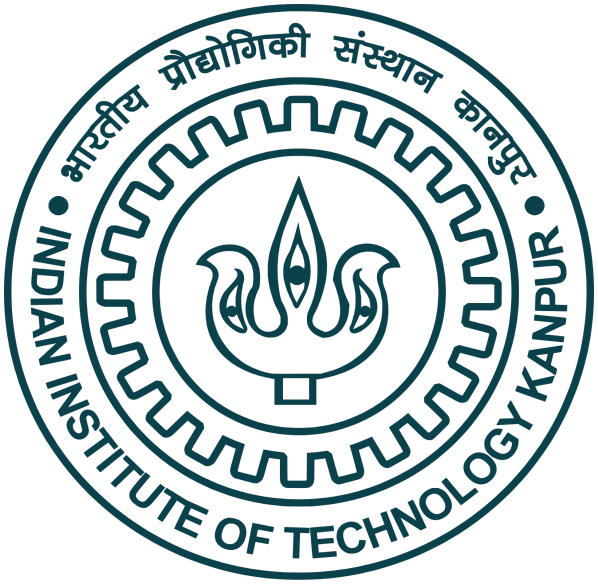


MBA786M - ASSIGNMENT

Group Members

Sabari S (208170829)
Deeksha Rawat (210303)
Shorya Agarwal (221023)
Harsh Nirmal (220431)
Kumar Aditya (220559)

Dataset Overview -

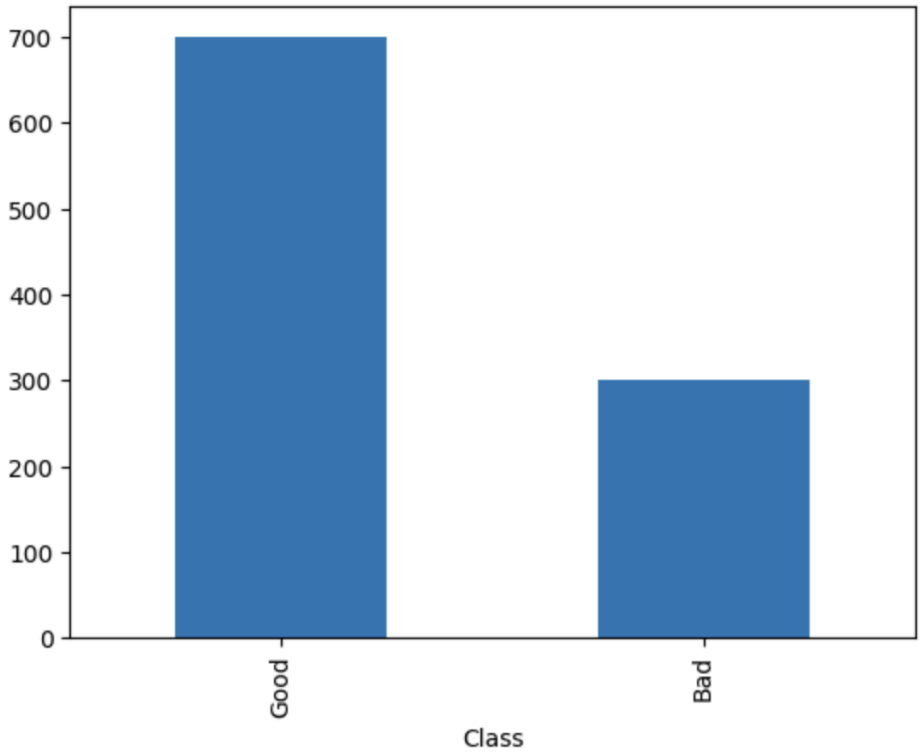
The dataset contains 62 columns, representing various features related to credit applicants (such as `Duration`, `Amount`, `Age`, etc.), and a target variable (`Class`), which indicates the credit risk classification as either **Good** or **Bad**. The target variable is crucial for binary classification tasks where we predict if a credit application is classified as "Good" or "Bad".

Class Distribution:

Upon inspecting the distribution of the target variable, it was found that the dataset is **imbalanced**. **70%** of the instances are labeled as **Good**, while only **30%** are labeled as **Bad**.

```
# Distribution of the target variable
data['Class'].value_counts().plot(kind='bar')
```

```
<Axes: xlabel='Class'>
```



Feature Selection

The target variable is first **label encoded**, with "Good" mapped to 0 and "Bad" to 1 for binary classification.

```
[53] data['Class_code'] = data['Class'].map({'Good': 0, 'Bad': 1})  # Target (0 for Good, 1 for Bad)

     features = data.drop(columns=['Class', 'Class_code'])
     target = data['Class_code']
```

Pearson correlation coefficients are then calculated between each feature and the target to assess linear relationships. **P-values** are computed for each feature to determine statistical significance, with features having p-values below 0.05 considered significant. These significant features are stored in a list. This method allows for the identification of variables with meaningful linear correlations to the target, aiding in feature selection for predictive modeling.

```
[47] # Initialize lists to store significant variables
     significant_vars = []
     p_values = []

     # Calculate Pearson correlation and p-values for each feature against the target
     for col in features.columns:
         correlation, p_value = stats.pearsonr(data[col], target)
         p_values.append(p_value)
         if p_value < 0.05:  # Assuming a significance level of 0.05
             significant_vars.append(col)
```

```
<ipython-input-47-223cfce55c85>:7: ConstantInputWarning: An input array is constant; the correlation coefficient is not defined.
  correlation, p_value = stats.pearsonr(data[col], target)
```

```
#The significant variables for Class
significant_vars
```

```
['Duration',
 'Amount',
 'InstallmentRatePercentage',
 'Age',
 'ForeignWorker',
 'CheckingAccountStatus.lt.0',
 'CheckingAccountStatus.0.to.200',
 'CheckingAccountStatus.none',
 'CreditHistory.NoCredit.AllPaid',
 'CreditHistory.ThisBank.AllPaid',
 'CreditHistory.Critical',
 'Purpose.NewCar',
 'Purpose.UsedCar',
 'Purpose.Radio.Television',
 'Purpose.Education',
 'SavingsAccountBonds.lt.100',
 'SavingsAccountBonds.500.to.1000',
 'SavingsAccountBonds.gt.1000',
 'SavingsAccountBonds.Unknown',
 'EmploymentDuration.lt.1',
 'EmploymentDuration.4.to.7',
 'Personal.Female.NotSingle',
 'Personal.Male.Single',
 'OtherDebtorsGuarantors.CoApplicant',
 'Property.RealEstate',
 'Property.Unknown',
 'OtherInstallmentPlans.Bank',
 'OtherInstallmentPlans.None',
 'Housing.Rent',
 'Housing.Own',
 'Housing.ForFree']
```

Why StandardScaler?

- **Equal Contribution of Features:** Algorithms that compute distances or gradients (like k-NN, SVM, and neural networks) are sensitive to the scale of the features. Without scaling, features with larger ranges could dominate those with smaller ranges, leading to suboptimal results.
- **Faster Convergence:** Gradient-based algorithms (like logistic regression, neural networks, and gradient descent) converge faster when the data is standardized.
- **Comparable Features:** By standardizing the features, each one is on the same scale, making it easier to interpret and compare their effects.

The purpose of `StandardScaler` is to **standardize the features** by removing the mean and scaling them to have unit variance. This is crucial for many machine learning algorithms that rely on distance-based calculations (like SVM, k-NN, and neural networks) to ensure that features with larger ranges do not

dominate the learning process.

For each feature X_i , the scaled value X_{scaled} is calculated as: $X_{scaled} = \frac{X_i - \mu_i}{\sigma_i}$

Why 'lbfgs' solver?

`lbfgs` is preferred here because it is **fast, accurate**, and **efficient for medium-sized datasets** like your Credit dataset. It supports **L2 regularization** and handles binary classification effectively, making it a strong candidate for logistic regression in this context.

TPR and FPR

TPR (Recall): Indicates the model's ability to correctly identify risky ("Bad") credit applicants. A high TPR means that fewer risky applicants are missed, which is critical for minimizing credit losses.

FPR: Reflects how often the model incorrectly identifies safe ("Good") credit applicants as risky. A low FPR ensures that fewer "Good" credit applicants are falsely classified as "Bad", avoiding potential negative impacts on legitimate customers.

In fraud detection, **False Positives** (legitimate transactions flagged as fraud) are usually less costly than **False Negatives** (fraudulent transactions missed). This is why preferring **high TPR** is often justified: the cost of missing fraud is higher than the cost of inconveniencing a few legitimate customers.

Q1. (a) Fit a logistic regression model on the dataset. Choose a probability of default threshold of 20%, 35%, and 50%, to assign an observation to the Bad class. Compute a confusion matrix for each of the models. How do the True Positive and False Positive rates vary over these models? Which model would you choose?

