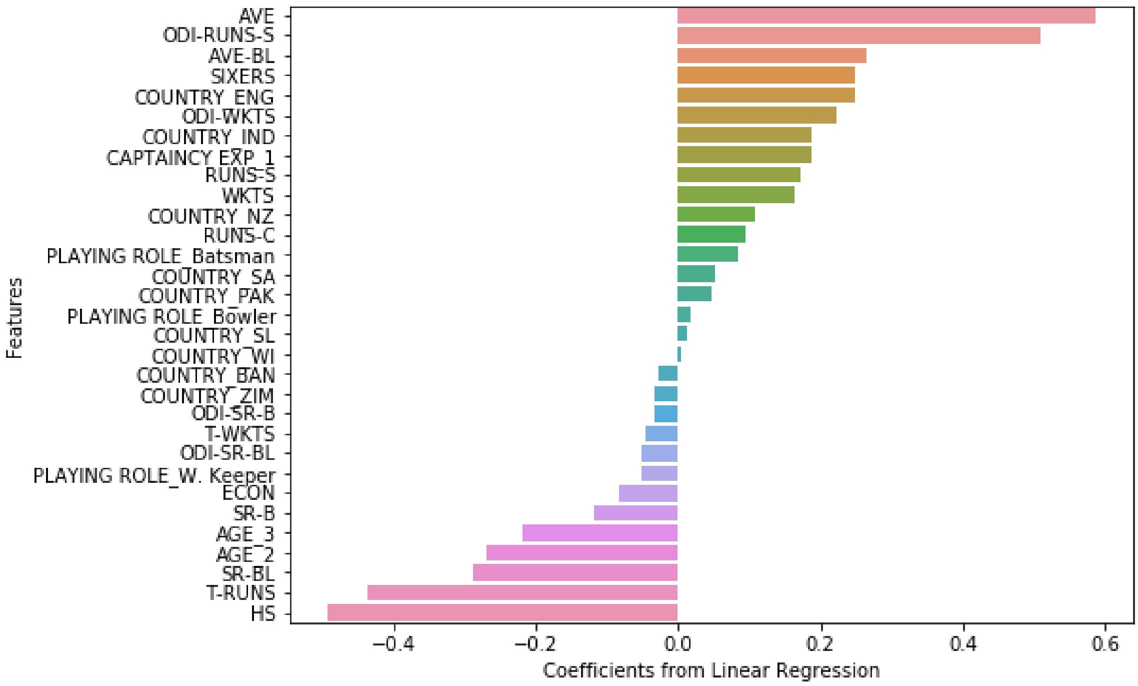
```
array([-0.43539611, -0.04632556,  0.50840867, -0.03323988,  0.222037
7 ,
       -0.05065703,  0.17282657, -0.49173336,  0.58571405, -0.116547
53,
        0.24880095,  0.09546057,  0.16428731,  0.26400753, -0.082533
41,
       -0.28643889, -0.26842214, -0.21910913, -0.02622351,  0.248178
98,
        0.18760332,  0.10776084,  0.04737488,  0.05191335,  0.012352
45,
        0.00547115, -0.03124706,  0.08530192,  0.01790803, -0.050774
54,
        0.18745577])
```

```
## 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

```
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")
```

```
Text(0,0.5,'Features')
```



6.4.1.6 Calculate R-Squared value

```
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
```

6.4.2 Applying Regularization

6.4.2.1 Ridge Regression

```
# 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, normal
ize=False,
   random_state=None, solver='auto', tol=0.001)
```

```
get_train_test_rmse( ridge )
```

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

```
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
```

6.4.2.2 Lasso Regression

```
# Importing Ridge 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=N
one,
   selection='cyclic', tol=0.0001, warm_start=False)
```

```
get_train_test_rmse( lasso )
```

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

```
## 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

6.4.2.3 Elastic Net Regression

```
0.01/1.01
```

```
0.009900990099009901
```

```
from sklearn.linear_model import ElasticNet

enet = ElasticNet(alpha = 1.01, l1_ratio = 0.0099, max_iter = 500)
enet.fit( X_train, y_train )
get_train_test_rmse( enet )
```

```
train:  0.794  test: 0.674
```

6.5 More Advanced Algorithms

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

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

```
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
```

6.5.1 Dealing with imbalanced datasets

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

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

```
## 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,     # sample with replacement
                                 n_samples=2000)

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

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

```
# 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']
```

```
## 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
```

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

```
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

```
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)
```

