Employee Attrition Prediction

In [ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import math
```

read CSV

In [ ]:
```
df = pd.read_csv('hr-employee-attrition-with-null.csv')
```

Dataset statistic

In [ ]:
```
df.describe()
```

Out[ ]:

	Unnamed: 0	Age	DailyRate	DistanceFromHome	Education	EmployeeCount	Empl
count	1470.000000	1176.000000	1176.000000	1176.00000	1176.000000	1176.0	
mean	734.500000	37.134354	798.875850	9.37500	2.920918	1.0	
std	424.496761	9.190317	406.957684	8.23049	1.028796	0.0	
min	0.000000	18.000000	102.000000	1.00000	1.000000	1.0	
25%	367.250000	30.000000	457.750000	2.00000	2.000000	1.0	
50%	734.500000	36.000000	798.500000	7.00000	3.000000	1.0	
75%	1101.750000	43.000000	1168.250000	15.00000	4.000000	1.0	
max	1469.000000	60.000000	1499.000000	29.00000	5.000000	1.0	

8 rows × 27 columns

In [ ]:
```
df.head(10)
```

Out[ ]:

	Unnamed: 0	Age	Attrition	BusinessTravel	DailyRate	Department	DistanceFromHome	Educatio
0	0	41.0	Yes	Travel_Rarely	NaN	NaN	1.0	Na
1	1	NaN	No	NaN	279.0	Research & Development	NaN	Na
2	2	37.0	Yes	NaN	1373.0	NaN	2.0	2.(
3	3	NaN	No	Travel_Frequently	1392.0	Research & Development	3.0	4.(
4	4	27.0	No	Travel_Rarely	591.0	Research & Development	2.0	1.(
5	5	32.0	No	NaN	1005.0	Research & Development	2.0	2.(
6	6	NaN	No	NaN	NaN	Research & Development	3.0	3.(
7	7	30.0	No	Travel_Rarely	1358.0	Research & Development	24.0	1.(
8	8	38.0	No	Travel_Frequently	216.0	Research & Development	NaN	3.(
9	9	NaN	No	Travel_Rarely	1299.0	Research & Development	NaN	3.(

10 rows × 36 columns

Feature transformation

In [ ]:

```
df.loc[df["Attrition"] == "No", "Attrition"] = 0.0
df.loc[df["Attrition"] == "Yes", "Attrition"] = 1.0
string_categorical_col = ['Department', 'Attrition', 'BusinessTravel', 'EducationFielc
                          'MaritalStatus', 'Over18', 'OverTime']



# ENCODE STRING COLUMNS TO CATEGORICAL COLUMNS
for col in string_categorical_col:
    # INSERT CODE HERE
    df[col]=pd.Categorical(df[col]).codes
# HANDLE NULL NUMBERS
# INSERT CODE HERE

df = df.loc[:, ~df.columns.isin(['EmployeeNumber', 'Unnamed: 0', 'EmployeeCount', 'Sta
```

In [ ]:

```
df.head(10)
```

Out[ ]:

	Age	Attrition	BusinessTravel	DailyRate	Department	DistanceFromHome	Education	EducationFi
0	41.0	1	2	NaN	-1	1.0	NaN	
1	NaN	0	-1	279.0	1	NaN	NaN	
2	37.0	1	-1	1373.0	-1	2.0	2.0	
3	NaN	0	1	1392.0	1	3.0	4.0	
4	27.0	0	2	591.0	1	2.0	1.0	
5	32.0	0	-1	1005.0	1	2.0	2.0	
6	NaN	0	-1	NaN	1	3.0	3.0	
7	30.0	0	2	1358.0	1	24.0	1.0	
8	38.0	0	1	216.0	1	NaN	3.0	
9	NaN	0	2	1299.0	1	NaN	3.0	

10 rows × 31 columns

Spliting data into train and test

In [ ]:

```
from sklearn.model_selection import train_test_split
```

In [ ]:

```
X_train, X_test, y_train, y_test=train_test_split(df.loc[:, ~df.columns.isin(['Attriti
print("X_train", X_train.shape)
print("X_test", X_test.shape)
print("y_train", y_train.shape)
print("y_test ", y_test.shape)
```

