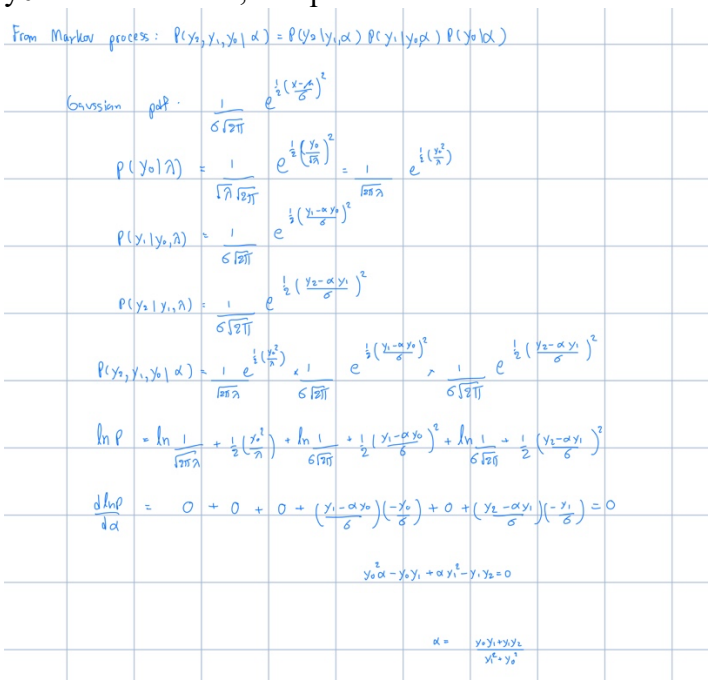
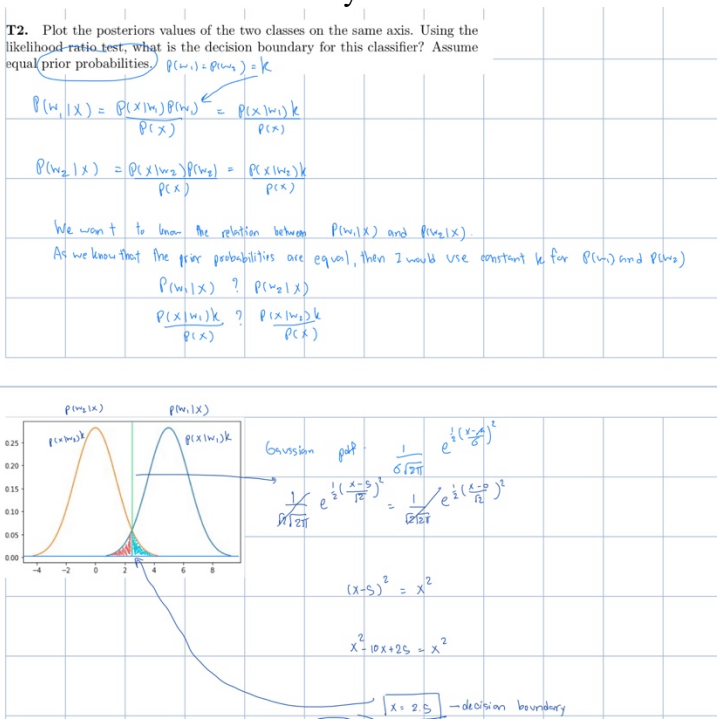
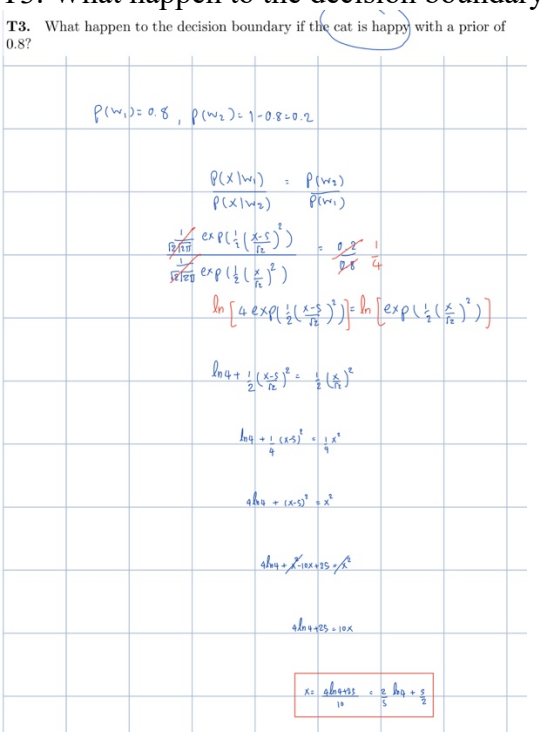
T1. Find the MLE of the rate of return, α, given the observed price at the end of each day y2, y1, y0. In other words, compute for the value of α that maximizes p(y2, y1, y0|α)