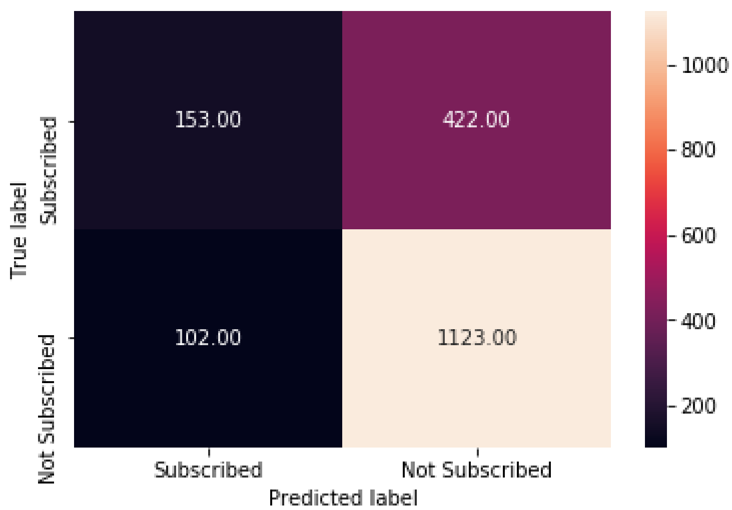
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_interce
pt=True,
          intercept_scaling=1, max_iter=100, multi_class='warn',
          n_jobs=None, penalty='l2', random_state=None, solver='war
n',
          tol=0.0001, verbose=0, warm_start=False)
```

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

6.5.2.2 Confusion Matrix

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

## Defining the matrix to draw the confusion metrix from actual and predicted cl
ass labels
def draw_cm( actual, predicted ):
    # Invoking confusion_matrix from metric package. The matrix will oriented as
[1,0] i.e.
    # the classes with label 1 will be reprensted the first row and 0 as second
 row
    cm = metrics.confusion_matrix( actual, predicted, [1,0] )
    ## Confustion will be plotted as heatmap for better visualization
    ## The lables 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()
```

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



6.5.2.3 Classification Report

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

```
              precision    recall  f1-score   support

           0       0.73      0.92      0.81      1225
           1       0.60      0.27      0.37       575

   micro avg       0.71      0.71      0.71      1800
   macro avg       0.66      0.59      0.59      1800
weighted avg       0.69      0.71      0.67      1800
```

6.5.2.4 ROC 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	0.704479	0.295521
1	0.853664	0.146336
2	0.666963	0.333037
3	0.588329	0.411671
4	0.707982	0.292018

```
## Initializing the DataFrame with actual class lables
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)
```

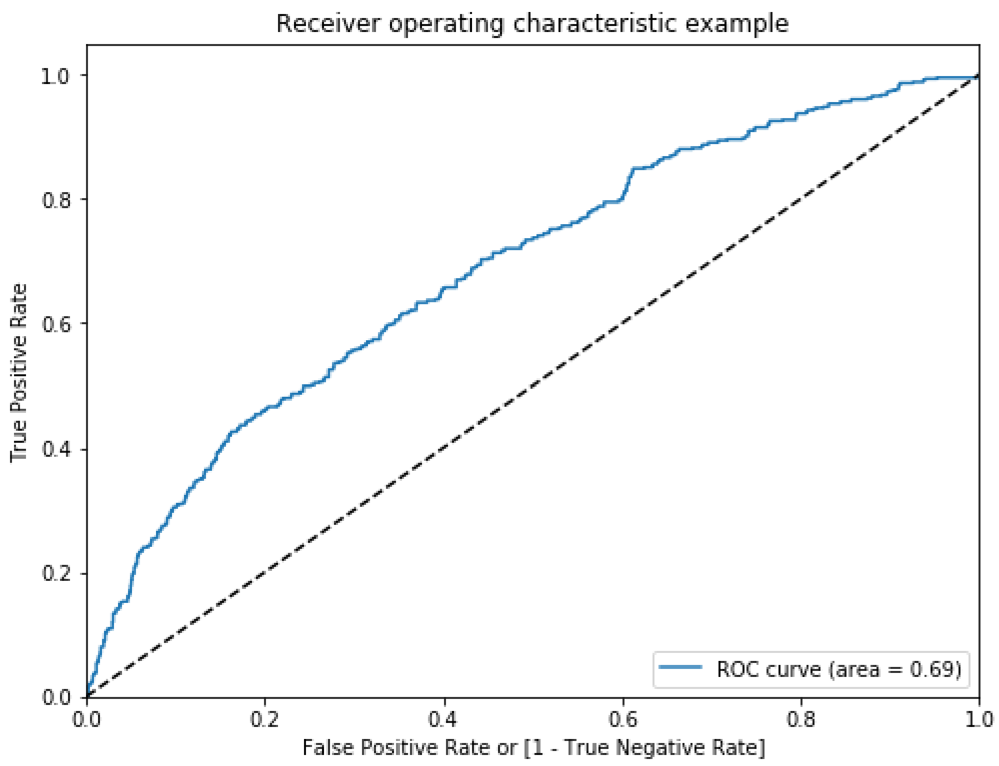
	index	actual	chd_1
0	1321	0	0.295521
1	3677	0	0.146336
2	1680	1	0.333037
3	821	0	0.411671
4	921	0	0.292018

```
# Passing actual class labels and the predicted probability values to compute RO
C AUC score.
auc_score = metrics.roc_auc_score( test_results_df.actual, test_results_df.chd_1
)
round( float( auc_score ), 2 )
```

```
0.69
```

```
## The method takes the three following 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 the 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 the 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
```

