Predicting Liver Disease:

The given dataset is related to Indian patients who have been tested for a liver disease. Based on chemical compounds (bilrubin,albumin,protiens,alkaline phosphatase) present in human body and tests like SGOT, SGPT the outcome mentioned is whether person is a patient i.e, whether he needs to be diagnosed further or not.

Objective:

Perform data cleansing, and required transformations and build a predictive model which will be able to predict most of the cases accurately.

Attributes:

Following are the feature names for the given data: Age,Gender,Total_Bilirubin,Direct_Bilirubin,Alkaline_Phosphotase, Alamine_Aminotransferase,Aspartate_Aminotransferase,Total_Protiens,Albumin, Albumin_and_Globulin_Ratio,Class.

Step 1:

Importing the libraries required -

In [54]:
```
import pandas as pd
import numpy as np
import seaborn as sns
from sklearn.utils import resample
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix, accuracy_score, classification_report
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
```

Step 2:

Loading the Data -

In [3]:
```
data = pd.read_csv("C:/Users/Rohit/Downloads/IndianLiverPatientData.txt", delimiter="\t", header=None, index_
col=0)
data.head()
```

Out[3]:

	1	2	3	4	5	6	7	8	9	10	11
0											
1	65	Female	0.7	0.1	187	16	18	6.8	3.3	0.90	No
2	62	Male	10.9	5.5	699	64	100	7.5	3.2	0.74	No
3	62	Male	7.3	4.1	490	60	68	7.0	3.3	0.89	No
4	58	Male	1.0	0.4	182	14	20	6.8	3.4	1.00	No
5	72	Male	3.9	2.0	195	27	59	7.3	2.4	0.40	No

As you can see we have no headings for the columns we give them manually

In [5]:
```
data.columns = ["Age", "Gender", "Total_Bilirubin", "Direct_Bilirubin", "Alkaline_Phosphotase",
                         "Alamine_Aminotransferase", "Aspartate_Aminotransferase", "Total_Protiens", "Albumi
n"
                         , "Albumin_and_Globulin_Ratio", "Class"]
data.head()
```

Out[5]:

	Age	Gender	Total_Bilirubin	Direct_Bilirubin	Alkaline_Phosphotase	Alamine_Aminotransferase	Aspartate_Aminotransferase	Total_P
0								
1	65	Female	0.7	0.1	187	16	18	
2	62	Male	10.9	5.5	699	64	100	
3	62	Male	7.3	4.1	490	60	68	
4	58	Male	1.0	0.4	182	14	20	
5	72	Male	3.9	2.0	195	27	59	

We can check the summary by -

In [8]:
```
summary = data.describe(include='all') # include all so that it takes categorical columns if there exist any
print(summary)
```

```
               Age Gender  Total_Bilirubin  Direct_Bilirubin  \
count   583.000000    563       583.000000        583.000000
unique         NaN      2              NaN               NaN
top            NaN   Male              NaN               NaN
freq           NaN    421              NaN               NaN
mean     44.746141    NaN         3.298799          1.486106
std      16.189833    NaN         6.209522          2.808498
min       4.000000    NaN         0.400000          0.100000
25%      33.000000    NaN         0.800000          0.200000
50%      45.000000    NaN         1.000000          0.300000
75%      58.000000    NaN         2.600000          1.300000
max      90.000000    NaN        75.000000         19.700000

        Alkaline_Phosphotase  Alamine_Aminotransferase  \
count             583.000000                583.000000
unique                   NaN                       NaN
top                      NaN                       NaN
freq                     NaN                       NaN
mean              290.576329                 80.713551
std               242.937989                182.620356
min                63.000000                 10.000000
25%               175.500000                 23.000000
50%               208.000000                 35.000000
75%               298.000000                 60.500000
max              2110.000000               2000.000000

        Aspartate_Aminotransferase  Total_Protiens     Albumin  \
count                   583.000000      568.000000  583.000000
unique                         NaN             NaN         NaN
top                            NaN             NaN         NaN
freq                           NaN             NaN         NaN
mean                    109.910806        6.483979    3.141852
std                     288.918529        1.084039    0.795519
min                      10.000000        2.700000    0.900000
25%                      25.000000        5.800000    2.600000
50%                      42.000000        6.600000    3.100000
75%                      87.000000        7.200000    3.800000
max                    4929.000000        9.600000    5.500000

        Albumin_and_Globulin_Ratio Class
count                   579.000000   583
unique                         NaN     2
top                            NaN    No
```

```
freq                          NaN   416
mean                     0.947064   NaN
std                      0.319592   NaN
min                      0.300000   NaN
25%                      0.700000   NaN
50%                      0.930000   NaN
75%                      1.100000   NaN
max                      2.800000   NaN
```

Step 3:

Checking for null values -

In [9]:
```
data.isnull().sum()
```

Out[9]:
```
Age                            0
Gender                        20
Total_Bilirubin                0
Direct_Bilirubin               0
Alkaline_Phosphotase           0
Alamine_Aminotransferase       0
Aspartate_Aminotransferase     0
Total_Protiens                15
Albumin                        0
Albumin_and_Globulin_Ratio     4
Class                          0
dtype: int64
```

In [10]:
```
# creating a copy of data if we mess something up
data1 = data.copy()
```

In [12]:
```
data1['Gender'].value_counts()
```

Out[12]:
```
Male      421
Female    142
Name: Gender, dtype: int64
```

In [13]:
```
# as we have more values for "Male" we fill the missing values in Gender column with "Male"
data1['Gender'].fillna(data1['Gender'].mode()[0], inplace=True)
```

In [14]:
```
# and we do the following for Albumin_and_Globulin_Ratio, Total_Protiens
for value in ['Albumin_and_Globulin_Ratio', 'Total_Protiens']:
    data1[value].fillna(data1[value].median(), inplace=True)
```

In [15]:
```
# to check if NAs still exist
data1.isnull().sum()
```

Out[15]:
```
Age                           0
Gender                        0
Total_Bilirubin               0
Direct_Bilirubin              0
Alkaline_Phosphotase          0
Alamine_Aminotransferase      0
Aspartate_Aminotransferase    0
Total_Protiens                0
Albumin                       0
Albumin_and_Globulin_Ratio    0
Class                         0
dtype: int64
```

Step 4:

Performing Label Encoding on categorical columns -

In [16]:
```
col_name = ['Gender', 'Class']
from sklearn import preprocessing
le={}
for value in col_name:
    le[value]=preprocessing.LabelEncoder()
for value in col_name:
    data1[value] = le[value].fit_transform(data1[value])

data1.info() # see if all values are either float or int
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 583 entries, 1 to 583
Data columns (total 11 columns):
Age                           583 non-null int64
Gender                        583 non-null int32
Total_Bilirubin               583 non-null float64
Direct_Bilirubin              583 non-null float64
Alkaline_Phosphotase          583 non-null int64
Alamine_Aminotransferase      583 non-null int64
Aspartate_Aminotransferase    583 non-null int64
Total_Protiens                583 non-null float64
Albumin                       583 non-null float64
Albumin_and_Globulin_Ratio    583 non-null float64
Class                         583 non-null int32
dtypes: float64(5), int32(2), int64(4)
memory usage: 50.1 KB
```

Step 5:

To check for Outliers -

In [19]:
```
summary2 = data1.describe(np.arange(0.9, 0.99, 0.01), include='all')
print(summary2)
```

```
             Age      Gender  Total_Bilirubin  Direct_Bilirubin  \
count  583.000000  583.000000       583.000000        583.000000   
mean    44.746141    0.756432         3.298799          1.486106   
std     16.189833    0.429603         6.209522          2.808498   
min      4.000000    0.000000         0.400000          0.100000   
50%     45.000000    1.000000         1.000000          0.300000   
90%     66.000000    1.000000         7.860000          4.080000   
91%     66.000000    1.000000         8.900000          4.500000   
92%     67.440000    1.000000        10.900000          5.144000   
93%     69.260000    1.000000        12.256000          6.252000   
94%     70.000000    1.000000        14.816000          7.608000   
95%     72.000000    1.000000        16.350000          8.400000   
96%     72.720000    1.000000        18.288000          8.972000   
97%     74.540000    1.000000        20.108000         10.108000   
98%     75.000000    1.000000        23.072000         11.736000   
max     90.000000    1.000000        75.000000         19.700000   

       Alkaline_Phosphotase  Alamine_Aminotransferase  \
count            583.000000                583.000000   
mean             290.576329                 80.713551   
std              242.937989                182.620356   
min               63.000000                 10.000000   
50%              208.000000                 35.000000   
90%              511.400000                140.000000   
91%              549.440000                153.240000   
92%              574.440000                166.880000   
93%              610.520000                183.080000   
94%              661.240000                198.560000   
95%              698.100000                232.000000   
96%              792.480000                341.440000   
97%              908.560000                409.700000   
98%             1103.600000                549.680000   
max             2110.000000               2000.000000   

       Aspartate_Aminotransferase  Total_Protiens     Albumin  \
count                  583.000000      583.000000  583.000000   
mean                   109.910806        6.486964    3.141852   
std                    288.918529        1.070136    0.795519   
min                     10.000000        2.700000    0.900000   
50%                     42.000000        6.600000    3.100000   
90%                    190.000000        7.900000    4.100000   
91%                    230.620000        7.900000    4.200000   
92%                    239.960000        8.000000    4.200000   
```

```
93%                   277.600000        8.000000    4.300000
94%                   351.360000        8.000000    4.300000
95%                   400.900000        8.100000    4.390000
96%                   507.920000        8.200000    4.400000
97%                   626.780000        8.300000    4.500000
98%                   846.160000        8.500000    4.636000
max                  4929.000000        9.600000    5.500000

       Albumin_and_Globulin_Ratio       Class
count                  583.000000  583.000000
mean                     0.946947    0.286449
std                      0.318495    0.452490
min                      0.300000    0.000000
50%                      0.930000    0.000000
90%                      1.300000    1.000000
91%                      1.380000    1.000000
92%                      1.400000    1.000000
93%                      1.400000    1.000000
94%                      1.400000    1.000000
95%                      1.500000    1.000000
96%                      1.507200    1.000000
97%                      1.600000    1.000000
98%                      1.700000    1.000000
max                      2.800000    1.000000
```

