**TASK:**

The task is to develop a classifier on the provided synthetic data composed by 1000 observation, 30 independent and 1 binomial dependent variable.

**GOAL:**
the goal is to build a predictve model for the dependent variable y and find the best bias-variance trade-off. In this project, **Logistic regression** is used to train and tune the data.

**Logistic regression:** This is an machine learning algorithm used to predict the probablity of categorical dependent variables.

# TASK 1

Importing the essential libraries to load and use the data

In [1]:
```
# Importing libraries
import numpy as np
import pandas as pd
from sklearn.model_selection import KFold
import seaborn as sb
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler from
sklearn.datasets import make_classification
from sklearn.metrics import roc_curve, precision_recall_curve
from sklearn.metrics import roc_auc_score
from sklearn.metrics import classification_report
```

In [3]:
```
# Importing synthetic dataset

address = 'D:/synthetic.csv' # Assigning the dataset address to a variable
called 'address'
syn_data = pd.read_csv(address)  # Reading data using pd.read_csv function
```

# Data exploration

In [4]:
```
syn_data.head() # pandas head() method is used to display first 5 rows of the data
```

Out[4]:

| | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | -14.698830 | 2.369710 | 1.089267 | -1.262030 | -15.650082 | -16.665997 | 15.909853 | -11.121045 | 18. |
| 1 | -8.457451 | 2.182712 | 0.972360 | -4.255289 | -11.524392 | -4.843399 | 9.557964 | -10.145921 | 6. |
| 2 | -6.541517 | 1.263892 | -0.494469 | -2.562072 | -8.979410 | -23.632245 | 15.740920 | -4.460916 | -16. |
| 3 | -18.139840 | 1.569545 | -3.286717 | -4.255045 | -16.146687 | -25.893126 | 12.005963 | -2.228017 | 5. |
| 4 | -12.500957 | 2.313632 | 5.227138 | 2.586718 | -15.022213 | -3.105726 | 18.070314 | -7.745197 | 0. |

5 rows × 31 columns

In [5]:
```
syn_data.columns    # check the feature name
```

Out[5]:
```
Index(['x1', 'x2', 'x3', 'x4', 'x5', 'x6', 'x7', 'x8', 'x9', 'x10', 'x11',
       'x12', 'x13', 'x14', 'x15', 'x16', 'x17', 'x18', 'x19', 'x20', 'x21',
       'x22', 'x23', 'x24', 'x25', 'x26', 'x27', 'x28', 'x29', 'x30', 'y'],
      dtype='object')
```

In [6]:
```
syn_data.ndim       # check dimension of data
```

Out[6]: 2

In [7]:
```
syn_data.shape      # returns total number of samples and features
```

Out[7]: (1000, 31)

Shape attribute returns number of rows and columns, there are 1000 rows which are the samples and 31 features. There are 30 predictor variables(x1,x2,x3..x30) and 1 target variable(y).

In [8]:
```
syn_data.info() # returns number of samples per features and type of each samples in the data
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 31 columns):
x1     1000 non-null float64
x2     1000 non-null float64
x3     1000 non-null float64
x4     1000 non-null float64
x5     1000 non-null float64
x6     1000 non-null float64
x7     1000 non-null float64
x8     1000 non-null float64
x9     1000 non-null float64
x10    1000 non-null float64
x11    1000 non-null float64
x12    1000 non-null float64
x13    1000 non-null float64
x14    1000 non-null float64
x15    1000 non-null float64
x16    1000 non-null float64
x17    1000 non-null float64
x18    1000 non-null float64
x19    1000 non-null float64
x20    1000 non-null float64
x21    1000 non-null float64
x22    1000 non-null float64
x23    1000 non-null float64
x24    1000 non-null float64
x25    1000 non-null float64
x26    1000 non-null float64
x27    1000 non-null float64
x28    1000 non-null float64
x29    1000 non-null float64
x30    1000 non-null float64
y      1000 non-null int64
dtypes: float64(30), int64(1)
memory usage: 242.3 KB
```

## Summary of the data

In [9]:
```
syn_data.describe()  # returns summary of the data
```

Out[9]:

| | x1 | x2 | x3 | x4 | x5 | x6 | |
|---|---|---|---|---|---|---|---|
| count | 1000.000000 | 1000.000000 | 1000.000000 | 1000.000000 | 1000.000000 | 1000.000000 | 1000.0000 |
| mean | -13.028746 | 2.182041 | -0.331036 | -1.501078 | -12.622918 | -10.854249 | 15.1999 |
| std | 3.659720 | 1.314388 | 4.259927 | 1.922640 | 3.604514 | 9.750920 | 7.2063 |
| min | -25.548066 | -1.599455 | -14.930338 | -10.215498 | -24.600418 | -55.753091 | -4.3209 |
| 25% | -15.588659 | 1.285855 | -3.149624 | -2.808884 | -15.109200 | -17.120274 | 10.2317 |
| 50% | -13.072938 | 2.170483 | -0.367062 | -1.510223 | -12.498793 | -11.170167 | 15.1962 |
| 75% | -10.534016 | 3.021294 | 2.485166 | -0.237209 | -10.214818 | -4.522221 | 19.9013 |
| max | -2.382520 | 6.026316 | 14.980421 | 5.101086 | 2.182904 | 23.826332 | 36.6469 |