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



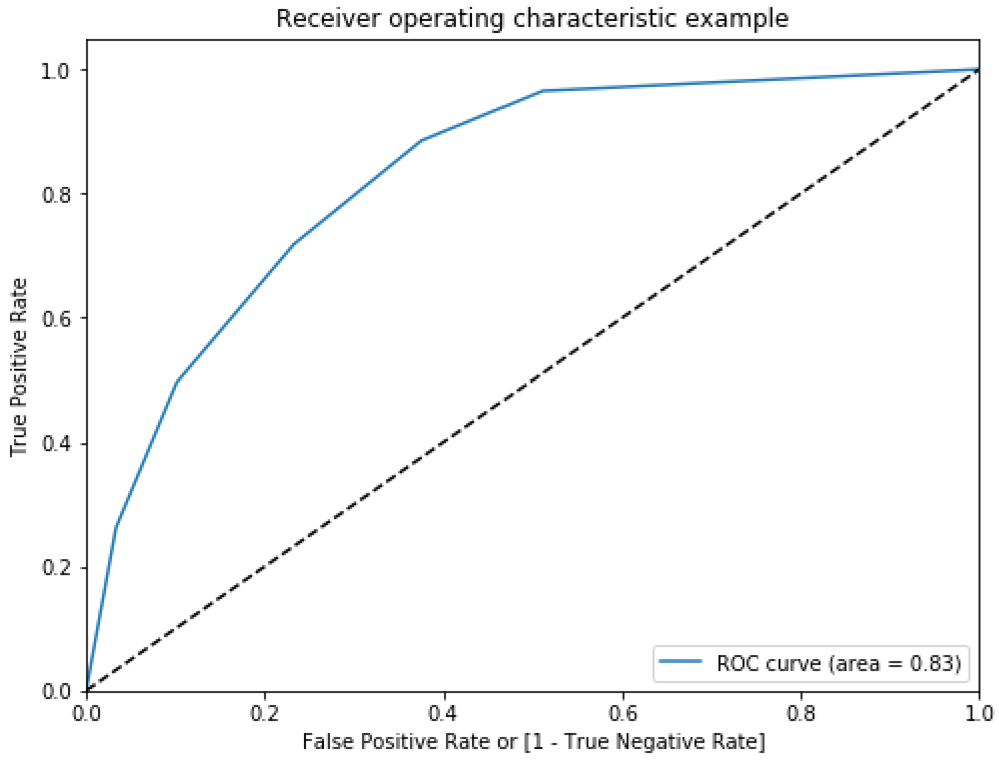
6.5.3 KNN Algorithm

```
## 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 )
```

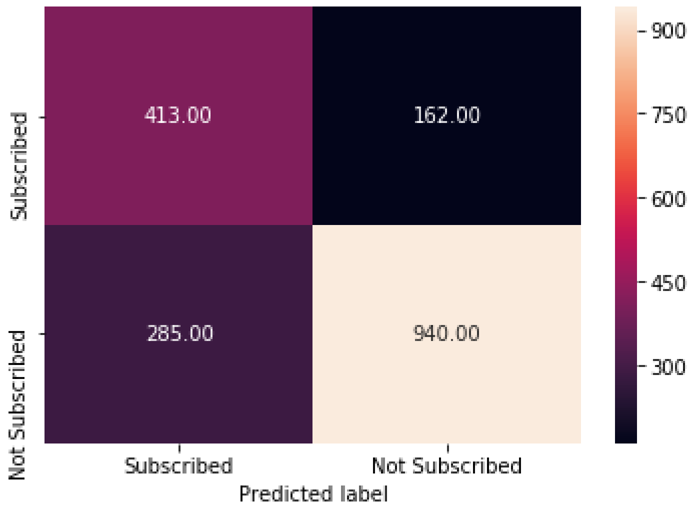
```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkows
ki',
           metric_params=None, n_jobs=None, n_neighbors=5, p=2,
           weights='uniform')
```

6.5.3.1 KNN Accuracy

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




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



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

```
              precision    recall  f1-score   support

           0       0.85      0.77      0.81      1225
           1       0.59      0.72      0.65       575

   micro avg       0.75      0.75      0.75      1800
   macro avg       0.72      0.74      0.73      1800
weighted avg       0.77      0.75      0.76      1800
```