T2. Plot the posteriors values of the two classes on the same axis. Using the likelihood ratio test, what is the decision boundary for this classifier? Assume equal prior probabilities.

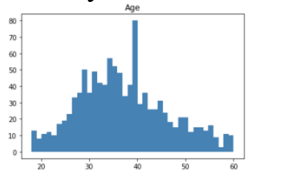


T3. What happen to the decision boundary if the cat is happy with a prior of 0.8?

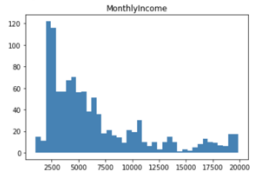


T4. Observe the histogram for Age, MonthlyIncome and DistanceFromHome. How many bins have zero counts? Do you think this is a good discretization? Why?

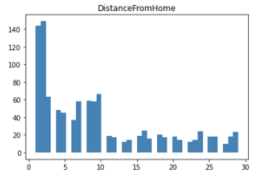
```
all_yes = all[all['Attrition'] == 1].reset_index(drop=True)
all_no = all[all['Attrition'] == 0].reset_index(drop=True)
all_yes_test_size, all_no_test_size = int(len(all_yes.index)*0.1), int(len(all_no.index)*0.1)
all_yes_test, all_yes_train = all_yes[:all_yes_test_size], all_yes[all_yes_test_size:]
all_no_test, all_no_train = all_no[:all_no_test_size], all_no[all_no_test_size:]
print(f'yes test size: {len(all_yes_test.index)}')
print(f'yes train size: {len(all_yes_train.index)}')
print(f'no test size: {len(all_no_test.index)}')
print(f'no train size: {len(all_no_train.index)}')
train_set = pd.concat([all_yes_train, all_no_train])
train_set = train_set.iloc[np.random.permutation(len(train_set))].reset_index(drop=True)
test_set = pd.concat([all_yes_test, all_no_test])
test_set = test_set.iloc[np.random.permutation(len(test_set))].reset_index(drop=True)
train_set2 = train_set.copy(deep=True)
test_set2 = test_set.copy(deep=True)
train_set3 = train_set.copy(deep=True)
test_set3 = test_set.copy(deep=True)
cols = ['Age', 'MonthlyIncome', 'DistanceFromHome']
for col in cols:
  train_col_no_nan = train_set[~np.isnan(train_set[col])][col]
  hist, bin_edge = np.histogram(train_col_no_nan, 40)
  bin_edge = np.array(bin_edge, dtype=float)
  zero_bin = len([e for e in hist if e == 0])
  # print(hist)
  plt.fill_between(bin_edge.repeat(2)[1:-1], hist.repeat(2),facecolor='steelblue')
  plt.title(col)
  plt.show()
  print(f'For {col}, the number of bins whose value is zero is {zero_bin}')
```




```
For Age, the number of bins whose value is zero is 0
```




```
For MonthlyIncome, the number of bins whose value is zero is 0
```




```
For DistanceFromHome, the number of bins whose value is zero is 11
```

