Compute performance metrics for the given Y and Y_score without sklearn

In [3]:

```
import numpy as np
import pandas as pd
# other than these two you should not import any other packages
```

In [144]:

```
!pip install tqdm
from tqdm import tqdm
```

```
Requirement already satisfied: tqdm in c:\users\hp\anaconda3\lib\site-packages (4.48.2)
```

```
mysql-connector-python 8.0.21 requires protobuf>=3.0.0, which is not installed.
distributed 1.21.8 requires msgpack, which is not installed.
You are using pip version 10.0.1, however version 20.2.3 is available.
You should consider upgrading via the 'python -m pip install --upgrade pip' command.
```

In [125]:

```
# confusion matrix function

def build_consfusion_matrix(y_actual, y_predicted):
    """This function will take actual class levels and predicted class levels and returns confusio
n matrix"""

    # Initialize matrix
    levels = np.unique(y_actual)
    size = levels.size
    confusion_matrix = np.zeros((size, size), dtype = int)

    #Assign values to the cells of matrix
    for i in range(size):
        for j in range(size):
            # get predicted and actual class levels in a cell
            class_p = levels[i]
            class_a = levels[j]
            count = 0
            for k in range(len(y_actual)):
                if(y_predicted[k] == class_p and y_actual[k] == class_a):
                    count += 1
            confusion_matrix[i,j] = count
    return confusion_matrix
```

In [32]:

```
# F1_score function
def computeF1Score(y_actual, y_predicted):
    """This function returns F1_score for binary classification.
    It takes actual class levels and predicted class levels as input
    """
    #Initialize TP(True Positive counts)
    TP = 0
    for k in range(len(y_actual)):
        if(y_actual[k] == y_predicted[k] == 1):
            TP += 1

    # Initialize FP(False positive)
    FP = 0
    for k in range(len(y_actual)):
        if(y_actual[k] == 0 and y_predicted[k] ==1):
            FP += 1

    # All +ve predicted counts
    TPFP = TP + FP
```

```
    precision = TP/TPFP

    # Initialize FNFalse negative
    FN = 0
    for k in range(len(y_actual)):
        if(y_actual[k] == 1 and y_predicted[k] ==0):
            FN += 1

    FNTP = TP + FN
    recall = TP/FNTP

    f1_score = (2*precision*recall)/(precision+recall)

# Alternative way using confusion matrix
#     confusion_matrix = build_consfusion_matrix(y_actual, y_predicted)
#     TP = confusion_matrix[1,1]
#     FP = confusion_matrix[1,0]
#     FN = confusion_matrix[0,1]
#     TPFP = TP + FP
#     precision = TP/TPFP
#     FNTP = TP + FN
#     recall = TP/FNTP
#     f1_score = (2*precision*recall)/(precision+recall)

    return f1_score
```

In [165]:

```
# AUC

def computeAUC(dataset):
    """This function computes AUC for a binary classification input(dataset of Yi and Y_proba)"""

    # Sort the values in descending order of the y_predicted probabilities.
    dataset_auc = dataset.sort_values('proba', ascending=False)
    size = len(dataset_auc.index)

    # Traverse through each point in descending dataset and make the probability score as
threshold

    # List of thresholds and list of its coresponding TPR and FPR
    TPR_list = []
    FPR_list = []

    for i in tqdm(range(size)):
        ds = dataset_auc.iloc[:,:2]
        threshold = dataset_auc.iloc[i,1]
        ds['proba'] = (dataset_auc['proba'] >= threshold).astype(int)

        # Compute TPR and FPR

        TP = len(ds[(ds['y'] == 1) & (ds['proba'] == 1)].index)
        P = len(ds[ds['y'] == 1].index)
        FP = len(ds[(ds['y'] == 0) & (ds['proba'] == 1)].index)
        N = len(ds[ds['y'] == 0].index)

        # TPR and FPR
        TPR = TP/P
        FPR = FP/N

        # Add threshold, TPR and FPR values to the list
        TPR_list.append(TPR)
        FPR_list.append(FPR)

    return np.trapz(TPR_list, FPR_list)
```