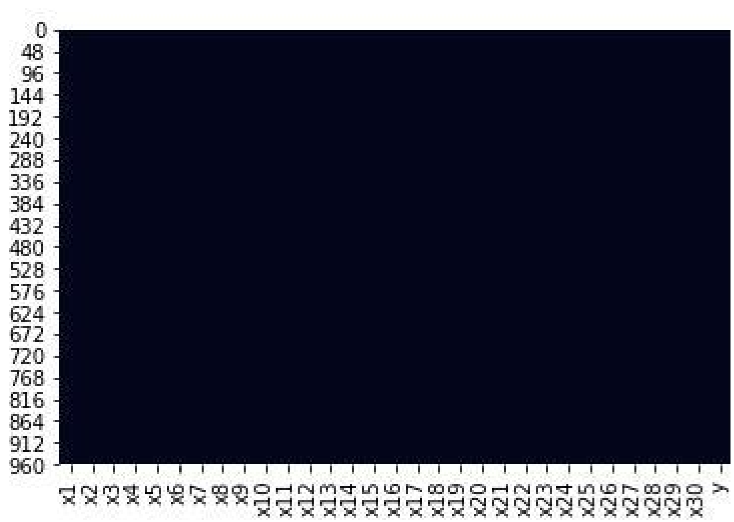
8 rows × 31 columns

- Describe() method returns statistical measures like mean, standard diviation.
- By looking at these values it is clear that the values are not standardized, so scaling should be applied to this data.
- Feature scaling is done before training the data to standardize the range of the independent variables.
- Standardization is done after the train and test split so that the test data is kept untouched till the very end. So that when test data is fitted it is not biased

## Checking for null values

In [10]:
```
sb.heatmap(syn_data.isnull(), cbar=False) # heatmap to show any missing values in the data
```

Out[10]: <matplotlib.axes._subplots.AxesSubplot at 0x202474bc848>



Heatmap shows there are no null values in the data.

In [11]:
```
syn_data.isnull().sum()    #returns sum of null values in the each columns
```

Out[11]:
```
x1      0
x2      0
x3      0
x4      0
x5      0
x6      0
x7      0
x8      0
x9      0
x10     0
x11     0
x12     0
x13     0
x14     0
x15     0
x16     0
x17     0
x18     0
x19     0
x20     0
x21     0
x22     0
x23     0
x24     0
x25     0
x26     0
x27     0
x28     0
x29     0
x30     0
y      0
dtype: int64
```

isnull().sum() method returns total number of null values in each features. By looking at the output it is clear that there are no null values in the data.

## Splitting independent and dependent variables

In [12]:
```
# defining x variables
x = syn_data.iloc[:, 0:30] # select entire rows and first 30 columns(x1 to x3 0)
x.head() # get first 5 rows from the x independent variables
```

Out[12]:

| | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | -14.698830 | 2.369710 | 1.089267 | -1.262030 | -15.650082 | -16.665997 | 15.909853 | -11.121045 | 18. |
| 1 | -8.457451 | 2.182712 | 0.972360 | -4.255289 | -11.524392 | -4.843399 | 9.557964 | -10.145921 | 6. |
| 2 | -6.541517 | 1.263892 | -0.494469 | -2.562072 | -8.979410 | -23.632245 | 15.740920 | -4.460916 | -16. |
| 3 | -18.139840 | 1.569545 | -3.286717 | -4.255045 | -16.146687 | -25.893126 | 12.005963 | -2.228017 | 5. |
| 4 | -12.500957 | 2.313632 | 5.227138 | 2.586718 | -15.022213 | -3.105726 | 18.070314 | -7.745197 | 0. |

5 rows × 30 columns

In [13]:
```
## defining y binomial variable
y=syn_data.iloc[:, [30]] # select all rows and last coloumnn(y)
y.tail(10) # get last 10 rows from the dependent variables
```

Out[13]:

| | y |
|---|---|
| 990 | 0 |
| 991 | 0 |
| 992 | 0 |
| 993 | 0 |
| 994 | 0 |
| 995 | 0 |
| 996 | 0 |
| 997 | 0 |
| 998 | 0 |
| 999 | 0 |

To define x and y from an array with index values iloc[ ] method from pandas library is used,it is location based indexing for selecting rows and columns in an array. iloc[ ] has two arguments [row selector, column selector].

## Inspecting independent and dependent variables

In [14]:
```
# check size of the predictor and the target variables
print(x.shape)
print(y.shape)
```

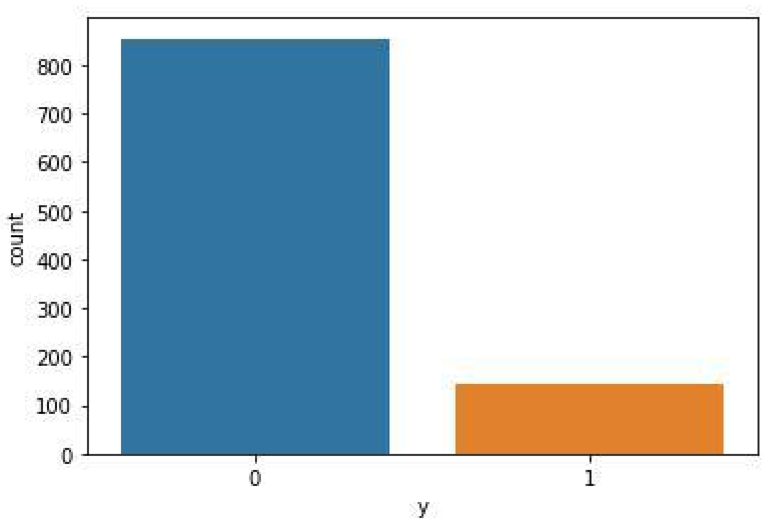
```
(1000, 30)
(1000, 1)
```

In [15]:
```
y = np.ravel(y)
```

**Check whether the target variable is categorical**

In [16]:
```
sb.countplot(x='y', data=syn_data) # check whether the target variable is cat
egorical
```

Out[16]: `<matplotlib.axes._subplots.AxesSubplot at 0x202477f4f48>`




Above bar chart shows that the target variable i.e., the dependent variable is binomial which has only two class, 0 and 1. **The classes are imbalanced. That is the occurence of class 0 is very higher than class 1.**