In [20]:
```
data2 = data1.loc[data1["Alkaline_Phosphotase"] < 909]
# here we took those samples where the specified column had value below 97th %ile

data3 = data2.loc[data2["Alamine_Aminotransferase"] < 500]
# here we took those samples where the specified column had value below 98th %ile

# we took this values because they seemed to have no outliers beyond those values
```

Step 6:

Checking multicollinearity and removing it -

In [23]:
```
# independent variables
x = data3[['Age', 'Gender', 'Total_Bilirubin', 'Direct_Bilirubin',
           'Alkaline_Phosphotase', 'Alamine_Aminotransferase',
           'Aspartate_Aminotransferase', 'Total_Protiens', 'Albumin',
           'Albumin_and_Globulin_Ratio']]

# dependent variables
y = data3[['Class']]


corr2 = x.corr(method='pearson')
corr2
```

Out[23]:

	Age	Gender	Total_Bilirubin	Direct_Bilirubin	Alkaline_Phosphotase	Alamine_Aminotransferase	As
Age	1.000000	0.063305	0.020704	0.012033	0.009075	-0.053116	
Gender	0.063305	1.000000	0.107551	0.121324	0.054975	0.117867	
Total_Bilirubin	0.020704	0.107551	1.000000	0.860373	0.184910	0.223902	
Direct_Bilirubin	0.012033	0.121324	0.860373	1.000000	0.209227	0.264239	
Alkaline_Phosphotase	0.009075	0.054975	0.184910	0.209227	1.000000	0.245415	
Alamine_Aminotransferase	-0.053116	0.117867	0.223902	0.264239	0.245415	1.000000	
Aspartate_Aminotransferase	-0.007257	0.124726	0.294169	0.328998	0.156771	0.774318	
Total_Protiens	-0.185952	-0.096330	0.006380	0.012968	0.024365	0.067497	
Albumin	-0.265396	-0.114106	-0.224084	-0.234582	-0.136099	-0.001841	
Albumin_and_Globulin_Ratio	-0.207842	-0.024835	-0.200034	-0.194982	-0.224555	-0.075860	

In [25]:
```
# We remove albumin as it is correlated to both 'total protein' and 'albumin and globulin ratio'

# and we check the multicollinearity again
x = data3[['Age', 'Gender', 'Total_Bilirubin', 'Direct_Bilirubin',
           'Alkaline_Phosphotase', 'Alamine_Aminotransferase',
           'Aspartate_Aminotransferase', 'Total_Protiens',
           'Albumin_and_Globulin_Ratio']]
coor3 = x.corr(method = "pearson")
coor3
```

Out[25]:

	Age	Gender	Total_Bilirubin	Direct_Bilirubin	Alkaline_Phosphotase	Alamine_Aminotransferase	As
Age	1.000000	0.063305	0.020704	0.012033	0.009075	-0.053116	
Gender	0.063305	1.000000	0.107551	0.121324	0.054975	0.117867	
Total_Bilirubin	0.020704	0.107551	1.000000	0.860373	0.184910	0.223902	
Direct_Bilirubin	0.012033	0.121324	0.860373	1.000000	0.209227	0.264239	
Alkaline_Phosphotase	0.009075	0.054975	0.184910	0.209227	1.000000	0.245415	
Alamine_Aminotransferase	-0.053116	0.117867	0.223902	0.264239	0.245415	1.000000	
Aspartate_Aminotransferase	-0.007257	0.124726	0.294169	0.328998	0.156771	0.774318	
Total_Protiens	-0.185952	-0.096330	0.006380	0.012968	0.024365	0.067497	
Albumin_and_Globulin_Ratio	-0.207842	-0.024835	-0.200034	-0.194982	-0.224555	-0.075860	

In [26]:
```
# 'Alamine_Aminotransferase' is more specific to liver than 'Aspartate_Aminotransferase' so we keep
#'Alamine_Aminotransferase' and drop 'Aspartate_Aminotransferase'

x = data3[['Age', 'Gender', 'Total_Bilirubin', 'Direct_Bilirubin',
           'Alkaline_Phosphotase', 'Alamine_Aminotransferase',
           'Total_Protiens', 'Albumin_and_Globulin_Ratio']]
coor4 = x.corr(method="pearson")
coor4
```

Out[26]:

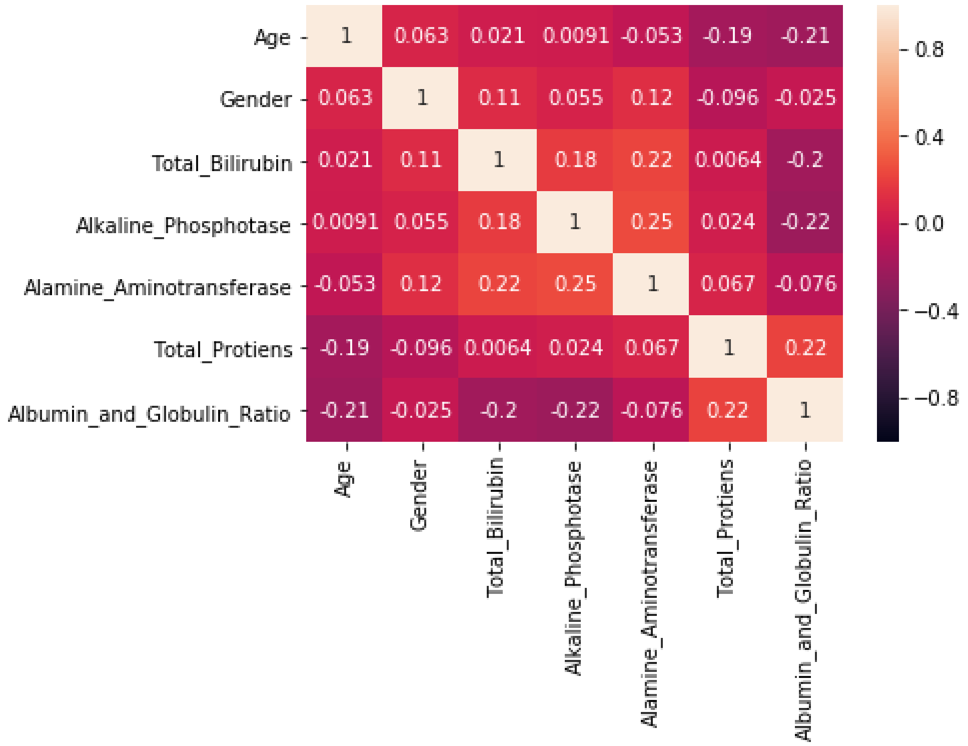
	Age	Gender	Total_Bilirubin	Direct_Bilirubin	Alkaline_Phosphotase	Alamine_Aminotransferase	Tot
Age	1.000000	0.063305	0.020704	0.012033	0.009075	-0.053116	
Gender	0.063305	1.000000	0.107551	0.121324	0.054975	0.117867	
Total_Bilirubin	0.020704	0.107551	1.000000	0.860373	0.184910	0.223902	
Direct_Bilirubin	0.012033	0.121324	0.860373	1.000000	0.209227	0.264239	
Alkaline_Phosphotase	0.009075	0.054975	0.184910	0.209227	1.000000	0.245415	
Alamine_Aminotransferase	-0.053116	0.117867	0.223902	0.264239	0.245415	1.000000	
Total_Protiens	-0.185952	-0.096330	0.006380	0.012968	0.024365	0.067497	
Albumin_and_Globulin_Ratio	-0.207842	-0.024835	-0.200034	-0.194982	-0.224555	-0.075860	

In [27]:
```
# As 'Total bilirubin' considers both water and fat soluble substances we drop 'direct bilirubin'

x = data3[['Age', 'Gender', 'Total_Bilirubin',
           'Alkaline_Phosphotase', 'Alamine_Aminotransferase',
           'Total_Protiens', 'Albumin_and_Globulin_Ratio']]
coor5 = x.corr(method="pearson")
```

In [28]:
```python
# we check the same with the help of heat map whether removing the above columns helped in reducing multicoll
inearity
sns.heatmap(coor5, vmin=-1, vmax=1, annot=True)
```

Out[28]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x123c6357f08>
```



	Age	Gender	Total_Bilirubin	Alkaline_Phosphotase	Alamine_Aminotransferase	Total_Protiens	Albumin_and_Globulin_Ratio
Age	1	0.063	0.021	0.0091	-0.053	-0.19	-0.21
Gender	0.063	1	0.11	0.055	0.12	-0.096	-0.025
Total_Bilirubin	0.021	0.11	1	0.18	0.22	0.0064	-0.2
Alkaline_Phosphotase	0.0091	0.055	0.18	1	0.25	0.024	-0.22
Alamine_Aminotransferase	-0.053	0.12	0.22	0.25	1	0.067	-0.076
Total_Protiens	-0.19	-0.096	0.0064	0.024	0.067	1	0.22
Albumin_and_Globulin_Ratio	-0.21	-0.025	-0.2	-0.22	-0.076	0.22	1

Step 7:

Resampling (*Class Balancing*)

In [29]:
```python
data3['Class'].value_counts()
```

Out[29]:
```
0    388
1    166
Name: Class, dtype: int64
```

We observe here that our dependent variable has class imbalanced as class 0 has 388 observations and class 1 has 166 observations. Hence to resolve this issue we upsample the class 1 to class 0 so that it matches perfectly and doesn't affect our predictions.

In [31]:
```
data4 = pd.concat([x, y], axis=1)
df_major = data4[data4['Class'] == 0]
df_minor = data4[data4['Class'] == 1]

