T1 find α , maximizes $p(y_2, y_1, y_0 | \alpha)$

จาก $P(y_n | y_{n-1}, y_{n-2}, \dots) = P(y_n | y_{n-1})$

แล: $y_2 = \alpha y_1 + w_1$

$y_1 = \alpha y_0 + w_0$

โดยที่ $y_0 \sim N(0, \lambda)$

และ $w_1, w_0 \sim N(0, \sigma^2)$

จะได้ $P(y_2, y_1, y_0 | \alpha) = P(y_2 | y_1, \alpha) \times P(y_1 | y_0, \alpha) \times P(y_0)$

$P(y_2 | y_1, \alpha) \sim N(\alpha y_1, \sigma^2)$

$P(y_1 | y_0, \alpha) \sim N(\alpha y_0, \sigma^2)$

จะได้ $P(y_2, y_1, y_0 | \alpha) \propto \left[\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_2 - \alpha y_1)^2}{2\sigma^2} \right) \right] \left[\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_1 - \alpha y_0)^2}{2\sigma^2} \right) \right]$

ตัด y_0 ได้ เพราะมาเป็นค่าคงที่ dif

$L(\alpha) = \log\left(\frac{1}{2\pi\sigma^2} \right) - \frac{(y_2 - \alpha y_1)^2}{2\sigma^2} - \frac{(y_1 - \alpha y_0)^2}{2\sigma^2}$

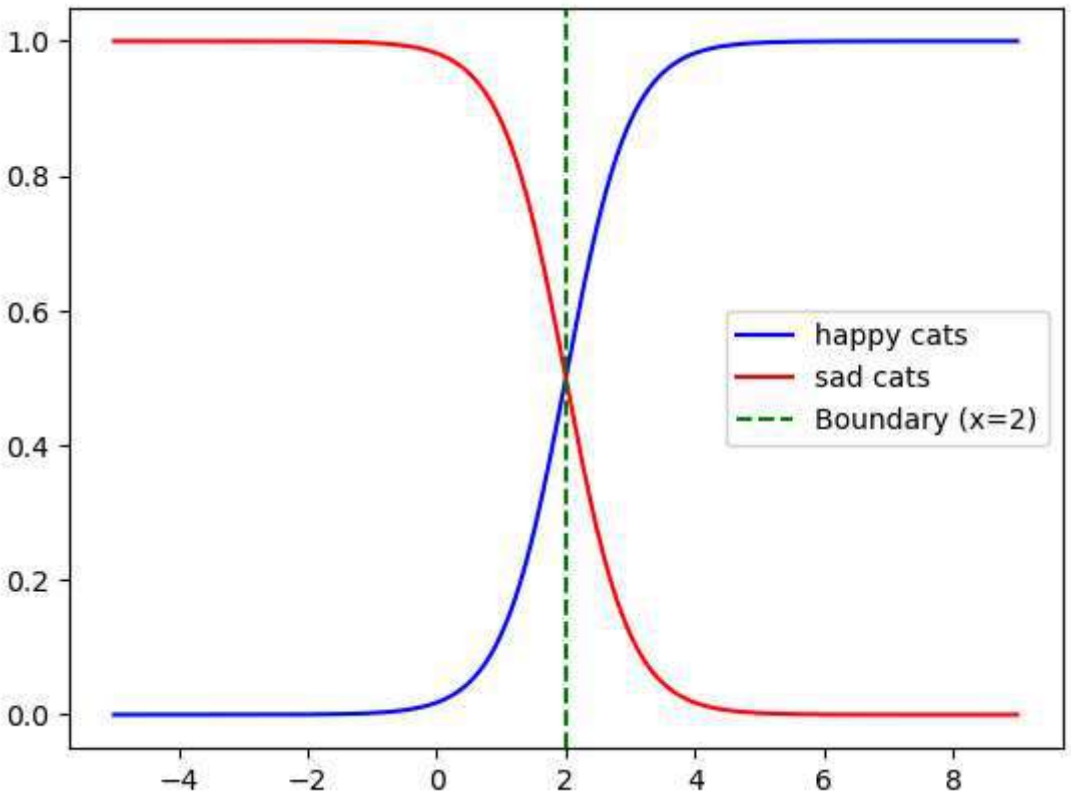
$\frac{d}{d\alpha} L(\alpha) = \frac{1}{2\sigma^2} \left(-2(y_2 - \alpha y_1)(-y_1) - 2(y_1 - \alpha y_0)(-y_0) \right)$

$0 = y_2 y_1 - \alpha y_1^2 + y_1 y_0 - \alpha y_0^2$

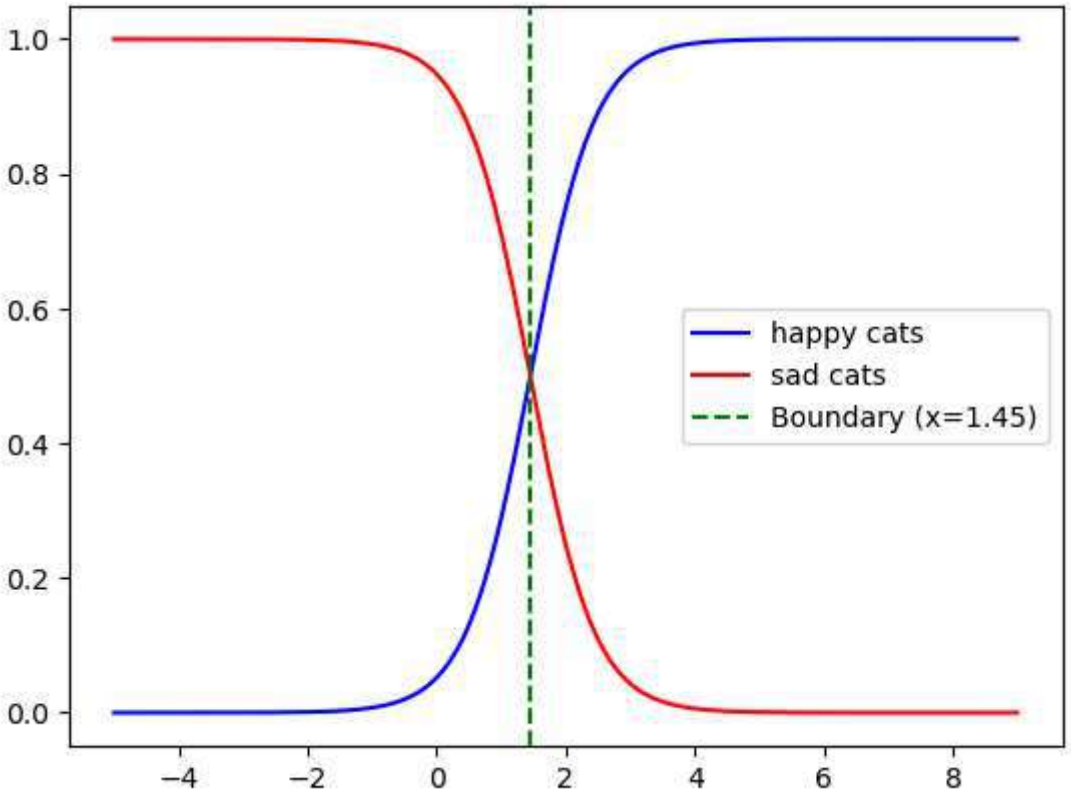
$\alpha = \frac{y_2 y_1 + y_1 y_0}{y_1^2 + y_0^2}$ ✳

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

In [2]:
```
#T2
'''
From the question, we assume equal prior probabilities. Then we can manually calcul
the decision boundary from the mean value of 2 dist which is x = 2.
'''
def norm_dis(x, mu, sigma):
    return (1 / (sigma * np.sqrt(2 * np.pi))) * np.exp(-0.5 * ((x - mu) / sigma)**2

mu1 = 4
mu2 = 0
sigma = np.sqrt(2)

x = np.linspace(-5, 9, 1000)

lh1 = norm_dis(x, mu1, sigma)
lh2 = norm_dis(x, mu2, sigma)

post1 = lh1 / (lh1 + lh2)
post2 = lh2 / (lh1 + lh2)

plt.plot(x, post1, label="happy cats", color='blue')
plt.plot(x, post2, label="sad cats", color='red')
plt.axvline(x = 2, color='green',linestyle='--', label='Boundary (x=2)')
plt.legend()
plt.show()
```



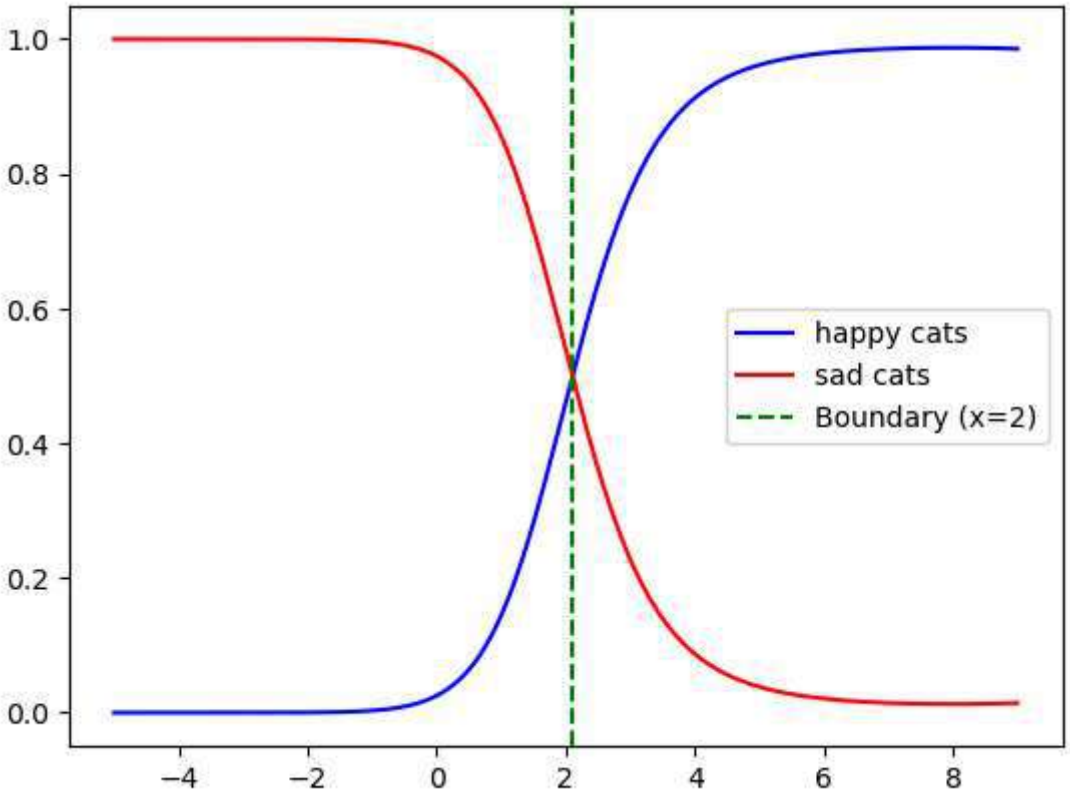
In [3]:
```
#T3
'''
From the question, we assume cat with the happy has a prior of 0.75 then
cat with the sad has a prior of 0.25.
After manually calculating the decision boundary, x is estimated to 1.45
'''
p1 = 0.75
p2 = 0.25
num1 = lh1*p1
num2 = lh2*p2

evidence = num1 + num2
post1 = num1 / evidence
post2 = num2 / evidence

plt.plot(x, post1, label="happy cats", color='blue')
plt.plot(x, post2, label="sad cats", color='red')
plt.axvline(x = 1.45, color='green',linestyle='--', label='Boundary (x=1.45)')
plt.legend()
plt.show()
```



In [ ]:
```
#OT2
```

In [107…
```
#OT3
'''
The decision boundary is around 2.105 found from the intersection.
'''
mu1 = 4
mu2 = 0
sigma1 = np.sqrt(2)
```

```
sigma2 = 2

x = np.linspace(-5, 9, 1000)

lh1 = norm_dis(x, mu1, sigma1)
lh2 = norm_dis(x, mu2, sigma2)

post1 = lh1 / (lh1 + lh2)
post2 = lh2 / (lh1 + lh2)

plt.plot(x, post1, label="happy cats", color='blue')
plt.plot(x, post2, label="sad cats", color='red')
plt.axvline(x = 2.105, color='green',linestyle='--', label='Boundary (x=2)')
plt.legend()
plt.show()
```



Employee Attrition Prediction

read CSV

In [4]:
```
df = pd.read_csv('hr-employee-attrition-with-null.csv')
```

Dataset statistic

In [5]:
```
df.describe()
```

Out[5]:

	Unnamed: 0	Age	DailyRate	DistanceFromHome	Education	EmployeeC
count	1470.000000	1176.000000	1176.000000	1176.00000	1176.000000	1
mean	734.500000	37.134354	798.875850	9.37500	2.920918	
std	424.496761	9.190317	406.957684	8.23049	1.028796	
min	0.000000	18.000000	102.000000	1.00000	1.000000	
25%	367.250000	30.000000	457.750000	2.00000	2.000000	
50%	734.500000	36.000000	798.500000	7.00000	3.000000	
75%	1101.750000	43.000000	1168.250000	15.00000	4.000000	
max	1469.000000	60.000000	1499.000000	29.00000	5.000000	

8 rows × 27 columns

In [6]:
```
df.head()
```

Out[6]:

	Unnamed: 0	Age	Attrition	BusinessTravel	DailyRate	Department	DistanceFromHome
0	0	41.0	Yes	Travel_Rarely	NaN	NaN	1.0
1	1	NaN	No	NaN	279.0	Research & Development	NaN
2	2	37.0	Yes	NaN	1373.0	NaN	2.0
3	3	NaN	No	Travel_Frequently	1392.0	Research & Development	3.0
4	4	27.0	No	Travel_Rarely	591.0	Research & Development	2.0

5 rows × 36 columns

In [7]:
```
# print(df.info())
# print(df["Attrition"].mode())
```

Feature transformation

In [8]:
```
df.loc[df["Attrition"] == "no", "Attrition"] = 0.0
df.loc[df["Attrition"] == "yes", "Attrition"] = 1.0
string_categorical_col = ['Department', 'Attrition', 'BusinessTravel', 'EducationFi
                          '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
for col in df.columns:
    if df[col].dtype == np.float64:
        df[col] = df[col].fillna(df[col].mean())
    else:
        df[col] = df[col].fillna(df[col].mode()[0])

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

Spliting data into train and test

In [9]:
```
from sklearn.model_selection import train_test_split
```

In [10]:
```
df_train, df_test = train_test_split(df, test_size = 0.1, random_state = 42, strati
```

Display histogram of each feature

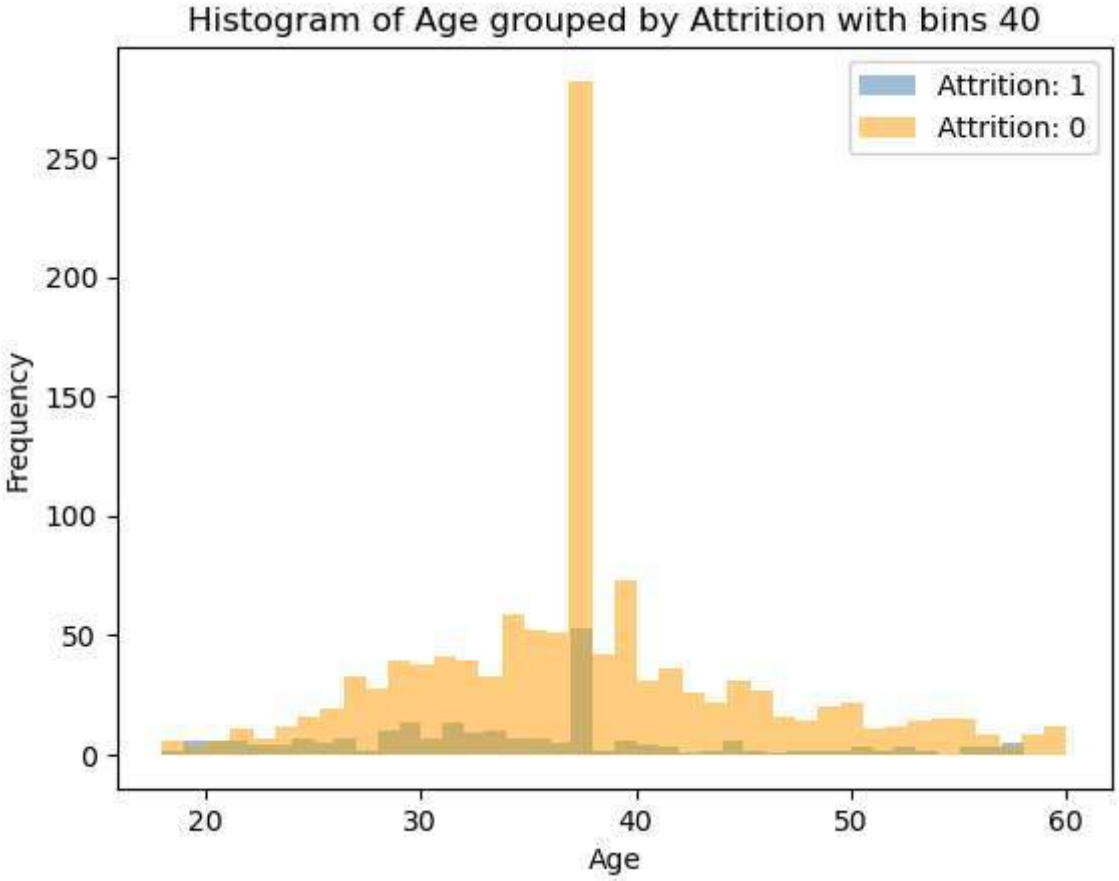
In [11]:
```
def display_histogram(df, col_name, cls, n_bin = 40):
    # INSERT CODE HERE
    classes = df[cls].unique()
    colors = ['steelblue', 'orange']

    for i, class_val in enumerate(classes):
        data_filtered = df[(df[cls] == class_val) & (df[col_name].notna())][col_nam
        hist, bin_edge = np.histogram(data_filtered, bins=n_bin)
        plt.fill_between(bin_edge.repeat(2)[1:-1],
                         hist.repeat(2),
                         facecolor=colors[i%len(colors)],
                         alpha=0.5,
                         label=f'{cls}: {class_val}')
        free_bin_indices = np.where(hist == 0)[0]
        print(f"Number of free bins: {len(free_bin_indices)}")
    plt.title(f'Histogram of {col_name} grouped by {cls} with bins {n_bin}')
    plt.xlabel(col_name)
    plt.ylabel('Frequency')
    plt.legend()
    plt.show()
```

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

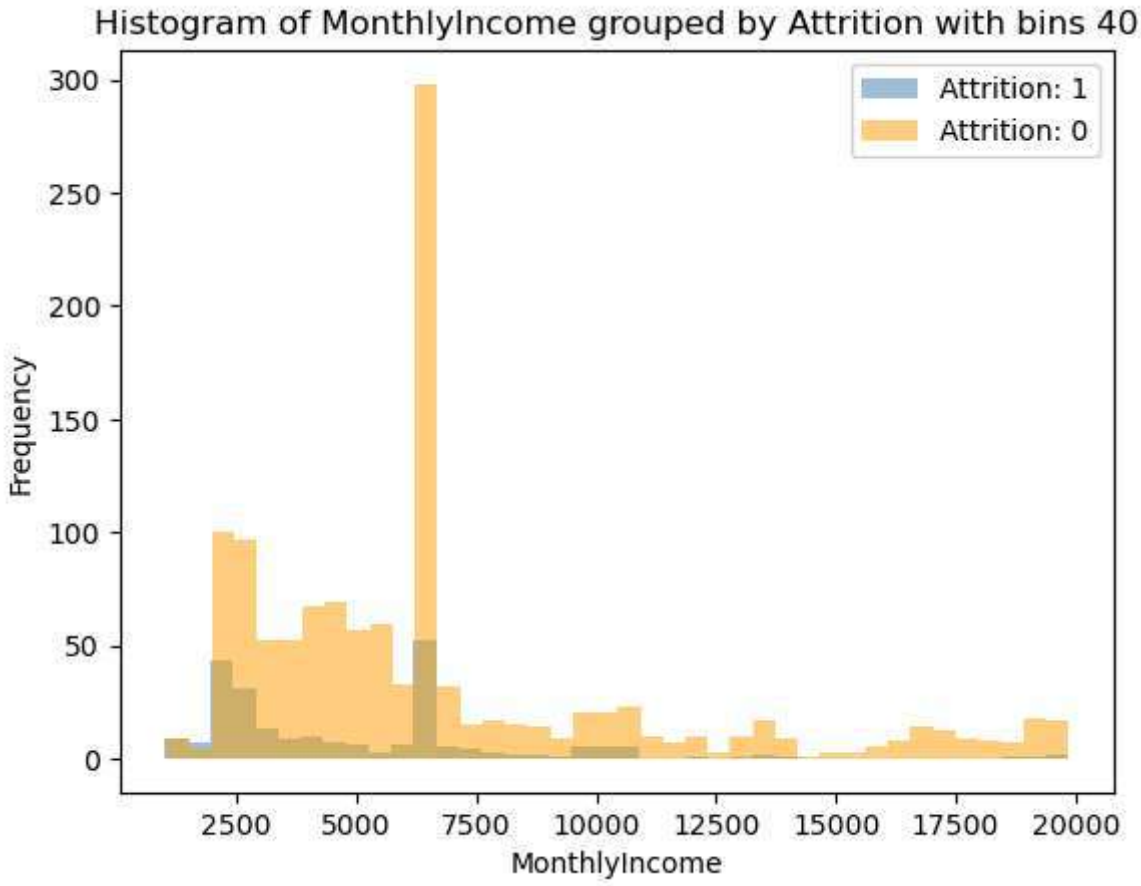
From the Age, There are no free bins.
From the MonthlyIncome, there are no free bins.
From the DistanceFromHome, there are 11 free bins.
Age and MonthlyIncome are good discretization but DistanceFromHome is not. So to solve this we must decrease bins size for DistanceFromHome.

In [12]:
```
display_histogram(df, 'Age', 'Attrition')
display_histogram(df, 'MonthlyIncome', 'Attrition')
display_histogram(df, 'DistanceFromHome', 'Attrition')
```

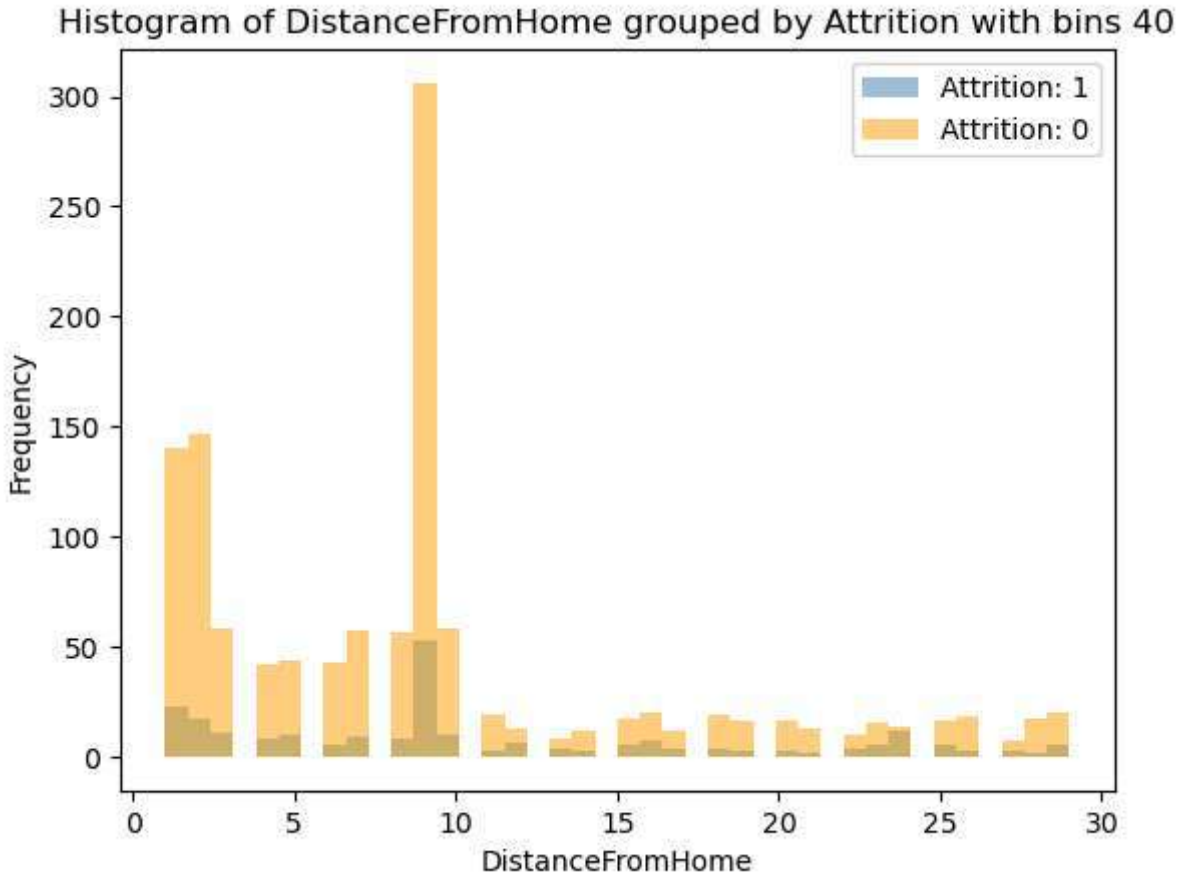
```
Number of free bins: 1
Number of free bins: 0
```