Here we train and test the model on the same dataset

Fitting a logistic regression model:

```
[52] # Logistic regression model
     model = LogisticRegression(solver ='lbfgs')
     model.fit(X, y)
```

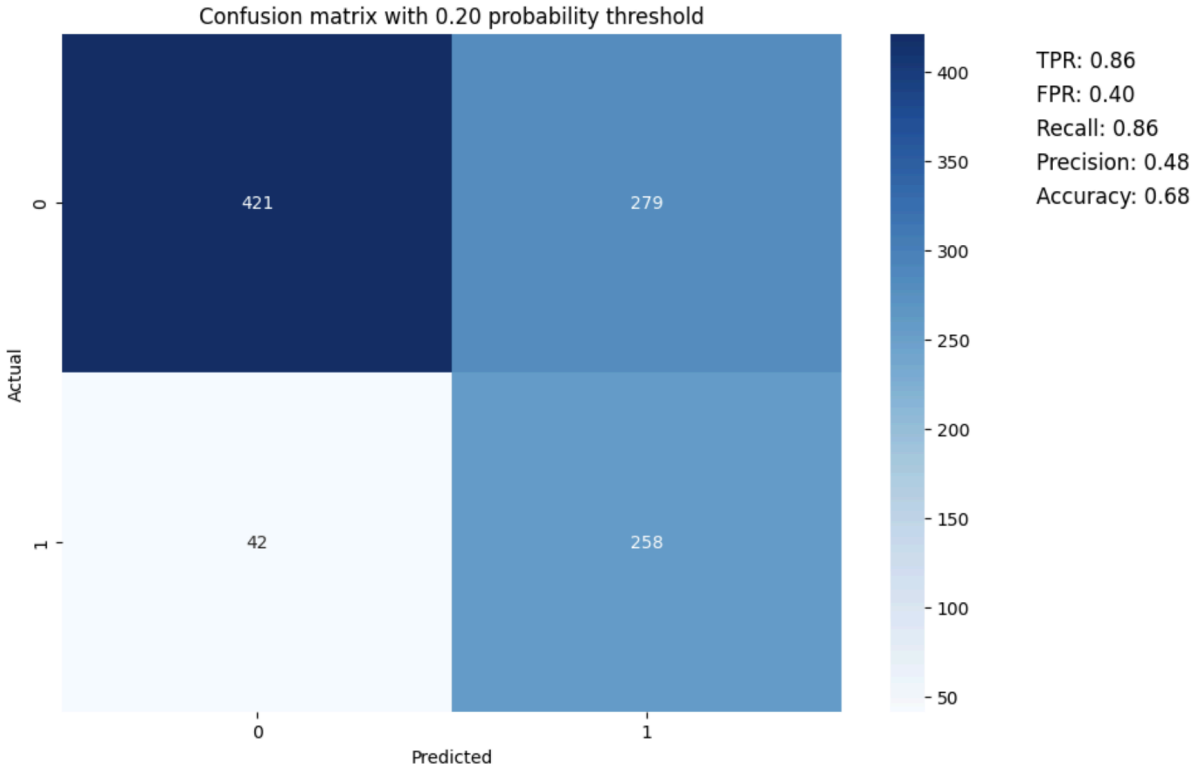
```
▾ LogisticRegression
LogisticRegression()
```

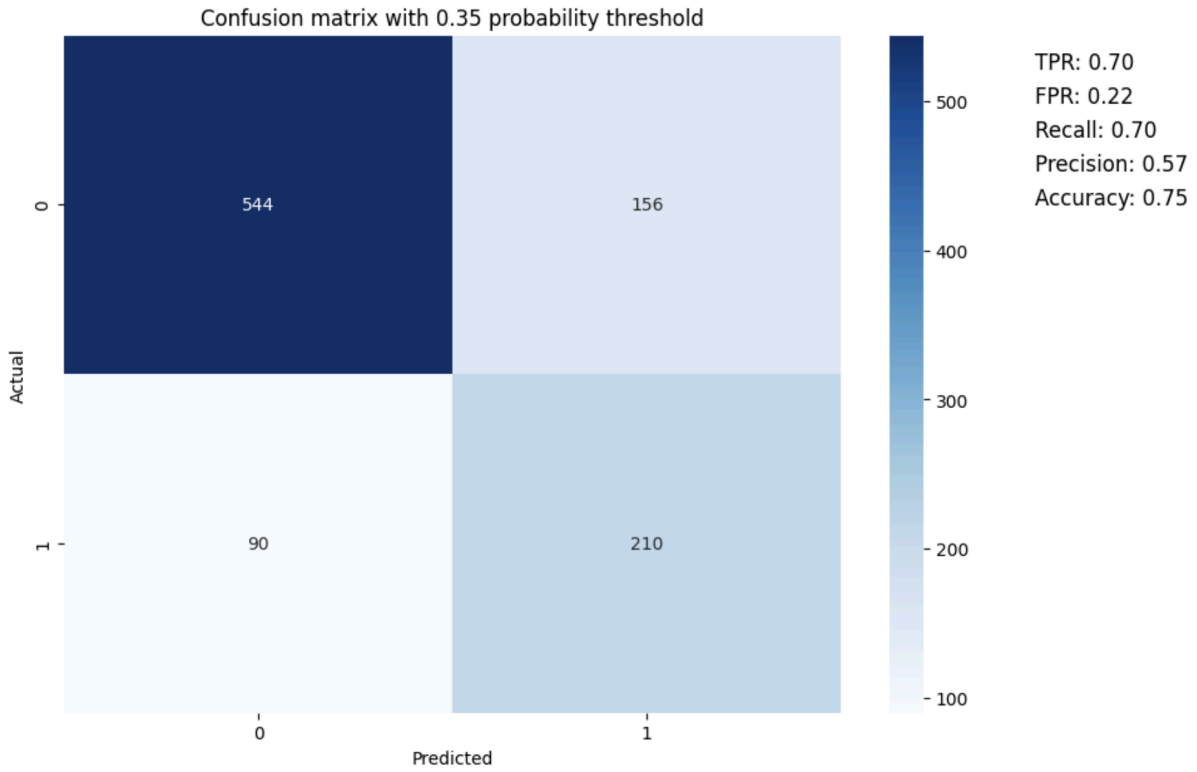
Varying the Probability Threshold

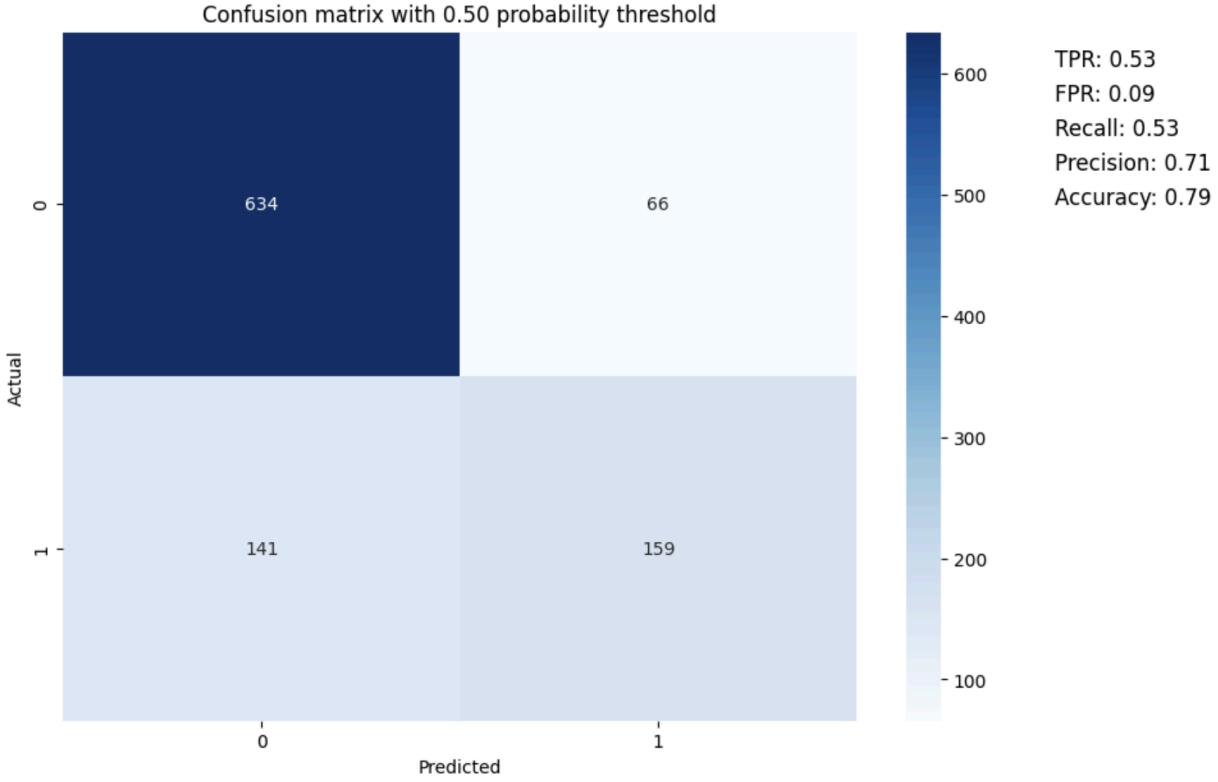
```
[83] prob_threshold = [0.20, 0.35, 0.50]
```

```
results = []
for prob_thres in prob_threshold:
  y_prob = model.predict_proba(X)[:,1]
  y_pred = (y_prob >= prob_thres).astype(int)
  conf_matrix = confusion_matrix(y, y_pred)
  # Plotting the confusion matrix
  plt.figure(figsize=(8, 6))
  sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues')
  plt.title(f"Confusion matrix with {prob_thres:.2f} probability threshold")
  plt.xlabel('Predicted')
  plt.ylabel('Actual')
  plt.show()
  #To calculate the TPR and FPR
  TN, FP, FN, TP = conf_matrix.ravel()

  # Calculating TPR and FPR
  TPR = TP / (TP + FN)
  FPR = FP / (FP + TN)

  # Display the results
  results.append({
      'Probability Threshold ':prob_thres,
      'True Positive Rate (TPR)': TPR,
      'False Positive Rate (FPR)': FPR
  })

results_df = pd.DataFrame(results)
```