In [17]:
```
## getting number of 0's and 1's
print(syn_data.groupby('y').size()) # returns number of 0 and 1 in the
depende nt variable.
```

```
y
0    855
1 145 dtype:
int64
```

**Having an imbalanced classes may affect the accuracy measures of the model. Model may give high accuracy but still the presicion, recall, roc rate will be less because of imbalanced classes. There are methods to handle imbalanced class like**

- **Resampling techniques**
    - **Random Under-Sampling**
    - **Random Over-Sampling**
    - **Cluster-Based Over Sampling**
    - **Tuning threshold and class_weight parameter and more techniques**

**In this model the iimbalanced classes are handeled using tuning the threshold and class_weight parameter**

## Splitting the data into train, test and validation

**The data is splitted into three parts**

- **Train set (A set for training the classifier)**
- **Test set (A set fro testing the classifier)**
- **Validation set (An untouched set which is used after model is build)**

    **First, the data is splitted into train and test using train_test_split() method. The train set is further used for training and fitting the model with k-fold. The test set is kept as validation set to fit the model after training and tuning the data. This validation data is kept untouched in order to**
    - **Avoid bias**
    - **Check how the classifier performs on new data set**

In [18]:
```
## splitting x and y training and test data
## train_test_split() is used to split the data into train and validation set.
X, X_val, Y, y_val = train_test_split(x, y, test_size = 0.2, train_size = 0.8,
random_state = 1, shuffle = False )
```

**The above statement splits the data into 4 parts**

- **X, Y : Training set which is used for training and testing the model with k-fold**
- **X_val,y_val : Validation set**

    **Now, the data can be standardized**

# Standardization of the data

```
In [19]: #Standardize the features
         scaler = StandardScaler() # creating instance
         scaler.fit(X)             # fit
         X_scale = scaler.transform(X) #transform
         X_scale = pd.DataFrame(X_scale)
         X_scale.describe()        # get statistical measures
```

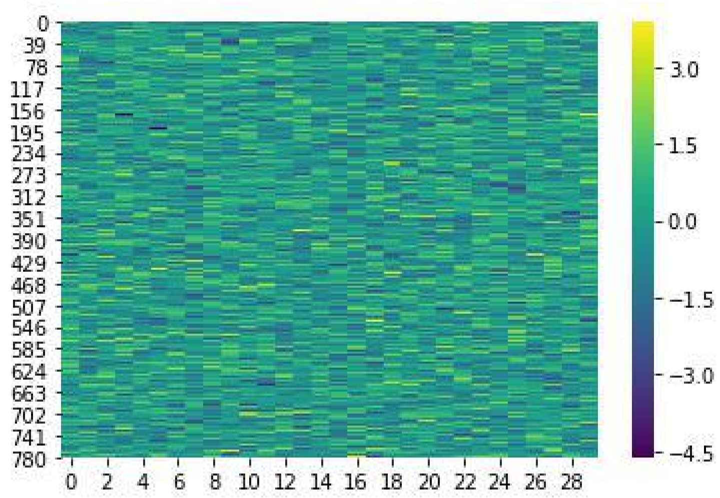
Out[19]:

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| count | 8.000000e+02 | 8.000000e+02 | 8.000000e+02 | 8.000000e+02 | 8.000000e+02 | 8.000000e+02 |
| mean | -4.053702e-16 | -3.330669e-18 | -4.024558e-17 | 5.224987e-17 | 3.393119e-17 | 1.662559e-16 |
| std | 1.000626e+00 | 1.000626e+00 | 1.000626e+00 | 1.000626e+00 | 1.000626e+00 | 1.000626e+00 |
| min | -3.390420e+00 | -2.859873e+00 | -3.147865e+00 | -4.526394e+00 | -3.337124e+00 | -4.622399e+00 |
| 25% | -6.802927e-01 | -6.911927e-01 | -6.618131e-01 | -6.815123e-01 | -7.085943e-01 | -6.140376e-01 |
| 50% | -3.814545e-02 | -1.203091e-02 | -1.313181e-02 | 2.368531e-04 | 4.888546e-02 | -5.766154e-02 |
| 75% | 6.961511e-01 | 6.354095e-01 | 6.448440e-01 | 6.819138e-01 | 6.889921e-01 | 6.320459e-01 |
| max | 2.825399e+00 | 2.920208e+00 | 3.616380e+00 | 3.420299e+00 | 2.760334e+00 | 3.591812e+00 |

8 rows × 30 columns

```
In [20]: sb.heatmap(X_scale, cmap='viridis')
```

Out[20]: <matplotlib.axes._subplots.AxesSubplot at 0x2024792f508>




**We can see that the data is standardized. Linear regression has assumptions of absence of multicollinearity. The next step is to check the collinearity of the feature variables.**

## Check the correlation between the variables

**The independent variables should be uncorrelated with each other because the correlated features in general do not improve models. If there is uncorrelated features then the interpretability of the model made easier and it decreases bias. To check the correlation between the variables**
--- **Variance_inflation_factor is used.**

--- **Variables which have VIF factor above 5 is said to be highly correlated with each other.**

In [21]:

```
# For each X, calculate VIF and save in dataframe
import statsmodels.api as sm
from statsmodels.stats.outliers_influence import variance_inflation_factor

vif = pd.DataFrame() # create a Dataframe for VIF values

vif["Features"] = X_scale.columns  # create a features column

vif["VIF Factor"] = [variance_inflation_factor(X_scale.values, i) for i in ran
ge(X_scale.shape[1])]
vif.round(1)
```