6.5.3.2 GridSerach for most optimal parameters

```
## 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 )
```

```
GridSearchCV(cv=10, error_score='raise-deprecating',
       estimator=KNeighborsClassifier(algorithm='auto', leaf_size=3
0, metric='minkowski',
           metric_params=None, n_jobs=None, n_neighbors=5, p=2,
           weights='uniform'),
       fit_params=None, iid='warn', n_jobs=None,
       param_grid=[{'n_neighbors': range(5, 10), 'metric': ['canberr
a', 'euclidean', 'minkowski']}],
       pre_dispatch='2*n_jobs', refit=True, return_train_score='war
n',
       scoring='roc_auc', verbose=0)
```

```
clf.best_score_
```

```
0.8368537419503068
```

```
clf.best_params_
```

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

6.5.4 Ensemble Methods

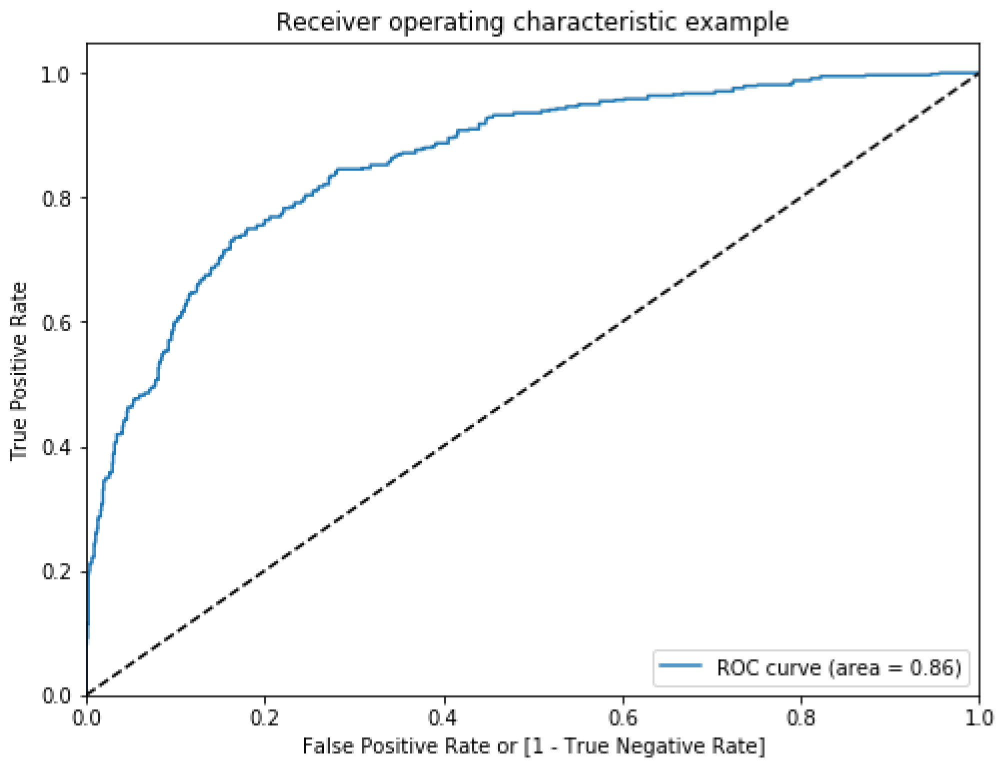
6.5.5 Random Forest

6.5.5.1 Buiding Random Forest Model

```
## Importing Random Forest Classifier from the sklearn.ensemble
from sklearn.ensemble import RandomForestClassifier

## Initializing the Random Forest Classifier with max_dept and n_estimators
radm_clf = RandomForestClassifier( max_depth=10, n_estimators=10)
radm_clf.fit( train_X, train_y )
```

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion
='gini',
            max_depth=10, max_features='auto', max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=No
ne,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False)
```

```
_, _, _, _ = draw_roc_curve( radm_clf, test_X, test_y );
```



6.5.5.2 Grid Search for Optimal Parameters

```
## Configuring parameters and values for searched
tuned_parameters = [{'max_depth': [10, 15],
                    'n_estimators': [10,20],
                    'max_features': ['sqrt', 'auto']}]

## Initializing the RF classifier
radm_clf = RandomForestClassifier()

## Configuring search with the tunable parameters
clf = GridSearchCV(radm_clf,
                  tuned_parameters,
                  cv=5,
                  scoring='roc_auc')

## Fitting the training set
clf.fit(train_X, train_y )
```

```
GridSearchCV(cv=5, error_score='raise-deprecating',
       estimator=RandomForestClassifier(bootstrap=True, class_weight
=None, criterion='gini',
            max_depth=None, max_features='auto', max_leaf_nodes=Non
e,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators='warn', n_job
s=None,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False),
       fit_params=None, iid='warn', n_jobs=None,
       param_grid=[{'n_estimators': [10, 20], 'max_depth': [10, 15],
'max_features': ['sqrt', 'auto']}],
       pre_dispatch='2*n_jobs', refit=True, return_train_score='war
n',
       scoring='roc_auc', verbose=0)
```