We get the following Confusion matrix for the above model:


Confusion matrix with 0.20 probability threshold
421
279
42
258
Actual
0
1
Predicted
0
1
400
350
300
250
200
150
100
50
TPR: 0.86
FPR: 0.40
Recall: 0.86
Precision: 0.48
Accuracy: 0.68


Confusion matrix with 0.35 probability threshold
544
156
90
210
Actual
Predicted
0
1
500
400
300
200
100
TPR: 0.70
FPR: 0.22
Recall: 0.70
Precision: 0.57
Accuracy: 0.75


Confusion matrix with 0.50 probability threshold
634
66
141
159
Actual
Predicted
0
1
600
500
400
300
200
100
TPR: 0.53
FPR: 0.09
Recall: 0.53
Precision: 0.71
Accuracy: 0.79

1. Model at 0.20 Probability Threshold:

- **TPR (Recall)**: 0.86 – The model correctly identifies 86% of bad credit customers. This is high recall, meaning it's very good at identifying customers with bad credit.
- **FPR**: 0.40 – It misclassifies 40% of good credit customers as bad, which is high. These customers could be denied credit unfairly.
- **Precision**: 0.48 – Only 48% of customers classified as having bad credit are actually bad, meaning the model is quite inaccurate in flagging bad credit customers.
- **Accuracy**: 0.68 – Overall, this model correctly classifies 68% of cases.

Interpretation: This model has **high recall** for bad credit (catching most bad credit customers), but it comes at the cost of many false positives, meaning a large number of good credit customers are misclassified as bad. This might lead to many good customers being denied credit, which is not ideal.

2. Model at 0.35 Probability Threshold:

- **TPR (Recall)**: 0.70 – The model correctly identifies 70% of bad credit customers. This is a bit lower than the first model, but still reasonable.
- **FPR**: 0.22 – The model misclassifies only 22% of good credit customers as bad, which is much better than the previous model.
- **Precision**: 0.57 – 57% of the customers classified as having bad credit are actually bad. This is a better precision than the first model.
- **Accuracy**: 0.75 – Overall, this model correctly classifies 75% of cases.

Interpretation: This model strikes a **balance** between precision and recall. It has a lower false positive rate, meaning fewer good credit customers are wrongly classified as bad. It also maintains a reasonably good recall, catching 70% of bad credit customers, which is still quite good. This model is a good compromise between minimizing risky lending and not unfairly denying credit to good customers.

3. Model at 0.50 Probability Threshold:

- **TPR (Recall)**: 0.53 – The model correctly identifies only 53% of bad credit customers, meaning it misses almost half of the bad credit cases.

- **FPR**: 0.09 – The model misclassifies only 9% of good credit customers as bad, meaning it's very cautious about labeling good credit customers as bad.
- **Precision**: 0.71 – 71% of the customers classified as having bad credit are actually bad, which is the highest precision among the three models.
- **Accuracy**: 0.79 – Overall, this model correctly classifies 79% of cases.

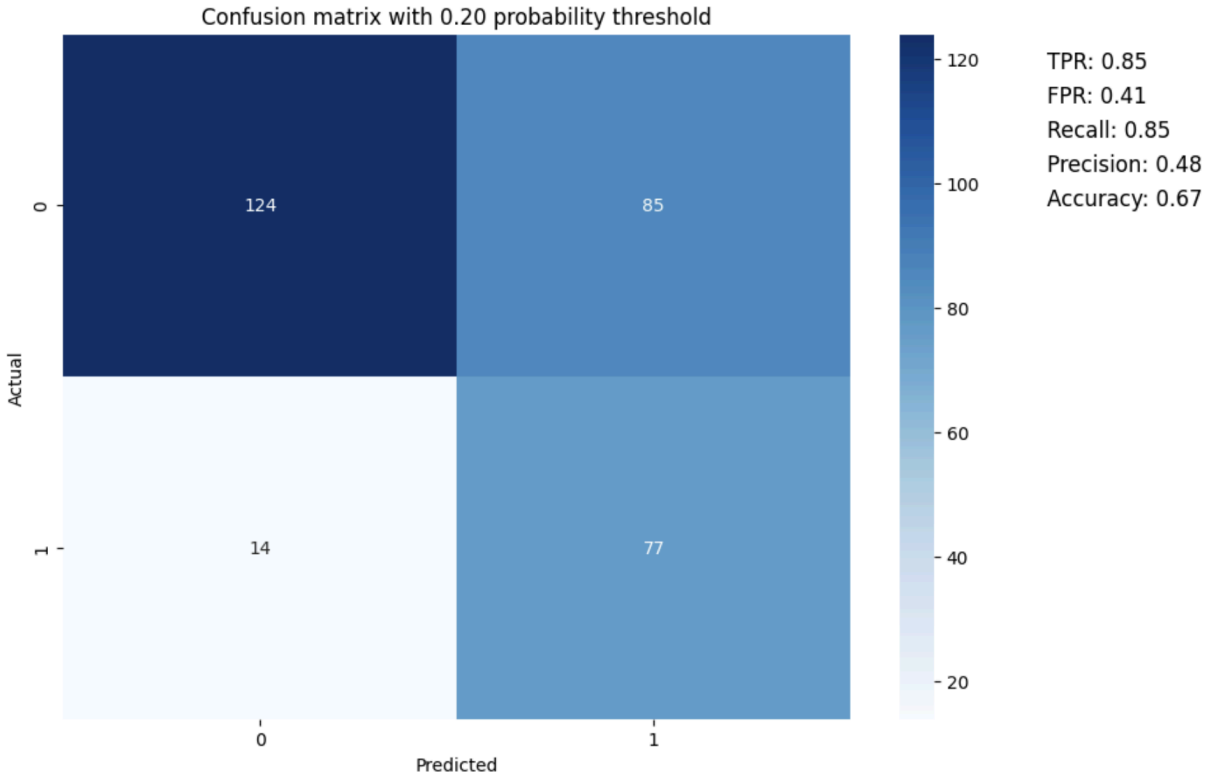
Interpretation: This model is very **conservative**, with high precision but low recall. While it is very good at not misclassifying good credit customers (low false positive rate), it misses many bad credit customers (low recall), which could be risky because those bad credit customers could be granted loans.
Since 1 is bad credit and 0 is good credit, the key priority is usually to **minimize false negatives (i.e., avoid misclassifying bad credit customers as good) while keeping false positives (denying good credit customers) reasonable**.
Best Model: The model with the 0.35 probability threshold is the most balanced for this use case. It correctly identifies 70% of bad credit customers (reasonably high recall). It keeps the false positive rate low (22% misclassification of good credit customers as bad), which means that while some good customers might be denied credit, it's a reasonable trade-off for financial safety. It has decent precision (57%), meaning that more than half of the flagged bad credit customers are actually bad credit.

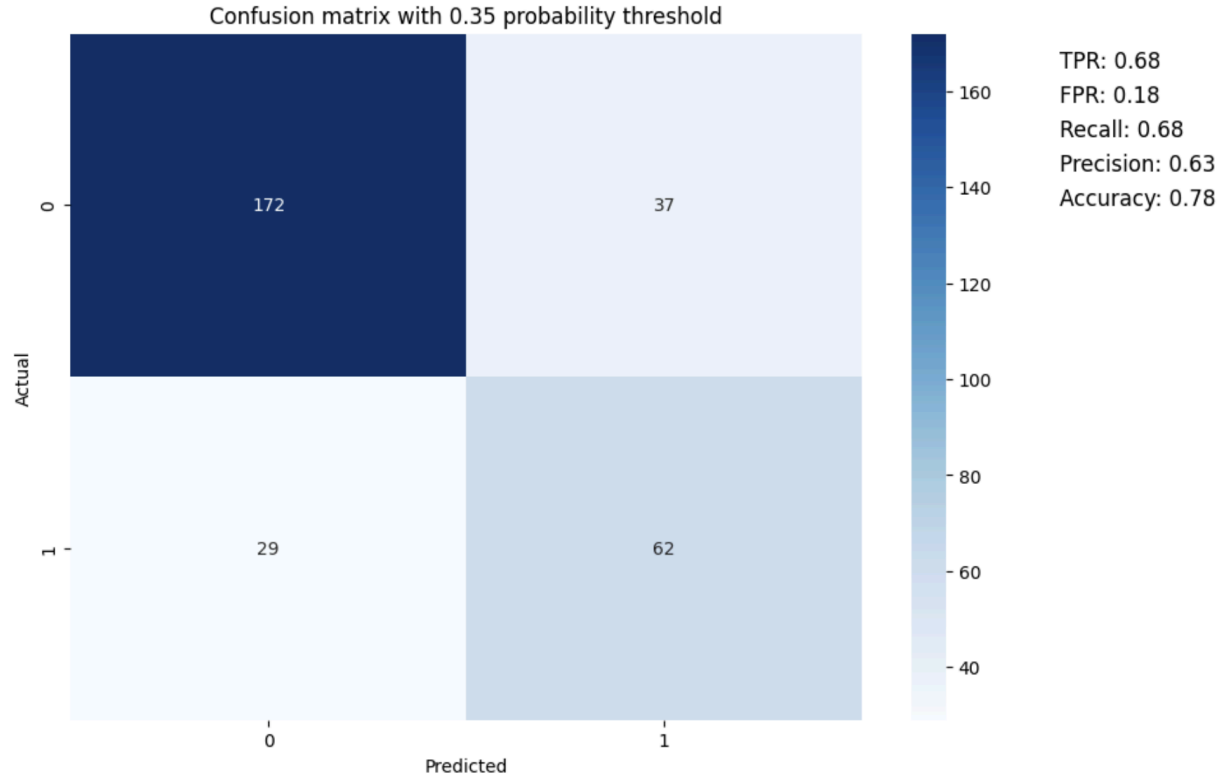
1 (b) Divide the dataset into training (70%) and test (30%) sets and repeat the above question and report the performance of these models on the test set.