Out[21]:

| | Features | VIF Factor |
|---|---|---|
| 0 | 0 | 1.0 |
| 1 | 1 | 1.0 |
| 2 | 2 | 1.0 |
| 3 | 3 | 1.0 |
| 4 | 4 | 1.0 |
| 5 | 5 | 1.1 |
| 6 | 6 | 1.0 |
| 7 | 7 | 1.0 |
| 8 | 8 | 1.0 |
| 9 | 9 | 1.0 |
| 10 | 10 | 1.0 |
| 11 | 11 | 1.0 |
| 12 | 12 | 1.0 |
| 13 | 13 | 1.0 |
| 14 | 14 | 1.0 |
| 15 | 15 | 1.0 |
| 16 | 16 | 1.0 |
| 17 | 17 | 1.0 |
| 18 | 18 | 1.0 |
| 19 | 19 | 1.0 |
| 20 | 20 | 1.1 |
| 21 | 21 | 1.0 |
| 22 | 22 | 1.0 |
| 23 | 23 | 1.0 |
| 24 | 24 | 1.0 |
| 25 | 25 | 1.0 |
| 26 | 26 | 1.0 |
| 27 | 27 | 1.0 |
| 28 | 28 | 1.0 |
| 29 | 29 | 1.0 |

## Check the shape of the variables

In [22]:
```
X_scale.shape
```

Out[22]: (800, 30)

In [23]:
```
Y.shape
```
Out[23]:
```
(800,)
```

**Preprocessing of data is done, next step is to train the model**

## K-fold cross validation

In [24]:
```
# k fold cross validation, KFold(n_splits=5, shuffle=False, random_state=None)
kf = KFold(n_splits = 10, shuffle= True ,random_state = 1 )
kf.get_n_splits(X) # returns the number of splitting iterations in the kfold c
ross validaton
```
Out[24]:
```
10
```

In [25]:
```
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
```

In [26]:
```
for train_index, test_index in kf.split(X):  # splitting data into train and t
est using kfold
    X_train, X_test = X_scale.iloc[train_index], X_scale.iloc[test_index]
    y_train, y_test = Y[train_index], Y[test_index]

    LogReg = LogisticRegression()             # instantiate the model (using th
e default parameters)

    LogReg.fit(X_train, y_train)              # fitting the model
    y_train_pred = LogReg.predict(X_train)    # predicting the model for train
 data
    y_test_pred = LogReg.predict(X_test)      # predicting the model for test d
ata

    from sklearn import metrics
    accuracy_score_train = metrics.accuracy_score(y_train, y_train_pred)  # ca
lculate accuracy score for train data
    accuracy_score_test = metrics.accuracy_score(y_test, y_test_pred)     # ca
lculate accuracy score for test data

print('mean  accuracy  score  for  train:',  accuracy_score_train.mean())  #
returns mean of the accuracy score for train
print('mean accuracy score for test:', accuracy_score_test.mean()) # returns
mean of the accuracy score for test
```

```
mean accuracy score for train: 0.8597222222222223
mean accuracy score for test: 0.8
```

### Confusion metrics, classification report and AUC curve score for train and test data set

In [27]:
```
confusion_train = metrics.confusion_matrix(y_train, y_train_pred) # confusion
metrics for train set
print(confusion_train)
print(('-')*50)

warnings.filterwarnings('ignore')  # ignore the warning

print(metrics.classification_report(y_train, y_train_pred)) # print
classifica tion report for train data

roc_auc = roc_auc_score(y_train, y_train_pred) # computing the auc curve
score print(('-')*50)
print('AUC: %.2f' % roc_auc)
```

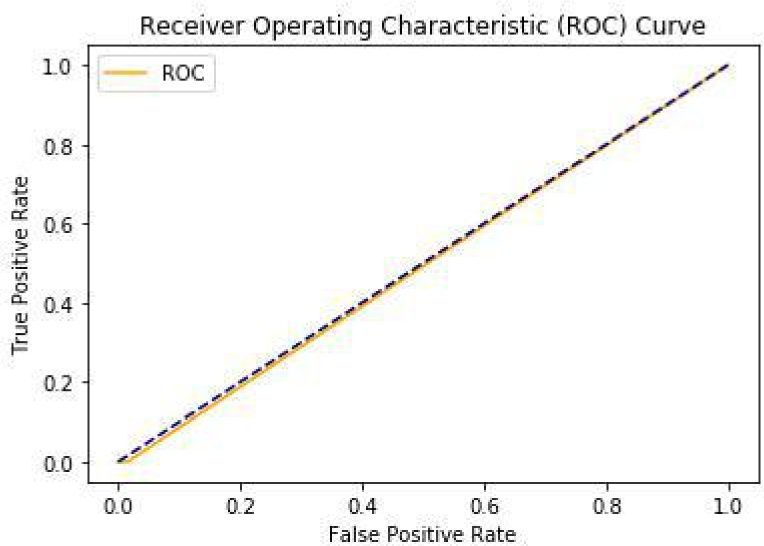
```
 [[618   0]
  [101   1]]
 --------------------------------------------------
               precision    recall  f1-score   support

            0       0.86      1.00      0.92       618
            1       1.00      0.01      0.02       102

     accuracy                           0.86       720
    macro avg       0.93      0.50      0.47       720
 weighted avg       0.88      0.86      0.80       720

--------------------------------------------------
AUC: 0.50
```

In [28]:
```
confusion_test = metrics.confusion_matrix(y_test, y_test_pred) # confusion met
rics for train set
print(confusion_test)
print(('-')*50)

print(classification_report(y_test, y_test_pred)) # print classification
repor t for test data
print(('-')*50)

roc_auc = roc_auc_score(y_test, y_test_pred) # computing the auc curve
score print('AUC: %.2f' % roc_auc)
```

```
[[64  1]
 [15  0]]
 --------------------------------------------------
               precision    recall  f1-score   support

            0       0.81      0.98      0.89        65
            1       0.00      0.00      0.00        15

     accuracy                           0.80        80
    macro avg       0.41      0.49      0.44        80
 weighted avg       0.66      0.80      0.72        80

--------------------------------------------------
AUC: 0.49
```