```
clf.best_score_
```

```
0.9399595384858543
```

```
clf.best_params_
```

```
{'max_depth': 15, 'max_features': 'auto', 'n_estimators': 20}
```

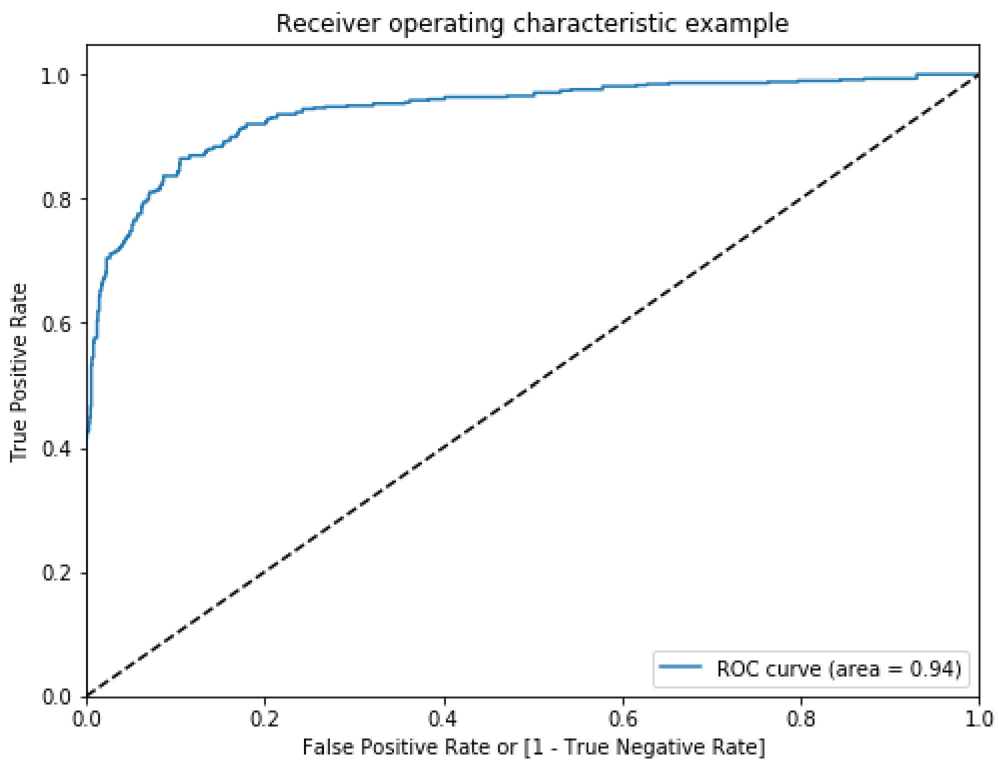
6.5.5.3 Building the final model with optimal parameter values

```
## Initializing the Random Forest Mode with the optimal values
radm_clf = RandomForestClassifier( max_depth=15, n_estimators=20, max_features =
'auto')
## Fitting the model with the training set
radm_clf.fit( train_X, train_y )
```

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion
='gini',
            max_depth=15, max_features='auto', max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=20, n_jobs=No
ne,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False)
```

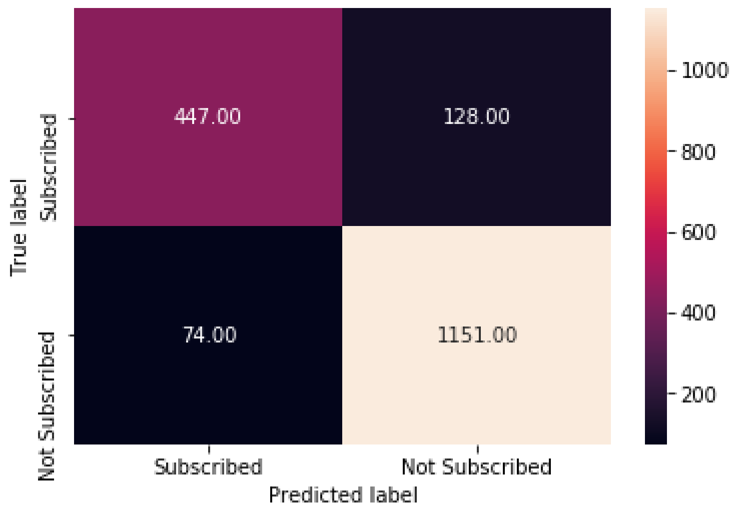
6.5.5.4 ROC AUC Score

```
_, _, _, _ = draw_roc_curve( clf, test_X, test_y )
```