```
Number of free bins: 12
Number of free bins: 0
```




```
Number of free bins: 11
Number of free bins: 11
```



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

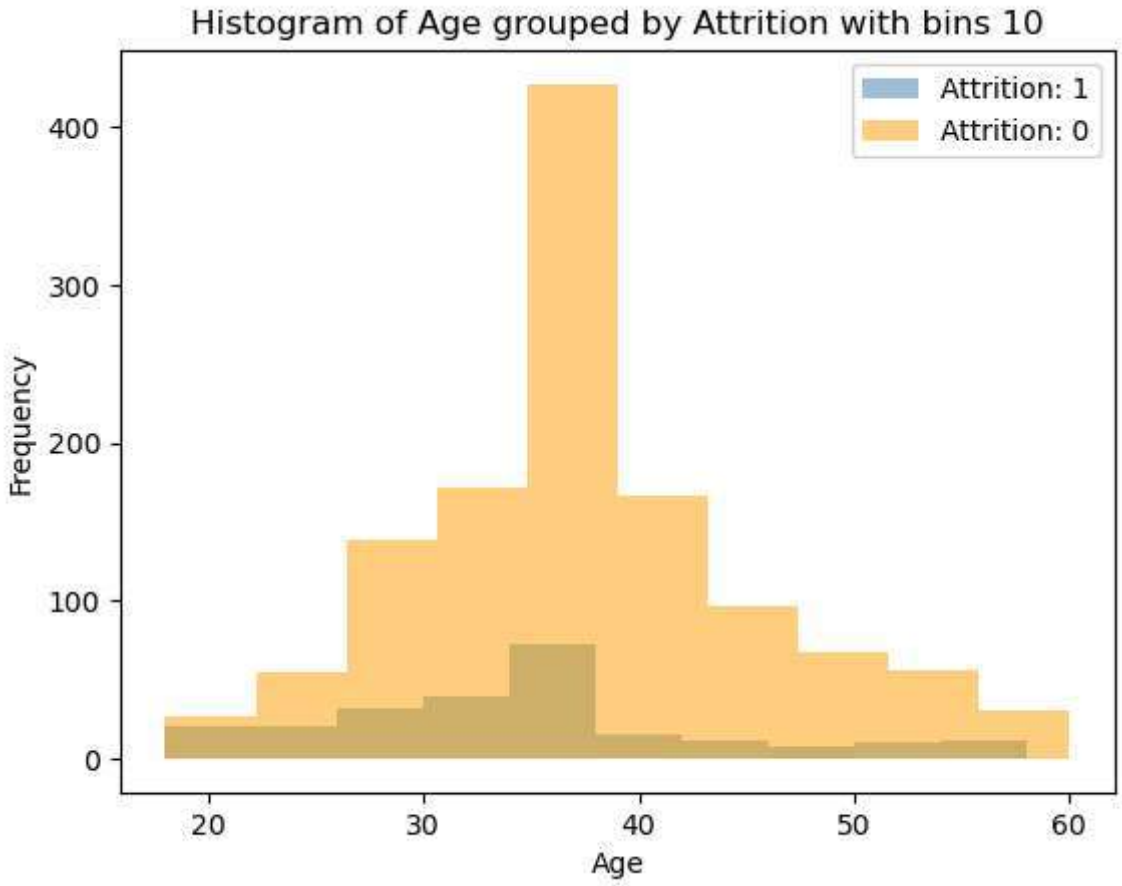
From the histogram from T4.
There are only Age that look like the normal distribution, and the rest is not.
So I think Age can use the gaussian to estimate while others use the GMM.

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?

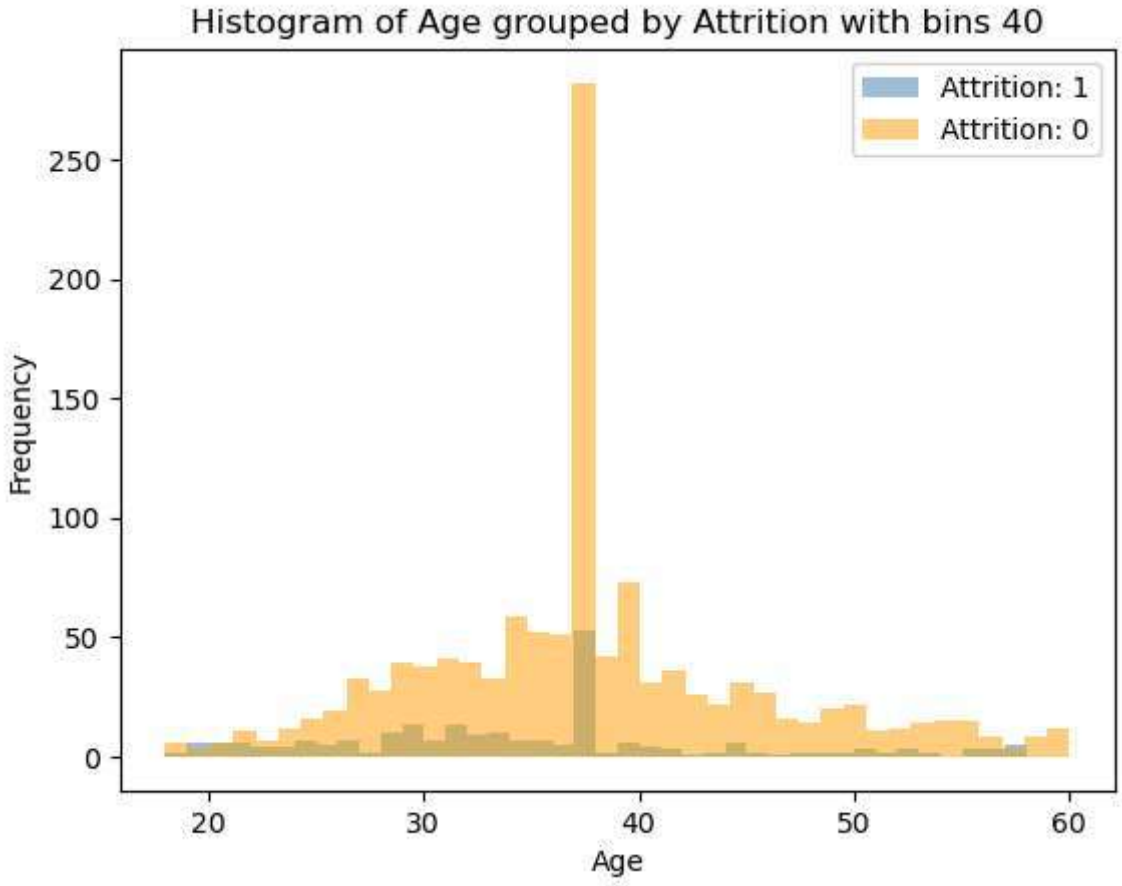
Age: Can use either 10 or 40, but I think 40 would be a better choices because 40 bins are detailed and the distribution is look like gaussian.
MonthlyIncome: I think 40 is the best because the data is not normal distribution and the 100 has a free bins which is bad discretization.
DistanceFromHome: The best is obivious 10 because all bins have data inside.

In [13]:
```
#bins = 10
for col in ['Age', 'MonthlyIncome', 'DistanceFromHome']:
    for bin in [10, 40, 100]:
        display_histogram(df, col, 'Attrition', bin)
```

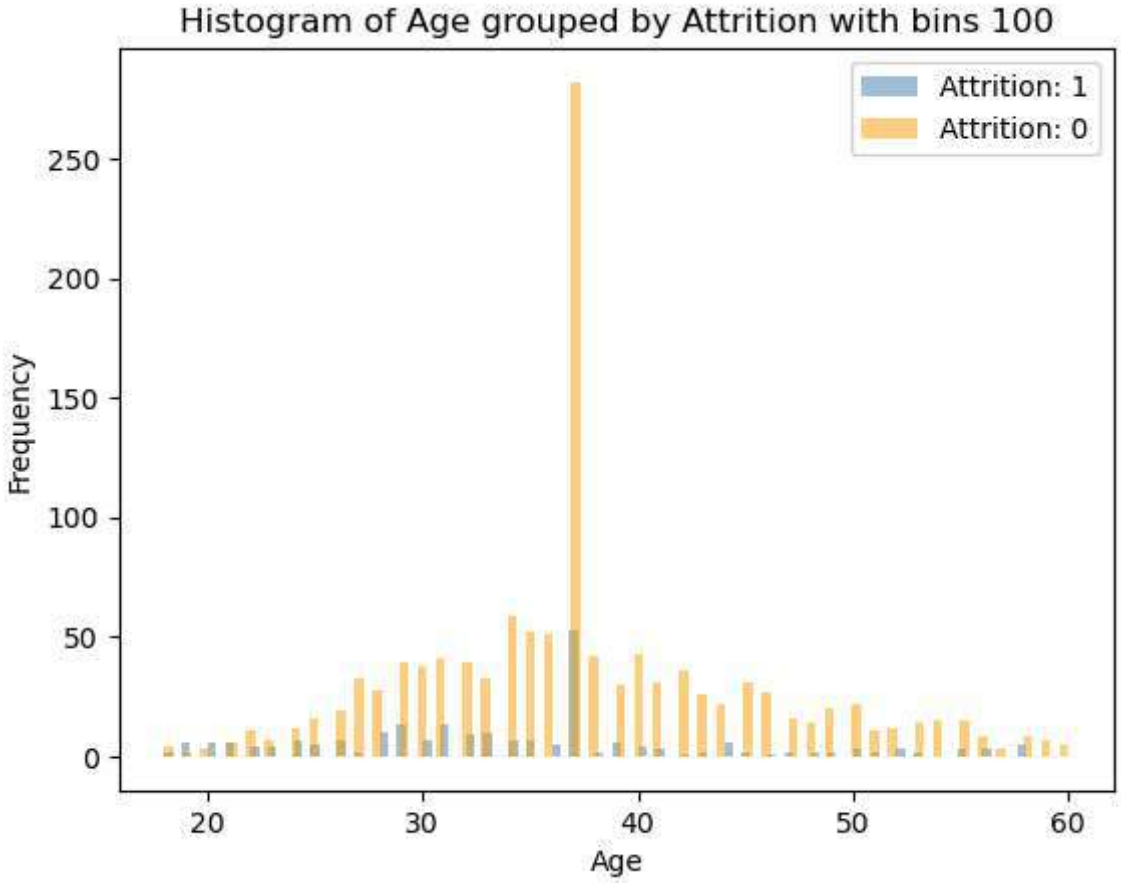
```
Number of free bins: 0
Number of free bins: 0
```



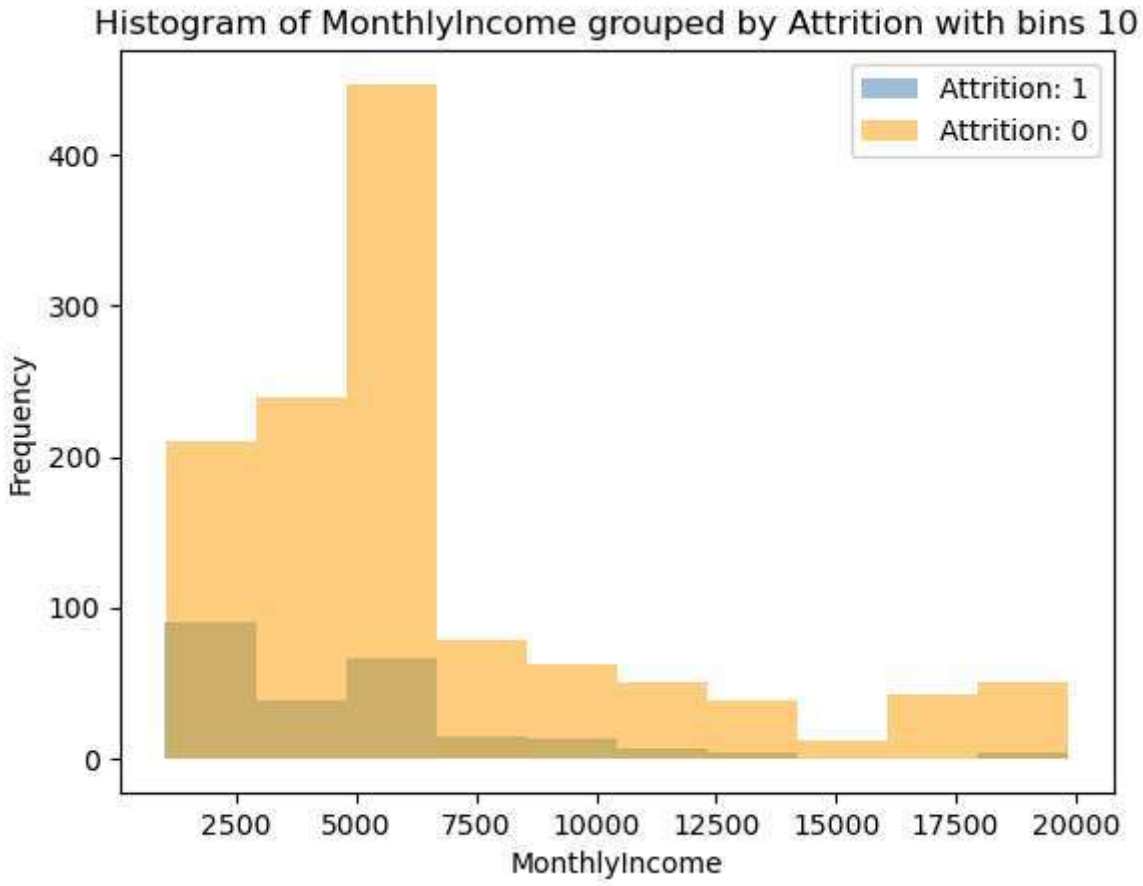

```
Number of free bins: 1
Number of free bins: 0
```



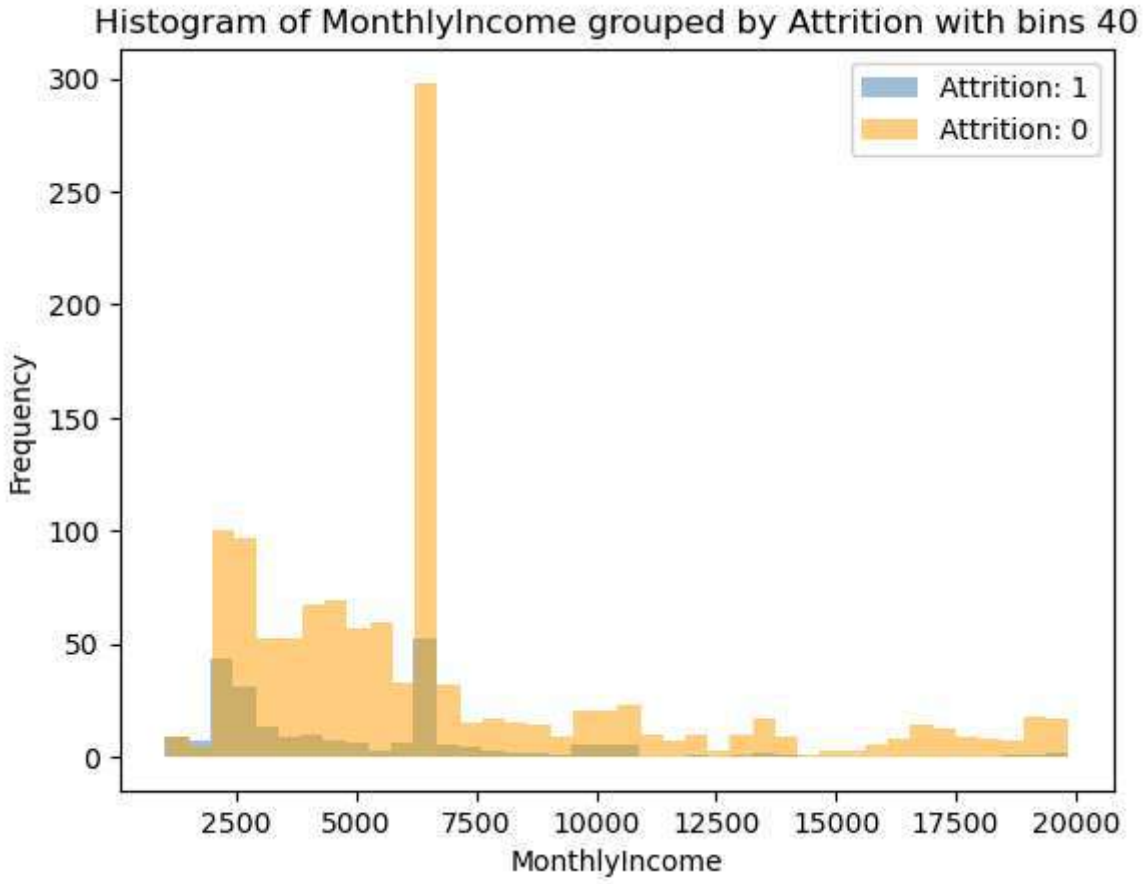
```
Number of free bins: 61
Number of free bins: 57
```



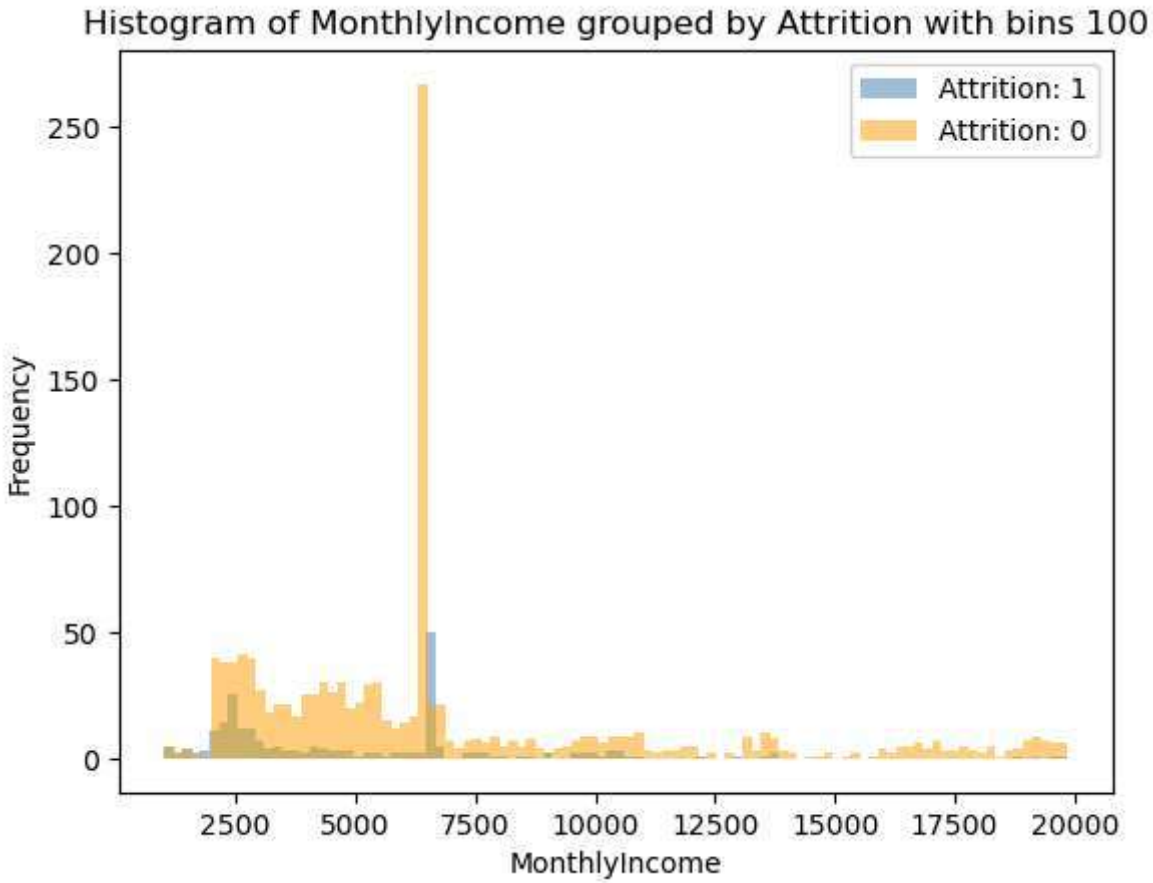