**Confusion metrics returns number of true positive, true negative, false positive and fasle negative. This confusion_metrics for the train data shows there are 0 true negative false positive.**
**This happens because of the unbalanced classes in data set. That is, the total number occurence of one class is very less than the another class.**
**Though the model gives accuracy of 80% the acuracy score for both train and test set is 50%.**

In [29]:
```
  # Plotting the ROC curve for test data
def plot_roc_curve(fpr, tpr):
    plt.plot(fpr, tpr, color='orange', label='ROC')
    plt.plot([0, 1], [0, 1], color='darkblue', linestyle='--')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver Operating Characteristic (ROC) Curve')
    plt.legend()
    plt.show()
## roc curve
fpr, tpr, thresholds = roc_curve(y_test, y_test_pred)
plot_roc_curve(fpr,tpr)
```




## Adjusting the classification threshold

**To handle the imbalanced classes the classification threshold can be tuned, the default threshold is 0.5 where if p>0.5 then model predicts class 1 and p<0.5 then model predicts class 0.**

In [30]:
```
# first 5 predicted probabilites of class 0 and 1
LogReg.predict_proba(X_test)[0:10]
```

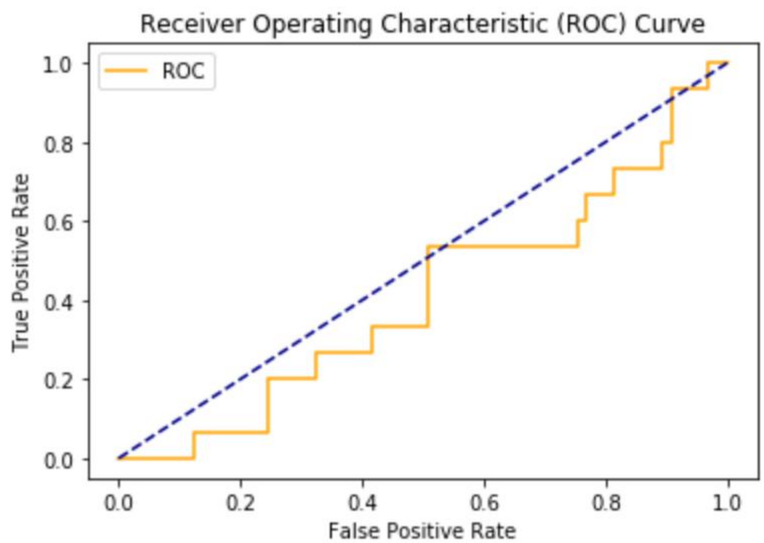
Out[30]:
```
array([[0.77816554, 0.22183446],
       [0.84976278, 0.15023722],
       [0.90851695, 0.09148305],
       [0.85708547, 0.14291453],
       [0.72490365, 0.27509635],
       [0.8526297 , 0.1473703 ],
       [0.94938535, 0.05061465],
       [0.93121547, 0.06878453],
       [0.7794568 , 0.2205432 ],
       [0.81422713, 0.18577287]])
```

**Above array has two column in which first column is observations of class 0 and the second column is observations of class 1**

In [31]:
```
# predicting probability of for the test data
#probabilities of the positive class only([:,1])
y_pred_prob = LogReg.predict_proba(X_test)[:, 1]# show probabilities of class
 1
y_pred_prob0 = LogReg.predict_proba(X_test)[:, 0] # show probabilities of
clas s 0
print("Probabilities of class 1:", y_pred_prob)
print("Probabilities of class 0:",y_pred_prob0)
```

```
Probabilities of class 1: [0.22183446 0.15023722 0.09148305 0.14291453 0.2750
9635  0.1473703
 0.05061465 0.06878453 0.2205432  0.18577287 0.18728472 0.0819783
 0.14154354 0.27157402 0.07921083 0.08950376 0.08171964 0.0523382
 0.02916283 0.24530279 0.15033355 0.14702853 0.18948451 0.08797888
 0.08531586 0.2056156  0.16812722 0.12518817 0.21115122 0.20470048
 0.16509124 0.26728421 0.1093177  0.08421851 0.07070385 0.06741648
 0.11472142 0.13390895 0.25833849 0.20977824 0.13593081 0.05690223
 0.06116937 0.06851801 0.19809053 0.09416589 0.10136562 0.19159134
 0.05943199 0.13350796 0.07688404 0.46116259 0.10089602 0.19401572
 0.44678576 0.1875883  0.18038183 0.19461344 0.29328556 0.15346138
 0.07827365 0.0610208  0.53310526 0.16774698 0.21063814 0.13297236
 0.18958258 0.19456774 0.13026297 0.07013985 0.17170151 0.24618437
 0.05499992 0.1193762  0.12441185 0.1730868  0.14264787 0.0489679
 0.37205531 0.05353411]
Probabilities of class 0: [0.77816554 0.84976278 0.90851695 0.85708547 0.7249
0365  0.8526297
 0.94938535 0.93121547 0.7794568  0.81422713 0.81271528 0.9180217
 0.85845646 0.72842598 0.92078917 0.91049624 0.91828036 0.9476618
 0.97083717 0.75469721 0.84966645 0.85297147 0.81051549 0.91202112
 0.91468414 0.7943844  0.83187278 0.87481183 0.78884878 0.79529952
 0.83490876 0.73271579 0.8906823  0.91578149 0.92929615 0.93258352
 0.88527858 0.86609105 0.74166151 0.79022176 0.86406919 0.94309777
 0.93883063 0.93148199 0.80190947 0.90583411 0.89863438 0.80840866
 0.94056801 0.86649204 0.92311596 0.53883741 0.89910398 0.80598428
 0.55321424 0.8124117  0.81961817 0.80538656 0.70671444 0.84653862
 0.92172635 0.9389792  0.46689474 0.83225302 0.78936186 0.86702764
 0.81041742 0.80543226 0.86973703 0.92986015 0.82829849 0.75381563
 0.94500008 0.8806238  0.87558815 0.8269132  0.85735213 0.9510321
 0.62794469 0.94646589]
```