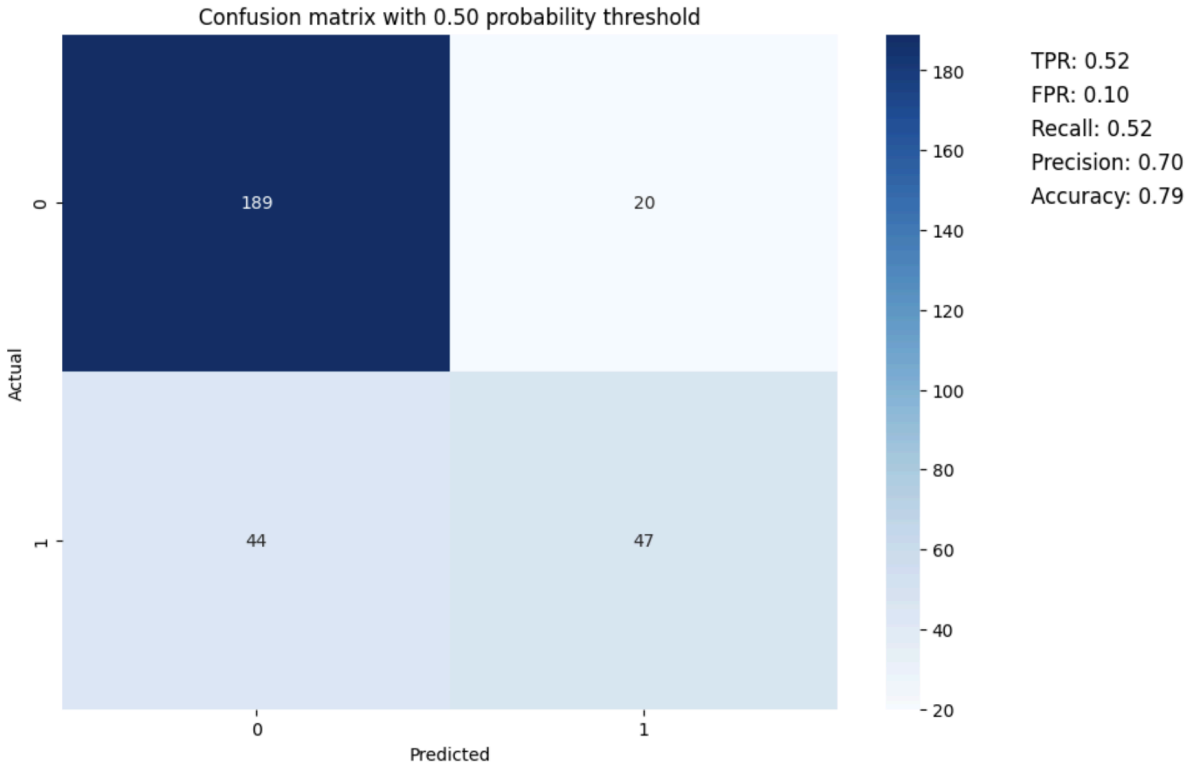
```
[64] # Splitting data into training and testing sets
     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
     # Logistic regression model
     model = LogisticRegression(solver ='lbfgs')
     model.fit(X_train, y_train)
```

```
▾ LogisticRegression
LogisticRegression()
```

Why Use `random_state`:
Reproducibility: When we split data into training and testing sets, it is done randomly. If we do not specify a `random_state`, each time you run the code, the data may be split differently. By setting a `random_state` (like `42`), we ensure that every time we run the code, the same split is used, which leads to consistent results across different runs.


Confusion matrix with 0.20 probability threshold
124
85
14
77
Actual
Predicted
0
1
120
100
80
60
40
20
TPR: 0.85
FPR: 0.41
Recall: 0.85
Precision: 0.48
Accuracy: 0.67


Confusion matrix with 0.35 probability threshold
172
37
29
62
Actual
Predicted
0
1
160
140
120
100
80
60
40
TPR: 0.68
FPR: 0.18
Recall: 0.68
Precision: 0.63
Accuracy: 0.78



Model at 0.20 Probability Threshold:

- **True Positive Rate (TPR/Recall)**: 0.85 – The model correctly identifies 85% of bad credit cases (positive class).
- **False Positive Rate (FPR)**: 0.41 – The model misclassifies 41% of good credit customers as bad, which is quite high.
- **Precision**: 0.48 – Only 48% of the customers classified as having bad credit are actually bad, meaning there is a lot of noise in the predictions for bad credit.
- **Accuracy**: 0.67 – Overall accuracy is 67%, but accuracy can be misleading in imbalanced datasets.

Interpretation:

- This model is **recall-focused**. It is good at catching most of the bad credit cases (high recall of 0.85), but it misclassifies many good credit customers

(high false positive rate of 0.41). The precision is quite low, which means that many predicted bad credit cases are false positives.
- **Downside**: Denying credit to many good customers due to the high false positive rate is a significant drawback.

Model at 0.35 Probability Threshold:

- **True Positive Rate (TPR/Recall)**: 0.68 – The model correctly identifies 68% of bad credit cases, lower than the previous model.
- **False Positive Rate (FPR)**: 0.18 – The model misclassifies only 18% of good credit customers as bad, which is a big improvement over the 0.20 threshold model.
- **Precision**: 0.63 – 63% of the customers classified as having bad credit are actually bad, meaning the model is more precise in identifying bad credit customers compared to the 0.20 threshold model.
- **Accuracy**: 0.78 – Accuracy has improved compared to the previous model.

Interpretation:

- This model provides a **balance between precision and recall**. The **false positive rate is much lower** (only 18%), so fewer good credit customers are misclassified as bad. The recall (68%) is still decent but lower than the 0.20 threshold model.
- **Advantage**: This model strikes a good balance between catching bad credit customers and not denying credit to too many good customers.

Model at 0.50 Probability Threshold:

- **True Positive Rate (TPR/Recall)**: 0.52 – The model correctly identifies only 52% of bad credit cases, which is quite low.
- **False Positive Rate (FPR)**: 0.10 – The model misclassifies only 10% of good credit customers as bad, meaning it's very cautious about labeling good credit customers as bad.
- **Precision**: 0.70 – 70% of the customers classified as having bad credit are actually bad, meaning the model is more accurate in predicting bad credit but at the cost of recall.
- **Accuracy**: 0.79 – Accuracy is slightly higher than the previous model, but the low recall is a concern.

Interpretation:

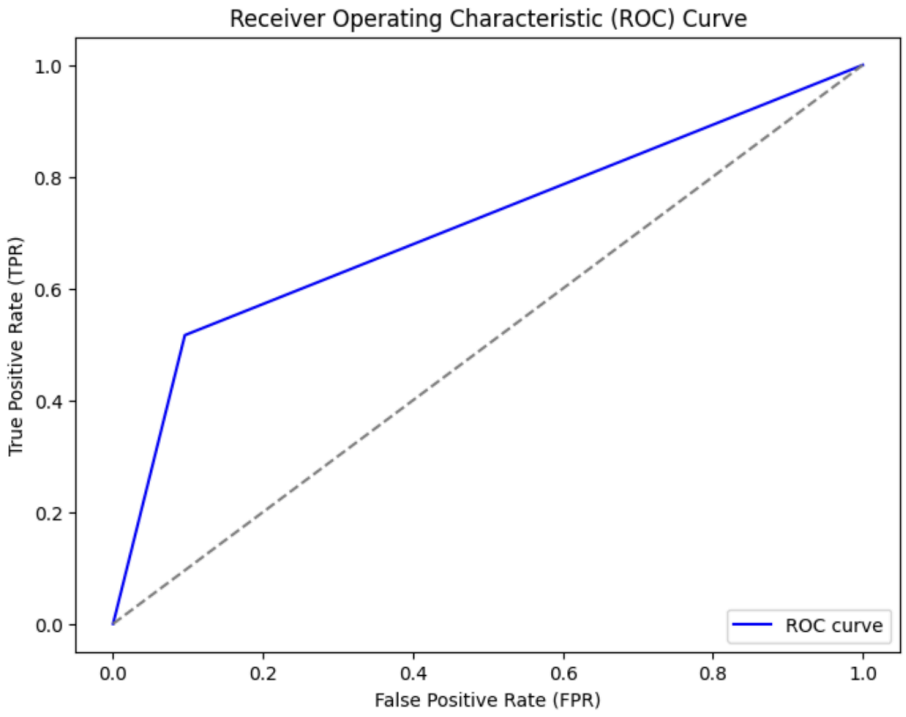
- This model is **precision-focused**, with the lowest false positive rate (0.10) and the highest precision (0.70). However, its recall is quite low (0.52), meaning it **misses a lot of bad credit customers**.
- **Downside**: While it minimizes the number of good credit customers denied credit, it misses many risky bad credit customers who might default.