```
X_train (1323, 30)
X_test (147, 30)
y_train (1323,)
y_test  (147,)
```

T4. Observe the histogram for Age, MonthlyIncome and DistanceFromHome. How many bins have zero counts? Do you think this is a good discretization? Why?

Display histogram of each feature

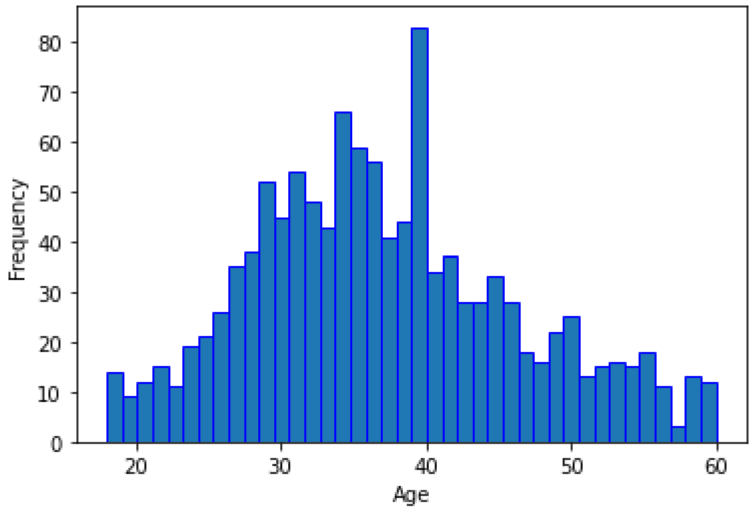
In [ ]:

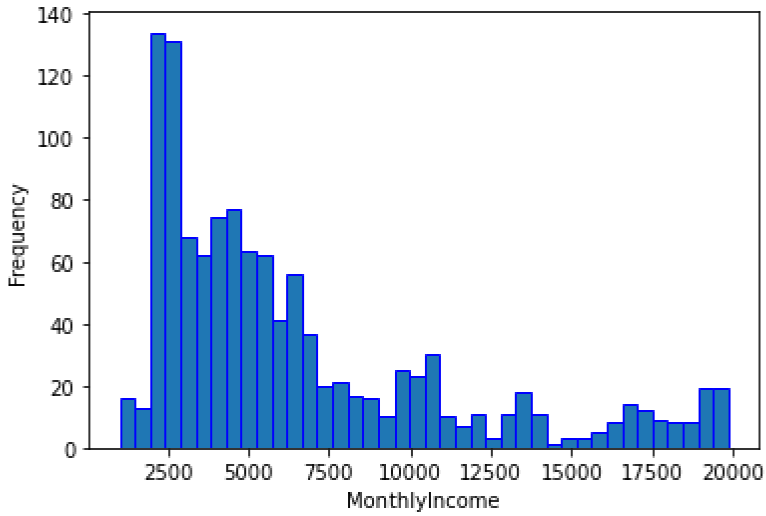
```
def display_histogram(df, col_name, n_bin = 40):
    for col in col_name:
        plt.hist(df[col],n_bin,edgecolor="blue")
        plt.xlabel(col)
        plt.ylabel("Frequency")
        plt.show()
```

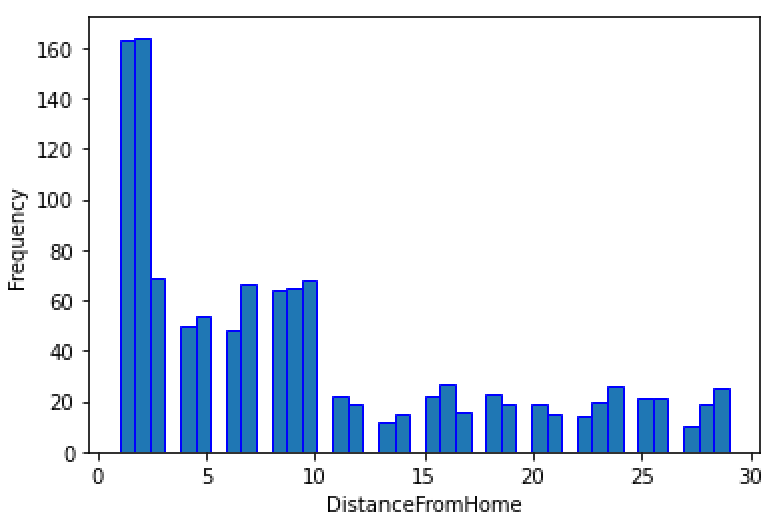
In [ ]:

```
print("T4")
display_histogram(df,["Age","MonthlyIncome","DistanceFromHome"])
```

T4







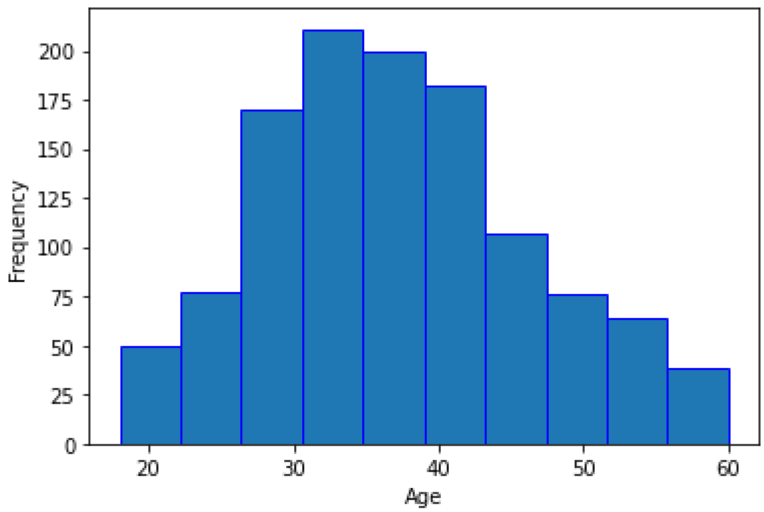
T5. Can we use a Gaussian to estimate this histogram? Why? What about a Gaussian Mixture Model (GMM)?

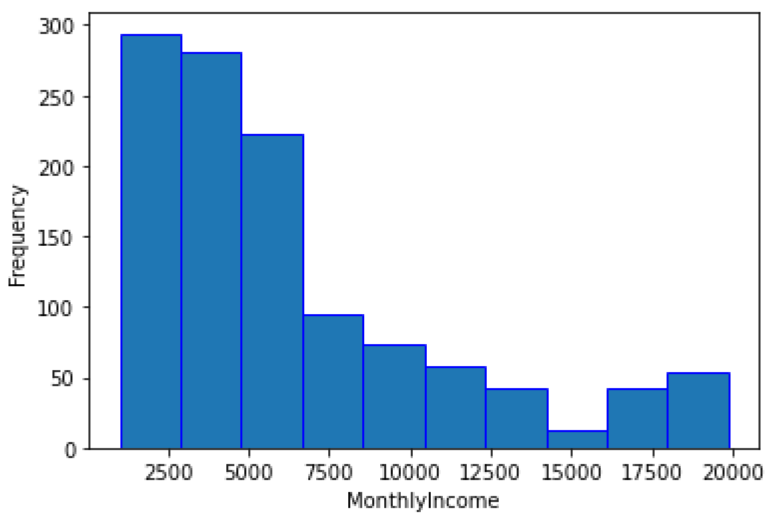
ANS: Ageสามารถใช้Gaussianได้เพราะโค้งค่อนข้างเข้สรูปแบบGaussian แต่MonthlyIncomeกับDistanceFromHomeไม่ได้ อาจต้องใช้ distributionอื่น หรือใช้ GMM มาสร้างโค้งที่เข้ารูปกับข้อมูล

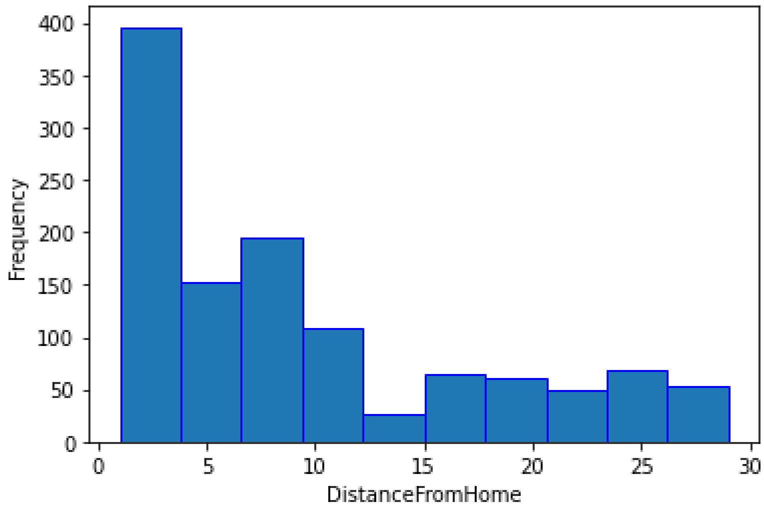
T6. Now plot the histogram according to the method described above (with 10, 40, and 100 bins) and show 3 plots each for Age, MonthlyIncome, and DistanceFromHome. Which bin size is most sensible for each features? Why?

In [ ]:
```
bins=[10,40,100]
for bin in bins:
    print("bin =",bin)
    display_histogram(df,["Age","MonthlyIncome","DistanceFromHome"],bin)
    print("-----------------------------------------------------------------------------
```