```
Number of free bins: 2
Number of free bins: 0
```



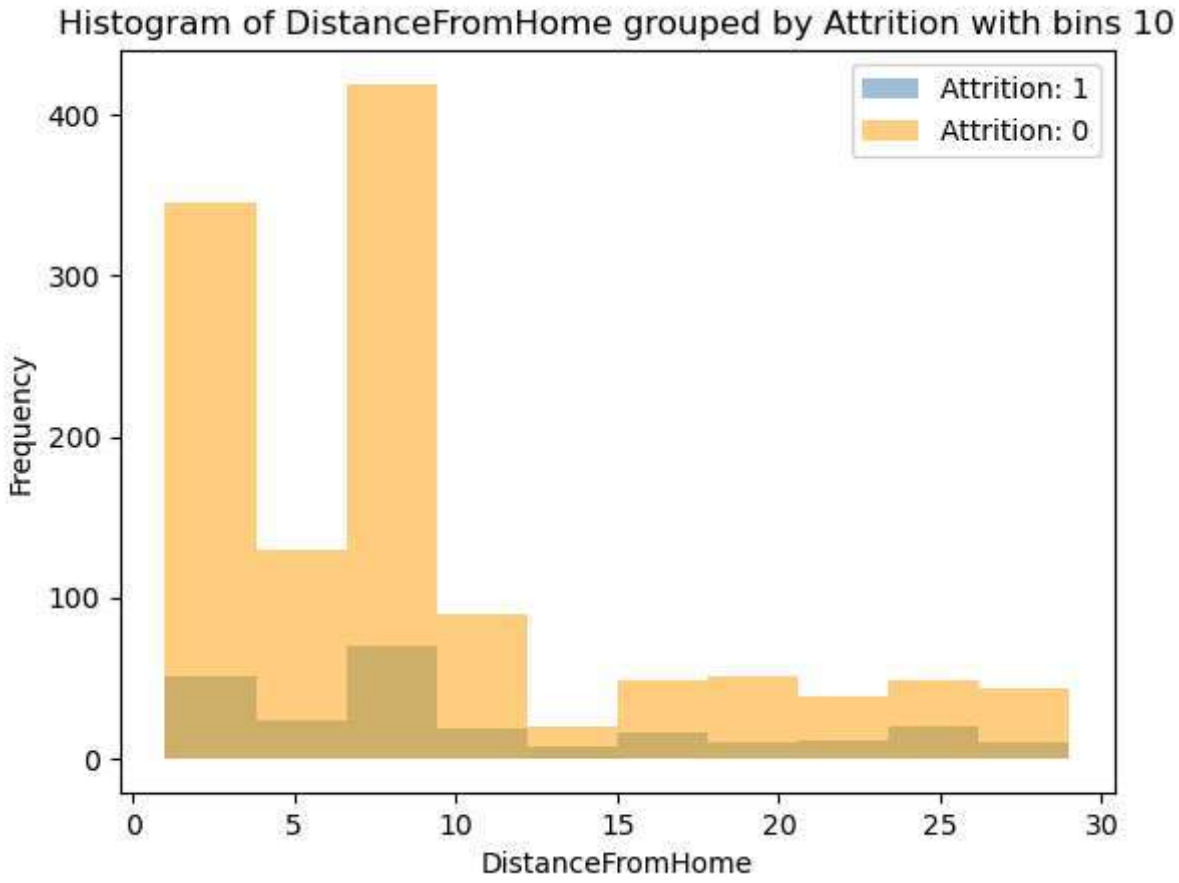

```
Number of free bins: 12
Number of free bins: 0
```



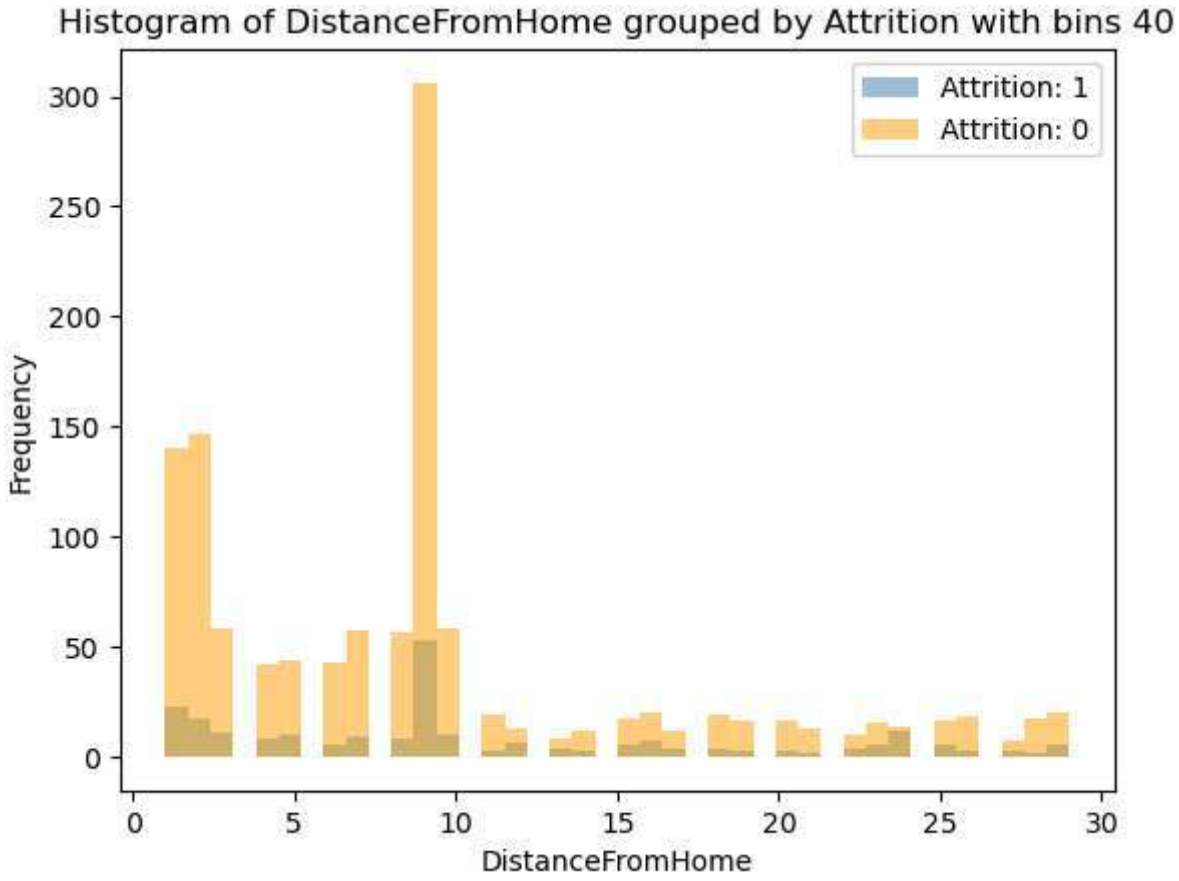
```
Number of free bins: 45
Number of free bins: 5
```



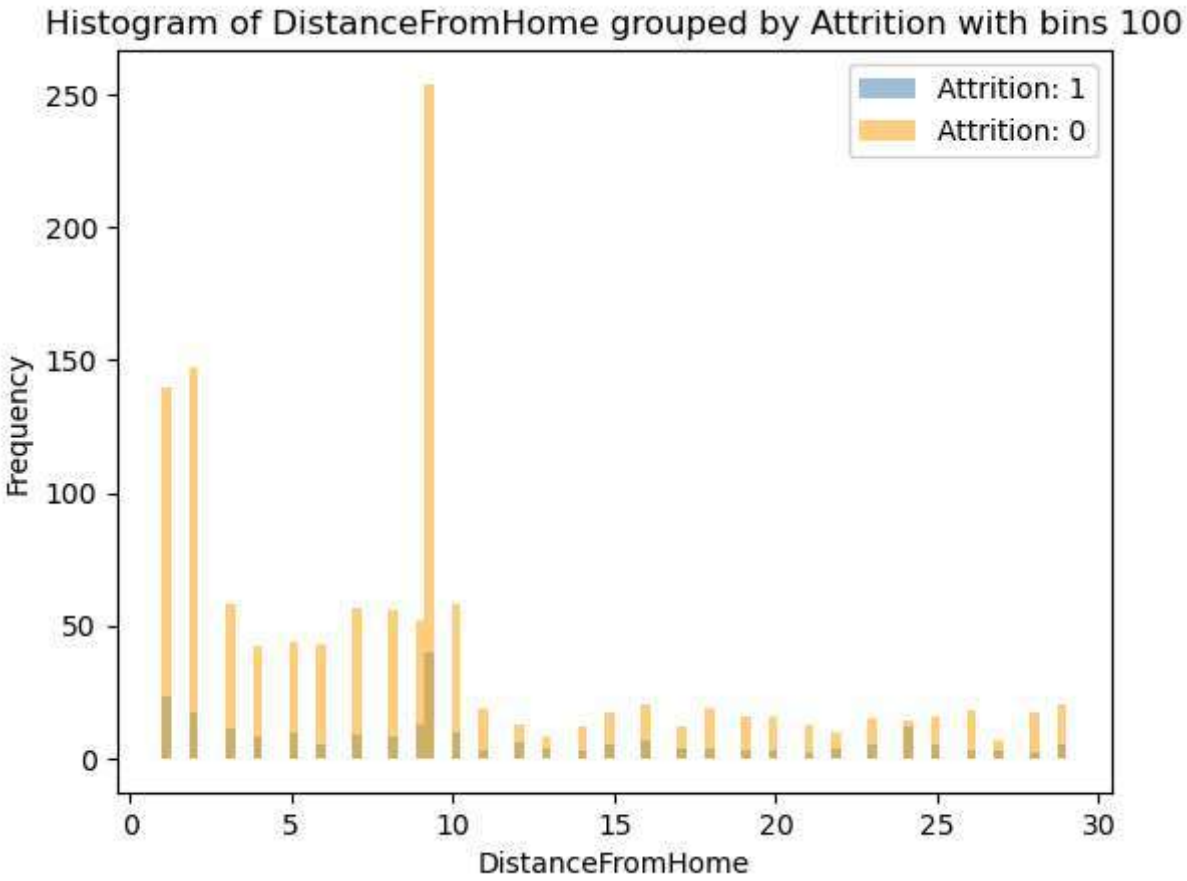

```
Number of free bins: 0
Number of free bins: 0
```




```
Number of free bins: 11
Number of free bins: 11
```



```
Number of free bins: 70
Number of free bins: 70
```

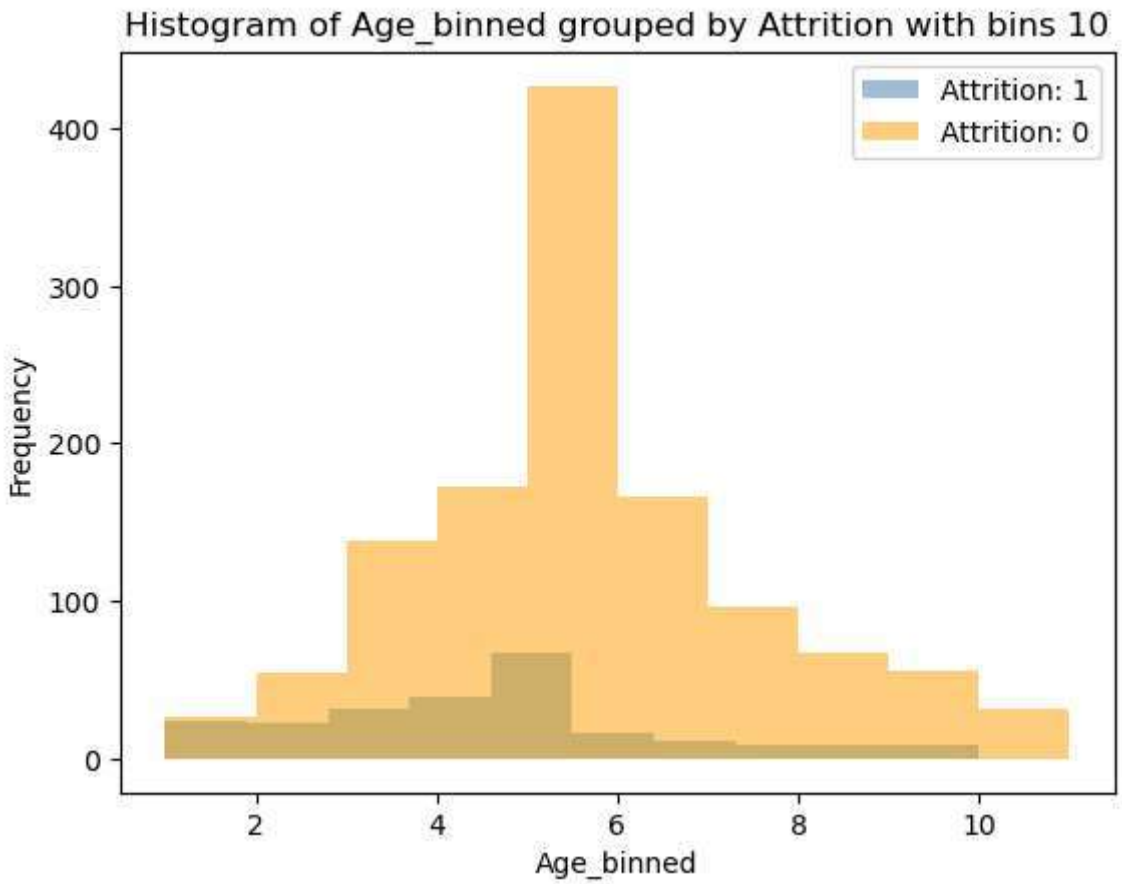


T7. For the rest of the features, which one should be discretized in order to be modeled by histograms? What are the criteria for choosing whether we should discretize a feature or not? Answer this and discretize those features into 10 bins each. In other words, figure out the bin edge for each feature,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

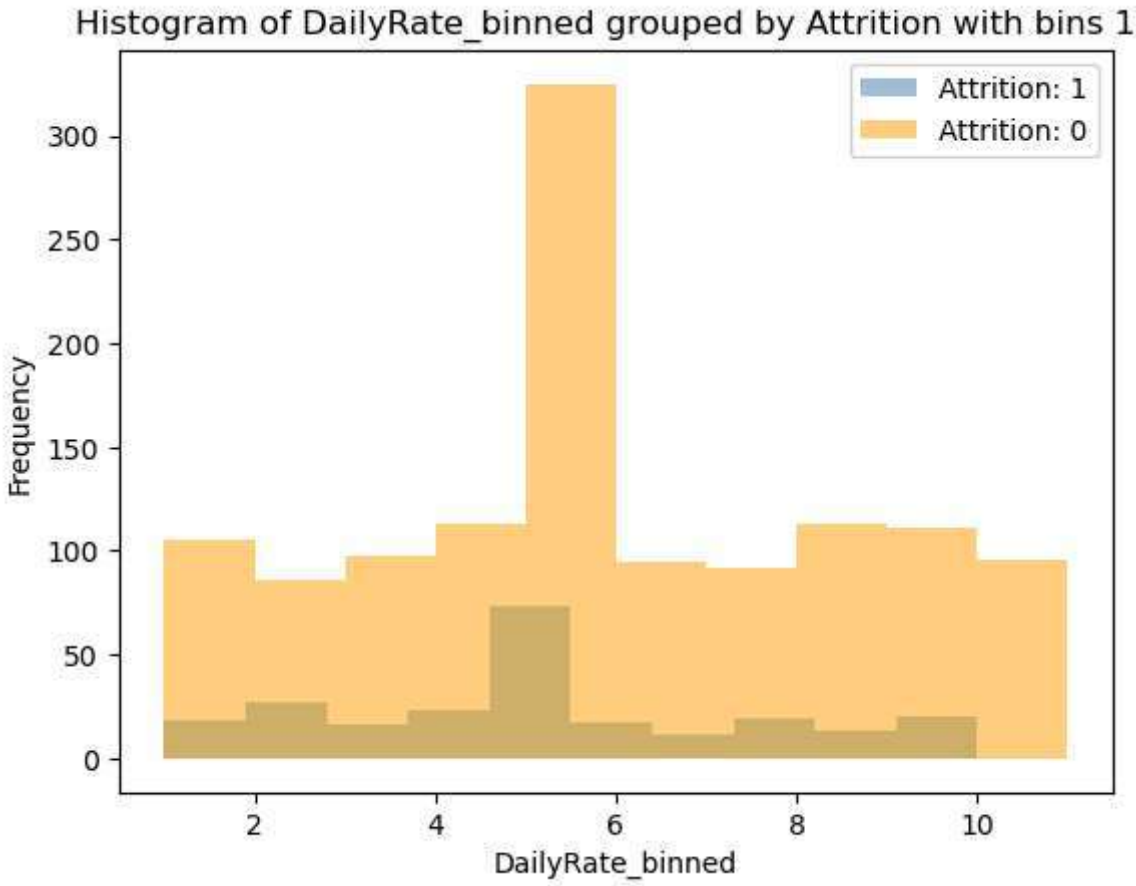
Features with less than 10 unique value should not discretized. but others should.

In [14]:
```
for col in df.columns:
    if col != 'Attrition':
        if df[col].nunique() < 10:
            print(f'{col} is not discretize')
        else:
            print(f'{col} is discretized.')
            new_fea = f'{col}_binned'
            bin_edges = np.linspace(df[col].min(), df[col].max(), 11)
            df[new_fea] = np.digitize(df[col], bin_edges)
            df[new_fea] = np.digitize(df[col], bin_edges)
            display_histogram(df, new_fea, 'Attrition', 10)
```

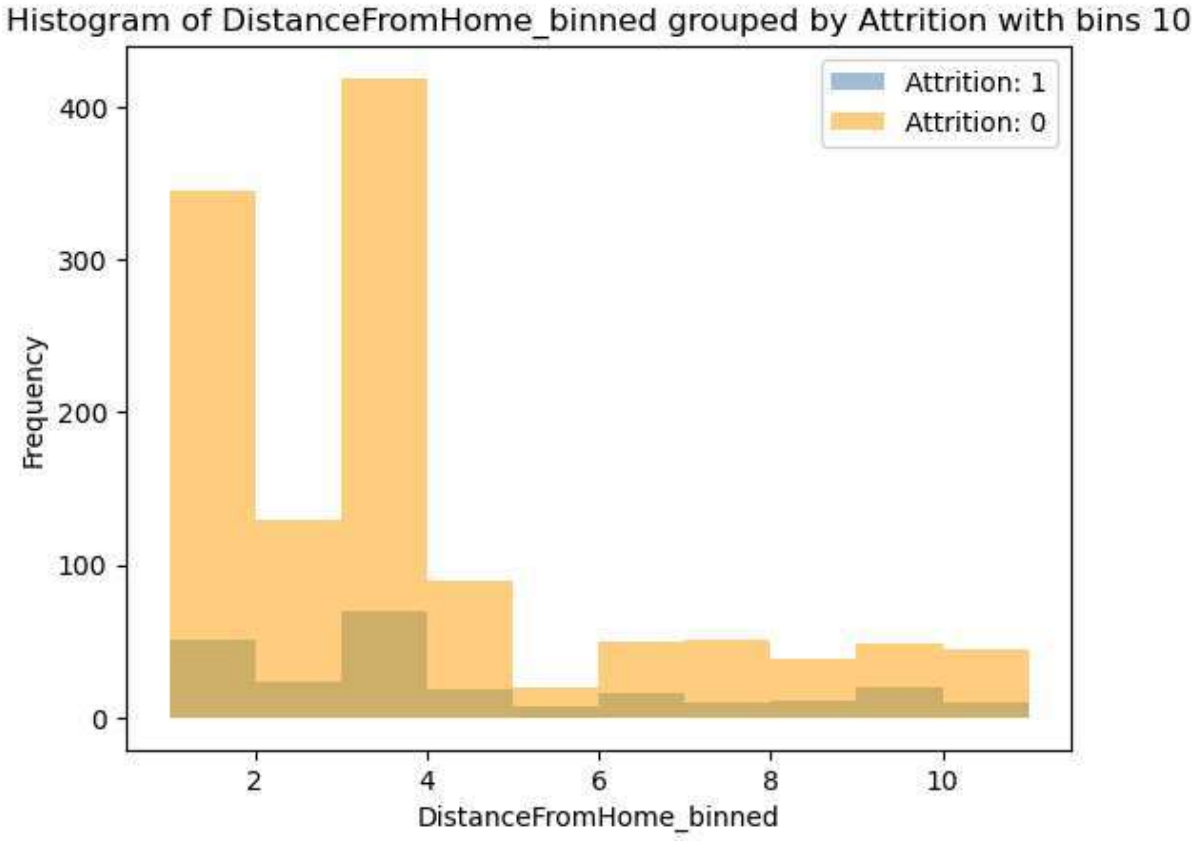
```
Age is discretized.
Number of free bins: 0
Number of free bins: 0
```



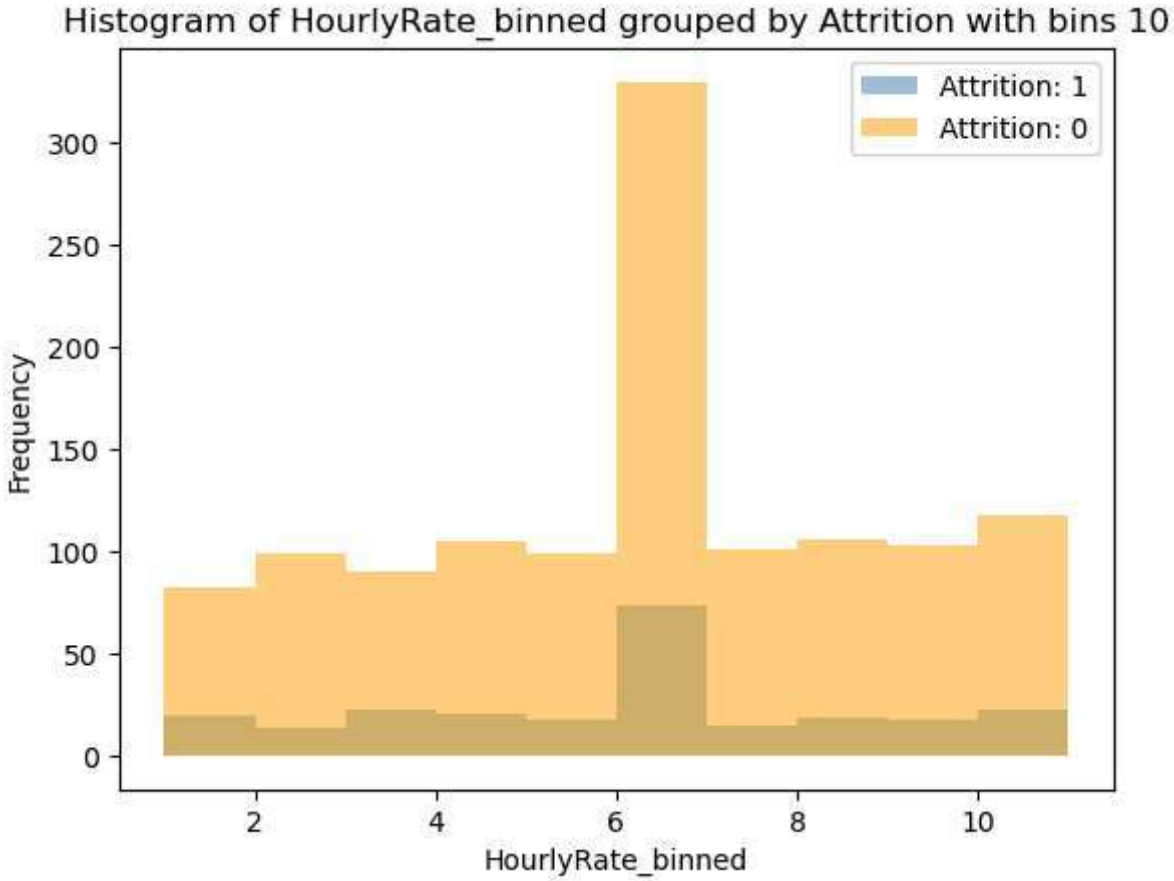