The **model with the 0.35 threshold** seems to be the best choice because it provides a **balance between precision and recall**. It reduces the false positive rate while still catching a reasonable number of bad credit customers. This balance makes it a solid option for minimizing financial risk without excessively rejecting good credit customers.

Overall, a probability threshold of 0.35 is preferred for practical uses.

1 (c) Plot the ROC for a logistic model on a graph and compute the AUC. Explain the information conveyed by the ROC and the AUC metrics.

```
# Compute ROC curve
fpr, tpr, thresholds = roc_curve(y_test, y_pred)

# Plot ROC curve
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='blue', label='ROC curve')
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')  # Diagonal line for random guessing
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.show()
```




```
[67] # Calculate AUC
     auc = roc_auc_score(y_test, y_pred)
     print(f"Area Under the Curve (AUC): {auc:.4f}")
```

```
Area Under the Curve (AUC): 0.7104
```

ROC Curve Analysis:

- The ROC curve demonstrates the model's ability to differentiate between **fraudulent transactions (positive class)** and **legitimate transactions (negative class)**.
- The curve starts from (0,0) and approaches (1,1), and the area under this curve (AUC) gives us a numerical measure of the model's overall performance.

Key Metrics:

- **AUC (Area Under the Curve)**: **0.7104** (71.04%)

Interpretation of AUC:

- **AUC of 0.7104** means that the model has a **71.04% chance of correctly distinguishing between a randomly chosen fraud case and a randomly chosen legitimate case**.
- This score is **moderate to good**, indicating that the model performs reasonably well in detecting fraud, but there is room for improvement.

Model Performance:

- The ROC curve's shape shows that the model does a decent job of balancing the trade-off between **True Positive Rate (TPR)** and **False Positive Rate (FPR)**.
- However, the model does not perfectly approach the top-left corner, meaning that while it can distinguish fraud from legitimate cases, there are still some misclassifications.

2. (a) Fit classification tree, bagging and random forest models on the dataset and comment on the performance of these models. Do you think we are overestimating the performance of these models by fitting them on to the whole dataset? If so, state your reasons.

Classification Tree

```
[29] from sklearn.tree import DecisionTreeClassifier
     from sklearn.ensemble import BaggingClassifier, RandomForestClassifier
     from sklearn.tree import plot_tree
```

```
[35] X_scaled = scaler.fit_transform(X)

     # Convert back to DataFrame for consistency in feature names
     X_scaled_df = pd.DataFrame(X_scaled, columns=X.columns)
     X_scaled_df= X_scaled_df[significant_vars]
```

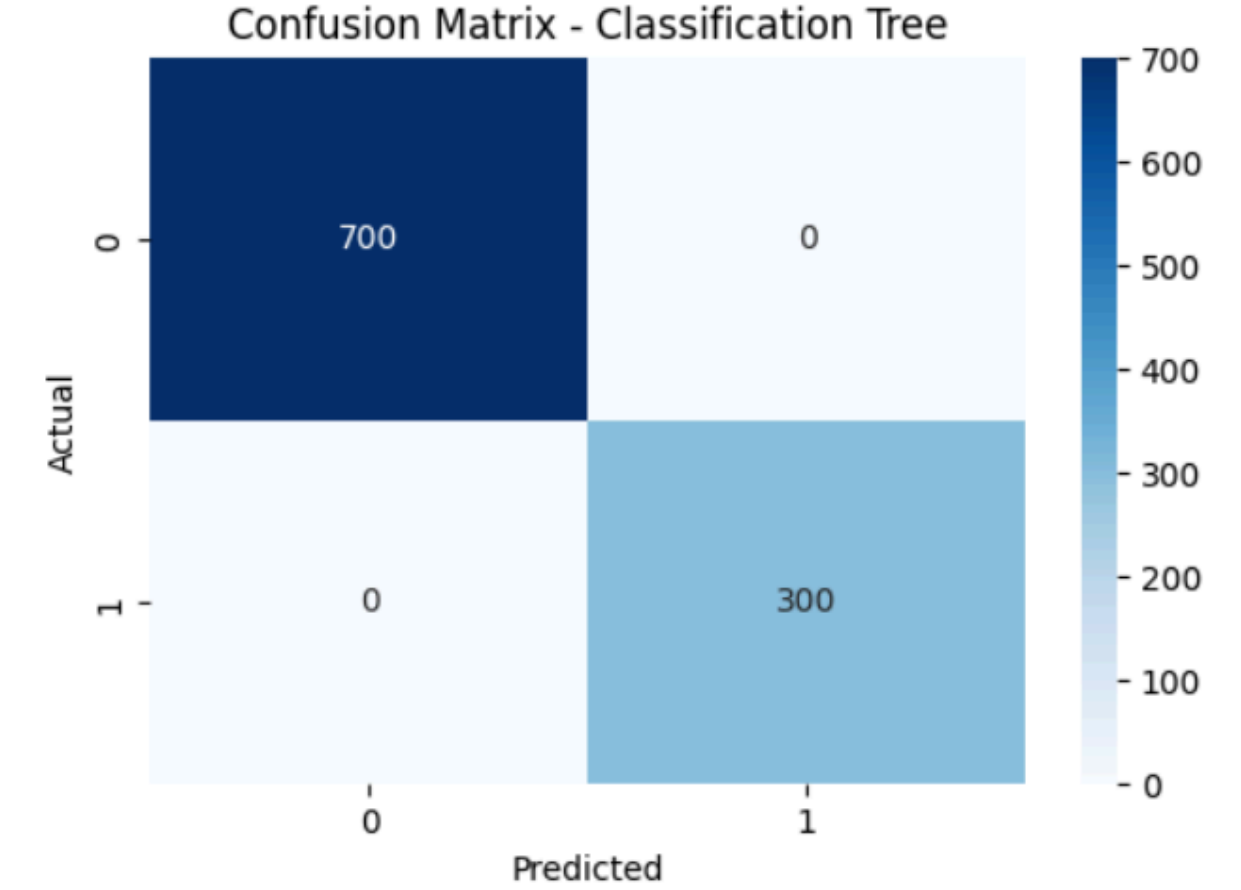
```
[36] tree_clf = DecisionTreeClassifier(criterion='gini',random_state=42,min_impurity_decrease=0.00)
     tree_clf.fit(X_scaled_df, y)
     y_pred_tree = tree_clf.predict(X_scaled_df)
     accuracy_tree = accuracy_score(y, y_pred_tree)
```