```
bin = 10
```



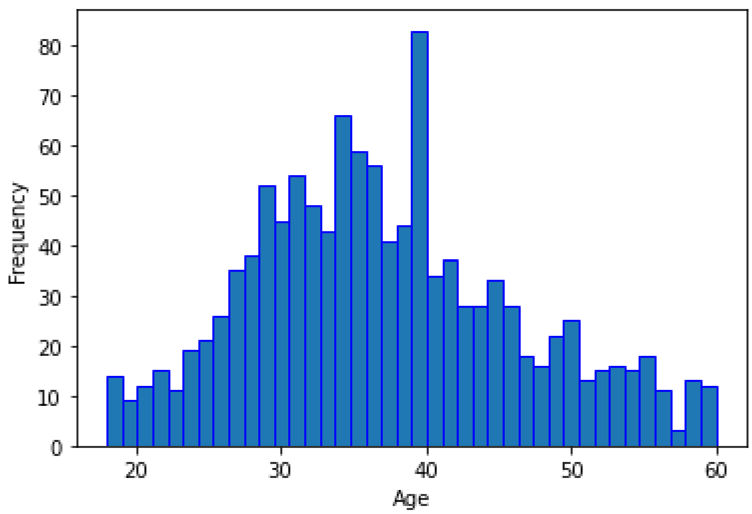




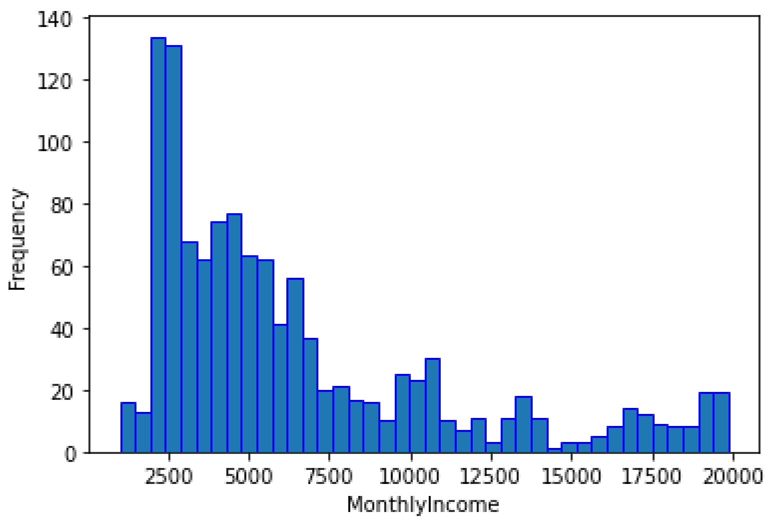
Frequency
DistanceFromHome

----

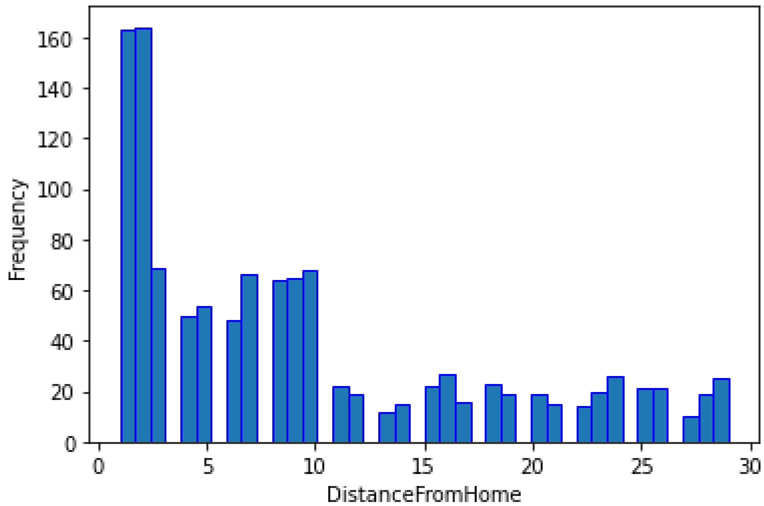
bin = 40



Frequency
Age



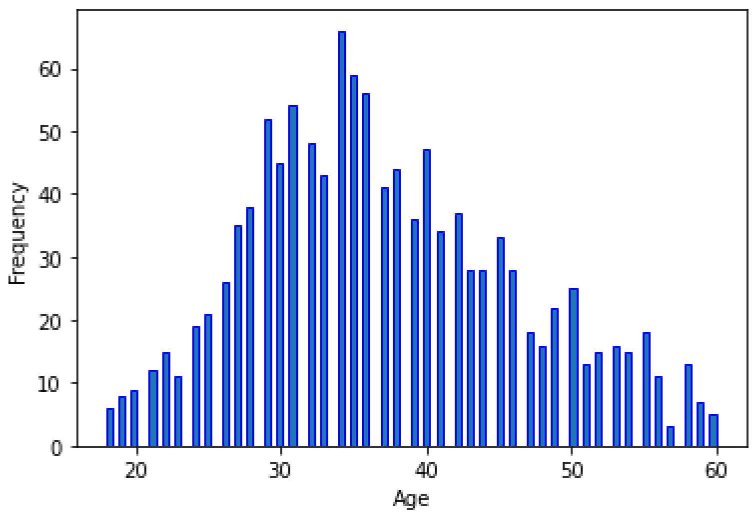
Frequency
MonthlyIncome



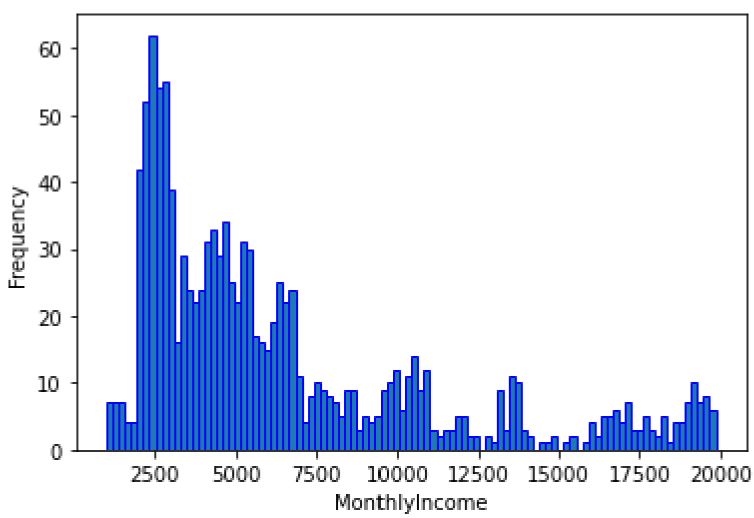
Frequency
DistanceFromHome

----

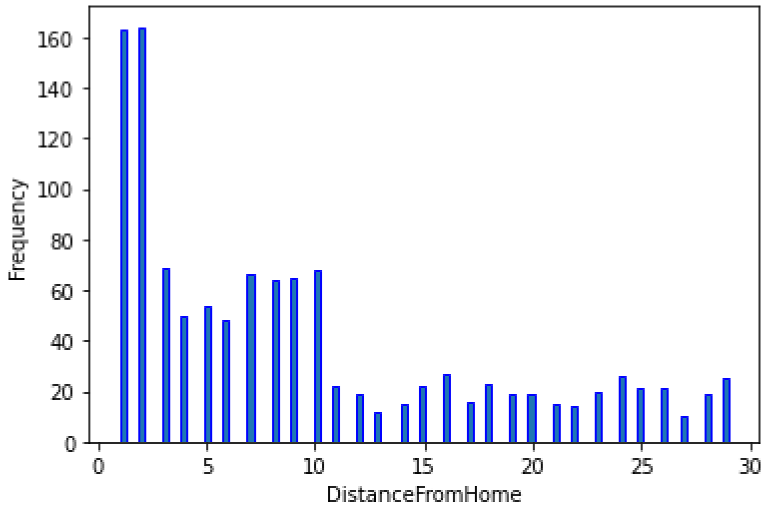
bin = 100



Frequency
Age



Frequency
MonthlyIncome



T7. For the rest of the features, which one should be discretized in order to be modeled by histograms? What are the criteria for choosing whether we should discretize a feature or not? Answer this and discretize those features into 10 bins each. In other words, figure out the bin edge for each feature, then use digitize() to convert the features to discrete values

T8. What kind of distribution should we use to model histograms? (Answer a distribution name) What is the MLE for the likelihood distribution? (Describe how to do the MLE). Plot the likelihood distributions of MonthlyIncome, JobRole, HourlyRate, and MaritalStatus for different Attrition values.

T9. What is the prior distribution of the two classes?

In [ ]:
```
def get_prior(df, value):
  return df.loc[df["Attrition"] == value, "Attrition"].count() / df.shape[0]