```
BusinessTravel is not discretize
DailyRate is discretized.
Number of free bins: 0
Number of free bins: 0
```



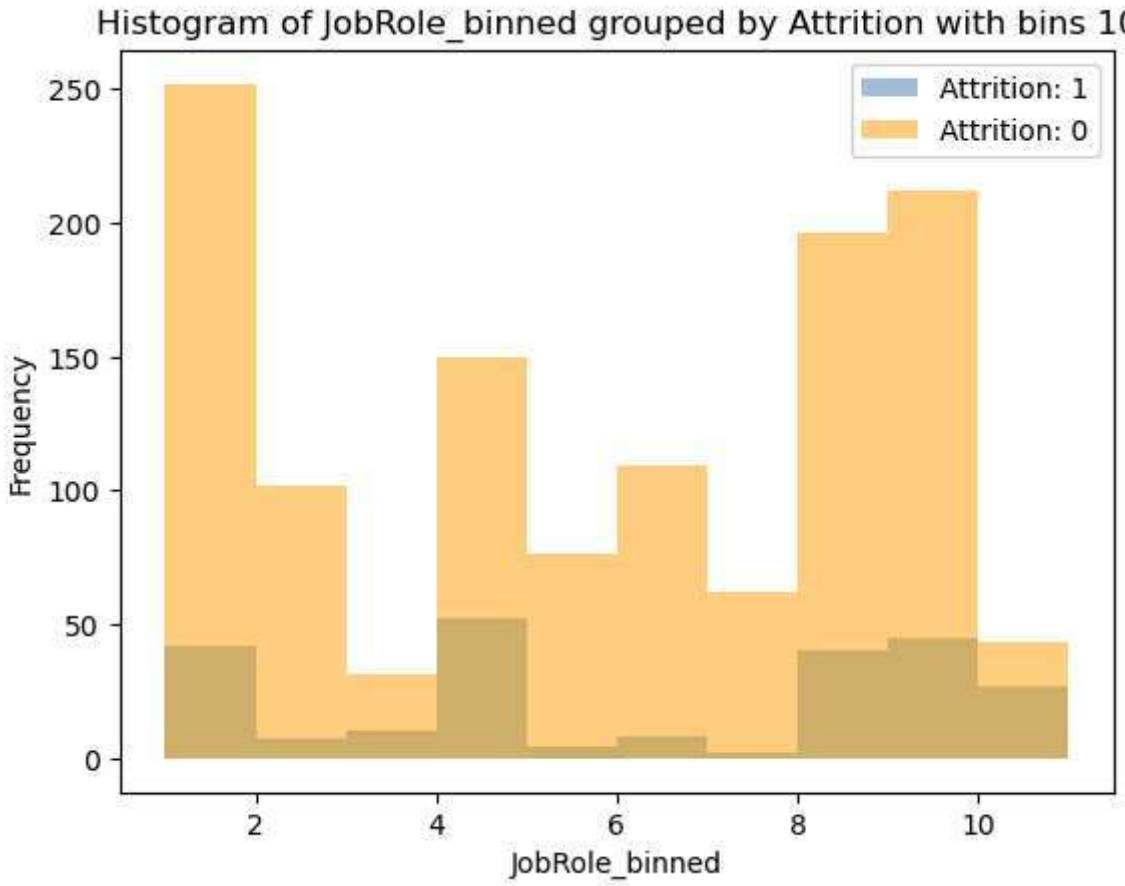

```
Department is not discretize
DistanceFromHome is discretized.
Number of free bins: 0
Number of free bins: 0
```



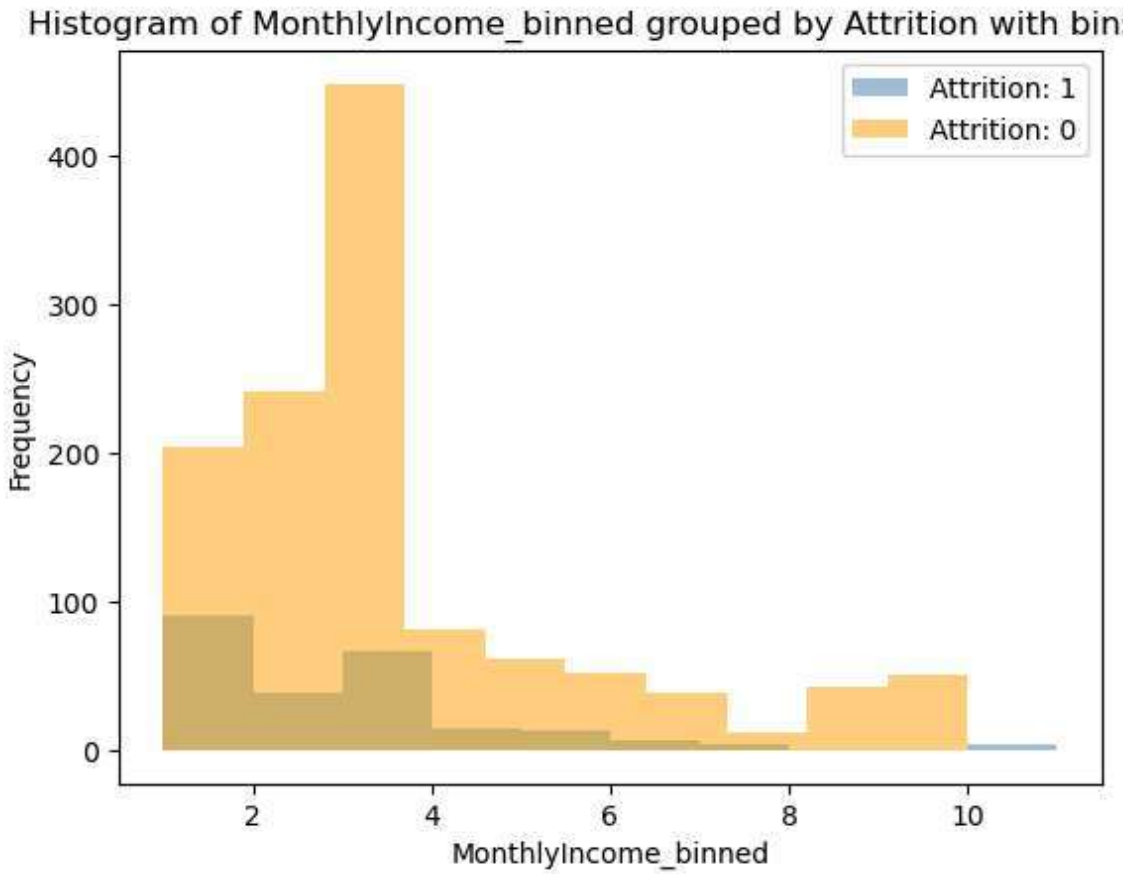
```
Education is not discretize
EducationField is not discretize
EnvironmentSatisfaction is not discretize
Gender is not discretize
HourlyRate is discretized.
Number of free bins: 0
Number of free bins: 0
```



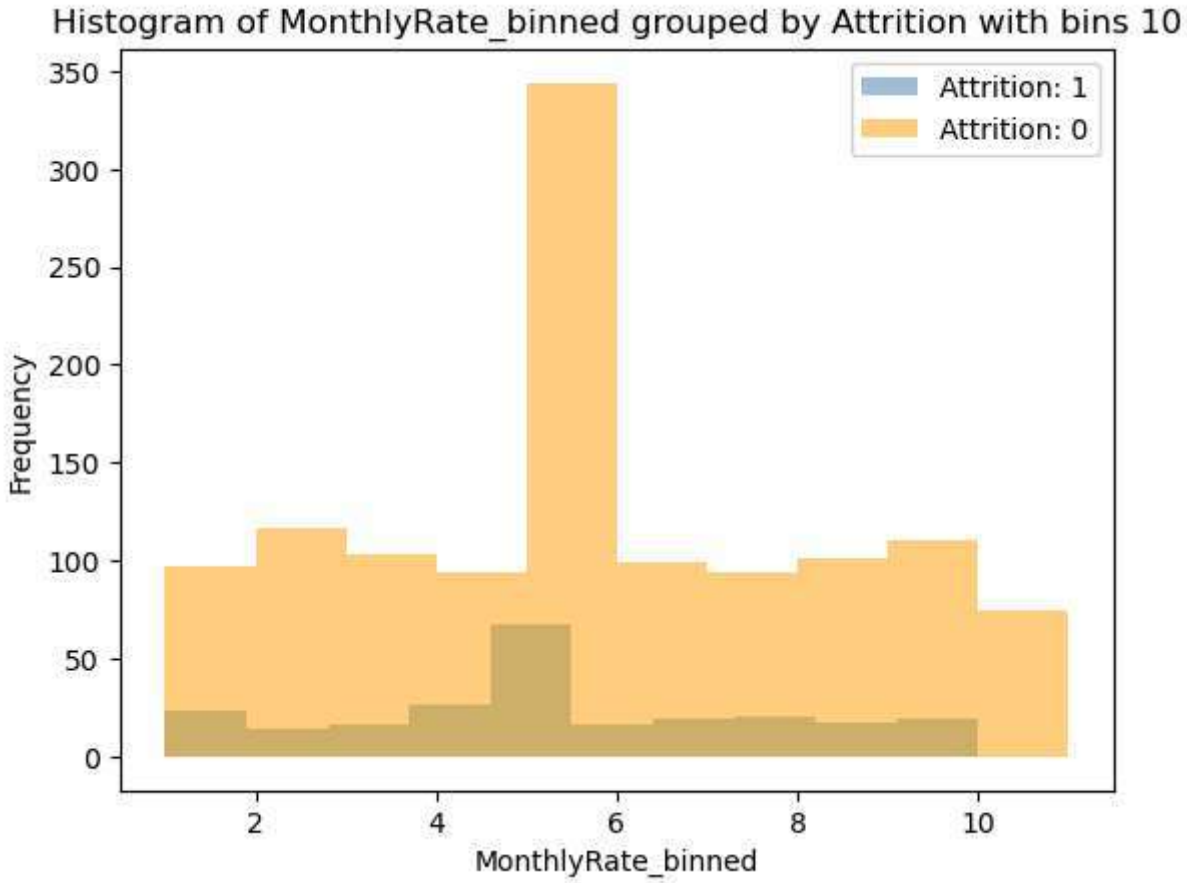

```
JobInvolvement is not discretize
JobLevel is not discretize
JobRole is discretized.
Number of free bins: 0
Number of free bins: 0
```



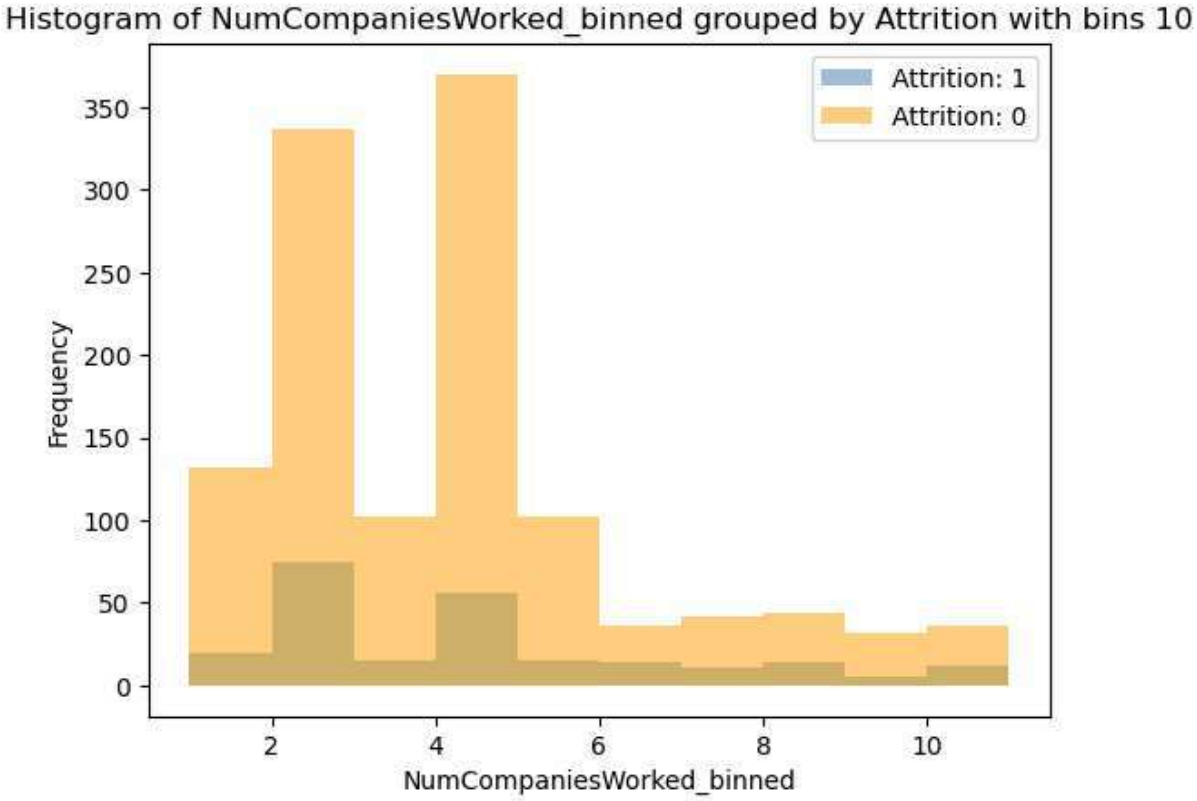

```
JobSatisfaction is not discretize
MaritalStatus is not discretize
MonthlyIncome is discretized.
Number of free bins: 2
Number of free bins: 0
```



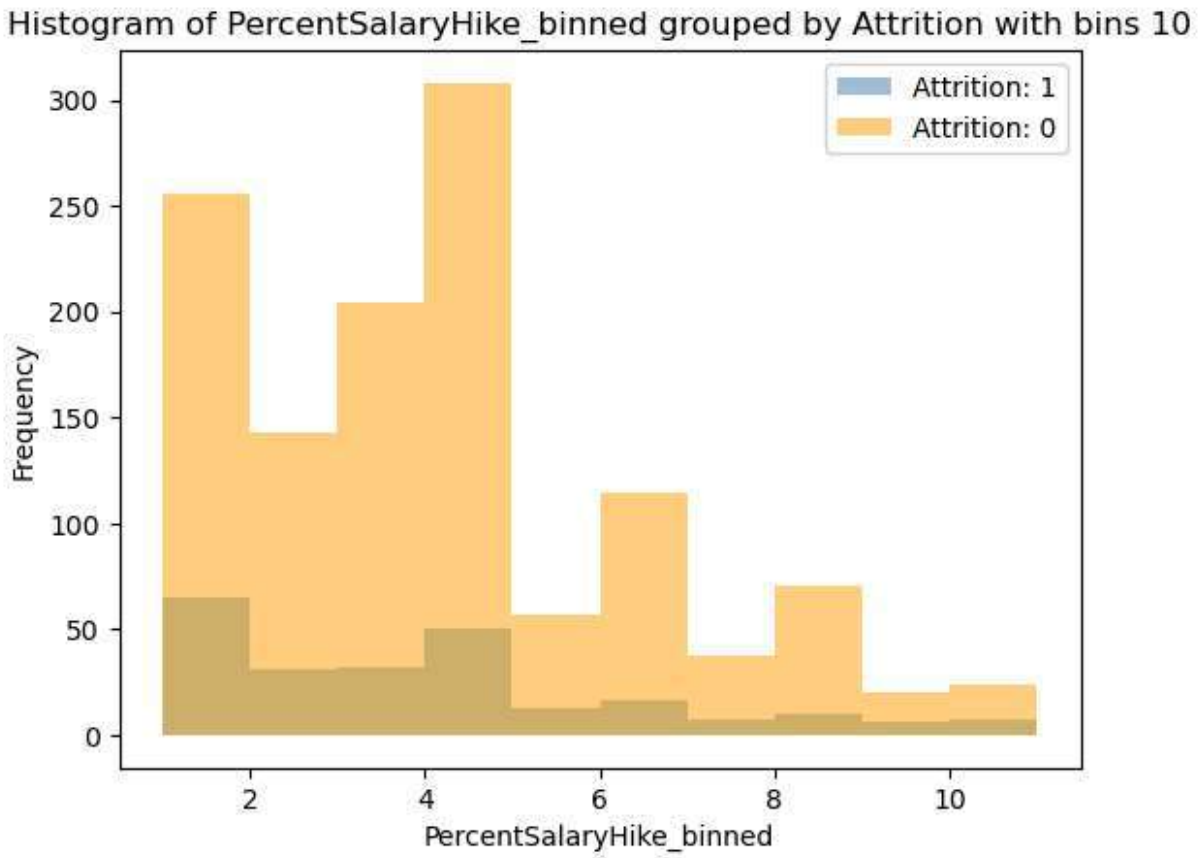

```
MonthlyRate is discretized.
Number of free bins: 0
Number of free bins: 0
```



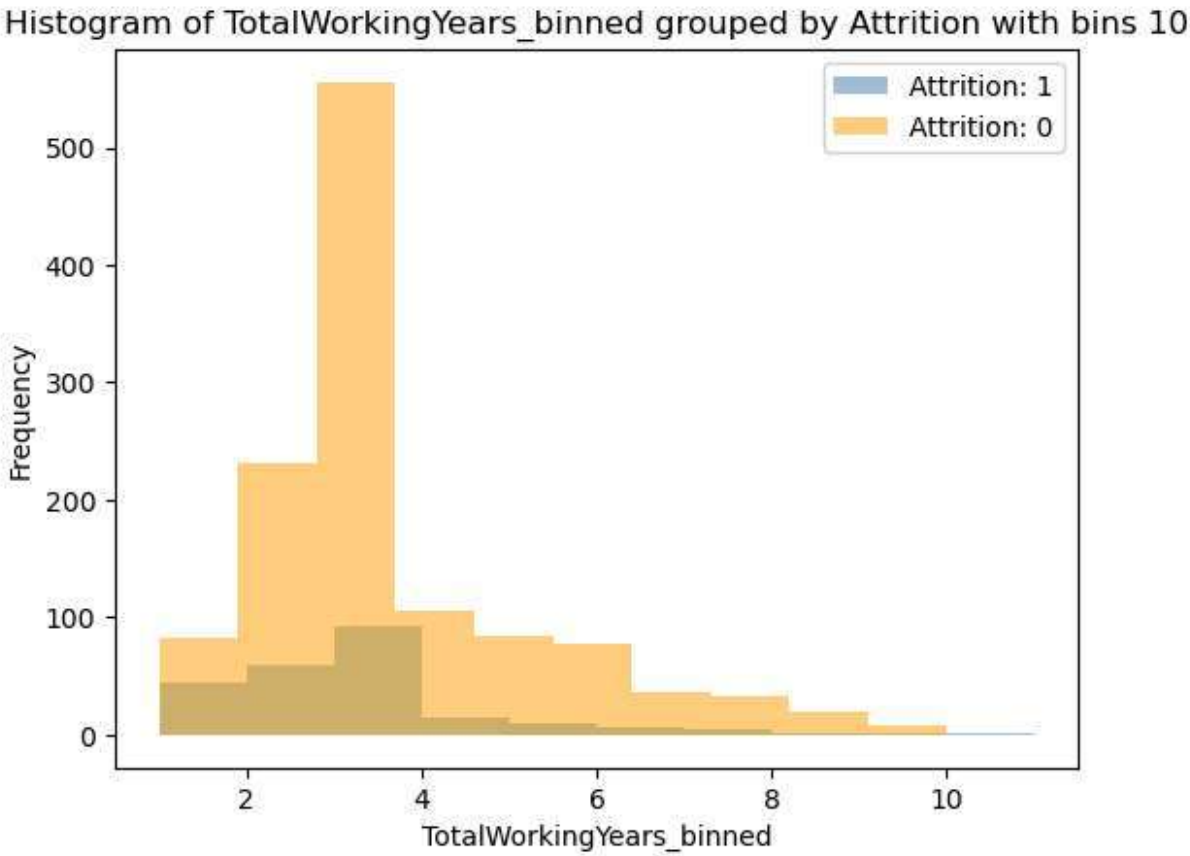
```
NumCompaniesWorked is discretized.
Number of free bins: 0
Number of free bins: 0
```



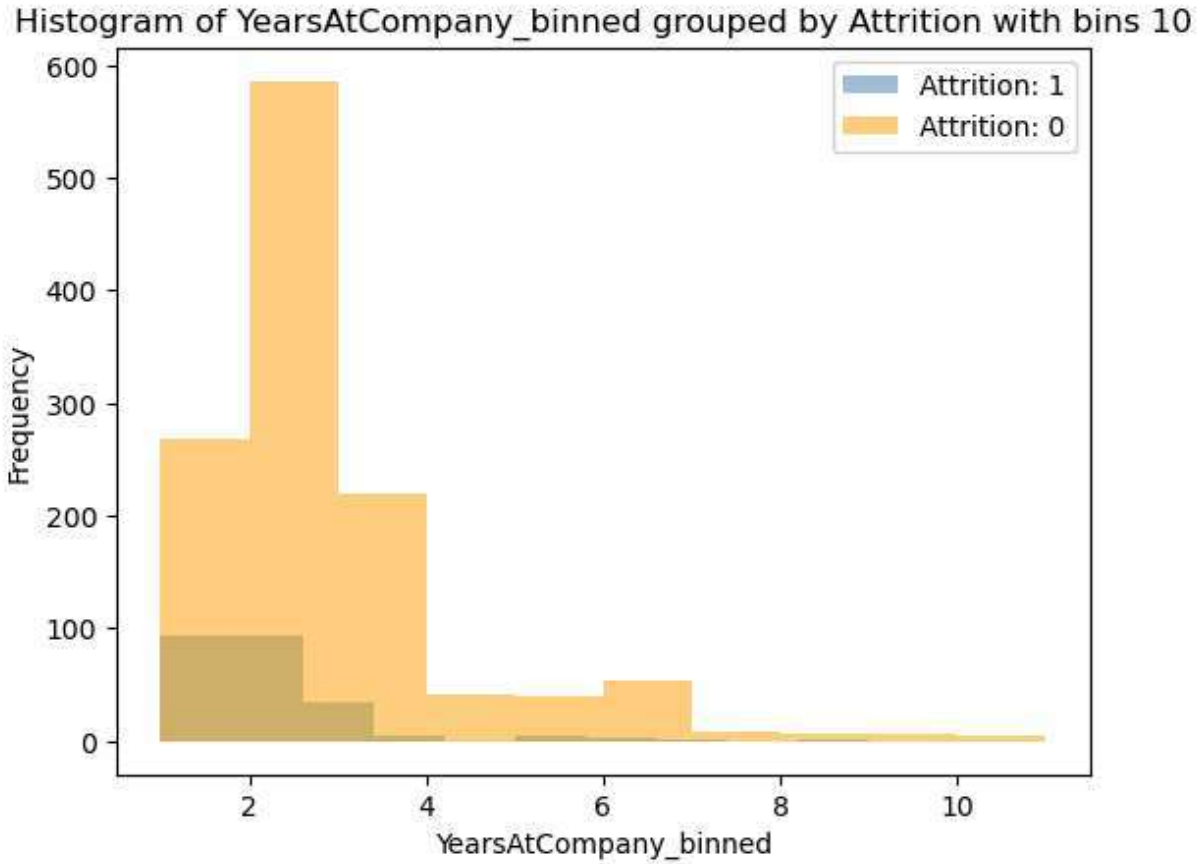