In [6]:

```
# accuracy

def compute_accuracy(y_actual, y_predict):
    """
    This function takes two paameters as actual class levels and predicted class levels (numpy arr
ays)
    and give the accuracy of the model
```

```
    and give the accuracy of the model
    """

    # Calculate total count
    total_count = y_actual.size
    correct_pred_count = 0

    # Calculate the correct predicted values
    for i in range(total_count):
        if(y_actual[i] == y_predict[i]):
            correct_pred_count += 1

    return correct_pred_count / total_count
```

A. Compute performance metrics for the given data **5_a.csv**

Note 1: in this data you can see number of positive points >> number of negatives points

Note 2: use pandas or numpy to read the data from **5_a.csv**

Note 3: you need to derive the class labels from given score

$y^{pred}= \text{[0 if y_score < 0.5 else 1]}$

1. Compute Confusion Matrix

2. Compute F1 Score

3. Compute AUC Score, you need to compute different thresholds and for each threshold compute tpr,fpr and then use numpy.trapz(tpr_array, fpr_array) https://stackoverflow.com/q/53603376/4084039, https://stackoverflow.com/a/39678975/4084039 Note: it should be numpy.trapz(tpr_array, fpr_array) not numpy.trapz(fpr_array, tpr_array)

4. Compute Accuracy Score

In [6]:

```
data_a_proba = pd.read_csv('5_a.csv')
data_a_proba.head(10)
```

Out[6]:

	y	proba
0	1.0	0.637387
1	1.0	0.635165
2	1.0	0.766586
3	1.0	0.724564
4	1.0	0.889199
5	1.0	0.601600
6	1.0	0.666323
7	1.0	0.567012
8	1.0	0.650230
9	1.0	0.829346

In [7]:

```
data_a = data_a_proba.iloc[:,:2]
data_a['proba'] = (data_a['proba'] > 0.5).astype(int)
data_a['y'] = data_a['y'].astype(int)
data_a.rename(columns= {'proba':'y_p'}, inplace=True)
```

```
data_a.rename(columns={'proba':'y_p'}, inplace=True)
data_a.head()
```

Out[7]:

	y	y_p
0	1	1
1	1	1
2	1	1
3	1	1
4	1	1

In [126]:

```
y_a_actual = data_a['y'].values
y_a_predicted = data_a['y_p'].values
```

In [127]:

```
# confusion matrix

build_consfusion_matrix(y_a_actual, y_a_predicted)
```

Out[127]:

```
array([[    0,     0],
       [  100, 10000]])
```

In [24]:

```
# F1 score

computeF1Score(y_a_actual, y_a_predicted)
```

Out[24]:

```
0.9950248756218906
```

In [160]:

```
# AUC score

computeAUC(data_a_proba)
```

```
100%|████████████████████████████████████████████████████████████| 10100/10100 [11
:32<00:00, 14.59it/s]
```

Out[160]:

```
0.48829900000000004
```

In [80]:

```
# Accuracy

compute_accuracy(y_a_actual, y_a_predicted)
```

Out[80]:

```
0.9900990099009901
```

B. Compute performance metrics for the given data **5_b.csv**
Note 1: in this data you can see number of positive points << number of negatives points
Note 2: use pandas or numpy to read the data from **5_b.csv**

Note 3: you need to derive the class labels from given score

$y^{pred}= \text{[0 if y_score < 0.5 else 1]}$

1. Compute Confusion Matrix

2. Compute F1 Score

3. Compute AUC Score, you need to compute different thresholds and for each threshold compute tpr,fpr and then use numpy.trapz(tpr_array, fpr_array) https://stackoverflow.com/q/53603376/4084039, https://stackoverflow.com/a/39678975/4084039