p_leave = get_prior(df, 1)
p_stay = get_prior(df, 0)
print("Prior(leave):", p_leave)
print("Prior(stay):", p_stay)
```

```
Prior(leave): 0.16122448979591836
Prior(stay): 0.8387755102040816
```

T10. If we use the current Naive Bayes with our current Maximum Likelihood Estimates, we will find that some P (x i |attrition) will be zero and will result in the entire product term to be zero. Propose a method to fix this problem.

ANS : 1)flooring : ถ้าเป็น 0 ให้เปลี่ยนเป็น ค่าที่น้อยมากๆแทนเช่น 1e-10 2)smoothing : เกลี่ยค่าจากข้างมา

T11. Implement your Naive Bayes classifier. Use the learned distributions to classify the test set. Don't forget to allow your classifier to handle missing values in the test set. Report the overall Accuracy. Then, report the Precision, Recall, and F score for detecting attrition. See Lecture 1 for the definitions of each metric.

In [ ]:

```
class SimpleBayesClassifier:

    def __init__(self, n_pos, n_neg):

        """
        Initializes the SimpleBayesClassifier with prior probabilities.

        Parameters:
        n_pos (int): The number of positive samples.
        n_neg (int): The number of negative samples.

        Returns:
        None: This method does not return anything as it is a constructor.
        """

        self.n_pos =n_pos
        self.n_neg =n_neg
        self.prior_pos = n_pos/(n_pos+n_neg)
        self.prior_neg =n_neg/(n_pos+n_neg)

    def fit_params(self, x, y, n_bins = 10):

        """
        Computes histogram-based parameters for each feature in the dataset.

        Parameters:
        x (np.ndarray): The feature matrix, where rows are samples and columns are fea
        y (np.ndarray): The target array, where each element corresponds to the label
        n_bins (int): Number of bins to use for histogram calculation.

        Returns:
        (stay_params, leave_params): A tuple containing two lists of tuples,
        one for 'stay' parameters and one for 'leave' parameters.
        Each tuple in the list contains the bins and edges of the histogram for a feat
        """

        self.stay_params = [(None, None) for _ in range(x.shape[1])]
        self.leave_params = [(None, None) for _ in range(x.shape[1])]

        # INSERT CODE HERE
        for i in range(x.shape[1]):
            x_stay = x[y == 0, i]
            x_stay=x_stay[~np.isnan(x_stay)]
            x_leave = x[y == 1, i]
            x_leave=x_leave[~np.isnan(x_leave)]
            a,b=np.histogram(x_stay,n_bins)
```

```
            a=a/x_stay.shape[0]
            self.stay_params[i]=(a,b)
            a,b=np.histogram(x_leave,n_bins)
            a=a/x_leave.shape[0]
            self.leave_params[i]=(a,b)
        return self.stay_params, self.leave_params

    def H(self,x):
        result=(self.prior_pos/self.prior_neg)
        for i in range(len(x)):
            if(math.isnan(x[i])):
                continue

            b=0
            if x[i]<self.leave_params[i][1][0]:
                b=0
            elif x[i]>=self.leave_params[i][1][-2]:
                b=len(self.leave_params[i][0])-1
            else:
                for j in range(1,len(self.leave_params[i][1])-1,1):
                    if(self.leave_params[i][1][j-1]<x[i]<=self.leave_params[i][1][j+1]
                        b=j
                        break

            if(self.leave_params[i][0][b]<=1e-10 or self.stay_params[i][0][b]<=1e-10):
                continue
            else:
                result=result*self.leave_params[i][0][b]
                result=result/self.stay_params[i][0][b]
        return result

    def predict(self, x, thresh = 1):

        """
        Predicts the class labels for the given samples using the non-parametric model

        Parameters:
        x (np.ndarray): The feature matrix for which predictions are to be made.
        thresh (float): The threshold for log probability to decide between classes.