```
OverTime is not discretize
PercentSalaryHike is discretized.
Number of free bins: 0
Number of free bins: 0
```



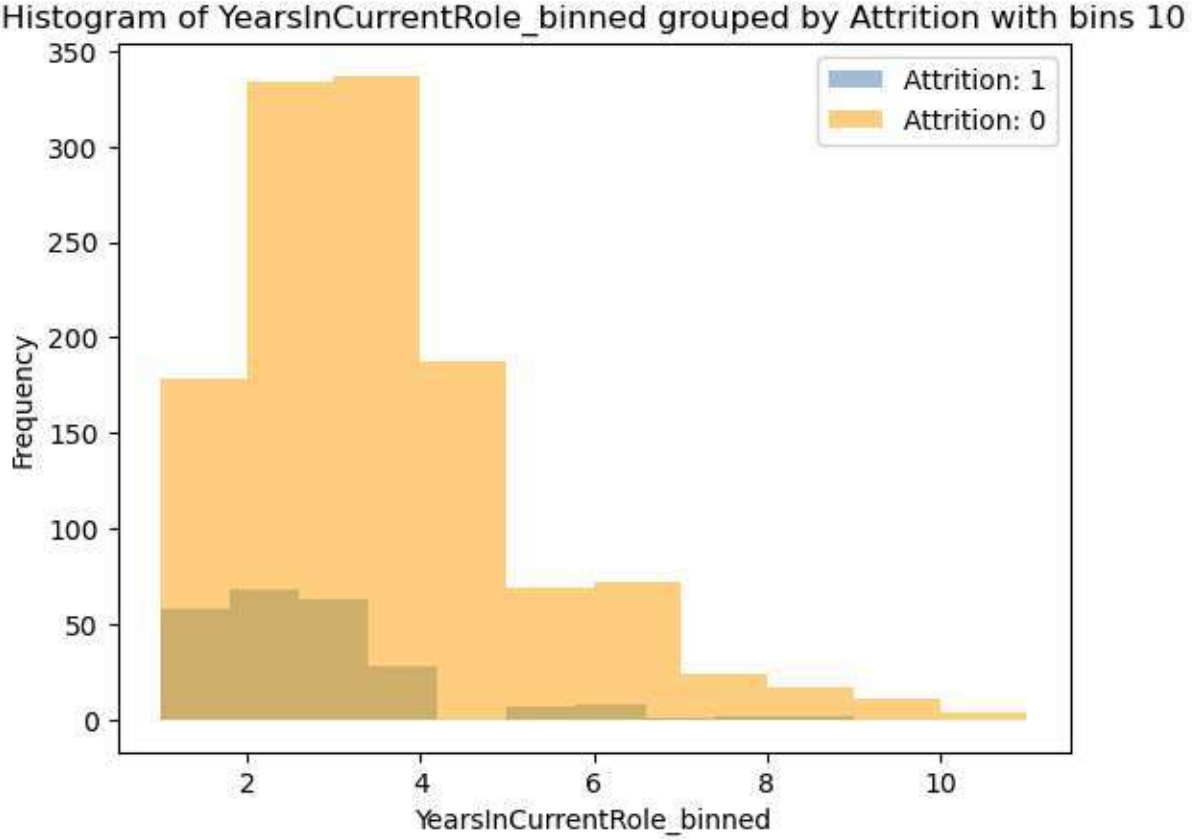

```
PerformanceRating is not discretize
RelationshipSatisfaction is not discretize
StockOptionLevel is not discretize
TotalWorkingYears is discretized.
Number of free bins: 0
Number of free bins: 0
```



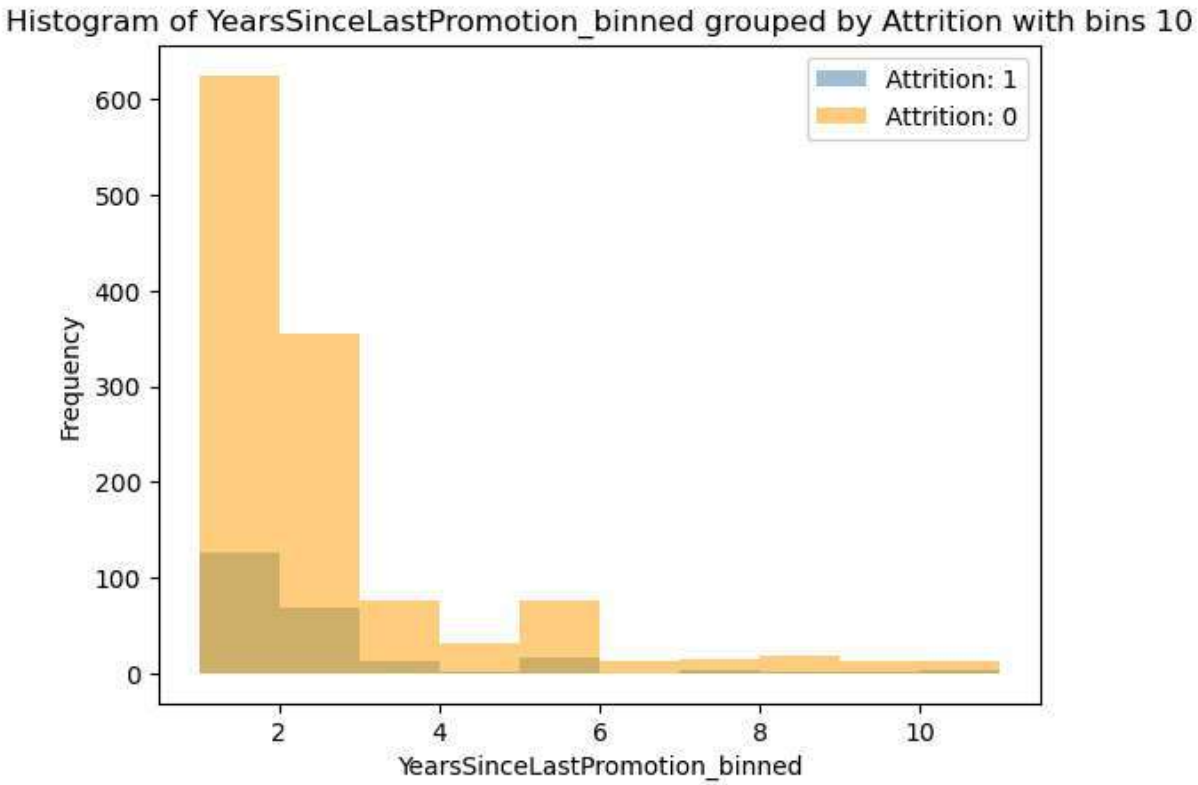

```
TrainingTimesLastYear is not discretize
WorkLifeBalance is not discretize
YearsAtCompany is discretized.
Number of free bins: 2
Number of free bins: 0
```



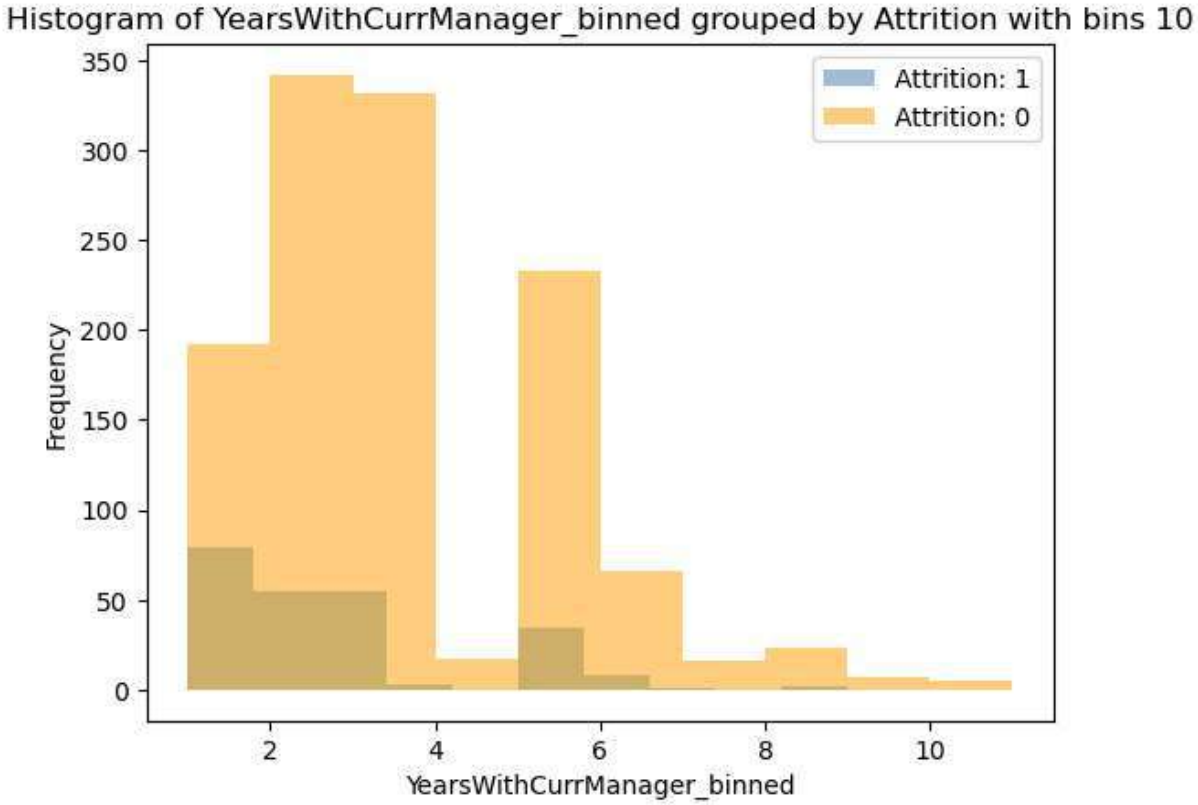

```
YearsInCurrentRole is discretized.
Number of free bins: 1
Number of free bins: 0
```



```
YearsSinceLastPromotion is discretized.
Number of free bins: 1
Number of free bins: 0
```




```
YearsWithCurrManager is discretized.
Number of free bins: 2
Number of free bins: 0
```



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.

1. We should use Categorical Distribution because we digitize data to the discrete bins.
2. MLE for the likelihood distribution.

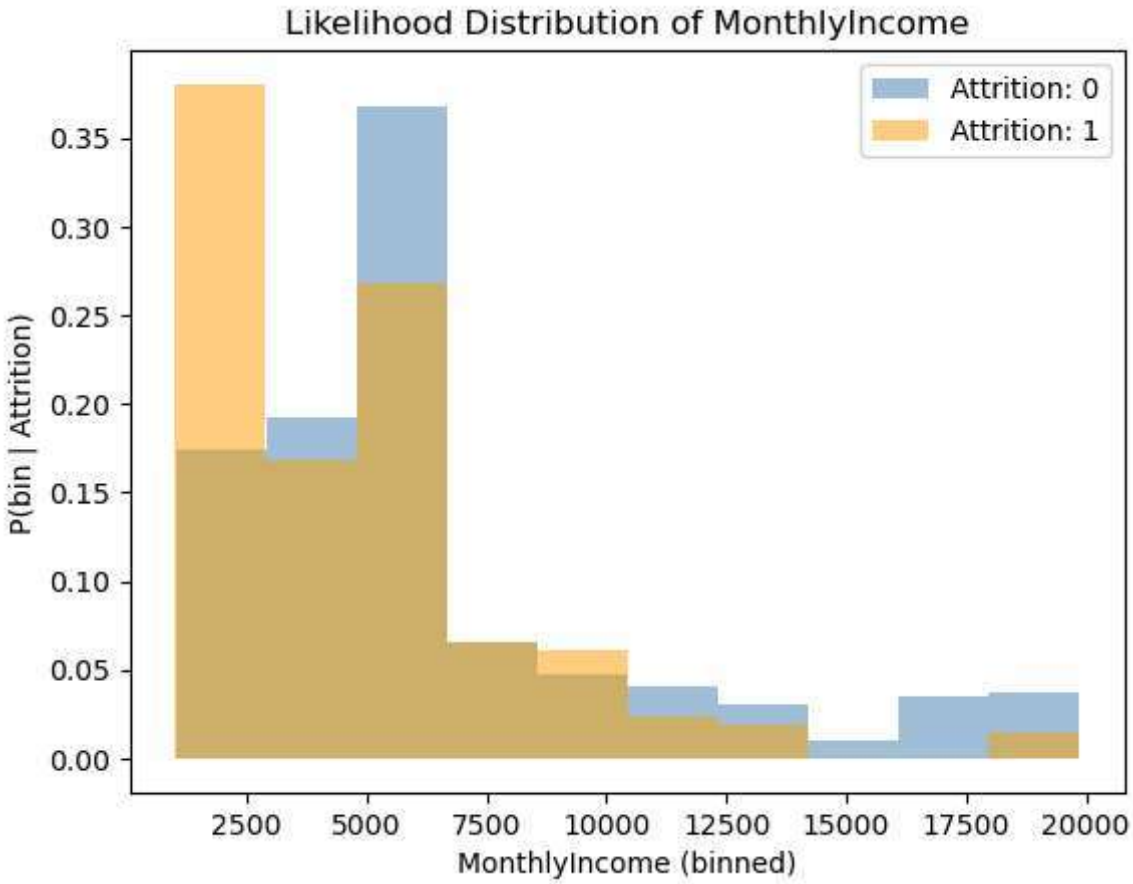
   Finding probability for each bins by

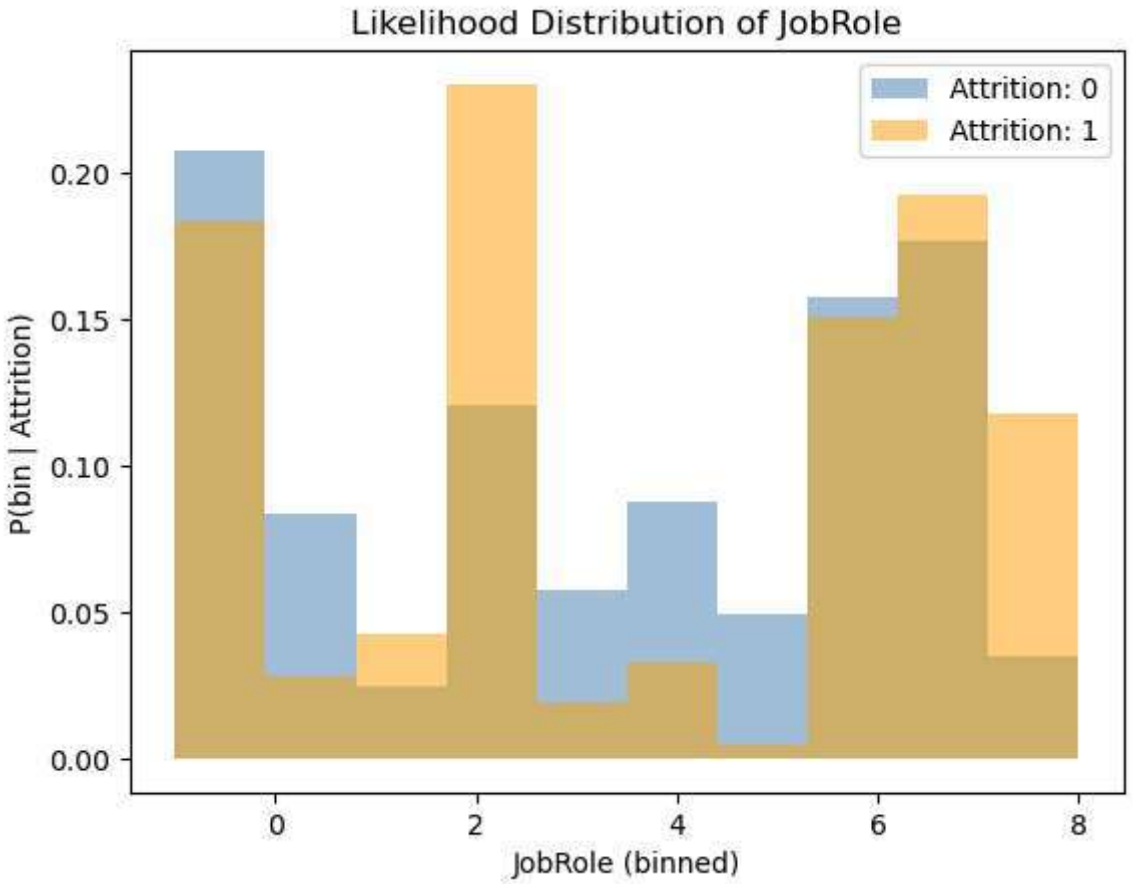
   A. Count data points in each bins.

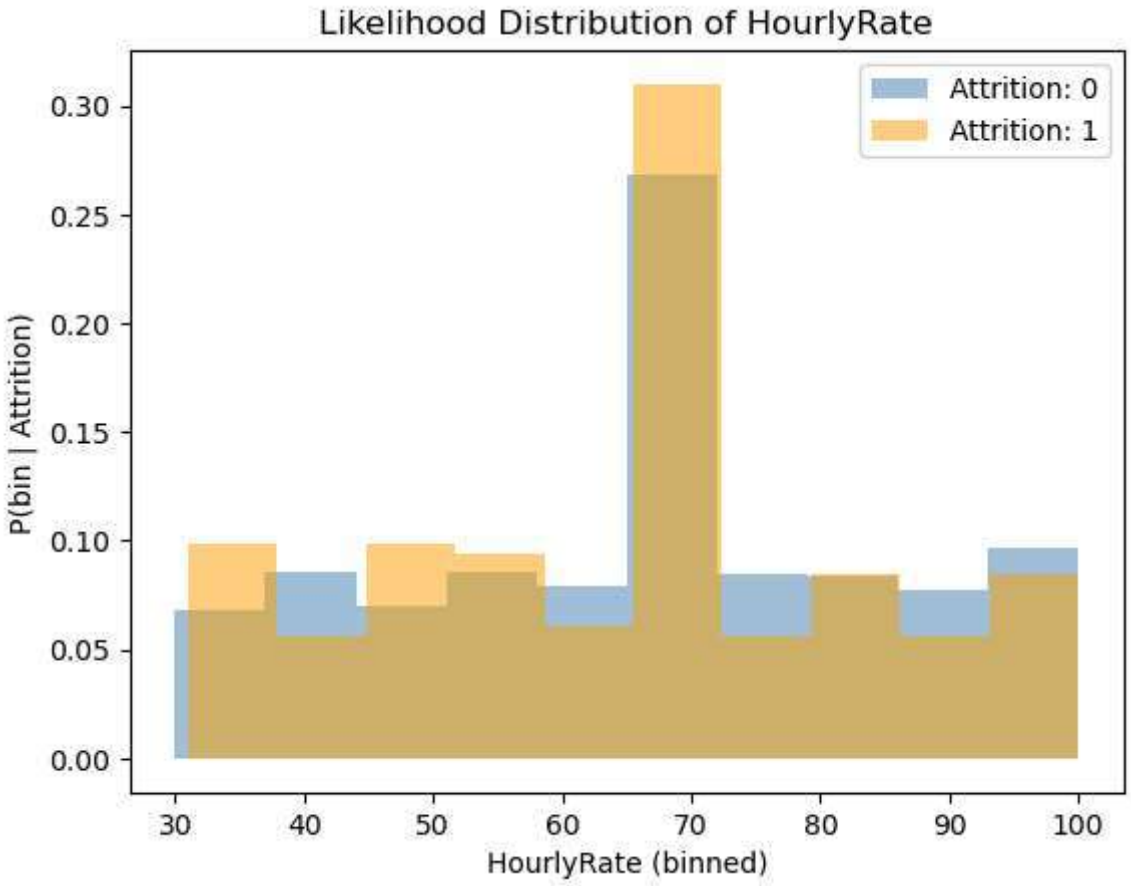
   B. Divided by total data points

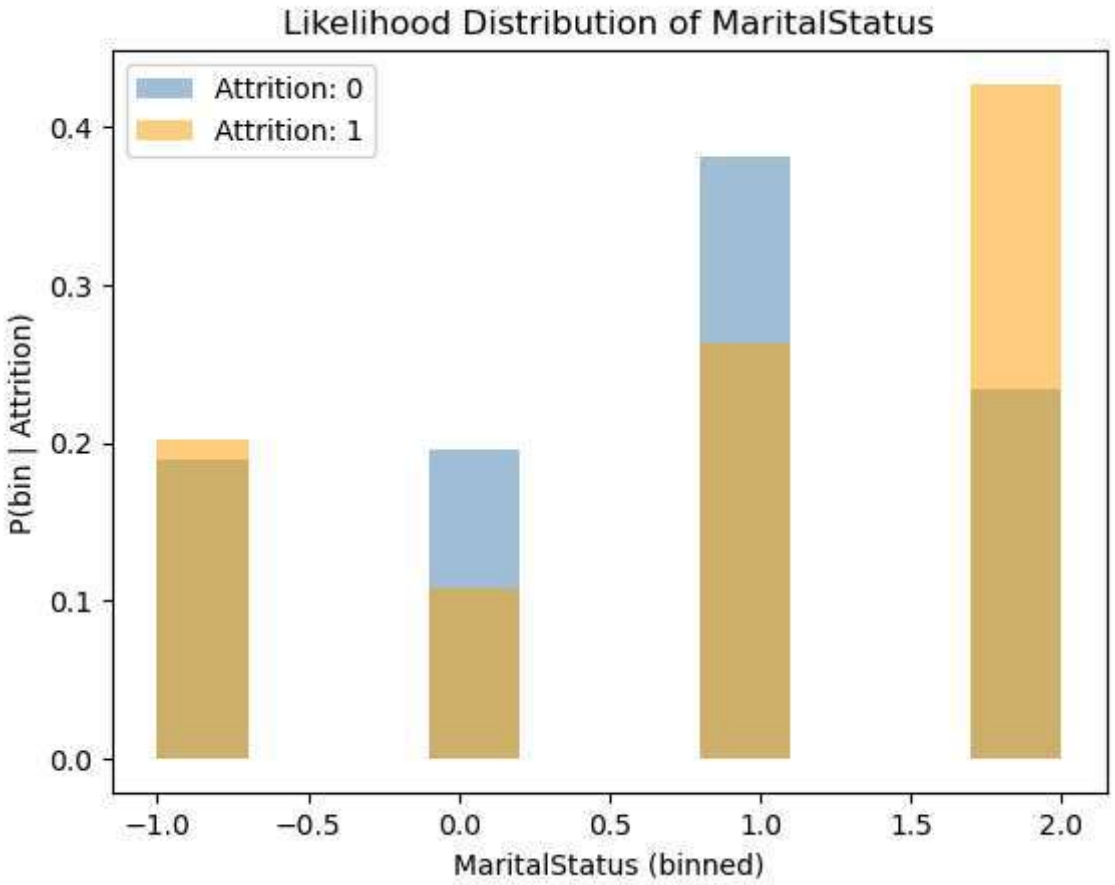
   C. P(bin) = count(bin)/total_count

In [15]:
```
for col in ['MonthlyIncome','JobRole','HourlyRate','MaritalStatus']:
    classes = df_train['Attrition'].unique()
    colors = ['steelblue', 'orange']

    for i, class_val in enumerate(classes):
        data_class = df_train[df_train['Attrition'] == class_val][col]
        data_class = data_class[~np.isnan(data_class)]
        hist, bin_edge = np.histogram(data_class, bins=10)

        total = hist.sum()
        lh = hist/total
        plt.fill_between(bin_edge.repeat(2)[1:-1],
                         lh.repeat(2),
                         facecolor=colors[i],
                         alpha=0.5,
                         label=f'Attrition: {class_val}')

    plt.title(f'Likelihood Distribution of {col}')
    plt.xlabel(f'{col} (binned)')
    plt.ylabel('P(bin | Attrition)')
    plt.legend()
    plt.show()
```


Likelihood Distribution of MonthlyIncome
Attrition: 0
Attrition: 1
P(bin | Attrition)
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
2500
5000
7500
10000
12500
15000
17500
20000
MonthlyIncome (binned)


Likelihood Distribution of JobRole
Attrition: 0
Attrition: 1
P(bin | Attrition)
0.00
0.05
0.10
0.15
0.20
0
2
4
6
8
JobRole (binned)





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

In [16]:
```
A0 = df.loc[df['Attrition'] == 0, 'Attrition'].count()
A1 = df.loc[df['Attrition'] == 1, 'Attrition'].count()

p0 = A0/(A0 + A1)
p1 = A1/(A0 + A1)

print(f'Attrition 0 : {p0}')
print(f'Attrition 1 : {p1}')
```

```
Attrition 0 : 0.8387755102040816
Attrition 1 : 0.16122448979591836
```

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.

add some small value (like 1e-10) to make non zero. Laplace smoothing

Instead of P = count_bin/total

add "a" which is a small value and use P = (count_bin + a)/(total_count + a*bins)

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 [17]:
```
from SimpleBayesClassifier import SimpleBayesClassifier
```

In [18]:
```
data_train = df_train.to_numpy()
data_test = df_test.to_numpy()
```

In [19]:
```
x_train = df_train.drop(columns=['Attrition']).to_numpy()
y_train = df_train['Attrition'].to_numpy()

x_test = df_test.drop(columns=['Attrition']).to_numpy()
y_test = df_test['Attrition'].to_numpy()
```

In [20]:
```
model = SimpleBayesClassifier(n_pos = (y_train==1).sum(), n_neg = (y_train==0).sum(
print((y_train==1).sum())
print((y_train==0).sum())
```