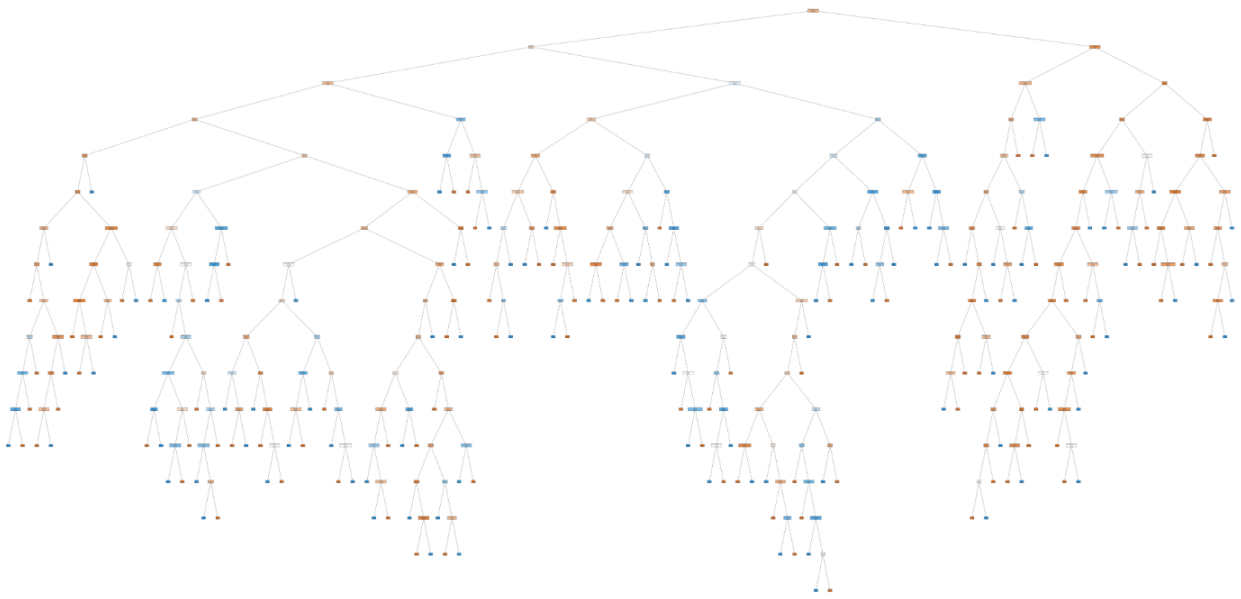
Bagging Model

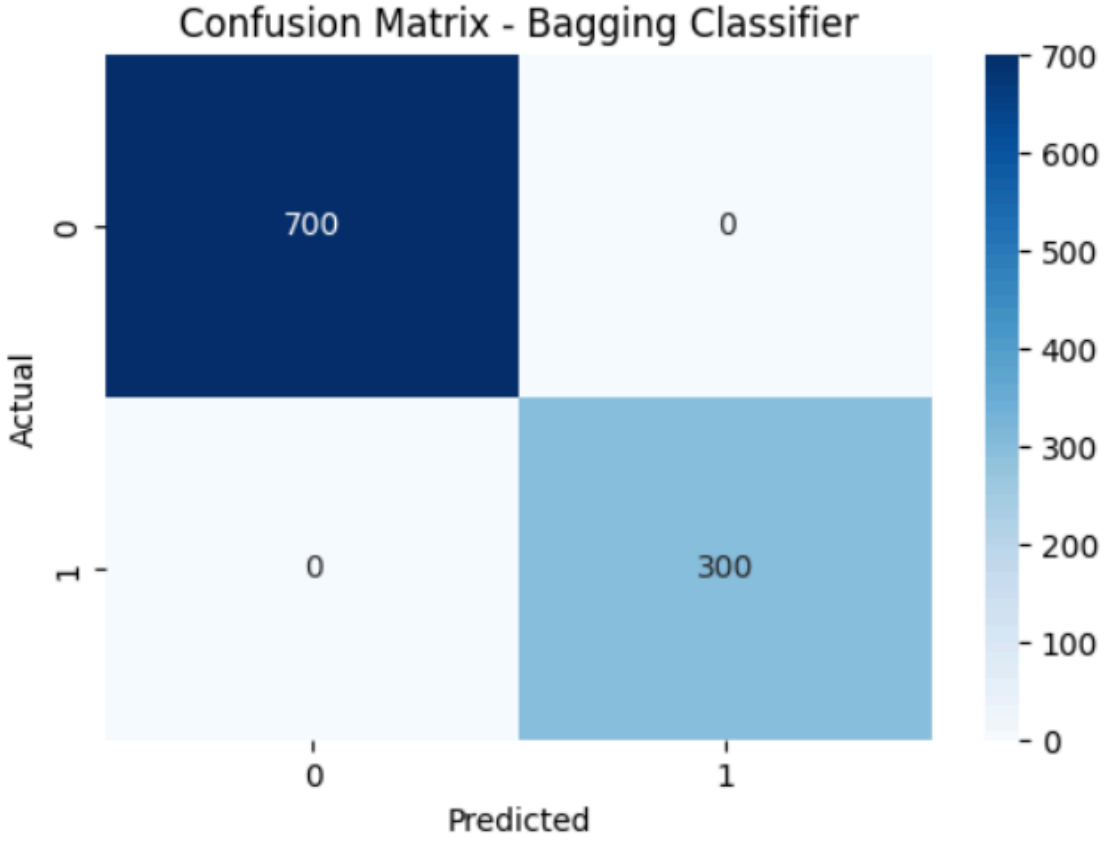
```
[38] bagging_clf = BaggingClassifier(
         estimator=DecisionTreeClassifier(criterion='gini', min_impurity_decrease=0.002, random_state=42),
         n_estimators=100,
         random_state=42
     )
     bagging_clf.fit(X_scaled_df, y)
     y_pred_bagging = bagging_clf.predict(X_scaled_df)
     accuracy_bagging = accuracy_score(y, y_pred_bagging)
```

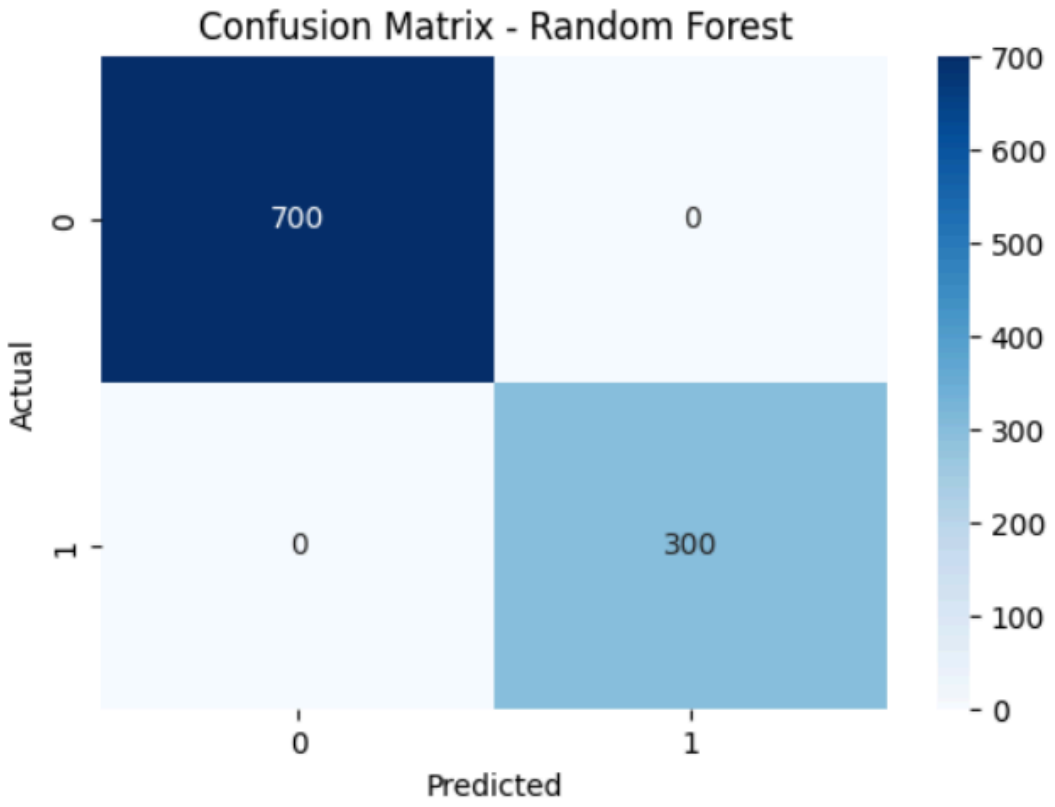
Random Forest

```
[39] rf_clf = RandomForestClassifier(n_estimators=100,criterion='gini', random_state=42)

     rf_clf.fit(X_scaled_df, y)
     y_pred_rf = rf_clf.predict(X_scaled_df)
     accuracy_rf = accuracy_score(y, y_pred_rf)
```


Confusion Matrix - Classification Tree
700
0
0
300
Actual
0
1
0
1
Predicted
700
600
500
400
300
200
100
0




Confusion Matrix - Bagging Classifier
700
0
0
300
Actual
0
1
Predicted
0
1
700
600
500
400
300
200
100
0



	Model	Accuracy	Precision	Recall	FPR	TPR
0	Classification Tree	1.0	1.0	1.0	0.0	1.0
1	Bagging Classifier	1.0	1.0	1.0	0.0	1.0
2	Random Forest	1.0	1.0	1.0	0.0	1.0

Comments on fitting the Entire Dataset

We are overestimating the performance of these models.

1. When we train models on the entire dataset without any validation or test set, the models have the opportunity to learn the noise and idiosyncrasies of the dataset. This results in overfitting, where the models perform exceptionally well on the training data but may not generalize well to new, unseen data.
2. A crucial aspect of model performance is its ability to generalize to unseen data. By using the whole dataset for training, we have no independent data to test how the model performs in practice.
3. Performance metrics such as accuracy, precision, recall, and AUC are computed on the same data the model was trained on. This can create a **biased estimate** of the model's true performance, as the model has already seen this data and has likely optimized specifically for it.
4. In models like Random Forest and Bagging, the models may inherently

become more complex and adapt to the dataset entirely, which gives a false impression of high accuracy and performance.

(b) Split the dataset in two parts: training (70%) and test sets (30%). Fit the models on the training dataset and evaluate their performance on the test set. Which model would you choose and why?

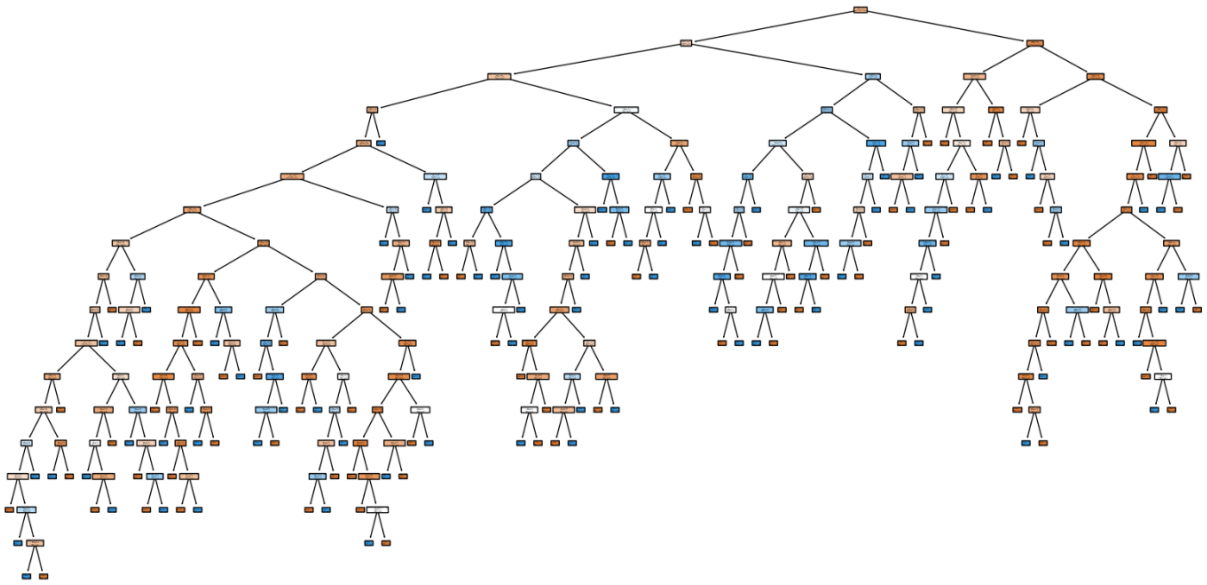
After splitting the dataset into a training set (70%) and a test set (30%), we evaluated the three models—Classification Tree, Bagging, and Random Forest—on the test set.