4. Compute Accuracy Score

In [25]:

```
data_b_proba = pd.read_csv('5_b.csv')
data_b_proba.head(10)
```

Out[25]:

	y	proba
0	0.0	0.281035
1	0.0	0.465152
2	0.0	0.352793
3	0.0	0.157818
4	0.0	0.276648
5	0.0	0.190260
6	0.0	0.320328
7	0.0	0.435013
8	0.0	0.284849
9	0.0	0.427919

In [26]:

```
data_b = data_b_proba.iloc[:,:2]
data_b['proba'] = (data_b['proba'] > 0.5).astype(int)
data_b['y'] = data_b['y'].astype(int)
data_b.rename(columns={'proba':'y_p'}, inplace=True)
data_b.head()
```

Out[26]:

	y	y_p
0	0	0
1	0	0
2	0	0
3	0	0
4	0	0

In [27]:

```
y_b_actual = data_b['y'].values
y_b_predicted = data_b['y_p'].values
```

In [28]:

```
# confusion matrix

build_consfusion_matrix(y_b_actual, y_b_predicted)
```

Out[28]:

```
array([[9761,   45],
       [ 239,   55]])
```

In [31]:

```
# F1 score

computeF1Score(y_b_actual, y_b_predicted)
```

Out[31]:

```
0.2791878172588833
```

In [161]:

```
# AUC score

computeAUC(data_b_proba)
```

```
100%|████████████████████████████████████████████████████████████████████████████| 10100/10100 [05
:04<00:00, 33.17it/s]
```

Out[161]:

```
0.9377570000000001
```

In [32]:

```
# Accuracy

compute_accuracy(data_b['y'].values, data_b['proba'].values)
```

Out[32]:

```
0.9718811881188119
```

C. Compute the best threshold (similarly to ROC curve computation) of probability which gives lowest values of metric **A** for the given data **5_c.csv**

you will be predicting label of a data points like this: $y^{pred}= \text{[0 if y_score < threshold else 1]}$

$ A = 500 \times \text{number of false negative} + 100 \times \text{numebr of false positive}$

> **Note 1:** in this data you can see number of negative points > number of positive points
> **Note 2:** use pandas or numpy to read the data from **5_c.csv**

In [162]:

```
data_c_proba = pd.read_csv('5_c.csv')
data_c_proba.head(10)
```

Out[162]:

	y	prob
0	0	0.458521
1	0	0.505037

	y	prob
2	0	0.418652
3	0	0.412057
4	0	0.375579
5	0	0.595387
6	0	0.370288
7	0	0.299273
8	0	0.297000
9	0	0.266479

In [168]:

```
def computeThreshold(dataset):
    """This function A matrix for all threshold value and return loweset one
    for a binary classification input(dataset of Yi and Y_proba)"""

    # Sort the values in descending order of the y_predicted probabilities.
    dataset_auc = dataset.sort_values('prob', ascending=False)
    size = len(dataset_auc.index)

    # Traverse through each point in descending dataset and make the probability score as
threshold

    # List of thresholds and list of its A value
    threshold_list = []
    A_list = []

    for i in tqdm(range(size)):
        ds = dataset_auc.iloc[:,:2]
        threshold = dataset_auc.iloc[i,1]
        ds['prob'] = (dataset_auc['prob'] >= threshold).astype(int)

        # Compute FP and FN
        FP = len(ds[(ds['y'] == 0) & (ds['prob'] == 1)].index)
        FN = len(ds[(ds['y'] == 1) & (ds['prob'] == 0)].index)

        # Compute A
        A = (500 * FN) + (100 * FP)

        # Add threshold, A values to the list
        threshold_list.append(threshold)
        A_list.append(A)

    A_array = np.array(A)
    return threshold_list[np.argmin(A_array)]
```

In [169]:

```
computeThreshold(data_c_proba)
```

```
100%|██████████████████████████████████████████████████████████████████████████████| 2852/2852
[00:18<00:00, 152.50it/s]
```