```
213
1110
```

In [21]:
```
def check_prior():
    """
    This function designed to test the implementation of the prior probability calc
    Specifically, it checks if the classifier correctly computes the prior probabil
```

```
    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).prio

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

check_prior()
```

In [22]:
```
model.fit_params(x_train, y_train)
```

```
Out[22]: ([(array([0.01875   , 0.04196429, 0.11428571, 0.14375   , 0.34285714,
                   0.13214286, 0.07589286, 0.05625   , 0.04910714, 0.025     ]),
            array([-inf, 22.2, 26.4, 30.6, 34.8, 39. , 43.2, 47.4, 51.6, 55.8,  inf])),
           (array([0.20714286, 0.00089286, 0.00089286, 0.08928571, 0.00089286,
                   0.00089286, 0.1375    , 0.00089286, 0.00089286, 0.56071429]),
            array([-inf, -0.7, -0.4, -0.1,  0.2,  0.5,  0.8,  1.1,  1.4,  1.7,  inf])),
           (array([0.08035714, 0.06607143, 0.08214286, 0.09196429, 0.26517857,
                   0.07767857, 0.07589286, 0.09285714, 0.09196429, 0.07589286]),
            array([  -inf,  241.7,  381.4,  521.1,  660.8,  800.5,  940.2, 1079.9,
                   1219.6, 1359.3,    inf])),
           (array([0.19285714, 0.00089286, 0.00089286, 0.03571429, 0.00089286,
                   0.00089286, 0.53839286, 0.00089286, 0.00089286, 0.22767857]),
            array([-inf, -0.7, -0.4, -0.1,  0.2,  0.5,  0.8,  1.1,  1.4,  1.7,  inf])),
           (array([0.26607143, 0.10446429, 0.34821429, 0.07232143, 0.01785714,
                   0.04196429, 0.04196429, 0.03214286, 0.03928571, 0.03571429]),
            array([-inf,  3.8,  6.6,  9.4, 12.2, 15. , 17.8, 20.6, 23.4, 26.2,  inf])),
           (array([0.09107143, 0.00089286, 0.15089286, 0.00089286, 0.19464286,
                   0.30267857, 0.00089286, 0.22678571, 0.00089286, 0.03035714]),
            array([-inf,  1.4,  1.8,  2.2,  2.6,  3. ,  3.4,  3.8,  4.2,  4.6,  inf])),
           (array([0.19464286, 0.01428571, 0.00089286, 0.33482143, 0.00089286,
                   0.08035714, 0.25625   , 0.00089286, 0.04642857, 0.07053571]),
            array([-inf, -0.4,  0.2,  0.8,  1.4,  2. ,  2.6,  3.2,  3.8,  4.4,  inf])),
           (array([0.14107143, 0.00089286, 0.00089286, 0.14732143, 0.00089286,
                   0.19196429, 0.26160714, 0.00089286, 0.00089286, 0.25357143]),
            array([-inf,  1.3,  1.6,  1.9,  2.2,  2.5,  2.8,  3.1,  3.4,  3.7,  inf])),
           (array([0.19910714, 0.00089286, 0.00089286, 0.00089286, 0.00089286,
                   0.33660714, 0.00089286, 0.00089286, 0.00089286, 0.45803571]),
            array([-inf, -0.8, -0.6, -0.4, -0.2,  0. ,  0.2,  0.4,  0.6,  0.8,  inf])),
           (array([0.06875   , 0.08571429, 0.07053571, 0.08571429, 0.07946429,
                   0.26696429, 0.08482143, 0.08392857, 0.07767857, 0.09642857]),
            array([-inf,  37.,  44.,  51.,  58.,  65.,  72.,  79.,  86.,  93.,  inf])),
           (array([0.03214286, 0.00089286, 0.00089286, 0.20446429, 0.00089286,
                   0.19553571, 0.48214286, 0.00089286, 0.00089286, 0.08125   ]),
            array([-inf,  1.3,  1.6,  1.9,  2.2,  2.5,  2.8,  3.1,  3.4,  3.7,  inf])),
           (array([0.25803571, 0.00089286, 0.50535714, 0.00089286, 0.00089286,
                   0.12321429, 0.00089286, 0.07232143, 0.00089286, 0.03660714]),
            array([-inf,  1.4,  1.8,  2.2,  2.6,  3. ,  3.4,  3.8,  4.2,  4.6,  inf])),
           (array([0.20625   , 0.08392857, 0.025     , 0.12053571, 0.05803571,
                   0.0875    , 0.05      , 0.15714286, 0.17589286, 0.03571429]),
            array([-inf, -0.1,  0.8,  1.7,  2.6,  3.5,  4.4,  5.3,  6.2,  7.1,  inf])),
           (array([0.15089286, 0.00089286, 0.00089286, 0.14375   , 0.00089286,
                   0.2       , 0.23392857, 0.00089286, 0.00089286, 0.26696429]),
            array([-inf,  1.3,  1.6,  1.9,  2.2,  2.5,  2.8,  3.1,  3.4,  3.7,  inf])),
           (array([0.18839286, 0.00089286, 0.00089286, 0.19464286, 0.00089286,
                   0.00089286, 0.37857143, 0.00089286, 0.00089286, 0.23303571]),
            array([-inf, -0.7, -0.4, -0.1,  0.2,  0.5,  0.8,  1.1,  1.4,  1.7,  inf])),
           (array([0.17321429, 0.19107143, 0.36517857, 0.06607143, 0.04821429,
                   0.04107143, 0.03125   , 0.01071429, 0.03571429, 0.0375    ]),
            array([   -inf,  2930.6,  4810.2,  6689.8,  8569.4, 10449. , 12328.6,
                   14208.2, 16087.8, 17967.4,     inf])),
           (array([0.08214286, 0.09196429, 0.08214286, 0.075     , 0.27767857,
                   0.07857143, 0.07767857, 0.08125   , 0.09017857, 0.06339286]),
            array([   -inf,  4584.3,  7074.6,  9564.9, 12055.2, 14545.5, 17035.8,
                   19526.1, 22016.4, 24506.7,     inf])),
           (array([0.10982143, 0.26785714, 0.08303571, 0.29375   , 0.0875    ,
                   0.03035714, 0.03482143, 0.03839286, 0.02767857, 0.02678571]),
```

```
 array([-inf,  0.9,  1.8,  2.7,  3.6,  4.5,  5.4,  6.3,  7.2,  8.1,  inf])),
(array([0.19464286, 0.00089286, 0.00089286, 0.00089286, 0.00089286,
        0.60625   , 0.00089286, 0.00089286, 0.00089286, 0.19285714]),
 array([-inf, -0.8, -0.6, -0.4, -0.2,  0. ,  0.2,  0.4,  0.6,  0.8,  inf])),
(array([0.20982143, 0.11517857, 0.16339286, 0.24642857, 0.04553571,
        0.09196429, 0.03125   , 0.05714286, 0.01875   , 0.02053571]),
 array([-inf, 12.4, 13.8, 15.2, 16.6, 18. , 19.4, 20.8, 22.2, 23.6,  inf])),
(array([0.675     , 0.19821429, 0.00089286, 0.00089286, 0.00089286,
        0.00089286, 0.00089286, 0.00089286, 0.00089286, 0.12053571]),
 array([-inf,  3.1,  3.2,  3.3,  3.4,  3.5,  3.6,  3.7,  3.8,  3.9,  inf])),
(array([0.14732143, 0.00089286, 0.00089286, 0.16696429, 0.00089286,
        0.19285714, 0.23928571, 0.00089286, 0.00089286, 0.24910714]),
 array([-inf,  1.3,  1.6,  1.9,  2.2,  2.5,  2.8,  3.1,  3.4,  3.7,  inf])),
(array([0.32142857, 0.00089286, 0.20267857, 0.35178571, 0.00089286,
        0.00089286, 0.07946429, 0.00089286, 0.00089286, 0.04017857]),
 array([-inf,  0.3,  0.6,  0.9,  1.2,  1.5,  1.8,  2.1,  2.4,  2.7,  inf])),
(array([0.0625    , 0.18928571, 0.25446429, 0.25982143, 0.07767857,
        0.06071429, 0.03125   , 0.02767857, 0.02410714, 0.0125    ]),
 array([-inf,  3.7,  7.4, 11.1, 14.8, 18.5, 22.2, 25.9, 29.6, 33.3,  inf])),
(array([0.02946429, 0.04464286, 0.00089286, 0.28660714, 0.20267857,
        0.26964286, 0.06339286, 0.00089286, 0.06964286, 0.03214286]),
 array([-inf,  0.6,  1.2,  1.8,  2.4,  3. ,  3.6,  4.2,  4.8,  5.4,  inf])),
(array([0.03392857, 0.00089286, 0.00089286, 0.17946429, 0.00089286,
        0.19464286, 0.50178571, 0.00089286, 0.00089286, 0.08571429]),
 array([-inf,  1.3,  1.6,  1.9,  2.2,  2.5,  2.8,  3.1,  3.4,  3.7,  inf])),
(array([0.21517857, 0.47410714, 0.17946429, 0.03303571, 0.03125   ,
        0.04285714, 0.00714286, 0.00625   , 0.00625   , 0.00446429]),
 array([-inf,  3.7,  7.4, 11.1, 14.8, 18.5, 22.2, 25.9, 29.6, 33.3,  inf])),
(array([0.14196429, 0.27142857, 0.26964286, 0.15267857, 0.05803571,
        0.05892857, 0.01964286, 0.01428571, 0.00982143, 0.00357143]),
 array([-inf,  1.8,  3.6,  5.4,  7.2,  9. , 10.8, 12.6, 14.4, 16.2,  inf])),
(array([0.50089286, 0.29196429, 0.06160714, 0.02410714, 0.06160714,
        0.01071429, 0.01071429, 0.01517857, 0.01071429, 0.0125    ]),
 array([-inf,  1.5,  3. ,  4.5,  6. ,  7.5,  9. , 10.5, 12. , 13.5,  inf])),
(array([0.15446429, 0.27857143, 0.26339286, 0.01339286, 0.18928571,
        0.05446429, 0.01428571, 0.01964286, 0.00714286, 0.00535714]),
 array([-inf,  1.7,  3.4,  5.1,  6.8,  8.5, 10.2, 11.9, 13.6, 15.3,  inf]))],
[(array([0.0941704 , 0.08071749, 0.13452915, 0.17488789, 0.28699552,
        0.06726457, 0.04035874, 0.03139013, 0.04484305, 0.04484305]),
 array([-inf,  22.,  26.,  30.,  34.,  38.,  42.,  46.,  50.,  54.,  inf])),
(array([0.1793722 , 0.0044843 , 0.0044843 , 0.03587444, 0.0044843 ,
        0.0044843 , 0.23766816, 0.0044843 , 0.0044843 , 0.52017937]),
 array([-inf, -0.7, -0.4, -0.1,  0.2,  0.5,  0.8,  1.1,  1.4,  1.7,  inf])),
(array([0.08071749, 0.11659193, 0.07174888, 0.10313901, 0.30044843,
        0.06278027, 0.04484305, 0.08071749, 0.06278027, 0.07623318]),
 array([  -inf,  242.3,  381.6,  520.9,  660.2,  799.5,  938.8, 1078.1,
        1217.4, 1356.7,    inf])),
(array([0.21973094, 0.0044843 , 0.0044843 , 0.04484305, 0.0044843 ,
        0.0044843 , 0.40358744, 0.0044843 , 0.0044843 , 0.30493274]),
 array([-inf, -0.7, -0.4, -0.1,  0.2,  0.5,  0.8,  1.1,  1.4,  1.7,  inf])),
(array([0.19282511, 0.0896861 , 0.30493274, 0.08071749, 0.03139013,
        0.07623318, 0.03587444, 0.05381166, 0.08520179, 0.04932735]),
 array([-inf,  3.8,  6.6,  9.4, 12.2, 15. , 17.8, 20.6, 23.4, 26.2,  inf])),
(array([0.11210762, 0.0044843 , 0.14349776, 0.0044843 , 0.17040359,
        0.34529148, 0.0044843 , 0.18834081, 0.0044843 , 0.02242152]),
 array([-inf,  1.4,  1.8,  2.2,  2.6,  3. ,  3.4,  3.8,  4.2,  4.6,  inf])),
```

```
(array([0.19282511, 0.03139013, 0.0044843 , 0.29596413, 0.0044843 ,
        0.12107623, 0.20627803, 0.0044843 , 0.04484305, 0.0941704 ]),
 array([-inf, -0.4,  0.2,  0.8,  1.4,  2. ,  2.6,  3.2,  3.8,  4.4,  inf])),
(array([0.21524664, 0.0044843 , 0.0044843 , 0.13452915, 0.0044843 ,
        0.23318386, 0.1838565 , 0.0044843 , 0.0044843 , 0.21076233]),
 array([-inf,  1.3,  1.6,  1.9,  2.2,  2.5,  2.8,  3.1,  3.4,  3.7,  inf])),
(array([0.1793722 , 0.0044843 , 0.0044843 , 0.0044843 , 0.0044843 ,
        0.29147982, 0.0044843 , 0.0044843 , 0.0044843 , 0.49775785]),
 array([-inf, -0.8, -0.6, -0.4, -0.2,  0. ,  0.2,  0.4,  0.6,  0.8,  inf])),
(array([0.09865471, 0.05829596, 0.09865471, 0.0941704 , 0.06278027,
        0.30044843, 0.05829596, 0.08520179, 0.05829596, 0.08520179]),
 array([-inf, 37.9, 44.8, 51.7, 58.6, 65.5, 72.4, 79.3, 86.2, 93.1,  inf])),
(array([0.0941704 , 0.0044843 , 0.0044843 , 0.23318386, 0.0044843 ,
        0.1838565 , 0.41704036, 0.0044843 , 0.0044843 , 0.04932735]),
 array([-inf,  1.3,  1.6,  1.9,  2.2,  2.5,  2.8,  3.1,  3.4,  3.7,  inf])),
(array([0.47982063, 0.0044843 , 0.34977578, 0.0044843 , 0.0044843 ,
        0.10313901, 0.0044843 , 0.02242152, 0.0044843 , 0.02242152]),
 array([-inf,  1.4,  1.8,  2.2,  2.6,  3. ,  3.4,  3.8,  4.2,  4.6,  inf])),
(array([0.1793722 , 0.03139013, 0.04484305, 0.22421525, 0.02242152,
        0.03587444, 0.00896861, 0.14798206, 0.18834081, 0.11659193]),
 array([-inf, -0.1,  0.8,  1.7,  2.6,  3.5,  4.4,  5.3,  6.2,  7.1,  inf])),
(array([0.17040359, 0.0044843 , 0.0044843 , 0.16143498, 0.0044843 ,
        0.18834081, 0.26008969, 0.0044843 , 0.0044843 , 0.19730942]),
 array([-inf,  1.3,  1.6,  1.9,  2.2,  2.5,  2.8,  3.1,  3.4,  3.7,  inf])),
(array([0.19730942, 0.0044843 , 0.0044843 , 0.10762332, 0.0044843 ,
        0.0044843 , 0.25560538, 0.0044843 , 0.0044843 , 0.41255605]),
 array([-inf, -0.7, -0.4, -0.1,  0.2,  0.5,  0.8,  1.1,  1.4,  1.7,  inf])),
(array([0.367713  , 0.16591928, 0.26008969, 0.06726457, 0.06278027,
        0.02690583, 0.02242152, 0.0044843 , 0.0044843 , 0.01793722]),
 array([  -inf,  2894.,  4779.,  6664.,  8549., 10434., 12319., 14204.,
        16089., 17974.,    inf])),
(array([0.10313901, 0.05381166, 0.07174888, 0.11210762, 0.28251121,
        0.06726457, 0.08071749, 0.07174888, 0.08520179, 0.07174888]),
 array([   -inf,  4884.3,  7321.6,  9758.9, 12196.2, 14633.5, 17070.8,
        19508.1, 21945.4, 24382.7,     inf])),
(array([0.07174888, 0.32735426, 0.06278027, 0.23318386, 0.06278027,
        0.06726457, 0.04035874, 0.06726457, 0.02690583, 0.04035874]),
 array([-inf,  0.9,  1.8,  2.7,  3.6,  4.5,  5.4,  6.3,  7.2,  8.1,  inf])),
(array([0.21973094, 0.0044843 , 0.0044843 , 0.0044843 , 0.0044843 ,
        0.35426009, 0.0044843 , 0.0044843 , 0.0044843 , 0.39461883]),
 array([-inf, -0.8, -0.6, -0.4, -0.2,  0. ,  0.2,  0.4,  0.6,  0.8,  inf])),
(array([0.26457399, 0.14349776, 0.12556054, 0.19282511, 0.05381166,
        0.07623318, 0.03587444, 0.04932735, 0.03139013, 0.02690583]),
 array([-inf, 12.4, 13.8, 15.2, 16.6, 18. , 19.4, 20.8, 22.2, 23.6,  inf])),
(array([0.6367713 , 0.20627803, 0.0044843 , 0.0044843 , 0.0044843 ,
        0.0044843 , 0.0044843 , 0.0044843 , 0.0044843 , 0.12556054]),
 array([-inf,  3.1,  3.2,  3.3,  3.4,  3.5,  3.6,  3.7,  3.8,  3.9,  inf])),
(array([0.18834081, 0.0044843 , 0.0044843 , 0.15246637, 0.0044843 ,
        0.21973094, 0.21973094, 0.0044843 , 0.0044843 , 0.19730942]),
 array([-inf,  1.3,  1.6,  1.9,  2.2,  2.5,  2.8,  3.1,  3.4,  3.7,  inf])),
(array([0.52466368, 0.0044843 , 0.17488789, 0.1838565 , 0.0044843 ,
        0.0044843 , 0.04932735, 0.0044843 , 0.0044843 , 0.04484305]),
 array([-inf,  0.3,  0.6,  0.9,  1.2,  1.5,  1.8,  2.1,  2.4,  2.7,  inf])),
(array([0.20627803, 0.21524664, 0.39013453, 0.06278027, 0.04484305,
        0.03139013, 0.01793722, 0.01345291, 0.0044843 , 0.01345291]),
 array([-inf,   4.,   8.,  12.,  16.,  20.,  24.,  28.,  32.,  36.,  inf])),
```

```
 (array([0.04484305, 0.03587444, 0.0044843 , 0.33632287, 0.1838565 ,
         0.22869955, 0.0896861 , 0.0044843 , 0.05381166, 0.01793722]),
  array([-inf,  0.6,  1.2,  1.8,  2.4,  3. ,  3.6,  4.2,  4.8,  5.4,  inf])),
 (array([0.08520179, 0.0044843 , 0.0044843 , 0.19282511, 0.0044843 ,
         0.20179372, 0.39461883, 0.0044843 , 0.0044843 , 0.10313901]),
  array([-inf,  1.3,  1.6,  1.9,  2.2,  2.5,  2.8,  3.1,  3.4,  3.7,  inf])),
 (array([0.39910314, 0.16591928, 0.2735426 , 0.08071749, 0.02690583,
         0.01793722, 0.00896861, 0.01345291, 0.0044843 , 0.00896861]),
  array([-inf,  3.1,  6.2,  9.3, 12.4, 15.5, 18.6, 21.7, 24.8, 27.9,  inf])),
 (array([0.25112108, 0.21524664, 0.31390135, 0.00896861, 0.10762332,
         0.03587444, 0.03587444, 0.0044843 , 0.01345291, 0.01345291]),
  array([-inf,  1.5,  3. ,  4.5,  6. ,  7.5,  9. , 10.5, 12. , 13.5,  inf])),
 (array([0.52466368, 0.28251121, 0.04932735, 0.01345291, 0.06726457,
         0.0044843 , 0.01793722, 0.01345291, 0.01345291, 0.01345291]),
  array([-inf,  1.5,  3. ,  4.5,  6. ,  7.5,  9. , 10.5, 12. , 13.5,  inf])),
 (array([0.33632287, 0.15695067, 0.28251121, 0.0044843 , 0.01793722,
         0.14349776, 0.01793722, 0.02242152, 0.0044843 , 0.01345291]),
  array([-inf,  1.4,  2.8,  4.2,  5.6,  7. ,  8.4,  9.8, 11.2, 12.6,  inf]))])
```

In [23]:

```python
def check_fit_params():

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

    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 : [0.16666667 0.08333333 0.08333333 0.08333333 0.08333333 0.08333333
 0.08333333 0.08333333 0.08333333 0.16666667]
EDGES : [-inf  0.2  0.4  0.6  0.8  1.   1.2  1.4  1.6  1.8  inf]
Feature : 1
BINS : [0.16666667 0.08333333 0.08333333 0.08333333 0.08333333 0.08333333
 0.08333333 0.08333333 0.08333333 0.16666667]
EDGES : [-inf  1.2  1.4  1.6  1.8  2.   2.2  2.4  2.6  2.8  inf]
Feature : 2
BINS : [0.16666667 0.08333333 0.08333333 0.08333333 0.08333333 0.08333333
 0.08333333 0.08333333 0.08333333 0.16666667]
EDGES : [-inf  2.2  2.4  2.6  2.8  3.   3.2  3.4  3.6  3.8  inf]
Feature : 3
BINS : [0.16666667 0.08333333 0.08333333 0.08333333 0.08333333 0.08333333
 0.08333333 0.08333333 0.08333333 0.16666667]
EDGES : [-inf  3.2  3.4  3.6  3.8  4.   4.2  4.4  4.6  4.8  inf]