6.5.5.5 Drawing the confusion matrix

```
pred_y = radm_clf.predict( test_X )
draw_cm( test_y, pred_y )
```

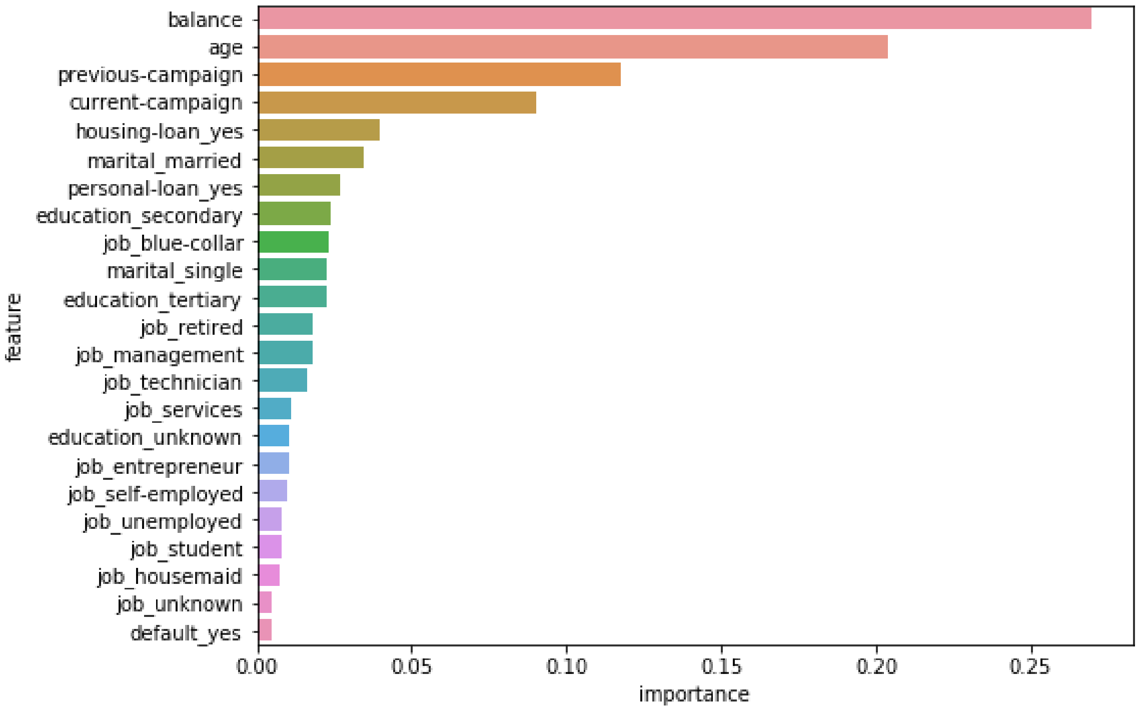



```
print( metrics.classification_report( test_y, pred_y ) )
```

```
              precision    recall  f1-score   support

           0       0.90      0.94      0.92      1225
           1       0.86      0.78      0.82       575

   micro avg       0.89      0.89      0.89      1800
   macro avg       0.88      0.86      0.87      1800
weighted avg       0.89      0.89      0.89      1800
```

6.5.5.6 Finding important features

```
import numpy as np

# Create a dataframe to store the featues and their corresponding importances
feature_rank = pd.DataFrame( { 'feature': train_X.columns,
                               'importance': radm_clf.feature_importances_ } )

## Sorting the features based on their importances with most important feature a
t top.
feature_rank = feature_rank.sort_values('importance', ascending = False)

plt.figure(figsize=(8, 6))
# plot the values
sn.barplot( y = 'feature', x = 'importance', data = feature_rank );
```



```
feature_rank['cumsum'] = feature_rank.importance.cumsum() * 100
feature_rank.head(10)
```

	feature	importance	cumsum
1	balance	0.269603	26.960282
0	age	0.203664	47.326707
3	previous-campaign	0.117525	59.079219
2	current-campaign	0.090085	68.087703
21	housing-loan_yes	0.039898	72.077486
15	marital_married	0.034329	75.510337
22	personal-loan_yes	0.027029	78.213244
17	education_secondary	0.023934	80.606690
4	job_blue-collar	0.023081	82.914811
16	marital_single	0.022495	85.164357