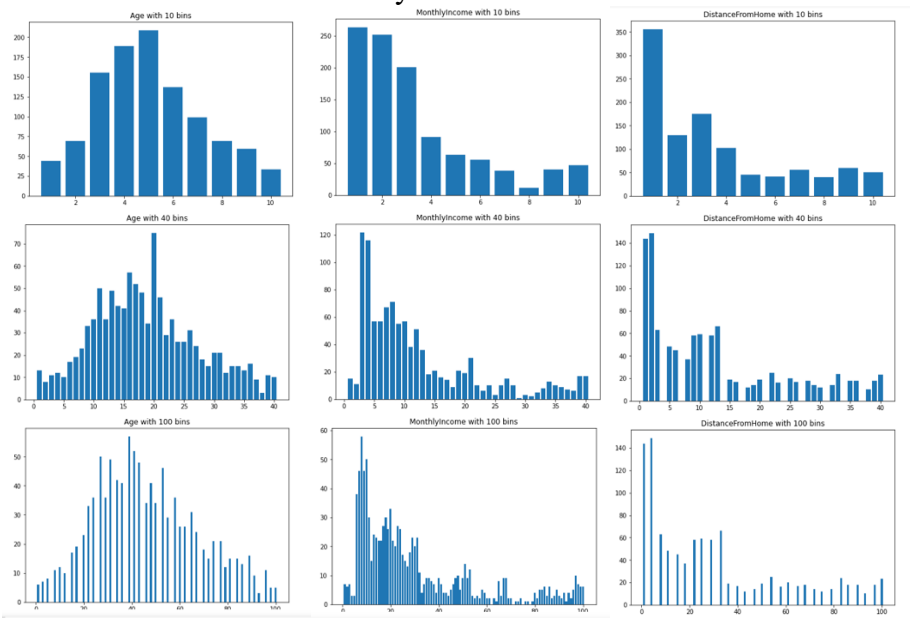
In my opinion, this is almost a good discretization except for the distancefromhome feature that made this discretization underperformed.

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

To my mind, it is possible to use a Gaussian to estimate the histogram. This could be done by using the MLE process. First, we need to find the parameter of the Gaussian which are mean and

standard deviation. From the data, it is quite straightforward to find these two values. Moreover, in order to use GMM, we need to define the number of the gaussian we want to fit with the data. In this case, only one Gaussian might be enough to describe the distribution of the data. As a result, it may be excessive to apply GMM.

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



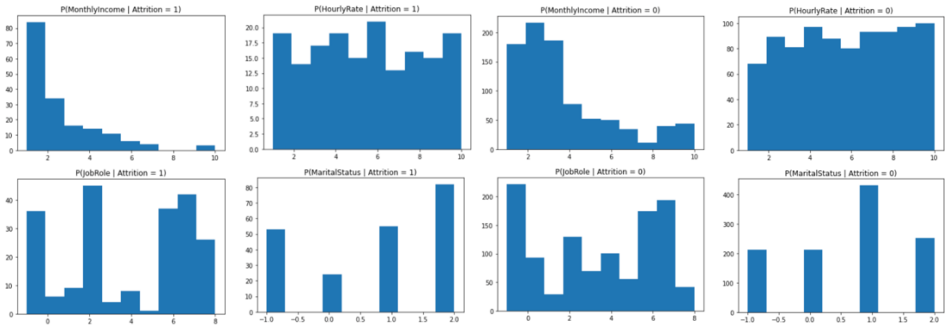
From the above figures, the 40 and 100 bins might not be appropriate since some of the bins contain the value of zero which is hard to infer about the real distribution of the data. As a result, the 10-bin is the best choice for binning.

T7. For the rest of the features, which one should be discretized? 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.