upSample = resample(df_minor, replace=True, n_samples=388, random_state=0)

data5 = pd.concat([df_major, upSample])
data5['Class'].value_counts()
```

Out[31]:
```
1    388
0    388
Name: Class, dtype: int64
```

As you can see we have matched both the classes

Step 8:

Scaling the data

In [32]:
```
x = data5[['Age', 'Gender', 'Total_Bilirubin',
           'Alkaline_Phosphotase', 'Alamine_Aminotransferase',
           'Total_Protiens', 'Albumin_and_Globulin_Ratio']]
y = data5['Class']

from sklearn.preprocessing import StandardScaler
scale = StandardScaler()
scale.fit(x)
x = scale.transform(x)
```

Step 9:

Splitting the data into training and testing

In [33]:
```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=0)
```

Step 10:

Performing Logistic Regression

In [35]:
```
model_lr = LogisticRegression()
model_lr.fit(x_train, y_train)

y_pred = model_lr.predict(x_test)
```

In [36]:
```
# confusion matrix and some othe measures
cfm = confusion_matrix(y_test, y_pred)

print(cfm)
print(classification_report(y_test, y_pred))
print(accuracy_score(y_test, y_pred))
```

```
[[41 30]
 [19 66]]
              precision    recall  f1-score   support

           0       0.68      0.58      0.63        71
           1       0.69      0.78      0.73        85

    accuracy                           0.69       156
   macro avg       0.69      0.68      0.68       156
weighted avg       0.69      0.69      0.68       156

0.6858974358974359
```

The catch here is that while dividing the classes as the probability of being 1, the GLM by default assumes the threshold as 0.5 Hence we need to decide which threshold suits better by checking for errors manually.

In [37]:
```
# this gives us a data frame for probabilty of each observation being 0 or 1
y_pred_prob = model_lr.predict_proba(x_test)
print(y_pred_prob) # first column is prob. of being 0 and 2nd column is prob. of beinf 1
```