        Returns:
        result (list): A list of predicted class labels (0 or 1) for each sample in th
        """

        y_pred = []

        # INSERT CODE HERE
        for i in range(x.shape[0]):
            if self.H(x[i])>=thresh :
                y_pred.append(1)
            else:
                y_pred.append(0)

        return y_pred

    def fit_gaussian_params(self, x, y):

        """
        Computes mean and standard deviation for each feature in the dataset.
```

```
        Parameters:
        x (np.ndarray): The feature matrix, where rows are samples and columns are fea
        y (np.ndarray): The target array, where each element corresponds to the label

        Returns:
        (gaussian_stay_params, gaussian_leave_params): A tuple containing two lists of
        one for 'stay' parameters and one for 'leave' parameters.
        Each tuple in the list contains the mean and standard deviation for a feature.
        """

        self.gaussian_stay_params = [(0, 0) for _ in range(x.shape[1])]
        self.gaussian_leave_params = [(0, 0) for _ in range(x.shape[1])]

        # INSERT CODE HERE
        for i in range(x.shape[1]):
            x_stay = x[y == 0, i]
            x_leave = x[y == 1, i]
            x_stay=x_stay[~np.isnan(x_stay)]
            x_leave=x_leave[~np.isnan(x_leave)]
            self.gaussian_stay_params[i]=(np.mean(x_stay),np.var(x_stay))
            self.gaussian_leave_params[i]=(np.mean(x_leave),np.var(x_leave))


        return self.gaussian_stay_params, self.gaussian_leave_params

    def gaussian(self,x,mean,sigma_2):
        return (1 / (math.sqrt(2 * math.pi * sigma_2))) * np.exp(-(x - mean)**2 / (2 *

    def calP(self,x):
        result = (self.prior_pos/self.prior_neg)
        for i in range(len(x)):
            if(math.isnan(x[i])):
                continue

            result*=self.gaussian(x[i],self.gaussian_leave_params[i][0],self.gaussian_
            result/=self.gaussian(x[i],self.gaussian_stay_params[i][0],self.gaussian_s
        return result

    def gaussian_predict(self, x, thresh = 1):

        """
        Predicts the class labels for the given samples using the parametric model.

        Parameters:
        x (np.ndarray): The feature matrix for which predictions are to be made.
        thresh (float): The threshold for log probability to decide between classes.

        Returns:
        result (list): A list of predicted class labels (0 or 1) for each sample in th
        """

        y_pred = []


        for i in range(x.shape[0]):
            if self.calP(x[i])>=thresh :
                y_pred.append(1)
            else:
                y_pred.append(0)
```

```
        return y_pred
```

```
In [ ]: X_train_leave = X_train.loc[df["Attrition"] == 1.0].copy()
        X_train_stay  = X_train.loc[df["Attrition"] == 0.0].copy()
```

```
In [ ]: model = SimpleBayesClassifier(n_pos =X_train_leave.shape[0] , n_neg = X_train_stay.sha
```

```
In [ ]: def check_prior():
            """
            This function designed to test the implementation of the prior probability calcula
            Specifically, it checks if the classifier correctly computes the prior probabiliti
            negative and positive classes based on given input counts.
            """

            # prior_neg = 5/(5 + 5) = 0.5 and # prior_pos = 5/(5 + 5) = 0.5
            assert (SimpleBayesClassifier(5, 5).prior_pos, SimpleBayesClassifier(5, 5).prior_r

            assert (SimpleBayesClassifier(3, 5).prior_pos, SimpleBayesClassifier(3, 5).prior_r
            assert (SimpleBayesClassifier(0, 1).prior_pos, SimpleBayesClassifier(0, 1).prior_r
            assert (SimpleBayesClassifier(1, 0).prior_pos, SimpleBayesClassifier(1, 0).prior_r

        check_prior()
```

```
In [ ]: a,b=model.fit_params(X_train.to_numpy(), y_train.to_numpy())
```

```
In [ ]: def check_fit_params():

            """
            This function is designed to test the fit_params method of a SimpleBayesClassifier
            This method is presumably responsible for computing parameters for a Naive Bayes c
            based on the provided training data. The parameters in this context is bins and ed
            """


            T = SimpleBayesClassifier(2, 2)
            X_TRAIN_CASE_1 = np.array([
                [0, 1, 2, 3],
                [1, 2, 3, 4],
                [2, 3, 4, 5],
                [3, 4, 5, 6]
            ])
            Y_TRAIN_CASE_1 = np.array([0, 1, 0, 1])
            STAY_PARAMS_1, LEAVE_PARAMS_1 = T.fit_params(X_TRAIN_CASE_1, Y_TRAIN_CASE_1)

            print("STAY PARAMETERS")
            for f_idx in range(len(STAY_PARAMS_1)):
                print(f"Feature : {f_idx}")
                print(f"BINS : {STAY_PARAMS_1[f_idx][0]}")
                print(f"EDGES : {STAY_PARAMS_1[f_idx][1]}")
            print("")
            print("LEAVE PARAMETERS")
            for f_idx in range(len(STAY_PARAMS_1)):
                print(f"Feature : {f_idx}")
                print(f"BINS : {LEAVE_PARAMS_1[f_idx][0]}")
                print(f"EDGES : {LEAVE_PARAMS_1[f_idx][1]}")

        check_fit_params()
```

```
STAY PARAMETERS
Feature : 0
BINS : [0.5 0.  0.  0.  0.  0.  0.  0.  0.  0.5]
EDGES : [0.  0.2 0.4 0.6 0.8 1.  1.2 1.4 1.6 1.8 2. ]
Feature : 1
BINS : [0.5 0.  0.  0.  0.  0.  0.  0.  0.  0.5]
EDGES : [1.  1.2 1.4 1.6 1.8 2.  2.2 2.4 2.6 2.8 3. ]
Feature : 2
BINS : [0.5 0.  0.  0.  0.  0.  0.  0.  0.  0.5]
EDGES : [2.  2.2 2.4 2.6 2.8 3.  3.2 3.4 3.6 3.8 4. ]
Feature : 3
BINS : [0.5 0.  0.  0.  0.  0.  0.  0.  0.  0.5]
EDGES : [3.  3.2 3.4 3.6 3.8 4.  4.2 4.4 4.6 4.8 5. ]