Basically, we want to inference the real distribution of the data. Consequently, we don't want any of the bin being zero. Hence, we will consider the interval and ratio data types to be transform by binning method. The features that we will transform are
`['Age', 'DailyRate', 'DistanceFromHome', 'Education', 'EmployeeCount', 'EnvironmentSatisfaction', 'HourlyRate', 'JobInvolvement', 'JobLevel', 'JobSatisfaction', 'MonthlyIncome', 'MonthlyRate', 'NumCompaniesWorked', 'PercentSalaryHike', 'PerformanceRating', 'RelationshipSatisfaction', 'StandardHours', 'StockOptionLevel', 'TotalWorkingYears', 'TrainingTimesLastYear', 'WorkLifeBalance', 'YearsAtCompany', 'YearsInCurrentRole', 'YearsSinceLastPromotion', 'YearsWithCurrManager'].`

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.

We will use Gaussian distribution to model the histograms. The parameters of the Gaussian distribution are mean and variance. To find the these attributes, we must consider the pdf of the Gaussian function. Then, we take the derivative respected to each parameter and set the whole derivative term to zero. Finally, we can obtain the formula to calculate for the mean and variance.



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

Bernoulli distribution because there are only two possible class which are 0 and 1.

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

Instead using zero, we will use the very low value for this case. We will apply 0.001 in spite of zero.

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, 6 Recall, and F score for detecting attrition. See Lecture 1 for the definitions of each metric.

Record the distribution of training data in each feature and each class.

```
features = [e for e in list(train_set.columns) if e != 'Attrition']
classes = [0, 1]
prob_feature_given_class = {}
for class_ in classes:
  for col in features:
    df_class = train_set[train_set['Attrition'] == class_]
    y = np.array(df_class[~np.isnan(df_class[col])][col])
    unique, counts = np.unique(y, return_counts=True)
    bucket_dict = dict(zip(unique, counts))
    number_element = sum(bucket_dict.values())
    normalize_bucket = {}
    for key, value in bucket_dict.items():
      normalize_bucket[key] = value / number_element
    print(normalize_bucket)
    prob_feature_given_class[(f'class_{class_}', col)] = normalize_bucket
    plt.hist(y)
    plt.title(f'P({col} | Attrition = {class_})')
    plt.show()
    plt.bar(normalize_bucket.keys(), normalize_bucket.values(), 1, color='g')
    plt.show()
```

Calculate the prior terms.

```
leave_number = train_set[train_set['Attrition'] == 1].shape[0]
stay_number = train_set[train_set['Attrition'] == 0].shape[0]
all_number =  train_set.shape[0]
prob_leave = leave_number / all_number
prob_stay = stay_number / all_number
print(f'leave number: {leave_number}')
print(f'stay number: {stay_number}')
print(f'all number: {all_number}')
print(f'p(leave): {prob_leave}')
print(f'p(stay): {prob_stay}')
```

```
leave number: 214
stay number: 1110
all number: 1324
p(leave): 0.16163141993957703
p(stay): 0.8383685800604229
```

Bin the test dataset as the same method as the training set. We also use the train_set to find the binning range.

```
for float_col in float_cols:
  bin_range = get_bin_range(train_set, float_col, 10)
  test_set[float_col] = bucketize(test_set, float_col, bin_range)
```