```
[[3.79015689e-01 6.20984311e-01]
 [3.15061664e-01 6.84938336e-01]
 [5.03658284e-01 4.96341716e-01]
 [8.76892943e-01 1.23107057e-01]
 [3.85386419e-01 6.14613581e-01]
 [3.42904387e-01 6.57095613e-01]
 [3.10812823e-01 6.89187177e-01]
 [3.84841870e-01 6.15158130e-01]
 [2.73809565e-01 7.26190435e-01]
 [3.88145628e-01 6.11854372e-01]
 [3.59816573e-01 6.40183427e-01]
 [3.36302097e-01 6.63697903e-01]
 [7.16060973e-01 2.83939027e-01]
 [3.10763336e-01 6.89236664e-01]
 [3.07018224e-01 6.92981776e-01]
 [3.88145628e-01 6.11854372e-01]
 [3.94509746e-01 6.05490254e-01]
 [5.69212388e-01 4.30787612e-01]
 [5.98679316e-01 4.01320684e-01]
 [5.93751541e-01 4.06248459e-01]
 [4.03036778e-01 5.96963222e-01]
 [2.20803501e-01 7.79196499e-01]
 [2.56669791e-01 7.43330209e-01]
 [2.73809565e-01 7.26190435e-01]
 [9.92174127e-01 7.82587320e-03]
 [3.90471903e-01 6.09528097e-01]
 [6.05807637e-01 3.94192363e-01]
 [2.43451824e-01 7.56548176e-01]
 [4.95418788e-01 5.04581212e-01]
 [5.69212388e-01 4.30787612e-01]
 [7.07644299e-01 2.92355701e-01]
 [9.06688507e-01 9.33114931e-02]
 [3.19267349e-01 6.80732651e-01]
 [4.13974047e-01 5.86025953e-01]
 [3.10812823e-01 6.89187177e-01]
 [3.81609614e-01 6.18390386e-01]
 [4.64449959e-01 5.35550041e-01]
 [5.98568958e-01 4.01431042e-01]
 [9.91743765e-01 8.25623548e-03]
 [4.86507424e-01 5.13492576e-01]
 [2.38835129e-01 7.61164871e-01]
 [4.87346008e-01 5.12653992e-01]
 [2.38397837e-01 7.61602163e-01]
 [4.03036778e-01 5.96963222e-01]
 [5.62671484e-01 4.37328516e-01]
 [9.98515433e-01 1.48456746e-03]
 [3.03010949e-01 6.96989051e-01]
 [3.79014877e-01 6.20985123e-01]
 [5.42738746e-01 4.57261254e-01]
 [8.73041399e-01 1.26958601e-01]
 [6.34869659e-01 3.65130341e-01]
 [6.85270531e-01 3.14729469e-01]
 [4.53845267e-01 5.46154733e-01]
 [4.00771570e-01 5.99228430e-01]
 [7.86112990e-01 2.13887010e-01]
 [9.99996035e-01 3.96501846e-06]
 [5.67778258e-01 4.32221742e-01]
 [9.96198740e-01 3.80125971e-03]
 [4.83863928e-01 5.16136072e-01]
 [5.00577378e-01 4.99422622e-01]
 [4.14810045e-01 5.85189955e-01]
 [2.32527928e-01 7.67472072e-01]
 [4.54921707e-01 5.45078293e-01]
 [7.60498846e-01 2.39501154e-01]
 [4.06605820e-01 5.93394180e-01]
 [5.21287336e-01 4.78712664e-01]
 [9.96963944e-01 3.03605569e-03]
 [2.50372469e-01 7.49627531e-01]
 [3.88145628e-01 6.11854372e-01]
 [9.18223301e-01 8.17766993e-02]
 [3.37235080e-01 6.62764920e-01]
 [2.75462494e-01 7.24537506e-01]
 [3.41451995e-01 6.58548005e-01]
 [2.93762536e-01 7.06237464e-01]
 [2.94212802e-01 7.05787198e-01]
 [3.13506319e-01 6.86493681e-01]
 [5.61831256e-01 4.38168744e-01]
 [2.56669791e-01 7.43330209e-01]
 [6.71117047e-01 3.28882953e-01]
 [3.56325539e-01 6.43674461e-01]
 [3.24748153e-01 6.75251847e-01]
 [2.43451824e-01 7.56548176e-01]
 [4.31327420e-01 5.68672580e-01]
 [3.35251797e-01 6.64748203e-01]
 [3.72372375e-01 6.27627625e-01]
 [9.98919167e-01 1.08083282e-03]
 [9.15354106e-01 8.46458940e-02]
 [2.95017267e-01 7.04982733e-01]
 [3.69020642e-01 6.30979358e-01]
 [9.81787406e-01 1.82125938e-02]
 [3.17188750e-01 6.82811250e-01]
 [3.33774386e-01 6.66225614e-01]
 [3.79014877e-01 6.20985123e-01]
 [3.61134351e-01 6.38865649e-01]
 [6.80960233e-01 3.19039767e-01]
 [3.40114822e-01 6.59885178e-01]
 [4.26154753e-01 5.73845247e-01]
 [5.42738746e-01 4.57261254e-01]
 [4.35508239e-01 5.64491761e-01]
 [3.28734792e-01 6.71265208e-01]
 [4.86507424e-01 5.13492576e-01]
 [3.26824059e-01 6.73175941e-01]
 [2.15863680e-01 7.84136320e-01]
 [3.33575970e-01 6.66424030e-01]
 [8.02320224e-01 1.97679776e-01]
 [5.27855795e-01 4.72144205e-01]
 [9.69709055e-01 3.02909451e-02]
 [5.18770701e-01 4.81229299e-01]
 [2.49224154e-01 7.50775846e-01]
 [1.52992519e-01 8.47007481e-01]
 [5.90766852e-01 4.09233148e-01]
 [2.88823981e-01 7.11176019e-01]
 [9.70865428e-01 2.91345724e-02]
 [6.31984730e-01 3.68015270e-01]
 [3.24748153e-01 6.75251847e-01]
 [9.99702595e-01 2.97405135e-04]
 [2.75462494e-01 7.24537506e-01]
 [3.67938666e-01 6.32061334e-01]
 [2.73520065e-01 7.26479935e-01]
 [2.67506832e-01 7.32493168e-01]
 [9.03930755e-01 9.60692452e-02]
 [3.73131893e-01 6.26868107e-01]
 [2.24104415e-01 7.75895585e-01]
 [4.35525277e-01 5.64474723e-01]
 [9.91571506e-01 8.42849427e-03]
 [9.99615116e-01 3.84884044e-04]
 [5.55836139e-01 4.44163861e-01]
 [7.13179984e-01 2.86820016e-01]
 [2.94107841e-01 7.05892159e-01]
 [4.51388281e-01 5.48611719e-01]
 [4.34263767e-01 5.65736233e-01]
 [4.53845267e-01 5.46154733e-01]
 [4.37018484e-01 5.62981516e-01]
 [2.16875029e-01 7.83124971e-01]
 [5.23328064e-01 4.76671936e-01]
 [4.16002870e-01 5.83997130e-01]
 [9.88541960e-01 1.14580400e-02]
 [2.95017267e-01 7.04982733e-01]
 [6.05229494e-01 3.94770506e-01]
 [2.50372469e-01 7.49627531e-01]
 [5.30404301e-01 4.69595699e-01]
 [5.16963147e-01 4.83036853e-01]
 [9.86161563e-01 1.38384374e-02]
 [3.13506319e-01 6.86493681e-01]
 [2.58742733e-01 7.41257267e-01]
 [3.16627167e-01 6.83372833e-01]
 [9.99977898e-01 2.21019831e-05]
 [9.81575573e-01 1.84244272e-02]
 [2.32527928e-01 7.67472072e-01]
 [8.69552135e-01 1.30447865e-01]
 [7.69999199e-01 2.30000801e-01]
 [2.43451824e-01 7.56548176e-01]
 [3.85386419e-01 6.14613581e-01]
 [9.16525795e-01 8.34742052e-02]
 [2.75227144e-01 7.24772856e-01]
 [8.08347056e-01 1.91652944e-01]]
```

In [38]:
```
# Now we check threshold wise which gives us less Type 2 error and significantly acceptable Total Error and T
ype 1 error
for a in np.arange(0.2, 0.8, 0.01):
    predict_mine = np.where(y_pred_prob[:, 1] > a, 1, 0)
    cfm = confusion_matrix(y_test, predict_mine)
    total_err = cfm[0, 1]+cfm[1, 0]
    print("Errors at threshold ", a, ":", total_err, " , type 2 error :",
          cfm[1, 0], " , type 1 error:", cfm[0, 1])
```