LEAVE PARAMETERS
Feature : 0
BINS : [0.16666667 0.08333333 0.08333333 0.08333333 0.08333333 0.08333333
 0.08333333 0.08333333 0.08333333 0.16666667]
EDGES : [-inf  1.2  1.4  1.6  1.8  2.   2.2  2.4  2.6  2.8  inf]
Feature : 1
BINS : [0.16666667 0.08333333 0.08333333 0.08333333 0.08333333 0.08333333
 0.08333333 0.08333333 0.08333333 0.16666667]
EDGES : [-inf  2.2  2.4  2.6  2.8  3.   3.2  3.4  3.6  3.8  inf]
Feature : 2
BINS : [0.16666667 0.08333333 0.08333333 0.08333333 0.08333333 0.08333333
 0.08333333 0.08333333 0.08333333 0.16666667]
EDGES : [-inf  3.2  3.4  3.6  3.8  4.   4.2  4.4  4.6  4.8  inf]
Feature : 3
BINS : [0.16666667 0.08333333 0.08333333 0.08333333 0.08333333 0.08333333
 0.08333333 0.08333333 0.08333333 0.16666667]
EDGES : [-inf  4.2  4.4  4.6  4.8  5.   5.2  5.4  5.6  5.8  inf]
```

In [65]:
```
y_pred = model.predict(x = x_test)
# print(y_test)
# print(y_pred)
```

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

In [70]:
```
def evaluate(y_true, y_pred, show_result = True):
    y_true = np.array(y_true).flatten()
    y_pred = np.array(y_pred).flatten()
    tp = np.sum((y_true == 1) & (y_pred == 1))
    tn = np.sum((y_true == 0) & (y_pred == 0))
```

```
    fp = np.sum((y_true == 0) & (y_pred == 1))
    fn = np.sum((y_true == 1) & (y_pred == 0))
    accuracy = (tp + tn)/(tp + tn + fp + fn + 1e-6)
    precision = tp/(tp + fp + 1e-6)
    recall = tp/(tp + fn + 1e-6)
    F1 = 2 * (precision * recall) / (precision + recall + 1e-6)
    fpr = fp/(tn + fp + 1e-6)
    if show_result:
        print(f'Accuracy: {accuracy}')
        print(f'Precision: {precision}')
        print(f'Recall: {recall}')
        print(f'F1: {F1}')
        print(f'fpr: {fpr}')
    return accuracy, precision, recall, F1, fpr
```

In [71]:
```
evaluate(y_test, y_pred)
```

```
Accuracy: 0.7959183619325282
Precision: 0.3333333148148158
Recall: 0.24999998958333375
F1: 0.2857137823137645
fpr: 0.09756097481657744
```

Out[71]:
```
(np.float64(0.7959183619325282),
 np.float64(0.3333333148148158),
 np.float64(0.24999998958333375),
 np.float64(0.2857137823137645),
 np.float64(0.09756097481657744))
```

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

In [72]:
```
model.fit_gaussian_params(x_train, y_train)
```

```
Out[72]: ([(np.float64(37.72092756021327), np.float64(7.88847682185423)),
           (np.float64(1.0594594594595), np.float64(1.2177647799009772)),
           (np.float64(808.949826867684), np.float64(361.25682681677)),
           (np.float64(0.8063063063063063), np.float64(1.000610559489571)),
           (np.float64(9.182545045045044), np.float64(7.164075868510379)),
           (np.float64(2.9385939510939507), np.float64(0.9271537464212607)),
           (np.float64(1.60990990990991), np.float64(1.7527561284838886)),
           (np.float64(2.7730112765827055), np.float64(0.9646770895907198)),
           (np.float64(0.26126126126126126), np.float64(0.7700674091437947)),
           (np.float64(65.67123552123552), np.float64(18.208229403188422)),
           (np.float64(2.759338420052706), np.float64(0.6075736122426656)),
           (np.float64(2.130216185573328), np.float64(0.9897227607048844)),
           (np.float64(3.291891891891892), np.float64(3.100047518820198)),
           (np.float64(2.7712255622969906), np.float64(0.989101685012613)),
           (np.float64(0.6603603603603604), np.float64(1.034954782428654)),
           (np.float64(6695.87816924067), np.float64(4221.853711213365)),
           (np.float64(14227.850730832874), np.float64(6407.70575233789)),
           (np.float64(2.687942023656309), np.float64(2.1914792033876367)),
           (np.float64(-0.0018018018018018018), np.float64(0.6238476919084147)),
           (np.float64(15.259137709137708), np.float64(3.2583758124930244)),
           (np.float64(3.1506871667585954), np.float64(0.32019424480110464)),
           (np.float64(2.7282581050438193), np.float64(0.976327748376568)),
           (np.float64(0.7847536311822025), np.float64(0.7262211190233104)),
           (np.float64(11.76081540724398), np.float64(6.99020008787738)),
           (np.float64(2.790050254335968), np.float64(1.1461985376736115)),
           (np.float64(2.793854568854569), np.float64(0.6057294770117945)),
           (np.float64(7.350251271679843), np.float64(5.51564707048728)),
           (np.float64(4.455428387571244), np.float64(3.246474563457173)),
           (np.float64(2.191220812649384), np.float64(2.8419599665904283)),
           (np.float64(4.302445302445302), np.float64(3.199460786156519))],
          [(np.float64(34.1485580147552), np.float64(9.01691537197167)),
           (np.float64(1.1408450704225352), np.float64(1.1337226227726505)),
           (np.float64(755.1428172207851), np.float64(355.93725009016947)),
           (np.float64(0.8215962441314554), np.float64(1.1074240661703678)),
           (np.float64(11.143779342723004), np.float64(7.8991831551993705)),
           (np.float64(2.845417744562613), np.float64(0.9170825756610679)),
           (np.float64(1.624413145539906), np.float64(1.8101864241960348)),
           (np.float64(2.5747700488646164), np.float64(1.0348346661517676)),
           (np.float64(0.3333333333333333), np.float64(0.7670850052769992)),
           (np.float64(64.67588866532527), np.float64(18.219577287618044)),
           (np.float64(2.5687777458401198), np.float64(0.6998837472476498)),
           (np.float64(1.7110679952732264), np.float64(0.8731681599741417)),
           (np.float64(3.7183098591549295), np.float64(3.2379053331190564)),
           (np.float64(2.6418550988470506), np.float64(0.9720481415504919)),
           (np.float64(0.9154929577464789), np.float64(1.1556736410009518)),
           (np.float64(5007.719651240778), np.float64(3255.6174754024587)),
           (np.float64(14308.666475040725), np.float64(6221.96642633736)),
           (np.float64(2.915764427836863), np.float64(2.3885094334245345)),
           (np.float64(0.18309859154929578), np.float64(0.7747759177169052)),
           (np.float64(15.041582830315226), np.float64(3.479720647876926)),
           (np.float64(3.1585584938200633), np.float64(0.32619883570862074)),
           (np.float64(2.593178116316949), np.float64(0.9910889570008495)),
           (np.float64(0.542708632748874), np.float64(0.7499145455643926)),
           (np.float64(8.933653508351698), np.float64(6.452654312527432)),
           (np.float64(2.654913608635942), np.float64(1.0821702076840374)),
           (np.float64(2.6896617802050398), np.float64(0.7436913276152447)),
```

```
(np.float64(5.32152837660886), np.float64(4.562600106098341)),
(np.float64(3.364640222286097), np.float64(2.9351285919645957)),
(np.float64(1.9100356104883265), np.float64(2.7333476474646305)),
(np.float64(3.055308038708441), np.float64(2.9560760612783836))])
```

In [73]:

```
def check_fit_gaussian_params():

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

    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_C

    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 [74]:
```
y_pred = model.gaussian_predict(x_test)
```

In [75]:
```
evaluate(y_test, y_pred)
```

```
Accuracy: 0.7278911515109445
Precision: 0.289473676592798
Recall: 0.4583333142361119
F1: 0.35483822372591944
fpr: 0.21951219333729924
```

Out[75]:
```
(np.float64(0.7278911515109445),
 np.float64(0.289473676592798),
 np.float64(0.4583333142361119),
 np.float64(0.35483822372591944),
 np.float64(0.21951219333729924))
```

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 [85]:
```
import random as rnd

y_pred_rnd = np.array([rnd.randint(0,1) for _ in range(len(y_test))])

evaluate(y_test, y_pred_rnd)
```

```
Accuracy: 0.4217687046138183
Precision: 0.14942528563878982
Recall: 0.5416666440972231
F1: 0.23423389108075413
fpr: 0.6016260113688943
```

Out[85]:
```
(np.float64(0.4217687046138183),
 np.float64(0.14942528563878982),
 np.float64(0.5416666440972231),
 np.float64(0.23423389108075413),
 np.float64(0.6016260113688943))
```

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 [84]:
```
n_stay = np.sum(y_train == 0)
n_leave = np.sum(y_train == 1)

majority_class = 0 if n_stay > n_leave else 1
y_pred_majority = np.full(len(y_test), majority_class)

evaluate(y_test, y_pred_majority)
```

```
Accuracy: 0.8367346881854784
Precision: 0.0
Recall: 0.0
F1: 0.0
fpr: 0.0
```

Out[84]:
```
(np.float64(0.8367346881854784),
 np.float64(0.0),
 np.float64(0.0),
 np.float64(0.0),
 np.float64(0.0))
```

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

From T11, I got the

- Accuracy: 0.7959183619325282
- Precision: 0.3333333148148158
- Recall: 0.24999998958333375
- F1: 0.2857137823137645
- fpr: 0.09756097481657744

From T12, I got the

- Accuracy: 0.7278911515109445
- Precision: 0.289473676592798

- Recall: 0.4583333142361119
- F1: 0.35483822372591944
- fpr: 0.21951219333729924

Compared with the rnd and majority, I can conclude the data that the different between leave and stay is to high made the accuracy high while other scores are low. While I have this data, I think the T11 is better than the T12.

T16. Use the following threshold values

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

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

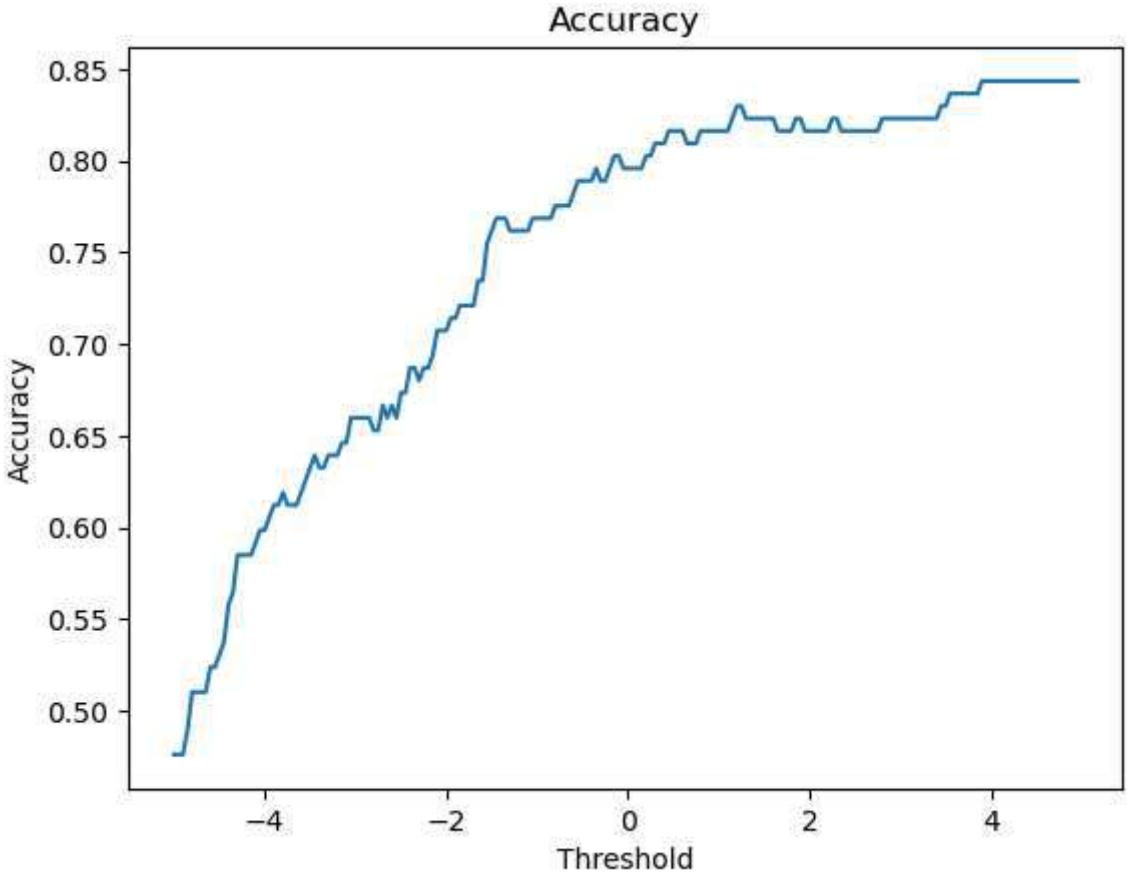
In [96]:
```
best_acc = float('-inf')
best_acc_t = -5
best_f1 = float('-inf')
best_f1_t = -5

hist_acc = []
hist_pre = []
hist_rec = []
hist_F1 = []
hist_fpr = []
for t in np.arange(-5,5,0.05):
    y_pred_t = model.predict(x = x_test, thresh = t)
    accuracy, precision, recall, F1, fpr = evaluate(y_test, y_pred_t, show_result=F
    hist_acc.append(accuracy)
    hist_pre.append(precision)
    hist_rec.append(recall)
    hist_F1.append(F1)
    hist_fpr.append(fpr)
    if accuracy > best_acc:
        best_acc = accuracy
        best_acc_t = t
    if F1 > best_f1:
        best_f1 = F1
        best_f1_t = t

print(f'Best Accuracy is {best_acc} with threshold: {best_acc_t}')
print(f'Best F1 is {best_f1} with threshold: {best_f1_t}')
```

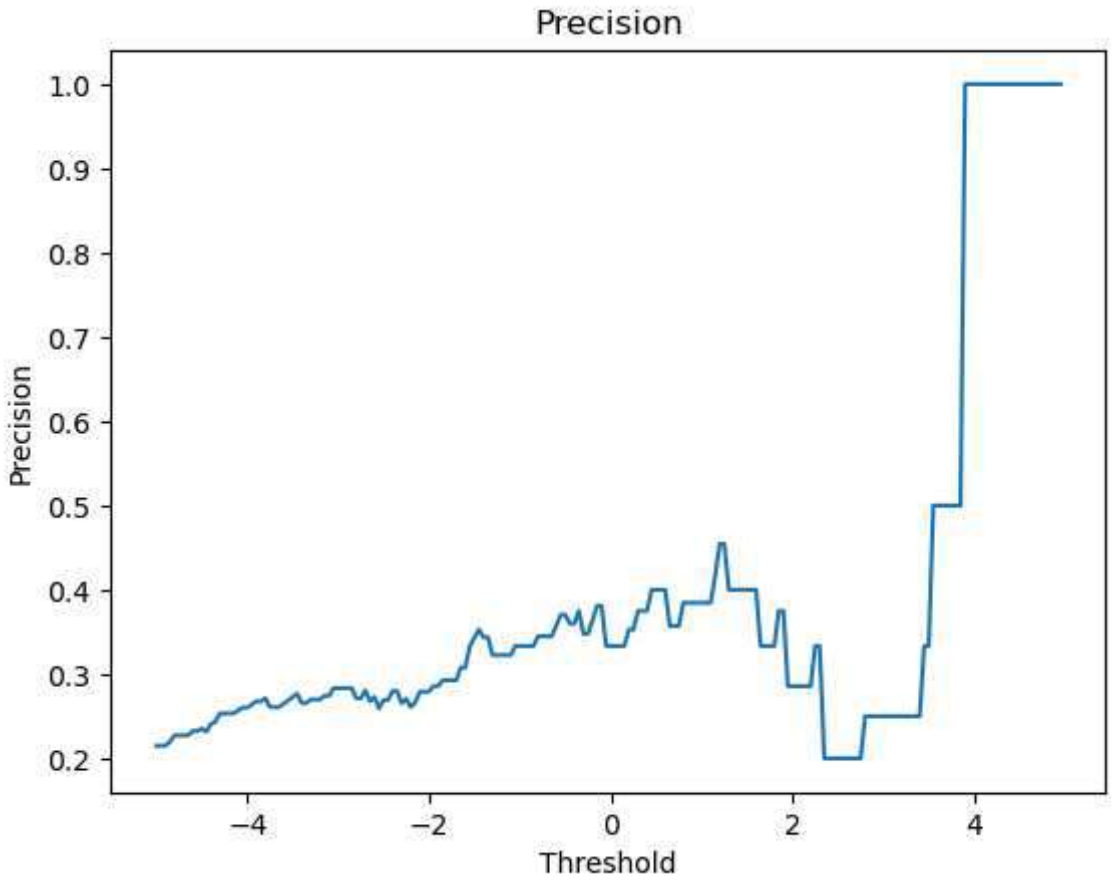
```
Best Accuracy is 0.8435374092276366 with threshold: 3.8999999999999684
Best F1 is 0.4137926040433744 with threshold: -1.4500000000000126
```

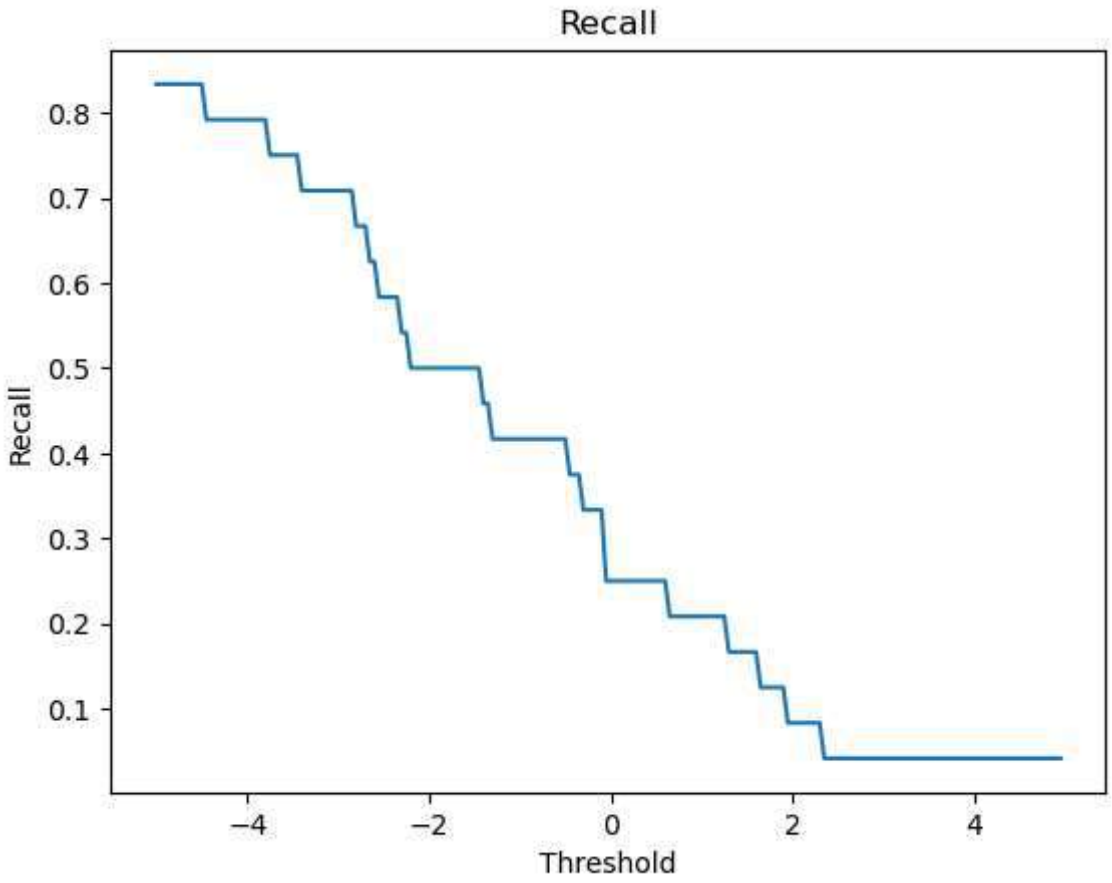
T17. Plot the RoC of your classifier.