Fit the model to the test dataset.

```
test_set_x, test_set_y = test_set.drop(columns = "Attrition"), test_set['Attrition']
test_result_prob = []
for index, row in test_set_x.iterrows():
  log_h_of_x = np.log(prob_leave) - np.log(prob_stay)
  for feature in features:
    if row[feature] == np.nan or str(row[feature]) == 'nan':
      continue
    key_leave = ('class_1', feature)
    leave_feature_dict = prob_feature_given_class[key_leave]
    p_feature_given_leave = leave_feature_dict[row[feature]] if row[feature] in leave_feature_dict else 1e-3
    key_stay = ('class_0', feature)
    stay_feature_dict = prob_feature_given_class[key_stay]
    p_feature_given_stay = stay_feature_dict[row[feature]] if row[feature] in stay_feature_dict else 1e-3
    log_h_of_x += (np.log(p_feature_given_leave) - np.log(p_feature_given_stay))
  test_result_prob.append(log_h_of_x)
test_result_prob = np.array(test_result_prob)
test_result_prob_model1 = test_result_prob
```

```
test_result_pred = (test_result_prob > 0) + 0
N = len(test_set_y)
accuracy = (test_set_y == test_result_pred).sum() / N
true_positive = ((test_set_y == 1) & (test_result_pred == 1)).sum()
false_postive = ((test_set_y == 0) & (test_result_pred == 1)).sum()
false_negative = ((test_set_y == 1) & (test_result_pred == 0)).sum()
precision = true_positive / (true_positive + false_postive)
recall = true_positive / (true_positive + false_negative)
f1 = (2*precision*recall) / (precision + recall)
print(f'accuracy: {accuracy}')
print(f'precision: {precision}')
print(f'recall: {recall}')
print(f'f1: {f1}')
```

```
accuracy: 0.8013698630136986
precision: 0.39285714285714285
recall: 0.4782608695652174
f1: 0.4313725490196078
```

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

Record the mean and standard deviation of the features that were originally be binned.

```
all_features = [e for e in list(train_set2.columns) if e != 'Attrition']
continuous_features = float_cols
discrete_features = [e for e in all_features if e not in continuous_features]
print(f'continuous features: {continuous_features}')
print(f'discrete features: {discrete_features}')
mean_std_continuous_features = {}
for class_ in [0, 1]:
  for feature in all_features:
    if feature in continuous_features:
      df_class = train_set2[train_set2['Attrition'] == class_]
      y = np.array(df_class[~np.isnan(df_class[feature])][feature])
      mean, std = np.mean(y), np.std(y)
      mean_std_continuous_features[(f'class_{class_}', feature)] = (mean, std)
```

Apply the new model to the test dataset.

```
test_set_x, test_set_y = test_set2.drop(columns = "Attrition"), test_set2['Attrition']
test_result_prob = []
import scipy
for index, row in test_set_x.iterrows():
  log_h_of_x = np.log(prob_leave) - np.log(prob_stay)
  for feature in features:
    if row[feature] == np.nan or str(row[feature]) == 'nan':
      continue
    key_leave = ('class_1', feature)
    key_stay = ('class_0', feature)
    if feature in continuous_features:
      mean_leave, std_leave = mean_std_continuous_features[key_leave]
      p_feature_given_leave = scipy.stats.norm(mean_leave, std_leave).pdf(row[feature])
      mean_stay, std_stay = mean_std_continuous_features[key_stay]
      p_feature_given_stay = scipy.stats.norm(mean_stay, std_stay).pdf(row[feature])
    elif feature in discrete_features:
      leave_feature_dict = prob_feature_given_class[key_leave]
      p_feature_given_leave = leave_feature_dict[row[feature]] if row[feature] in leave_feature_dict else 1e-3
      stay_feature_dict = prob_feature_given_class[key_stay]
      p_feature_given_stay = stay_feature_dict[row[feature]] if row[feature] in stay_feature_dict else 1e-3
    log_h_of_x += (np.log(p_feature_given_leave) - np.log(p_feature_given_stay))
  test_result_prob.append(log_h_of_x)
test_result_prob = np.array(test_result_prob)
test_result_prob_model2 = test_result_prob
```

```
/usr/local/lib/python3.7/dist-packages/scipy/stats/_distn_infrastructure.py:1740: RuntimeWarning: invalid value
  x = np.asarray((x - loc)/scale, dtype=dtyp)
```

```
test_result_pred = (test_result_prob > 0) + 0
N = len(test_set_y)
accuracy = (test_set_y == test_result_pred).sum() / N
true_positive = ((test_set_y == 1) & (test_result_pred == 1)).sum()
false_postive = ((test_set_y == 0) & (test_result_pred == 1)).sum()
false_negative = ((test_set_y == 1) & (test_result_pred == 0)).sum()
precision = true_positive / (true_positive + false_postive)
recall = true_positive / (true_positive + false_negative)
f1 = (2*precision*recall) / (precision + recall)
print(f'accuracy: {accuracy}')
print(f'precision: {precision}')
print(f'recall: {recall}')
print(f'f1: {f1}')
```

```
accuracy: 0.8424657534246576
precision: 0.5
recall: 0.043478260869565216
f1: 0.08
```