LEAVE PARAMETERS
Feature : 0
BINS : [0.5 0.  0.  0.  0.  0.  0.  0.  0.  0.5]
EDGES : [1.  1.2 1.4 1.6 1.8 2.  2.2 2.4 2.6 2.8 3. ]
Feature : 1
BINS : [0.5 0.  0.  0.  0.  0.  0.  0.  0.  0.5]
EDGES : [2.  2.2 2.4 2.6 2.8 3.  3.2 3.4 3.6 3.8 4. ]
Feature : 2
BINS : [0.5 0.  0.  0.  0.  0.  0.  0.  0.  0.5]
EDGES : [3.  3.2 3.4 3.6 3.8 4.  4.2 4.4 4.6 4.8 5. ]
Feature : 3
BINS : [0.5 0.  0.  0.  0.  0.  0.  0.  0.  0.5]
EDGES : [4.  4.2 4.4 4.6 4.8 5.  5.2 5.4 5.6 5.8 6. ]
```

In [ ]:

```python
y_pred = model.predict(X_test.to_numpy())
print("T11")
print(y_pred)
```

```
T11
[0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
0, 1, 0, 0, 0, 0, 0]
```

In [ ]:

```python
def evaluate(y_test, y_pred):
  tp,tn,fp,fn=0,0,0,0
  for i in range(len(y_pred)):
    if(y_pred[i]==0 and y_test[i]==0):
      tp+=1
    elif(y_pred[i]==0 and y_test[i]==1):
      fp+=1
    elif(y_pred[i]==1 and y_test[i]==0):
      fn+=1
    else:
      tn+=1


  return  tp,tn,fp,fn
```

In [ ]:

```python
tp,tn,fp,fn=evaluate(y_test.to_numpy(), y_pred)
accuracy = (tp+tn)/(tp+tn+fp+fn)
precision = tp/(tp+fp)
recall = tp/(tp+fn)
f1=(2*(precision)*(recall))/(precision+recall)
```

```python
print("accuracy : ",accuracy)
print("precision : ",precision)
print("recall : ",recall)
print("f1 : ",f1)
```

```
accuracy :  0.8775510204081632
precision :  0.9296875
recall :  0.9296875
f1 :  0.9296875
```

T12. Use the learned distributions to classify the test set. Report the results using the same metric as the previous question.

In [ ]:

```python
print("T12")
model.fit_gaussian_params(X_train.to_numpy(), y_train)
```

```
T12
```

```
Out[ ]: ([(37.923863636363635, 80.18624870867768),
  (1.071493212696832, 1.4763366843430723),
  (808.623172103487, 162078.18533481963),
  (0.7981900452488688, 0.9936618824348398),
  (9.026136363636363, 65.89136234504133),
  (2.93609865470852, 1.0799973355587285),
  (1.602714932126697, 3.1064179685100632),
  (2.778275475923852, 1.135608672153329),
  (0.2642533936651584, 0.5889936733482115),
  (66.07613636363637, 412.1953396177685),
  (2.781426953567384, 0.46524960593262177),
  (2.183371298405467, 1.2704751947115256),
  (3.2841628959276017, 9.343685837718311),
  (2.7771493212669682, 1.213912235621711 4),
  (0.6479638009049774, 1.0787854466534261),
  (6884.5, 23374549.48755656),
  (14218.990980834273, 50121922.50499194),
  (2.6980703745743475, 6.126772667011096),
  (0.005429864253393665, 0.38006101431174627),
  (15.262857142857143, 13.060048979591835),
  (3.1455981941309257, 0.12439935999673883),
  (2.7317620650953987, 1.1727173216136928),
  (0.8038548752834467, 0.6588060016145536),
  (11.940045248868778, 62.13780814274891),
  (2.8277027027027026, 1.6854037314341368),
  (2.790909090909091, 0.46309917355371905),
  (7.417995444191344, 39.09748937583346),
  (4.437570303712036, 13.067249889601822),
  (2.200458190148912, 10.14881995042834),
  (4.318493150684931, 12.737603208857196)],
 [(33.548022598870055, 98.69967123112771),
  (1.114678899082569, 1.3308854473529164),
  (750.6768292682926, 169437.25531677576),
  (0.7660550458715596, 1.2342605841259153),
  (10.88950276243094, 71.75574616159457),
  (2.8131868131868134, 1.0530129211447894),
  (1.651376146788991, 3.4656173722750614),
  (2.4850299401197606, 1.3515723044928105),
  (0.29357798165137616, 0.6018853631849169),
  (65.16279069767442, 411.415359653867),
  (2.519553072625698 5, 0.5736400237196092),
  (1.6519337016574585, 0.9230487469857453),
  (3.7660550458715596, 10.133343152933254),
  (2.5714285714285716, 1.1820408163265308),
  (0.8440366972477065, 1.3976937968184495),
  (4690.156976744186, 13461904.725358304),
  (14162.290697674418, 51622148.9736344),
  (2.9497206703910615, 7.299147966667706),
  (0.16972477064220184, 0.6088081811295345),
  (14.983333333333333, 13.938611111111 11),
  (3.153409090909091, 0.12987474173553723),
  (2.5654761904761907, 1.281427154195011),
  (0.47752808988764045, 0.6652253503345538),
  (8.549707602339181, 54.890804008070866),
  (2.6089385474860336, 1.567741331419119),
  (2.697674418604651, 0.699296917252569),
  (5.362068965517241, 31.909135949266748),
  (3.309941520467836, 11.360076604767277),
  (1.9942857142857142, 10.131395918367348),
  (2.9887005649717513, 10.564844074180474)])
```

In [ ]:
```python
def check_fit_gaussian_params():

    """
    This function is designed to test the fit_gaussian_params method of a SimpleBayesC
    This method is presumably responsible for computing parameters for a Naive Bayes c
    based on the provided training data. The parameters in this context is mean and ST
    """