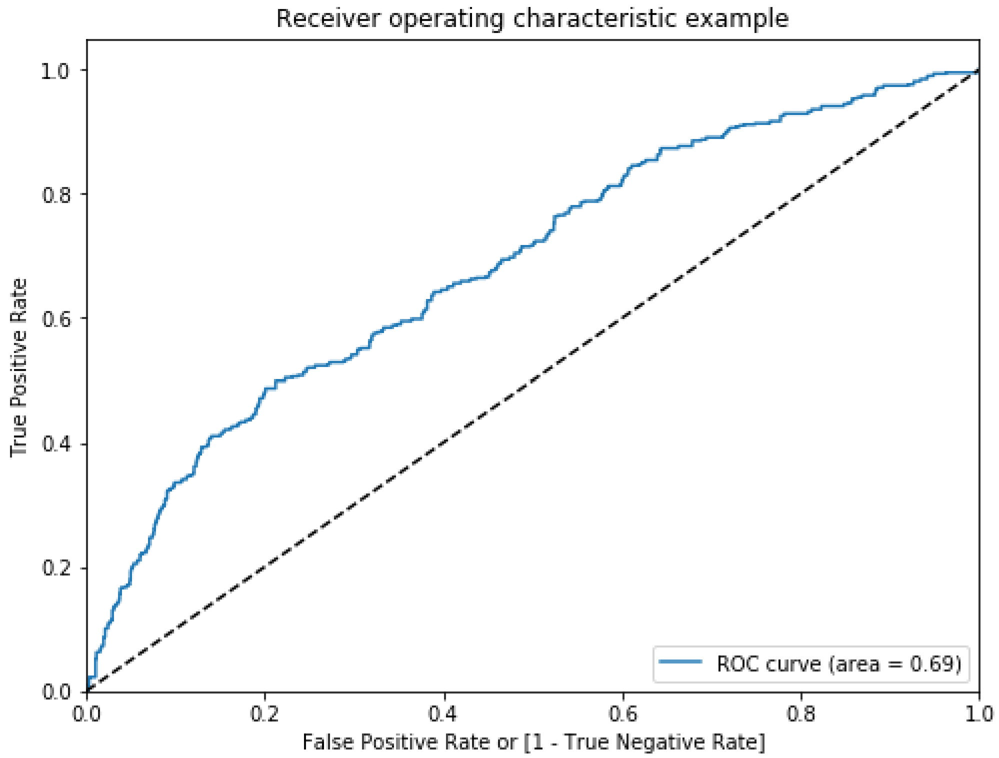
6.5.6 Boosting

6.5.6.1 Adaboost

```
## Importing Adaboost classifier
from sklearn.ensemble import AdaBoostClassifier

## Initializing logistic regression to use as base classifier
logreg_clf = LogisticRegression()

## Initilizing adaboost classifier with 50 classifers
ada_clf = AdaBoostClassifier(logreg_clf, n_estimators=50)

## Fitting adaboost model to training set
ada_clf.fit(train_X, train_y )
```

```
AdaBoostClassifier(algorithm='SAMME.R',
          base_estimator=LogisticRegression(C=1.0, class_weight=Non
e, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='warn',
          n_jobs=None, penalty='l2', random_state=None, solver='war
n',
          tol=0.0001, verbose=0, warm_start=False),
          learning_rate=1.0, n_estimators=50, random_state=None)
```

```
_, _, _, _ = draw_roc_curve( ada_clf, test_X, test_y )
```