In [32]:
```
#Defining a python function to plot the ROC curves.
def plot_roc_curve(fpr, tpr):
    plt.plot(fpr, tpr, color='orange', label='ROC')
    plt.plot([0, 1], [0, 1], color='darkblue', linestyle='--')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver Operating Characteristic (ROC) Curve')
    plt.legend()
    plt.show()
## roc curve
fpr, tpr, thresholds = roc_curve(y_test, y_pred_prob)
plot_roc_curve(fpr,tpr)
```

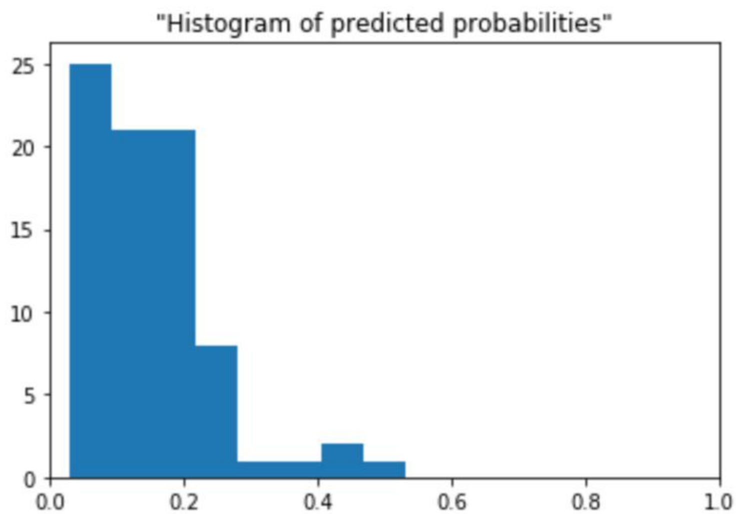



In [33]:
```
## computing the auc curve score

roc_auc = roc_auc_score(y_test, y_pred_prob)
print('AUC: %.2f' % roc_auc)
```

```
AUC: 0.41
```

In [34]:
```python
# plot histogram fot predicted probabilities

plt.hist(y_pred_prob, bins = 8)

plt.xlim(0,1)
plt.title('"Histogram of predicted probabilities"')
```

Out[34]:
```
Text(0.5, 1.0, '"Histogram of predicted probabilities"')
```




In [35]:
```python
# results are 2D so we slice out the first column
from sklearn.preprocessing import binarize
y_pred_class = binarize(y_pred_prob.reshape(-1, len(y_pred_prob)), 0.20)[0]
y_pred_class
```

Out[35]:
```
array([1., 0., 0., 0., 1., 0., 0., 0., 1., 0., 0., 0., 0., 1., 0., 0., 0.,
0., 0., 1., 0., 0., 0., 0., 0., 1., 0., 0., 1., 1., 0., 1., 0., 0., 0., 0., 0., 0.,
        1., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       1., 0., 0., 1., 0., 0., 0., 1., 0., 0., 0., 1., 0., 1., 0., 0., 0.,
       0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 1., 0.])
```

In [36]:
```python
# printing the classes with lower threshold
y_pred_class = y_pred_class.astype(int)
y_pred_class
```

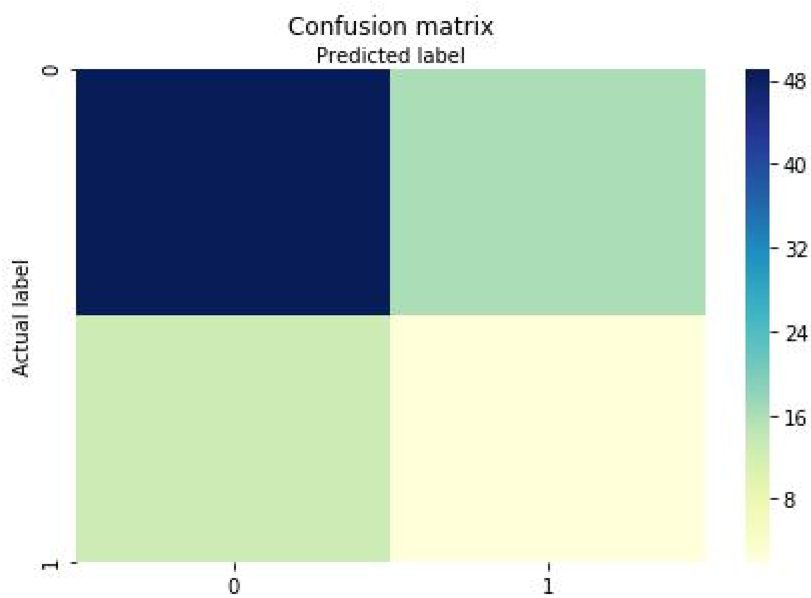
Out[36]:
```
array([1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0,
       0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0,
       0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0])
```

In [37]:
```python
confusion1 = metrics.confusion_matrix(y_test, y_pred_class)
print(confusion1)
```