    T = SimpleBayesClassifier(2, 2)
    X_TRAIN_CASE_1 = np.array([
        [0, 1, 2, 3],
        [1, 2, 3, 4],
        [2, 3, 4, 5],
        [3, 4, 5, 6]
    ])
    Y_TRAIN_CASE_1 = np.array([0, 1, 0, 1])
    STAY_PARAMS_1, LEAVE_PARAMS_1 = T.fit_gaussian_params(X_TRAIN_CASE_1, Y_TRAIN_CASE

    print("STAY PARAMETERS")
    for f_idx in range(len(STAY_PARAMS_1)):
        print(f"Feature : {f_idx}")
        print(f"Mean : {STAY_PARAMS_1[f_idx][0]}")
        print(f"STD. : {STAY_PARAMS_1[f_idx][1]}")
    print("")
    print("LEAVE PARAMETERS")
    for f_idx in range(len(STAY_PARAMS_1)):
        print(f"Feature : {f_idx}")
        print(f"Mean : {LEAVE_PARAMS_1[f_idx][0]}")
        print(f"STD. : {LEAVE_PARAMS_1[f_idx][1]}")

check_fit_gaussian_params()
```

```
STAY PARAMETERS
Feature : 0
Mean : 1.0
STD. : 1.0
Feature : 1
Mean : 2.0
STD. : 1.0
Feature : 2
Mean : 3.0
STD. : 1.0
Feature : 3
Mean : 4.0
STD. : 1.0

LEAVE PARAMETERS
Feature : 0
Mean : 2.0
STD. : 1.0
Feature : 1
Mean : 3.0
STD. : 1.0
Feature : 2
Mean : 4.0
STD. : 1.0
Feature : 3
Mean : 5.0
STD. : 1.0
```

In [ ]:
```
y_pred = model.gaussian_predict(X_test.to_numpy())
print("T12")
print(y_pred)
```

```
T12
[0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0,
0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0,
1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0,
0, 1, 0, 0, 0, 0, 0]
```

In [ ]:
```
tp,tn,fp,fn=evaluate(y_test.to_numpy(), y_pred)

accuracy = (tp+tn)/(tp+tn+fp+fn)
precision = tp/(tp+fp)
recall = tp/(tp+fn)
f1=(2*(precision)*(recall))/(precision+recall)

print("accuracy : ",accuracy)
print("precision : ",precision)
print("recall : ",recall)
print("f1 : ",f1)
```

```
accuracy :  0.7891156462585034
precision :  0.9217391304347826
recall :  0.828125
f1 :  0.8724279835390947
```

T13 : The random choice baseline is the accuracy if you make a random guess for each test sample. Give random guess (50% leaving, and 50% staying) to the test samples. Report the overall Accuracy. Then, report the Precision, Recall, and F score for attrition prediction using the random choice baseline.

In [ ]:
```
rand_y_pred= np.random.randint(2, size=y_test.shape[0])
tp,tn,fp,fn=evaluate(y_test.to_numpy(), rand_y_pred)
accuracy = (tp+tn)/(tp+tn+fp+fn)
precision = tp/(tp+fp)
recall = tp/(tp+fn)
f1=(2*(precision)*(recall))/(precision+recall)

print("T13)random")
print("accuracy : ",accuracy)
print("precision : ",precision)
print("recall : ",recall)
print("f1 : ",f1)
```

```
T13)random
accuracy :  0.48299319727891155
precision :  0.8714285714285714
recall :  0.4765625
f1 :  0.6161616161616161
```

T14. The majority rule is the accuracy if you use the most frequent class from the training set as the classification decision. Report the overall Accuracy. Then, report the Precision, Recall,

and F score for attrition prediction using the majority rule baseline.

In [ ]:
```
zero_y_pred= np.zeros(y_test.shape[0])

tp,tn,fp,fn=evaluate(y_test.to_numpy(), zero_y_pred)
accuracy = (tp+tn)/(tp+tn+fp+fn)
precision = tp/(tp+fp)
recall = tp/(tp+fn)
f1=(2*(precision)*(recall))/(precision+recall)
print("T14)random")
print("accuracy : ",accuracy)
print("precision : ",precision)
print("recall : ",recall)
print("f1 : ",f1)
```

```
T14)random
accuracy :  0.8707482993197279
precision :  0.8707482993197279
recall :  1.0
f1 :  0.9309090909090908
```

T15. Compare the two baselines with your Naive Bayes classifier.

T16. Use the following threshold values

$t = np.arange(-5, 5, 0.05)$

find the best accuracy, and F score (and the corresponding thresholds)

In [ ]:
```
t = np.arange(1,10,0.05)
maxacc,maxpreci,maxrecal,maxf1=0,0,0,0
threacc,threpreci,threrecal,thref1=0,0,0,0
for thre in t:
    y_pred = model.predict(X_test.to_numpy(),thre)
    tp,tn,fp,fn=evaluate(y_test.to_numpy(), y_pred)[0:4]
    accuracy = (tp+tn)/(tp+tn+fp+fn)
    precision = tp/(tp+fp)
    recall = tp/(tp+fn)
    f1=(2*(precision)*(recall))/(precision+recall)
    if(accuracy>=maxacc):
        maxacc=accuracy
        threacc=thre
    if(precision>=maxpreci):
        maxpreci=precision
        threpreci=thre
    if(recall>=maxrecal):
        maxrecal=recall
        threrecal=thre
    if(f1>=maxf1):
        maxf1=f1
        thref1=thre
print("T16) max")

print("max accuracy , thresholds : ",maxacc,threacc)
```

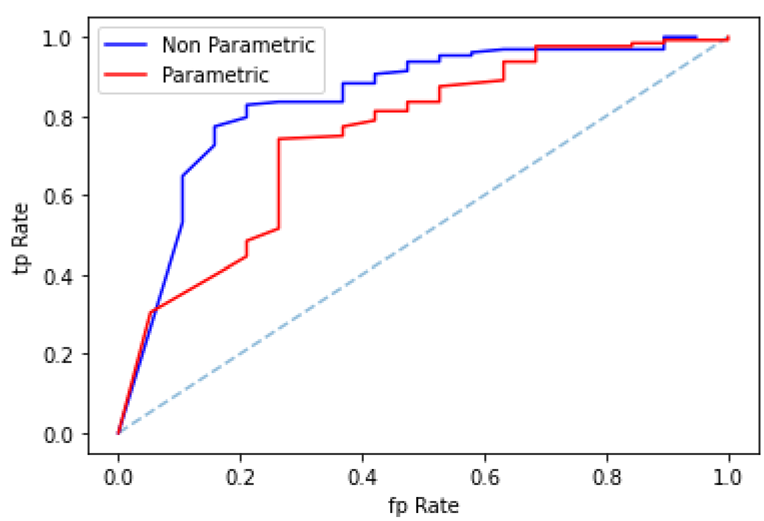
```
print("max precision , thresholds : ",maxpreci,threpreci)
print("max recall , thresholds : ",maxrecal,threrecal)
print("max f1 , thresholds : ",maxf1,thref1)
```