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.

```
base_line = []
import random
for i in range(N):
  base_line.append(random.randrange(0,2))
base_line = np.array(base_line)
N = len(test_set_y)
accuracy = (test_set_y == base_line).sum() / N
true_positive = ((test_set_y == 1) & (base_line == 1)).sum()
false_postive = ((test_set_y == 0) & (base_line == 1)).sum()
false_negative = ((test_set_y == 1) & (base_line == 0)).sum()
precision = true_positive / (true_positive + false_postive)
recall = true_positive / (true_positive + false_negative)
f1 = (2*precision*recall) / (precision + recall)
print(f'accuracy: {accuracy}')
print(f'precision: {precision}')
print(f'recall: {recall}')
print(f'f1: {f1}')
```

```
accuracy: 0.5
precision: 0.1323529411764706
recall: 0.391304347826087
f1: 0.1978021978021978
```

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.

```
majority = np.zeros(N)
N = len(test_set_y)
accuracy = (test_set_y == majority).sum() / N
true_positive = ((test_set_y == 1) & (majority == 1)).sum()
false_postive = ((test_set_y == 0) & (majority == 1)).sum()
false_negative = ((test_set_y == 1) & (majority == 0)).sum()
precision = true_positive / (true_positive + false_postive)
recall = true_positive / (true_positive + false_negative)
f1 = (2*precision*recall) / (precision + recall)
print(f'accuracy: {accuracy}')
print(f'precision: {precision}')
print(f'recall: {recall}')
print(f'f1: {f1}')
```

```
accuracy: 0.8424657534246576
precision: nan
recall: 0.0
f1: nan
```

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

	Accuracy	Precision	Recall	F1-score
model from T11	0.8013	0.3928	0.4782	0.4313
model from T12	0.8424	0.5000	0.0434	0.0800
Random baseline	0.5000	0.1323	0.3913	0.1978
Majority baseline	0.8424	nan	0.0	nan

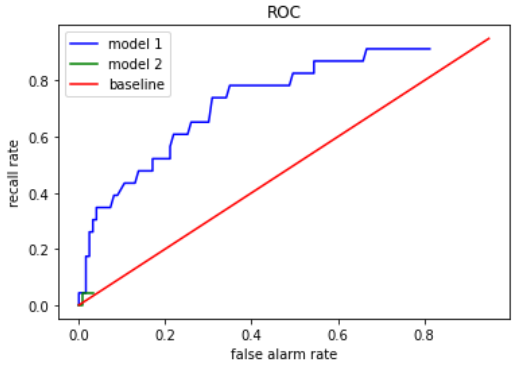
The models seem to perform better than the baselines.

T16. Use the following threshold values
t = np.arange(-5,5,0.05)
find the best accuracy, and F score (and the corresponding thresholds)