```
Errors at threshold  0.2 : 46  , type 2 error : 1  , type 1 error: 45
Errors at threshold  0.21000000000000002 : 46  , type 2 error : 1  , type 1 error: 45
Errors at threshold  0.22000000000000003 : 45  , type 2 error : 1  , type 1 error: 44
Errors at threshold  0.23000000000000004 : 45  , type 2 error : 1  , type 1 error: 44
Errors at threshold  0.24000000000000005 : 43  , type 2 error : 1  , type 1 error: 42
Errors at threshold  0.25000000000000006 : 43  , type 2 error : 1  , type 1 error: 42
Errors at threshold  0.26000000000000006 : 43  , type 2 error : 1  , type 1 error: 42
Errors at threshold  0.2700000000000001 : 43  , type 2 error : 1  , type 1 error: 42
Errors at threshold  0.2800000000000001 : 43  , type 2 error : 1  , type 1 error: 42
Errors at threshold  0.2900000000000001 : 43  , type 2 error : 2  , type 1 error: 41
Errors at threshold  0.3000000000000001 : 44  , type 2 error : 3  , type 1 error: 41
Errors at threshold  0.3100000000000001 : 44  , type 2 error : 3  , type 1 error: 41
Errors at threshold  0.3200000000000001 : 46  , type 2 error : 5  , type 1 error: 41
Errors at threshold  0.3300000000000001 : 47  , type 2 error : 6  , type 1 error: 41
Errors at threshold  0.34000000000000014 : 47  , type 2 error : 6  , type 1 error: 41
Errors at threshold  0.35000000000000014 : 47  , type 2 error : 6  , type 1 error: 41
Errors at threshold  0.36000000000000015 : 47  , type 2 error : 6  , type 1 error: 41
Errors at threshold  0.37000000000000016 : 47  , type 2 error : 7  , type 1 error: 40
Errors at threshold  0.38000000000000017 : 47  , type 2 error : 7  , type 1 error: 40
Errors at threshold  0.3900000000000002 : 47  , type 2 error : 7  , type 1 error: 40
Errors at threshold  0.4000000000000002 : 47  , type 2 error : 8  , type 1 error: 39
Errors at threshold  0.4100000000000002 : 49  , type 2 error : 11  , type 1 error: 38
Errors at threshold  0.4200000000000002 : 49  , type 2 error : 11  , type 1 error: 38
Errors at threshold  0.4300000000000002 : 49  , type 2 error : 11  , type 1 error: 38
Errors at threshold  0.4400000000000002 : 48  , type 2 error : 13  , type 1 error: 35
Errors at threshold  0.45000000000000023 : 47  , type 2 error : 13  , type 1 error: 34
Errors at threshold  0.46000000000000024 : 49  , type 2 error : 15  , type 1 error: 34
Errors at threshold  0.47000000000000025 : 48  , type 2 error : 15  , type 1 error: 33
Errors at threshold  0.48000000000000026 : 47  , type 2 error : 16  , type 1 error: 31
Errors at threshold  0.49000000000000027 : 47  , type 2 error : 17  , type 1 error: 30
Errors at threshold  0.5000000000000002 : 49  , type 2 error : 19  , type 1 error: 30
Errors at threshold  0.5100000000000002 : 48  , type 2 error : 19  , type 1 error: 29
Errors at threshold  0.5200000000000002 : 48  , type 2 error : 21  , type 1 error: 27
Errors at threshold  0.5300000000000002 : 48  , type 2 error : 21  , type 1 error: 27
Errors at threshold  0.5400000000000003 : 49  , type 2 error : 22  , type 1 error: 27
Errors at threshold  0.5500000000000003 : 51  , type 2 error : 25  , type 1 error: 26
Errors at threshold  0.5600000000000003 : 51  , type 2 error : 25  , type 1 error: 26
Errors at threshold  0.5700000000000003 : 52  , type 2 error : 28  , type 1 error: 24
Errors at threshold  0.5800000000000003 : 53  , type 2 error : 29  , type 1 error: 24
Errors at threshold  0.5900000000000003 : 52  , type 2 error : 30  , type 1 error: 22
Errors at threshold  0.6000000000000003 : 56  , type 2 error : 34  , type 1 error: 22
Errors at threshold  0.6100000000000003 : 58  , type 2 error : 36  , type 1 error: 22
Errors at threshold  0.6200000000000003 : 63  , type 2 error : 42  , type 1 error: 21
Errors at threshold  0.6300000000000003 : 62  , type 2 error : 44  , type 1 error: 18
Errors at threshold  0.6400000000000003 : 63  , type 2 error : 46  , type 1 error: 17
Errors at threshold  0.6500000000000004 : 63  , type 2 error : 47  , type 1 error: 16
Errors at threshold  0.6600000000000004 : 62  , type 2 error : 48  , type 1 error: 14
Errors at threshold  0.6700000000000004 : 61  , type 2 error : 50  , type 1 error: 11
Errors at threshold  0.6800000000000004 : 61  , type 2 error : 52  , type 1 error: 9
Errors at threshold  0.6900000000000004 : 66  , type 2 error : 59  , type 1 error: 7
Errors at threshold  0.7000000000000004 : 66  , type 2 error : 60  , type 1 error: 6
Errors at threshold  0.7100000000000004 : 71  , type 2 error : 65  , type 1 error: 6
Errors at threshold  0.7200000000000004 : 72  , type 2 error : 66  , type 1 error: 6
Errors at threshold  0.7300000000000004 : 74  , type 2 error : 70  , type 1 error: 4
Errors at threshold  0.7400000000000004 : 73  , type 2 error : 70  , type 1 error: 3
Errors at threshold  0.7500000000000004 : 76  , type 2 error : 74  , type 1 error: 2
Errors at threshold  0.7600000000000005 : 80  , type 2 error : 78  , type 1 error: 2
Errors at threshold  0.7700000000000005 : 84  , type 2 error : 82  , type 1 error: 2
Errors at threshold  0.7800000000000005 : 86  , type 2 error : 84  , type 1 error: 2
Errors at threshold  0.7900000000000005 : 84  , type 2 error : 84  , type 1 error: 0
Errors at threshold  0.8000000000000005 : 84  , type 2 error : 84  , type 1 error: 0
```

In [53]:
```
# Now we decide the threshold to be 0.35 as it has lower Type 2 error and considerably acceptable Total error
# While detecting a disease we Type 2 error says that he/she doesn't have a disease in fact when he/she has t
he disease
# Hence we look for a small Type 2 error