Out[169]:

```
0.9577467989277196
```

D. Compute performance metrics(for regression) for the given data **5_d.csv**

Note 2: use pandas or numpy to read the data from **5_d.csv**

Note 1: **5_d.csv** will having two columns Y and predicted_Y both are real valued features

1. Compute Mean Square Error

2. Compute MAPE: https://www.youtube.com/watch?v=ly6ztgIkUxk

3. Compute R^2 error: https://en.wikipedia.org/wiki/Coefficient_of_determination#Definitions

In [170]:

```
data_d_proba = pd.read_csv('5_d.csv')
data_d_proba.head(10)
```

Out[170]:

	y	pred
0	101.0	100.0
1	120.0	100.0
2	131.0	113.0
3	164.0	125.0
4	154.0	152.0
5	133.0	153.0
6	148.0	139.0
7	172.0	145.0
8	153.0	162.0
9	162.0	154.0

In [175]:

```
# MSE
def computeMSE(y_actual, y_predicted):
    """This function takes Y_actual and Y_predicted for regression and calculates mean sqare error
"""

    n = y_actual.size
    # Initialize mean sqare error
    MSE = 0

    for i in tqdm(range(n)):
        term = (y_actual[i] - y_predicted[i])**2
        MSE += term

    MSE /= n
    return MSE
```

In [199]:

```
# MAPE

def computeMAPE(y_actual, y_predicted):
    """This function returns MAPE(Mean Absolute Percentage Error) for a set of actual and predicte
d values"""

    n = y_actual.size
    # Compute sum of actual values
    y_sum = 0

    for i in tqdm(range(n)):
        y_sum += abs(y_actual[i])

    # Iitialize error sum
    e_sum = 0

    # Compute sum of absolute errors
    for i in tqdm(range(n)):
        e_sum += abs(y_predicted[i] - y_actual[i])
```

```
    MAPE = e_sum / y_sum
    return MAPE
```

In [201]:

```
# R^2

def computeRSqr(y_actual, y_predicted):
    """This function calculates R-sqare value for actual and predicted Y values in regression"""

    # Initialize SS_total and SS_res
    SS_total = 0
    SS_res = 0

    # Mean value of Y_actual
    n = y_actual.size
    y_mean = y_actual.sum() / n

    # Compute SS_total
    for i in tqdm(range(n)):
        term = (y_actual[i] - y_mean)**2
        SS_total += term

    SS_total /= n

    # Compute SS_res
    for i in tqdm(range(n)):
        term = (y_actual[i] - y_predicted[i])**2
        SS_res += term

    SS_res /= n

    # Compute R-sqare from SS_total and SS_resudue
    R_sqr = 1 - (SS_res / SS_total)

    return R_sqr
```

In [172]:

```
y_d_actual = data_d_proba['y'].values
y_d_predicted = data_d_proba['pred'].values
```

In [173]:

```
# MSE

computeMSE(y_d_actual, y_d_predicted)
```

```
100%|████████████████████████████████████████████████████████████████████████| 157200/157200
[00:00<00:00, 362646.67it/s]
```

Out[173]:

```
177.16569974554707
```

In [200]:

```
# MAPE

computeMAPE(y_d_actual, y_d_predicted)
```

```
100%|████████████████████████████████████████████████████████████████████████| 157200/157200
[00:00<00:00, 465132.64it/s]
100%|████████████████████████████████████████████████████████████████████████| 157200/157200
[00:00<00:00, 366323.51it/s]
```

Out[200]:

```
0.1291202994009687
```

In [202]:

```
# R-Sqare

computeRSqr(y_d_actual, y_d_predicted)
```

```
100%|██████████████████████████████████████████████████████████████████████| 157200/157200
[00:00<00:00, 398752.35it/s]
100%|██████████████████████████████████████████████████████████████████████| 157200/157200
[00:00<00:00, 397754.79it/s]
```

Out[202]:

```
0.9563582786990964
```