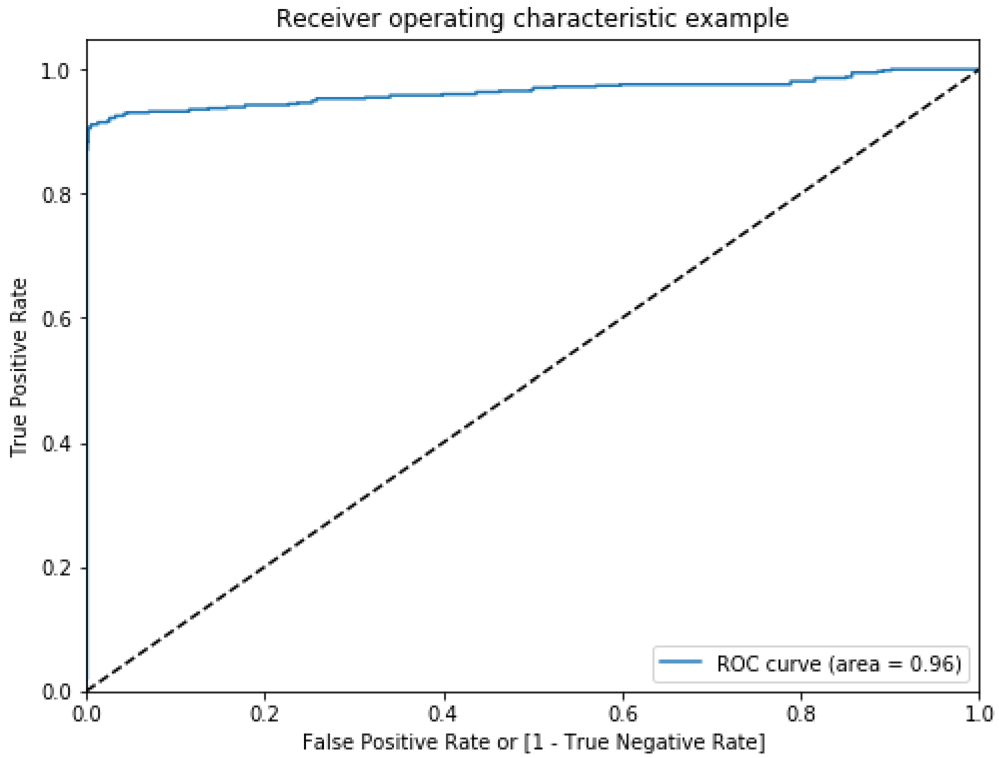


6.5.6.2 Gradient Boosting

```
## Importing Gradient Boosting classifier
from sklearn.ensemble import GradientBoostingClassifier

## Initializing Gradient Boosting with 500 estimators and max depth as 10.
gboost_clf = GradientBoostingClassifier( n_estimators=500, max_depth=10)

## Fitting gradient boosting model to training set
gboost_clf.fit(train_X, train_y )
```

```
GradientBoostingClassifier(criterion='friedman_mse', init=None,
              learning_rate=0.1, loss='deviance', max_depth=10,
              max_features=None, max_leaf_nodes=None,
              min_impurity_decrease=0.0, min_impurity_split=None,
              min_samples_leaf=1, min_samples_split=2,
              min_weight_fraction_leaf=0.0, n_estimators=500,
              n_iter_no_change=None, presort='auto', random_state=No
ne,
              subsample=1.0, tol=0.0001, validation_fraction=0.1,
              verbose=0, warm_start=False)
```

```
_, _, _, _ = draw_roc_curve( gboost_clf, test_X, test_y )
```

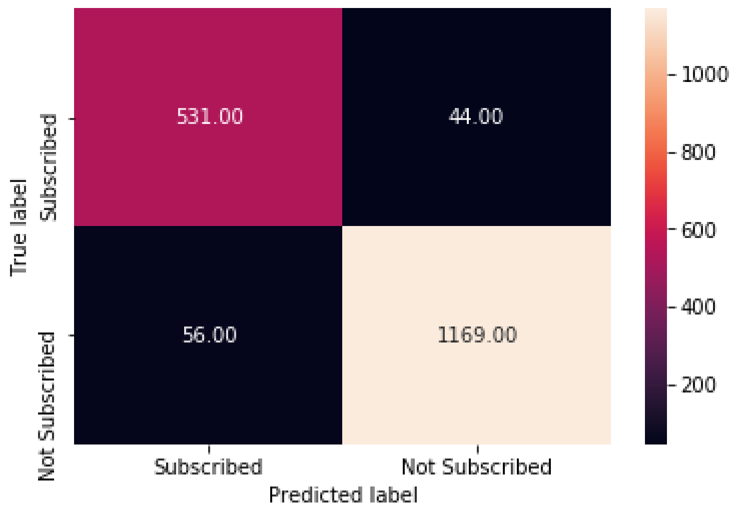



```
from sklearn.model_selection import cross_val_score

gboost_clf = GradientBoostingClassifier( n_estimators=500, max_depth=10)
cv_scores = cross_val_score( gboost_clf, train_X, train_y, cv = 10, scoring = 'r
oc_auc' )
```

```
print( cv_scores )
print( "Mean Accuracy: ", np.mean(cv_scores), " with standard deviation of: ",
np.std(cv_scores))
```

```
[0.98241686 0.98105851 0.98084469 0.9585199  0.95482216 0.96667006
 0.95342452 0.97368689 0.95937357 0.98174607]
Mean Accuracy:  0.969256322542174  with standard deviation of:  0.01
1406249012935668
```

```
gboost_clf.fit(train_X, train_y )
pred_y = gboost_clf.predict( test_X )
draw_cm( test_y, pred_y )
```




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

```
              precision    recall  f1-score   support

           0       0.96      0.95      0.96      1225
           1       0.90      0.92      0.91       575

   micro avg       0.94      0.94      0.94      1800
   macro avg       0.93      0.94      0.94      1800
weighted avg       0.94      0.94      0.94      1800
```

```
import numpy as np

# Create a dataframe to store the featues and their corresponding importances
feature_rank = pd.DataFrame( { 'feature': train_X.columns,
                               'importance': gboost_clf.feature_importances_ } )

## Sorting the features based on their importances with most important feature a
t top.
feature_rank = feature_rank.sort_values('importance', ascending = False)

plt.figure(figsize=(8, 6))
# plot the values
sn.barplot( y = 'feature', x = 'importance', data = feature_rank );
```