```
T16) max
max accuracy , thresholds :  0.891156462585034 3.200000000000002
max precision , thresholds :  0.9302325581395349 1.05
max recall , thresholds :  1.0 9.950000000000008
max f1 , thresholds :  0.9393939393939394 3.200000000000002
```

T17. Plot the RoC of your classifier.

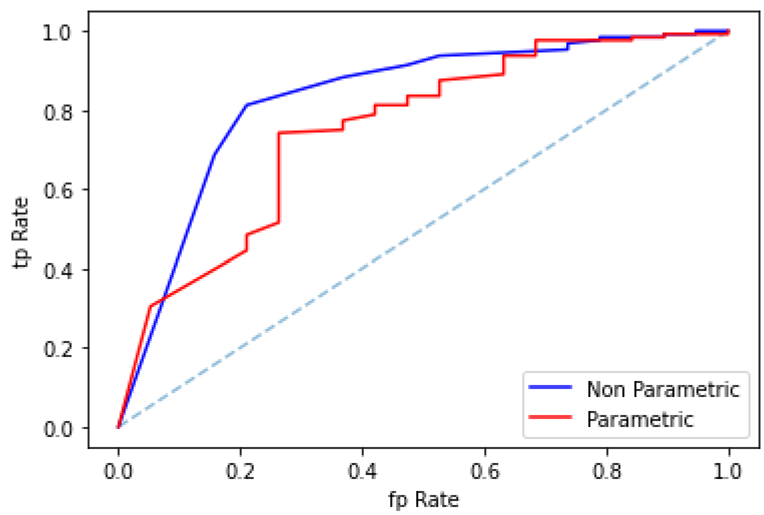
In [ ]:

```
thresholds=np.arange(-60,30,0.05)
xNonPara,yNonPara=[],[]
xPara,yPara=[],[]
for thre in thresholds:
    y_pred=model.predict(X_test.to_numpy(),thre)
    tp,tn,fp,fn=evaluate(y_test.to_numpy(),y_pred)
    xNonPara.append(fp/(fp+tn))
    yNonPara.append(tp/(tp+fn))
    y_pred=model.gaussian_predict(X_test.to_numpy(),thre)
    tp,tn,fp,fn=evaluate(y_test.to_numpy(),y_pred)
    xPara.append(fp/(fp+tn))
    yPara.append(tp/(tp+fn))
plt.plot([0, 1], [0, 1], "--", alpha=0.5)
plt.plot(xNonPara,yNonPara,'-',color="blue",label="Non Parametric")
plt.plot(xPara,yPara,'-',color="red",label="Parametric")
plt.xlabel("fp Rate")
plt.ylabel("tp Rate")
plt.legend()
plt.show()
```



T18. Change the number of discretization bins to 5. What happens to the RoC curve? Which discretization is better? The number of discretization bins can be considered as a hyperparameter, and must be chosen by comparing the final performance.

```
In [ ]: a,b=model.fit_params(X_train.to_numpy(), y_train.to_numpy(),5)
        xNonPara,yNonPara=[],[]
        xPara,yPara=[],[]
        for thre in thresholds:
            y_pred=model.predict(X_test.to_numpy(),thre)
            tp,tn,fp,fn=evaluate(y_test.to_numpy(),y_pred)
            xNonPara.append(fp/(fp+tn))
            yNonPara.append(tp/(tp+fn))
            y_pred=model.gaussian_predict(X_test.to_numpy(),thre)
            tp,tn,fp,fn=evaluate(y_test.to_numpy(),y_pred)
            xPara.append(fp/(fp+tn))
            yPara.append(tp/(tp+fn))
        plt.plot([0, 1], [0, 1], "--", alpha=0.5)
        plt.plot(xNonPara,yNonPara,'-',color="blue",label="Non Parametric")
        plt.plot(xPara,yPara,'-',color="red",label="Parametric")
        plt.xlabel("fp Rate")
        plt.ylabel("tp Rate")
        plt.legend()
        plt.show()
```



OT3.Shuffle the database, and create new test and train sets. Redo the entire training and evaluation process 10 times (each time with a new training and test set). Calculate the mean and variance of the accuracy rate.

```
In [ ]: n_round=10
        lst_accuracy=[]
        for i in range(n_round):
            X_train, X_test, y_train, y_test=train_test_split(df.loc[:, ~df.columns.isin(['Att
            X_train_leave = X_train.loc[df["Attrition"] == 1.0].copy()
            X_train_stay  = X_train.loc[df["Attrition"] == 0.0].copy()
            OT3model = SimpleBayesClassifier(n_pos =X_train_leave.shape[0] , n_neg = X_train_s
            a,b=model.fit_params(X_train.to_numpy(), y_train.to_numpy())
            y_pred = model.predict(X_test.to_numpy())
            tp,tn,fp,fn=evaluate(y_test.to_numpy(),y_pred)
            accuracy = (tp+tn)/(tp+tn+fp+fn)
            print("round",i+1,"accuracy : ",accuracy)
            lst_accuracy.append(accuracy)
        print(np.mean(lst_accuracy))
```

```
print(np.var(lst_accuracy))
```

```
round 1 accuracy :  0.8367346938775511
round 2 accuracy :  0.8095238095238095
round 3 accuracy :  0.8299319727891157
round 4 accuracy :  0.8571428571428571
round 5 accuracy :  0.8435374149659864
round 6 accuracy :  0.8095238095238095
round 7 accuracy :  0.8707482993197279
round 8 accuracy :  0.8503401360544217
round 9 accuracy :  0.8299319727891157
round 10 accuracy :  0.7891156462585034
0.8326530612244898
0.0005479198482113928
```