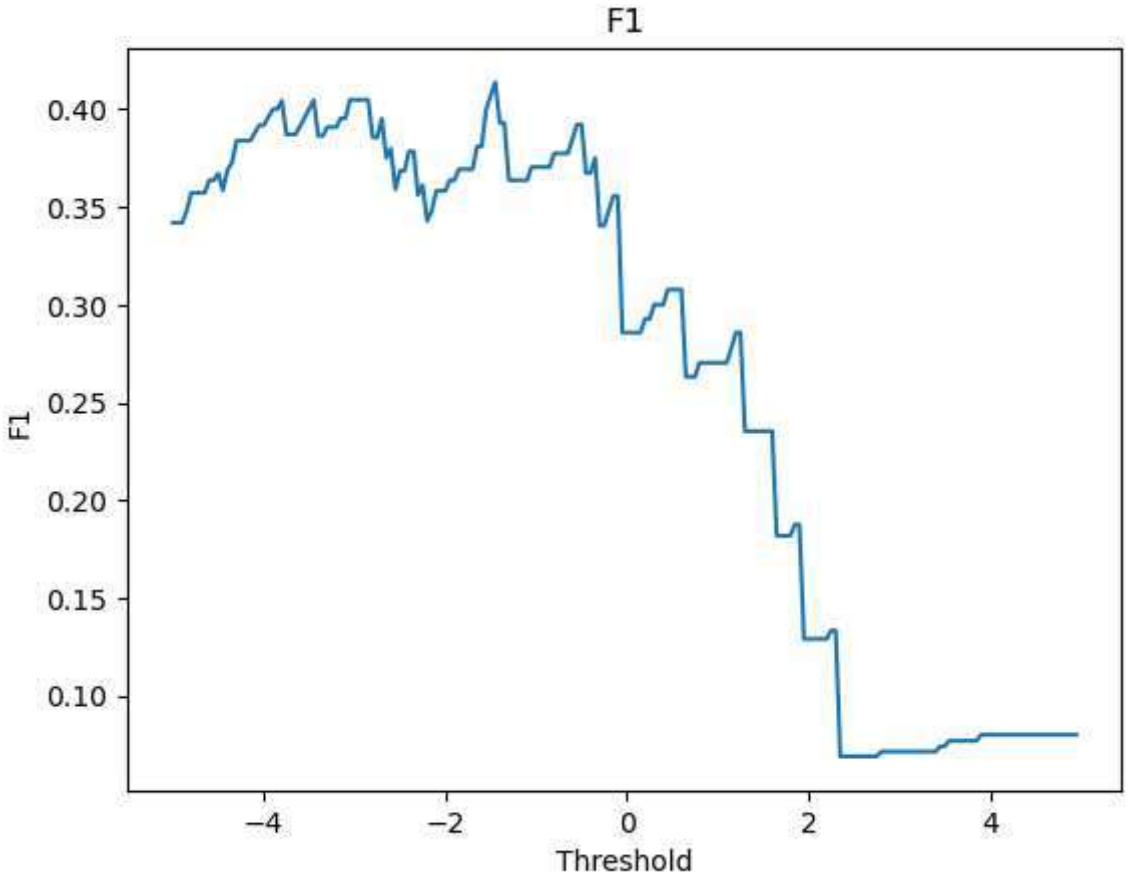
In [97]:
```
t = np.arange(-5,5,0.05)
plt.plot(t, hist_acc)
```

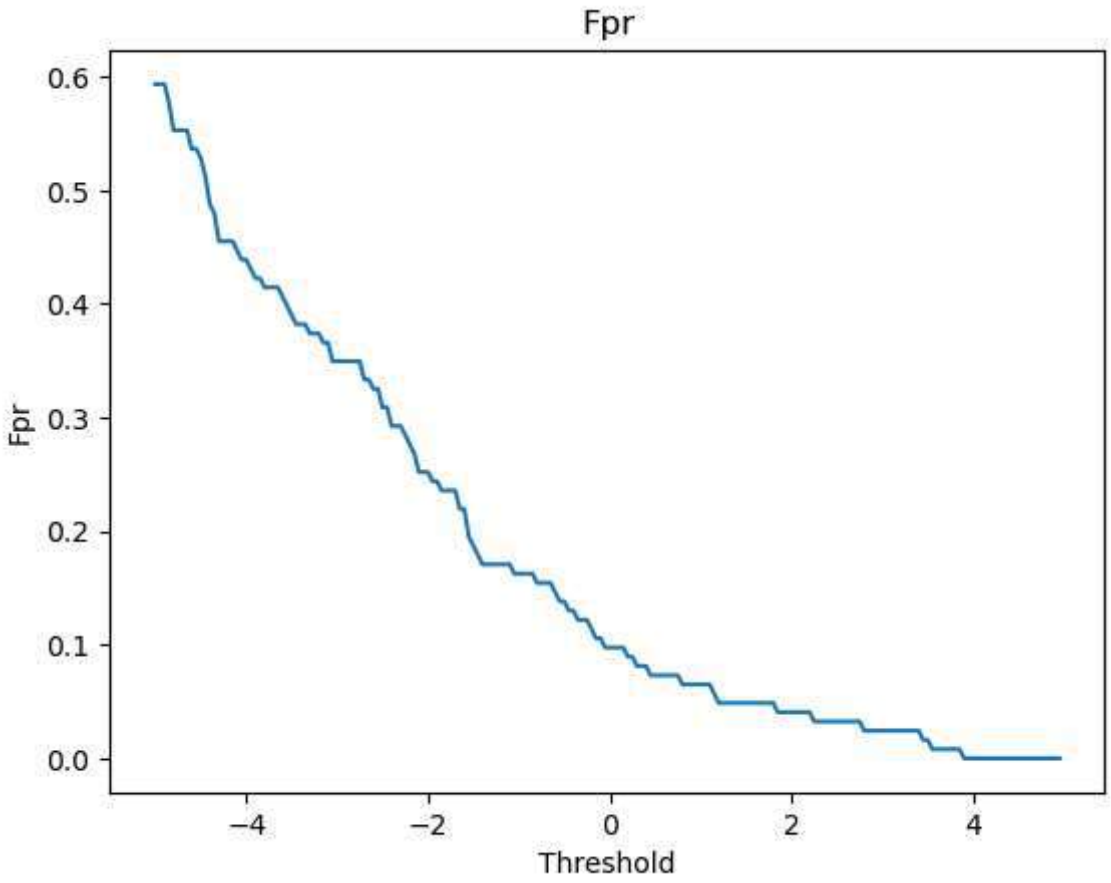
```
plt.title('Accuracy')
plt.xlabel('Threshold')
plt.ylabel('Accuracy')
plt.show()

plt.plot(t, hist_pre)
plt.title('Precision')
plt.xlabel('Threshold')
plt.ylabel('Precision')
plt.show()

plt.plot(t, hist_rec)
plt.title('Recall')
plt.xlabel('Threshold')
plt.ylabel('Recall')
plt.show()

plt.plot(t, hist_F1)
plt.title('F1')
plt.xlabel('Threshold')
plt.ylabel('F1')
plt.show()

plt.plot(t, hist_fpr)
plt.title('Fpr')
plt.xlabel('Threshold')
plt.ylabel('Fpr')
plt.show()
```




Precision
Precision
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
−4
−2
0
2
4
Threshold


Recall
Recall
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
−4
−2
0
2
4
Threshold


F1
0.40
0.35
0.30
0.25
0.20
0.15
0.10
F1
−4
−2
0
2
4
Threshold


Fpr
0.6
0.5
0.4
0.3
0.2
0.1
0.0
Fpr
−4
−2
0
2
4
Threshold

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.

From the result, bins = 5 has higher accuracy and F1 score but only a slightly. So I should conclude that 5 bins is better.

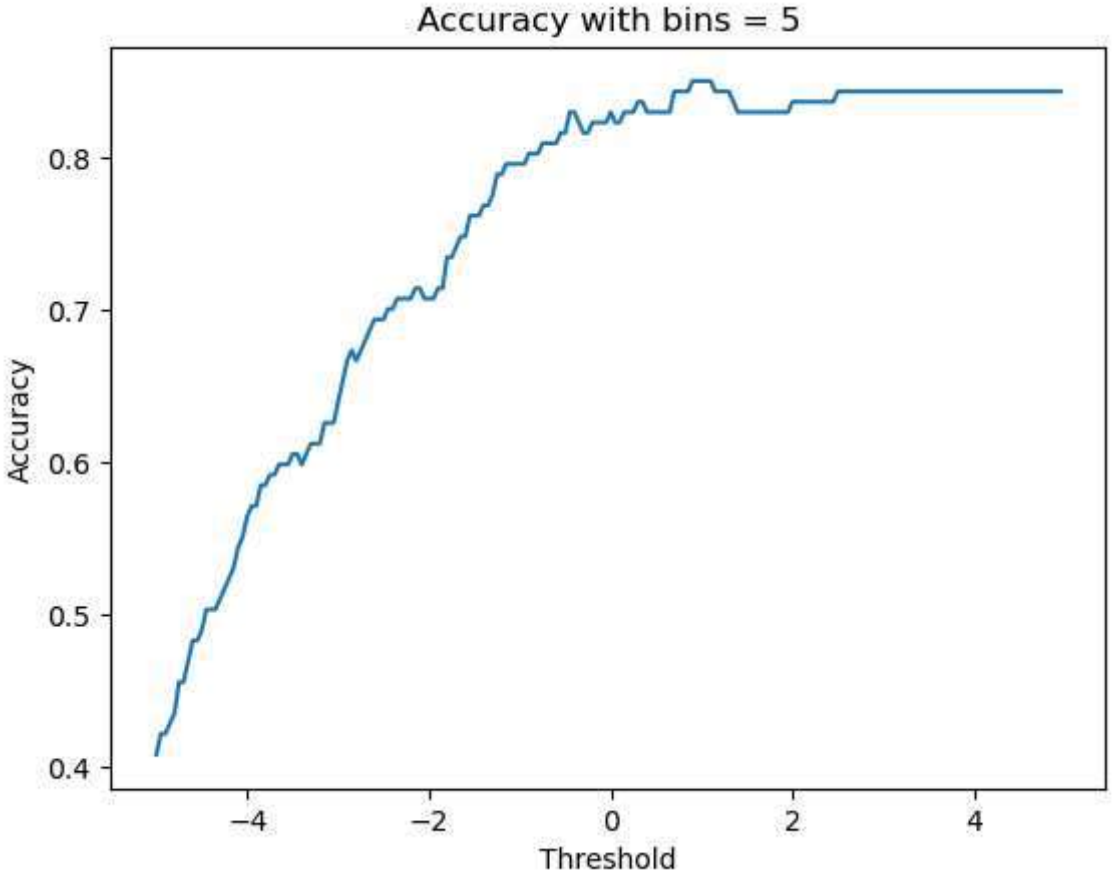
In [102…
```
model.fit_params(x_train, y_train, 5)
y_pred = model.predict(x = x_test)
```

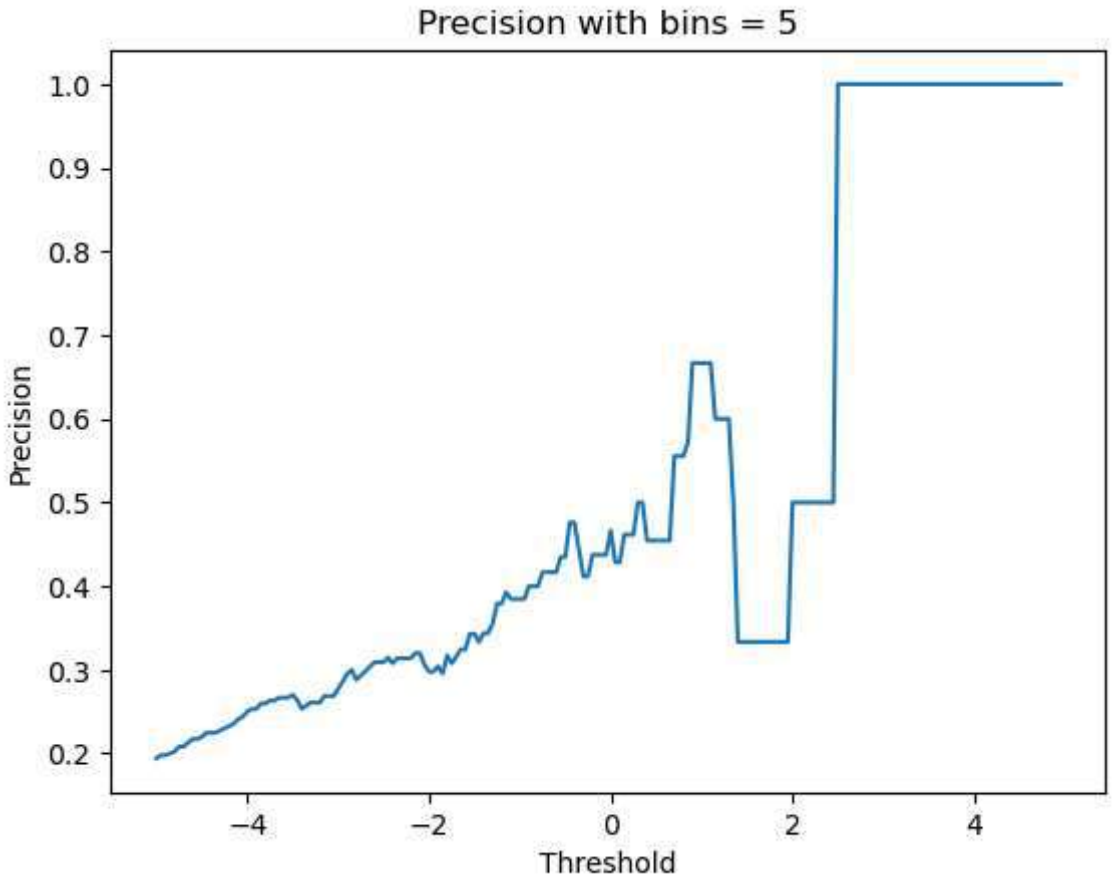
In [103…
```
best_acc = float('-inf')
best_acc_t = -5
best_f1 = float('-inf')
best_f1_t = -5

hist_acc_5 = []
hist_pre_5 = []
hist_rec_5 = []
hist_F1_5 = []
hist_fpr_5 = []
for t in np.arange(-5,5,0.05):
    y_pred_t = model.predict(x = x_test, thresh = t)
    accuracy, precision, recall, F1, fpr = evaluate(y_test, y_pred_t, show_result=F
    hist_acc_5.append(accuracy)
    hist_pre_5.append(precision)
    hist_rec_5.append(recall)
    hist_F1_5.append(F1)
    hist_fpr_5.append(fpr)
    if accuracy > best_acc:
        best_acc = accuracy
        best_acc_t = t
    if F1 > best_f1:
        best_f1 = F1
        best_f1_t = t

print(f'Best Accuracy is {best_acc} with threshold: {best_acc_t}')
print(f'Best F1 is {best_f1} with threshold: {best_f1_t}')
```

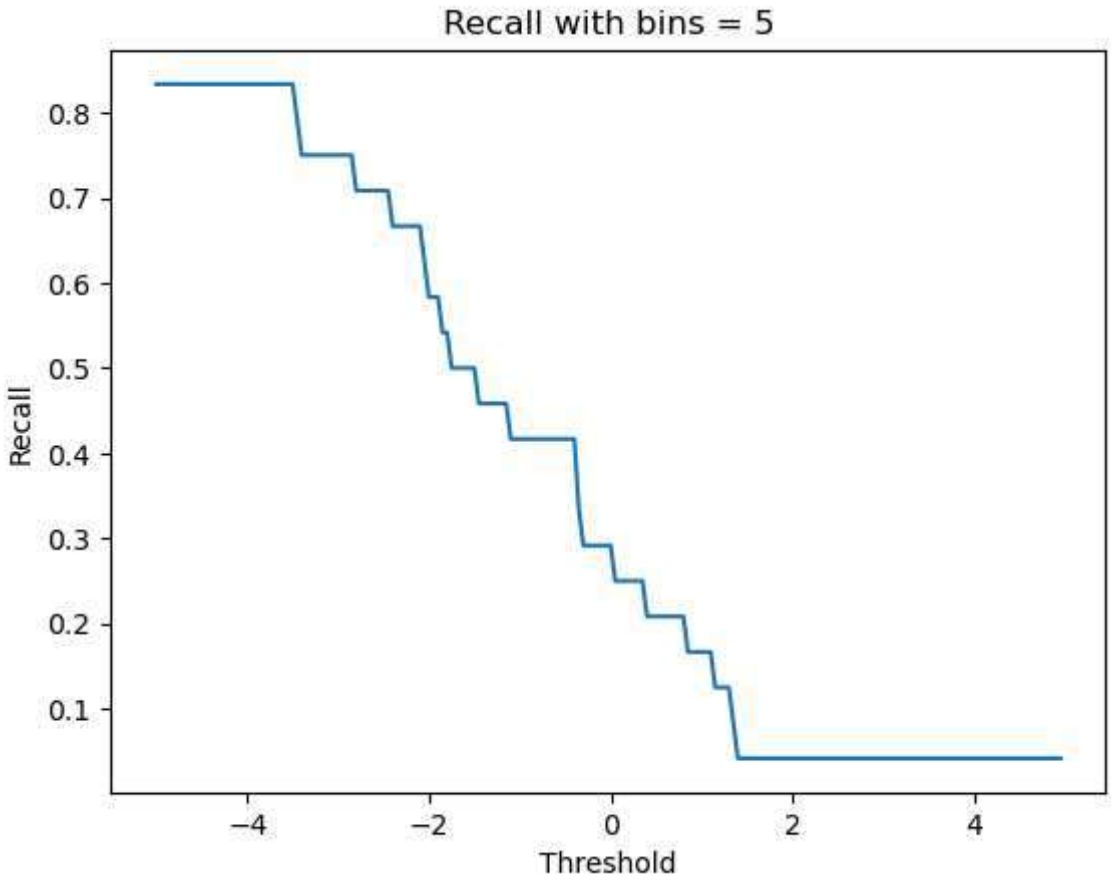
```
Best Accuracy is 0.850340130269795 with threshold: 0.899999999999979
Best F1 is 0.4444439269141384 with threshold: -0.45000000000001616
```

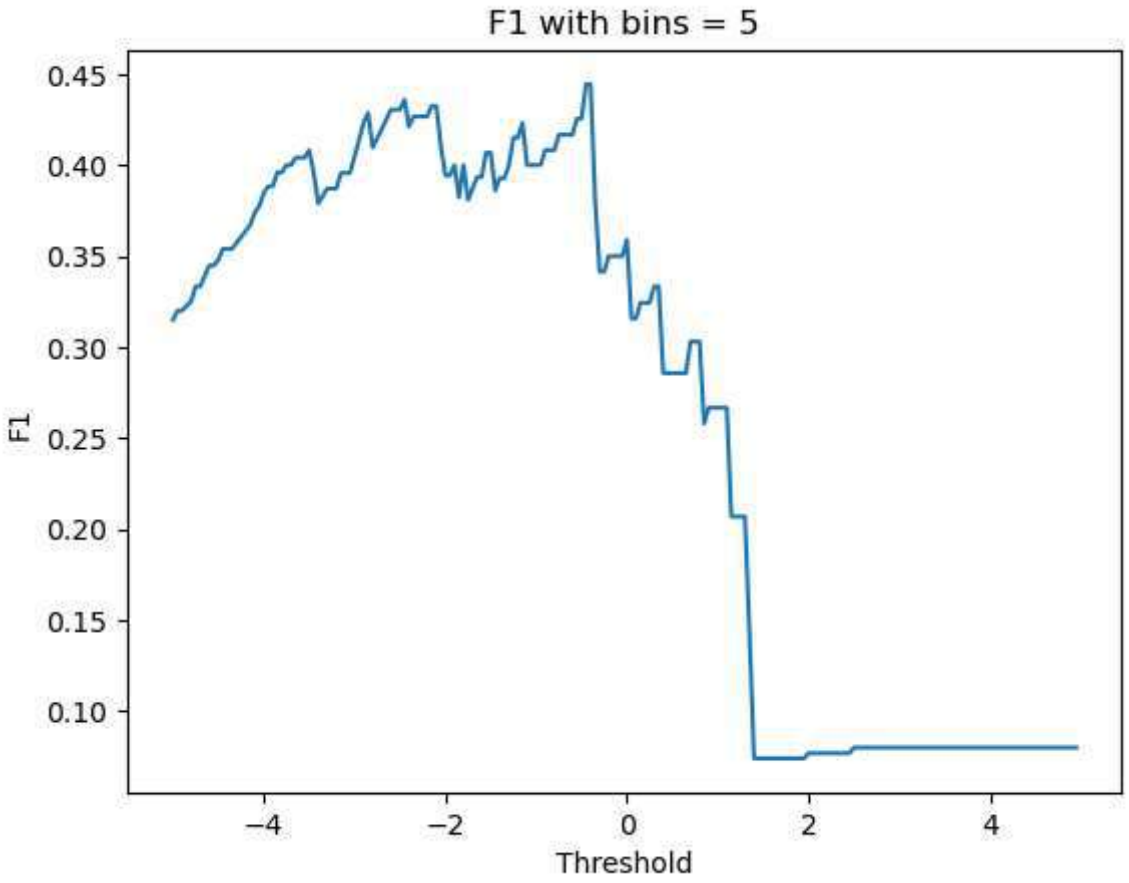
In [104…
```
t = np.arange(-5,5,0.05)
plt.plot(t, hist_acc_5)
plt.title('Accuracy with bins = 5')
plt.xlabel('Threshold')
plt.ylabel('Accuracy')
plt.show()

plt.plot(t, hist_pre_5)
plt.title('Precision with bins = 5')
plt.xlabel('Threshold')
```

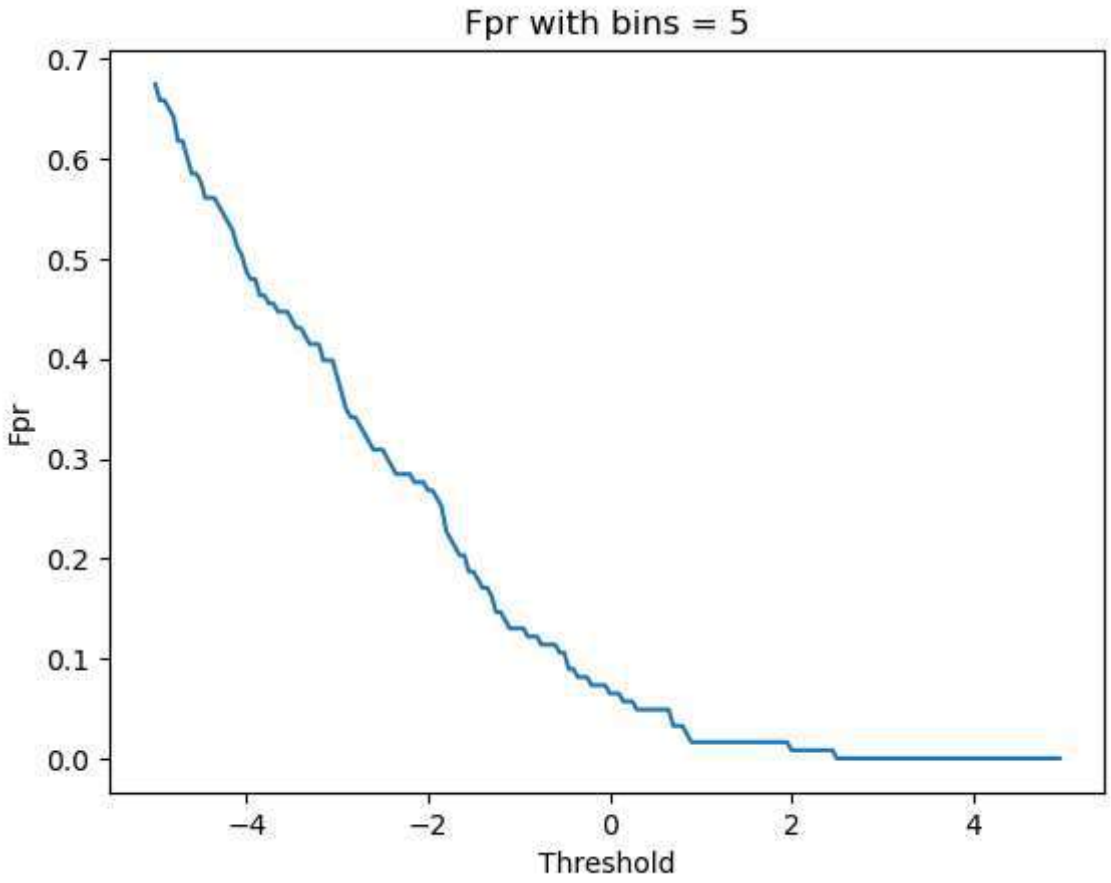
```
plt.ylabel('Precision')
plt.show()

plt.plot(t, hist_rec_5)
plt.title('Recall with bins = 5')
plt.xlabel('Threshold')
plt.ylabel('Recall')
plt.show()

plt.plot(t, hist_F1_5)
plt.title('F1 with bins = 5')
plt.xlabel('Threshold')
plt.ylabel('F1')
plt.show()

plt.plot(t, hist_fpr_5)
plt.title('Fpr with bins = 5')
plt.xlabel('Threshold')
plt.ylabel('Fpr')
plt.show()
```




Precision with bins = 5
Precision
Threshold
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
−4
−2
0
2
4


Recall with bins = 5
Recall
Threshold
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
−4
−2
0
2
4


F1 with bins = 5
F1
Threshold
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
−4
−2
0
2
4


Fpr with bins = 5
Fpr
Threshold
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
−4
−2
0
2
4

In [105…

```
#OT4
all_accu = []
for i in range(10):
    df_train, df_test = train_test_split(df, test_size = 0.1, random_state = 42, st
    data_train = df_train.to_numpy()
    data_test = df_test.to_numpy()
    x_train = df_train.drop(columns=['Attrition']).to_numpy()
    y_train = df_train['Attrition'].to_numpy()

    x_test = df_test.drop(columns=['Attrition']).to_numpy()
    y_test = df_test['Attrition'].to_numpy()
    model = SimpleBayesClassifier(n_pos = (y_train==1).sum(), n_neg = (y_train==0).
    model.fit_params(x_train, y_train)
    y_pred = model.predict(x = x_test)
    accuracy, precision, recall, F1, fpr = evaluate(y_test, y_pred_t, show_result=F
    all_accu.append(accuracy)
print(f'mean: {np.mean(all_accu)}')
print(f'variance: {np.var(all_accu)}')
```

```
mean: 0.8435374092276365
variance: 1.232595164407831e-32
```