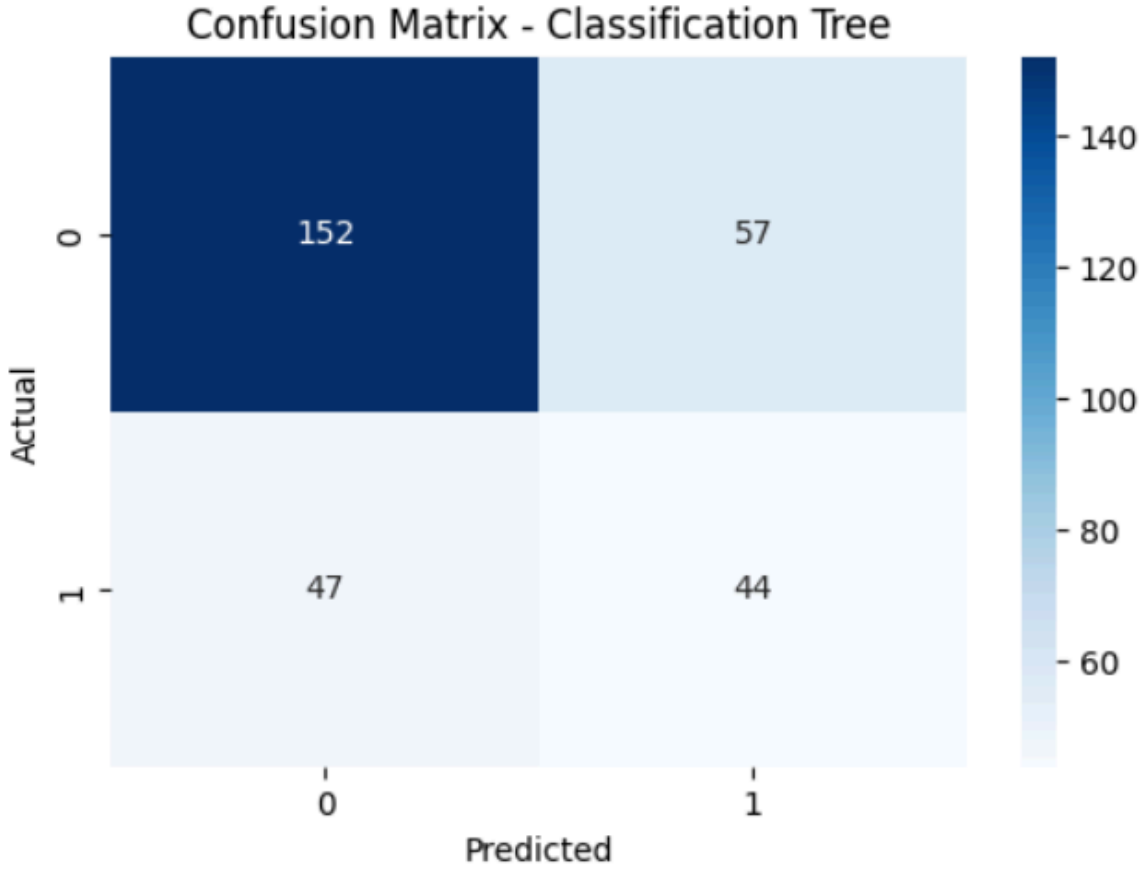
Splitting the Data

```
[41] # Split the data into training (70%) and testing sets (30%)
     X_train, X_test, y_train, y_test = train_test_split(X_scaled_df, y, test_size=0.3, random_state=42)

     # 1. Classification Tree on Training Data

     tree_clf.fit(X_train, y_train)
     y_pred_tree_test = tree_clf.predict(X_test)
     accuracy_tree_test = accuracy_score(y_test, y_pred_tree_test)

     # 2. Bagging Classifier on Training Data
     bagging_clf.fit(X_train, y_train)
     y_pred_bagging_test = bagging_clf.predict(X_test)
     accuracy_bagging_test = accuracy_score(y_test, y_pred_bagging_test)

     # 3. Random Forest on Training Data
     rf_clf.fit(X_train, y_train)
     y_pred_rf_test = rf_clf.predict(X_test)
     accuracy_rf_test = accuracy_score(y_test, y_pred_rf_test)
```

We fit three models—Classification Tree, Bagging, and Random Forest—on the entire dataset using all available features.

Classification Tree:

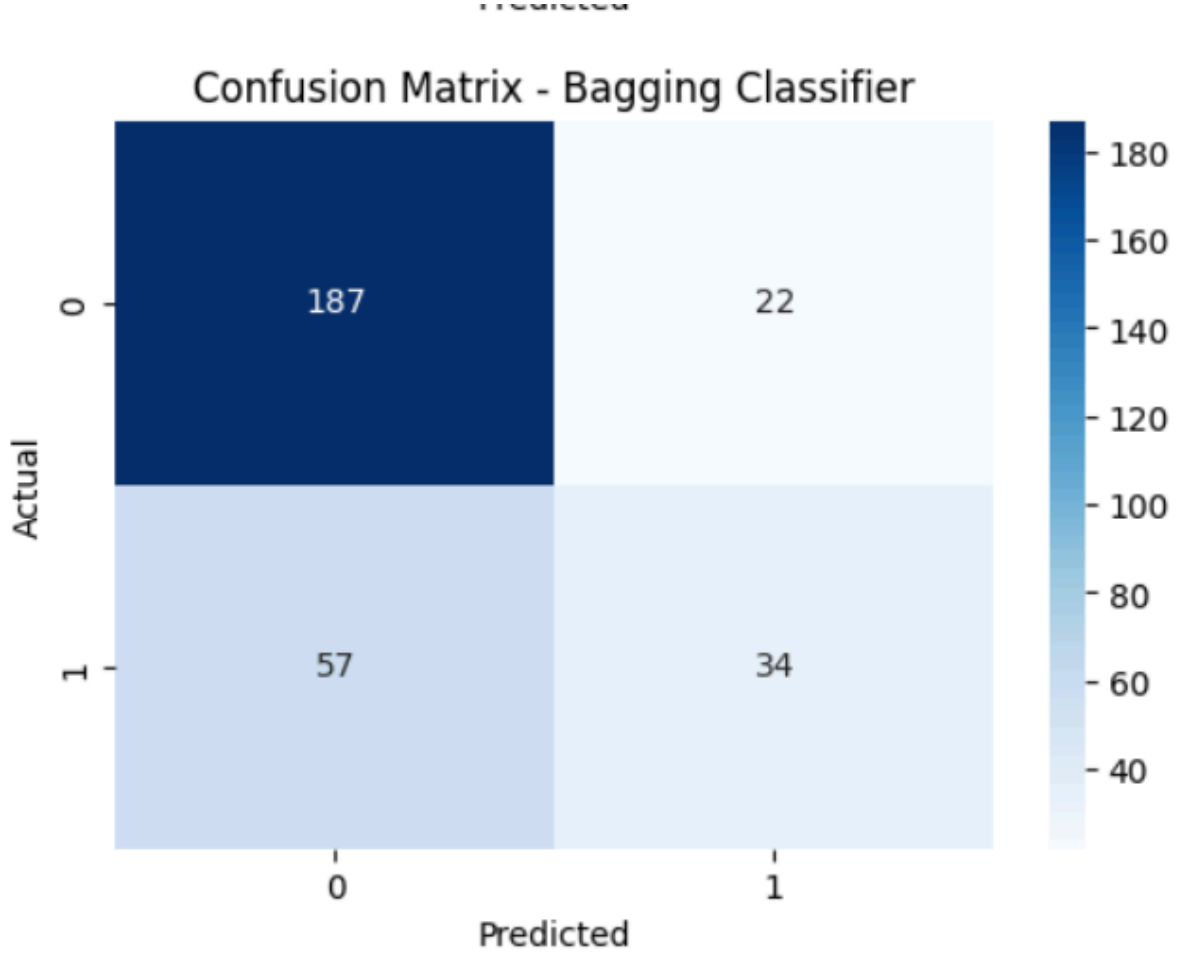




- **Accuracy**: 63.53%
- **Precision**: 0.45
- **Recall**: 0.48
- **False Positive Rate (FPR)**: 0.27