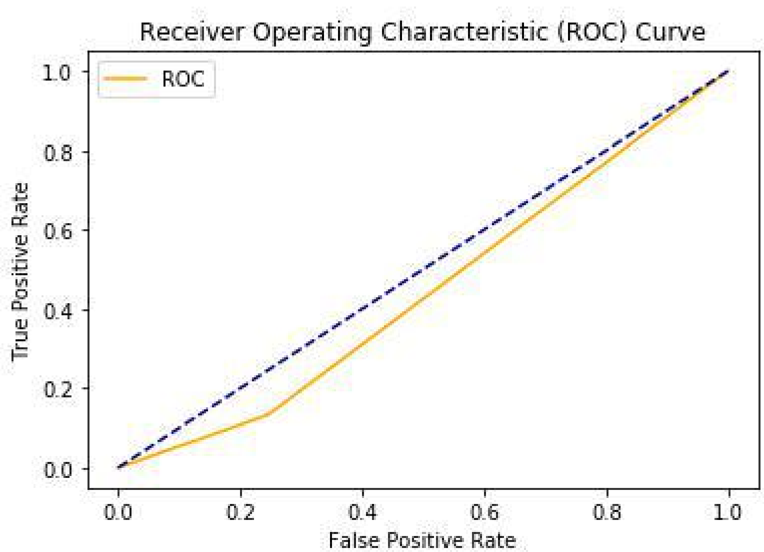
```
[[49 16]
 [13  2]]
```

In [38]:
```
%matplotlib inline
class_names=[0,1] # name  of classes
fig, ax = plt.subplots()
tick_marks = np.arange(len(class_names))
plt.xticks(tick_marks, class_names)
plt.yticks(tick_marks, class_names)
# create heatmap
sb.heatmap(pd.DataFrame(confusion1), cmap="YlGnBu" ,fmt='g')
ax.xaxis.set_label_position("top")
plt.tight_layout()
plt.title('Confusion matrix', y=1.1)
plt.ylabel('Actual label')
plt.xlabel('Predicted label')
```

Out[38]:
```
Text(0.5, 257.44, 'Predicted label')
```




In [39]:
```
## roc curve
fpr, tpr, thresholds = roc_curve(y_test, y_pred_class)
plot_roc_curve(fpr,tpr)
```



In [40]:
```python
# printing previous classification report
print(classification_report(y_test, y_test_pred))
print(('-'*50))
print(metrics.accuracy_score(y_test, y_test_pred))
```

```
              precision    recall  f1-score   support

           0       0.81      0.98      0.89        65
           1       0.00      0.00      0.00        15

    accuracy                           0.80        80
   macro avg       0.41      0.49      0.44        80
weighted avg       0.66      0.80      0.72        80

--------------------------------------------------
0.8
```

In [42]:
```python
from sklearn.metrics import classification_report
print(classification_report(y_test, y_pred_class))
print(('-'*50))
print(metrics.accuracy_score(y_test, y_pred_class))
```

```
              precision    recall  f1-score   support

           0       0.79      0.75      0.77        65
           1       0.11      0.13      0.12        15

    accuracy                           0.64        80
   macro avg       0.45      0.44      0.45        80
weighted avg       0.66      0.64      0.65        80

--------------------------------------------------
0.6375
```

# TUNING THE MODEL

**All machine learning algorithms have default parameters which can be tuned to get best model.**
**Logistic regression hyperparameters used for tuning in this model are penality: l1, l2 regularization.**

**max_iter:Maximum number of iterations taken for the solvers to converge.**
**C:Inverse of regularization strength.**
**class_weight:Weights associated with classes**

## RandomizedSearchCV

**This does an randomized search on hyperparameters**

In [43]:
```
from sklearn.model_selection import RandomizedSearchCV
import time

#setting value for parameters
penalty = ["l1", "l2"]
max_iter=[100,110,120,130,140,150,200]
multi_class = ['ovr', 'multinomial']
C_range = [0.001, 0.01, 0.1, 1, 10, 100, 1000]
class_weight = ['balanced']

param_grid = dict(max_iter=max_iter, penalty=penalty, C = C_range,
class_weigh t = class_weight )
start_time = time.time()

#instantiate RandomizedSearchCV with 10 cross validation
random = RandomizedSearchCV(estimator=LogReg, param_distributions=param_grid,
cv = 10, n_jobs=-1, random_state=0)
random_result= random.fit(X_scale, Y)

# Summarize results
print("Best score:" ,random_result.best_score_)
print("The best performing max_iter is:
{}".format(random_result.best_params_[ 'max_iter']))
print("The best performing penalty is:
{}".format(random_result.best_params_[ 'penalty']))
print("The best performing C is: {}".format(random_result.best_params_['C']))
#print("The best performing multi_class_option is:
{}".format(random_result.be st_params_['multi_class']))
print("Execution time: " + str((time.time() - start_time)) + ' ms')
```

```
Best score: 0.85375
The best performing max_iter is: 200
The best performing penalty is: l1
The best performing C is: 0.01
Execution time: 3.8219752311706543 ms
```

In [44]:
```
#getting the best parameter values
random_result.cv_results_['params'][random_result.best_index_]
```

Out[44]:
```
{'penalty': 'l1', 'max_iter': 200, 'class_weight': 'balanced', 'C': 0.01}
```

## Fitting the model for VALIDATION DATA

In [45]:
```
# Check the size of the validation dataset
print(X_val.shape)
print(y_val.shape)
```

```
(200, 30)
(200,)
```

In [46]:
```
#Standardizng the data
scaler = StandardScaler()
scaler.fit(X_val)
X_val_scale = scaler.transform(X_val)
X_val_scale = pd.DataFrame(X_val_scale)
X_val_scale.describe()
```

Out[46]:

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| count | 2.000000e+02 | 2.000000e+02 | 2.000000e+02 | 2.000000e+02 | 2.000000e+02 | 2.000000e+02 |
| mean | -9.353629e-17 | -1.380840e-16 | -1.429412e-17 | 1.221245e-16 | -3.204381e-16 | -1.740795e-16 |
| std | 1.002509e+00 | 1.002509e+00 | 1.002509e+00 | 1.002509e+00 | 1.002509e+00 | 1.002509e+00 |
| min | -2.567176e+00 | -2.759298e+00 | -3.316784e+00 | -2.327616e+00 | -2.731530e+00 | -2.838090e+00 |
| 25% | -7.549189e-01 | -5.873509e-01 | -7.321253e-01 | -7.178847e-01 | -6.237717e-01 | -7.586302e-01 |
| 50% | 9.470151e-02 | 7.576154e-02 | 1.186628e-02 | -3.998367e-02 | 6.203768e-03 | 7.964492e-02 |
| 75% | 6.367520e-01 | 7.075186e-01 | 7.073822e-01 | 6.395343e-01 | 5.219278e-01 | 6.838536e-01 |
| max | 2.884106e+00 | 2.502259e+00 | 2.336392e+00 | 2.878581e+00 | 4.047909e+00 | 2.207956e+00 |

8 rows × 30 columns

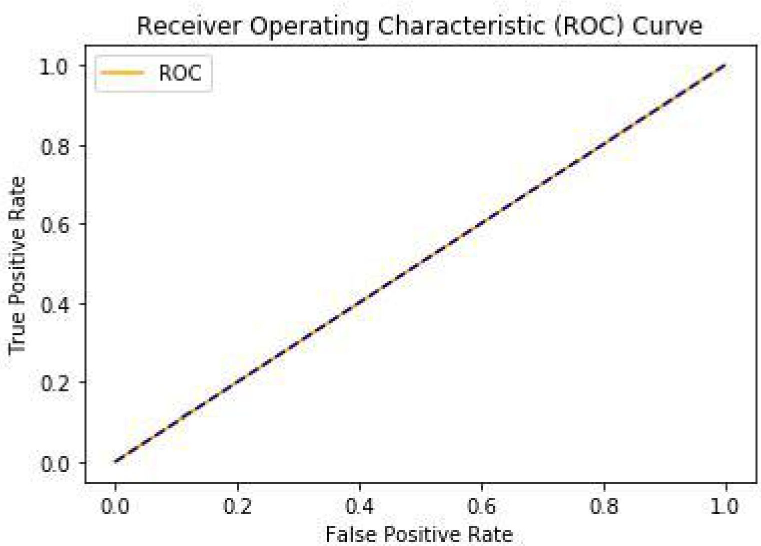
In [47]:
```
# Logistic regression
for train_index, test_index in kf.split(X): # splitting data into train and t
est using kfold
    X_train1, X_test1 = X_scale.iloc[train_index], X_scale.iloc[test_index]
    y_train1, y_test1 = Y[train_index], Y[test_index]

    LogReg1 = LogisticRegression(penalty = 'l1', max_iter = 200, C = 0.01,
cla ss_weight = 'balanced') # passing the best parameters

    LogReg1.fit(X_train1, y_train1)         # fitting the model
    y_val_pred = LogReg1.predict(X_val_scale) # predicting the model for
valid ation data

from sklearn import metrics
accuracy_score = metrics.accuracy_score(y_val, y_val_pred)
print("Accuracy score of the validation ", accuracy_score)
```

```
Accuracy score of the validation  0.86
```

In [48]:
```
print(classification_report(y_val, y_val_pred))
print('-'*50)
confusion2 = metrics.confusion_matrix(y_val, y_val_pred)
print(confusion2)
```

```
              precision    recall  f1-score   support

           0       0.86      1.00      0.92       172
           1       0.00      0.00      0.00        28

    accuracy                           0.86       200
   macro avg       0.43      0.50      0.46       200
weighted avg       0.74      0.86      0.80       200

--------------------------------------------------
[[172   0]
 [ 28   0]]
```

In [49]:
```
## roc curve
fpr, tpr, thresholds = roc_curve(y_val, y_val_pred)
plot_roc_curve(fpr,tpr)
```




## Comparing the baseline model with the tuned model

In [50]:
```python
from sklearn.metrics import classification_report
print(classification_report(y_test, y_pred_class))
print(('-'*50))
print(metrics.accuracy_score(y_test, y_pred_class))
```

```
              precision    recall  f1-score   support

           0       0.79      0.75      0.77        65
           1       0.11      0.13      0.12        15

    accuracy                           0.64        80
   macro avg       0.45      0.44      0.45        80
weighted avg       0.66      0.64      0.65        80

--------------------------------------------------
0.6375
```

In [51]:
```python
# printing tuned model classification report
print(classification_report(y_val, y_val_pred))
print('-'*50)
print(metrics.accuracy_score(y_val, y_val_pred))
```

```
              precision    recall  f1-score   support

           0       0.86      1.00      0.92       172
           1       0.00      0.00      0.00        28

    accuracy                           0.86       200
   macro avg       0.43      0.50      0.46       200
weighted avg       0.74      0.86      0.80       200

--------------------------------------------------
0.86
```

In [54]:
```python
## computing the auc curve score
roc_auc = roc_auc_score(y_test, y_pred_prob)
roc_auc_1 = roc_auc_score(y_val, y_val_pred)
print('AUC_before_tuning: %.2f' % roc_auc)
print('AUC_after_tuning: %.2f' % roc_auc_1)
```

```
AUC_before_tuning: 0.41
AUC_after_tuning: 0.50
```

**By looking at the weighted avg of tuned model and baseline model, tuned model performs better than baseline model. The accuracy has increased in tuned model. Because of the imbalanced classes the presicion, recall and f1 score is affected. The AUC curve score is also increased after the tuning(Higher the AUC, better the model predicts classes correctly).**
**Even though accuracy of the tuned model is better, the presicion, recall and f1-score is very less. These measures should also be higher so that the model can predict the classes correctly**

**In this project the algorithm is fitted using k-fold cross validation which is good practise to avoid overfitting and the data is splitted into 3 parts, train, test and validation where validation dataset is kept untouched till the model tuned and got best params.**