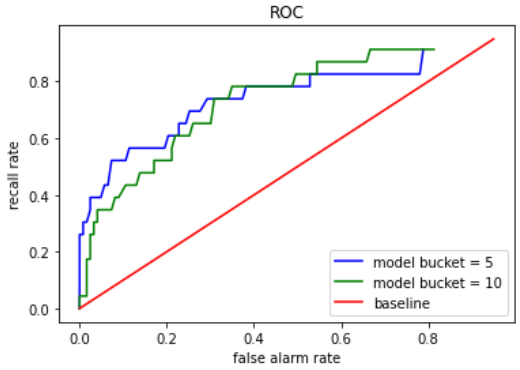
```
thresholds = np.arange(-5,5,0.05)
max_f1_model_1 = 0
max_acc_model_1 = 0
max_f1_model_2 = 0
max_acc_model_2 = 0
recall_rate_model_1 = []
false_alarm_rate_model_1 = []
recall_rate_model_2 = []
false_alarm_rate_model_2 = []
best_acc_model_1_dix = 0
best_f1_model_1_dix = 0
best_acc_model_2_dix = 0
best_f1_model_2_dix = 0
for t in thresholds:
  test_result_pred = (test_result_prob_model1 > t) + 0
  N = len(test_set_y)
  # print(len(test_set_y))
  accuracy = (test_set_y == test_result_pred).sum() / N
  true_positive = ((test_set_y == 1) & (test_result_pred == 1)).sum()
  false_postive = ((test_set_y == 0) & (test_result_pred == 1)).sum()
  false_negative = ((test_set_y == 1) & (test_result_pred == 0)).sum()
  actual_no = (test_set_y == 0).sum()
  precision = true_positive / (true_positive + false_postive)
  recall = true_positive / (true_positive + false_negative)
  f1 = (2*precision*recall) / (precision + recall)
  false_alarm_rate = false_postive / actual_no
  if accuracy > max_acc_model_1:
    max_acc_model_1 = accuracy
    best_acc_model_1_dix = t
  if f1 > max_f1_model_1:
    max_f1_model_1 = f1
    best_f1_model_1_dix = t
  max_acc_model_1 = max(max_acc_model_1, accuracy)
  max_f1_model_1 = max(max_f1_model_1, f1)
  recall_rate_model_1.append(recall)
  false_alarm_rate_model_1.append(false_alarm_rate)


  test_result_pred = (test_result_prob_model2 > t) + 0
  N = len(test_set_y)
  accuracy = (test_set_y == test_result_pred).sum() / N
  true_positive = ((test_set_y == 1) & (test_result_pred == 1)).sum()
  false_postive = ((test_set_y == 0) & (test_result_pred == 1)).sum()
  false_negative = ((test_set_y == 1) & (test_result_pred == 0)).sum()
  actual_no = (test_set_y == 0).sum()
  precision = true_positive / (true_positive + false_postive)
  recall = true_positive / (true_positive + false_negative)
  f1 = (2*precision*recall) / (precision + recall)
  false_alarm_rate = false_postive / actual_no
  max_acc_model_2 = max(max_acc_model_2, accuracy)
  max_f1_model_2 = max(max_f1_model_2, f1)
  recall_rate_model_2.append(recall)
  false_alarm_rate_model_2.append(false_alarm_rate)
  if accuracy > max_acc_model_2:
    max_acc_model_2 = accuracy
    best_acc_model_2_dix = t
  if f1 > max_f1_model_2:
    max_f1_model_2 = f1
    best_f1_model_2_dix = t

print(f'model 1 max acc: {max_acc_model_1} on t={max_acc_model_1}')
print(f'model 1 max f1: {max_f1_model_1} on t={best_f1_model_1_dix}')
print(f'model 2 max acc: {max_acc_model_2} on t={best_acc_model_2_dix}')
print(f'model 2 max f1: {max_f1_model_2} on t={best_f1_model_2_dix}')
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:48: RuntimeWarning: invalid value encountered in double_scalars
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:46: RuntimeWarning: invalid value encountered in long_scalars
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:23: RuntimeWarning: invalid value encountered in long_scalars
model 1 max acc: 0.863013698630137 on t=0.863013698630137
model 1 max f1: 0.4444444444444444 on t=0.7499999999999796
model 2 max acc: 0.8424657534246576 on t=0
model 2 max f1: 0.08 on t=0
```

T17. Plot the RoC of your classifier.



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.



Considering the area under RoC, I believe that the 5-bucket is better than 10-bucket.