y_pred1 = np.where(y_pred_prob[:, 1] > 0.35, 1, 0)

cfm = confusion_matrix(y_test, y_pred1)
print(cfm)
print(classification_report(y_test, y_pred1))
print("Accuracy is",round(accuracy_score(y_test, y_pred1)*100, 2) , "%")
```

```
[[30 41]
 [ 6 79]]
              precision    recall  f1-score   support

           0       0.83      0.42      0.56        71
           1       0.66      0.93      0.77        85

    accuracy                           0.70       156
   macro avg       0.75      0.68      0.67       156
weighted avg       0.74      0.70      0.68       156

Accuracy is 69.87 %
```

A slight increase in the accuracy but not worth considering

Step 11:

Performing SVC

In [52]:
```
model = SVC(C=1, kernel='rbf')  # i.e cost = 1

model.fit(x_train, y_train)

y_pred = model.predict(x_test)

print(cfm)
print(classification_report(y_test, y_pred))
print("Accuracy is", round(accuracy_score(y_test, y_pred)*100, 2) , "%")
```

```
[[30 41]
 [ 6 79]]
              precision    recall  f1-score   support

           0       0.78      0.51      0.62        71
           1       0.68      0.88      0.77        85

    accuracy                           0.71       156
   macro avg       0.73      0.69      0.69       156
weighted avg       0.73      0.71      0.70       156

Accuracy is 71.15 %
```

Step 12:

Applying Random Forest

In [55]:
```
rf = RandomForestClassifier(n_estimators=500, criterion='entropy', min_samples_split=4)
rf.fit(x_train, y_train)
ypred_rf = rf.predict(x_test)
confusion_matrix(y_test, ypred_rf)
print("Accuracy is", round(accuracy_score(y_test, ypred_rf)*100, 2) , "%")
```

```
Accuracy is 83.97 %
```

Conclusion:

Out of all the models we applied, Random Forest gave us the best accuracy. Hence we find this model suitable for our data and will use this for further evaluation if new data comes in.

```
In [ ]:
```