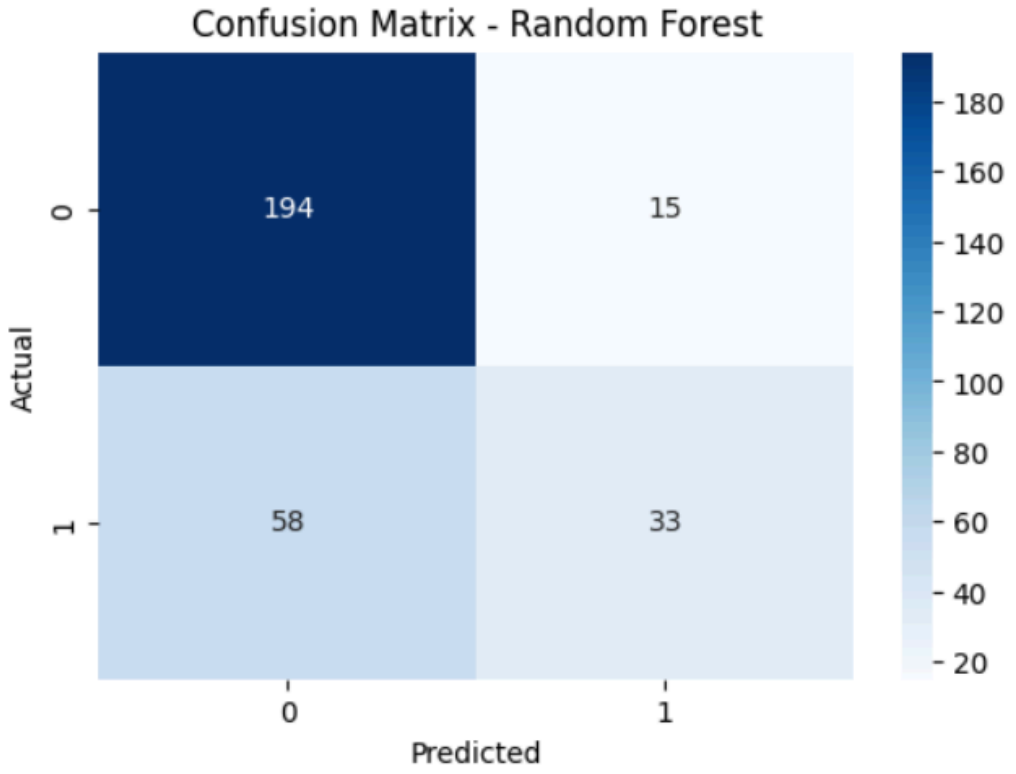
- **True Positive Rate (TPR)**: 0.48
- **AUC**: 0.6054

Bagging Classifier:



- **Accuracy**: 73.67%
- **Precision**: 0.61
- **Recall:** 0.37
- **FPR**: 0.10
- **TPR**: 0.37
- **AUC**: 0.7678

Random Forest:

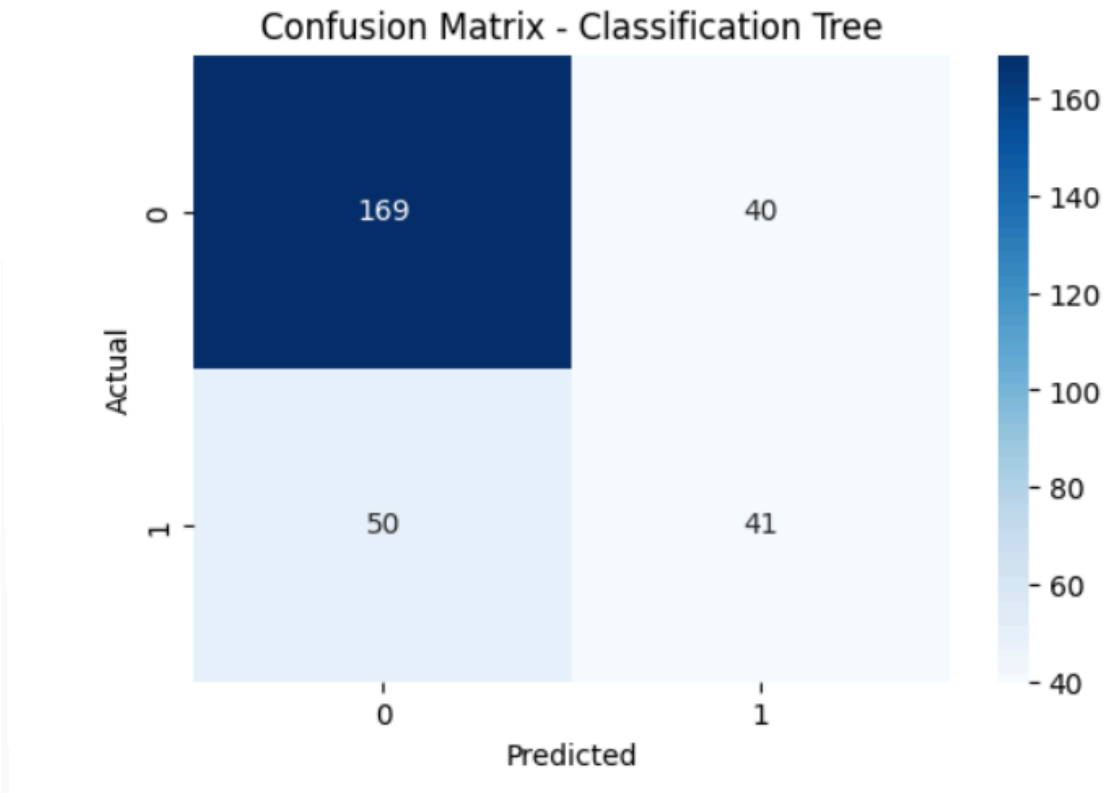


	Model	Accuracy	Precision	Recall	FPR	TPR
0	Classification Tree	0.653333	0.435644	0.483516	0.272727	0.483516
1	Bagging Classifier	0.736667	0.607143	0.373626	0.105263	0.373626
2	Random Forest	0.756667	0.687500	0.362637	0.071770	0.362637

- **Accuracy**: 75.67%
- **Precision**: 0.69
- **Recall**: 0.36
- **FPR**: 0.07
- **TPR**: 0.36
- **AUC**: 0.7931

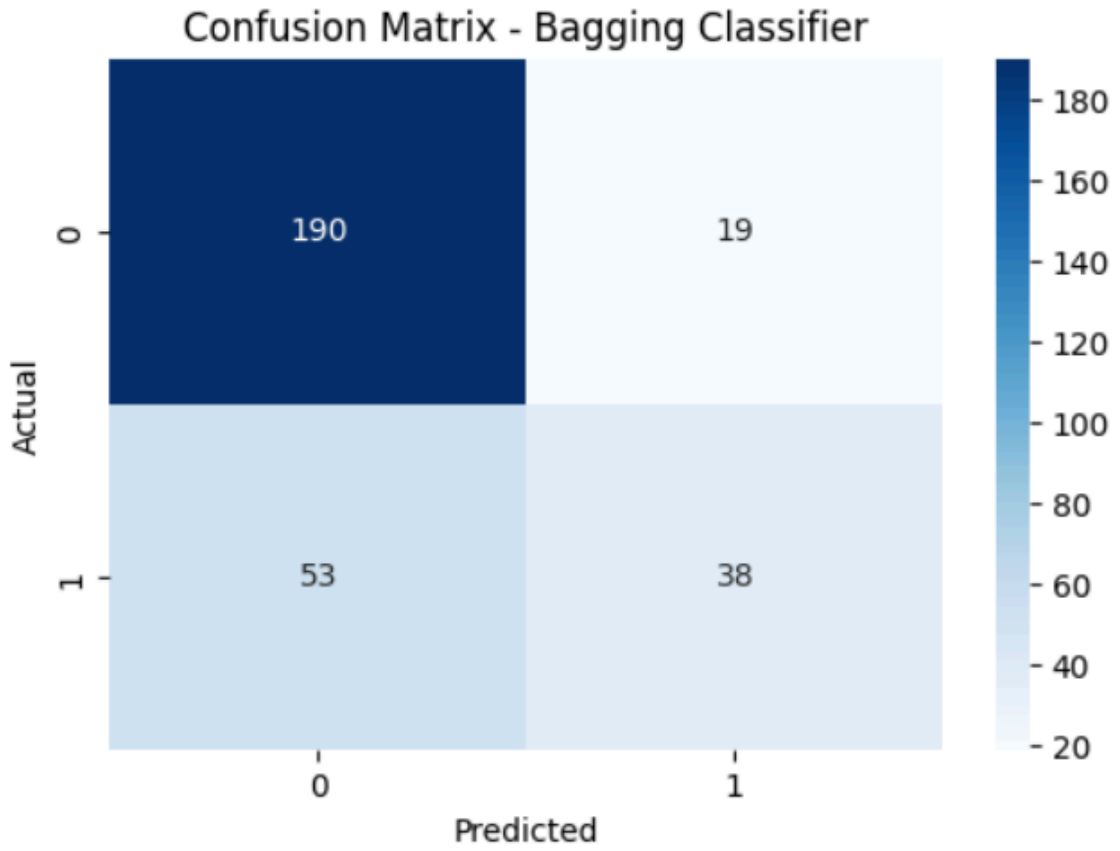
Now ,We fit three models—Classification Tree, Bagging, and Random Forest—on the entire dataset using specific features identified earlier.

Classification Tree:



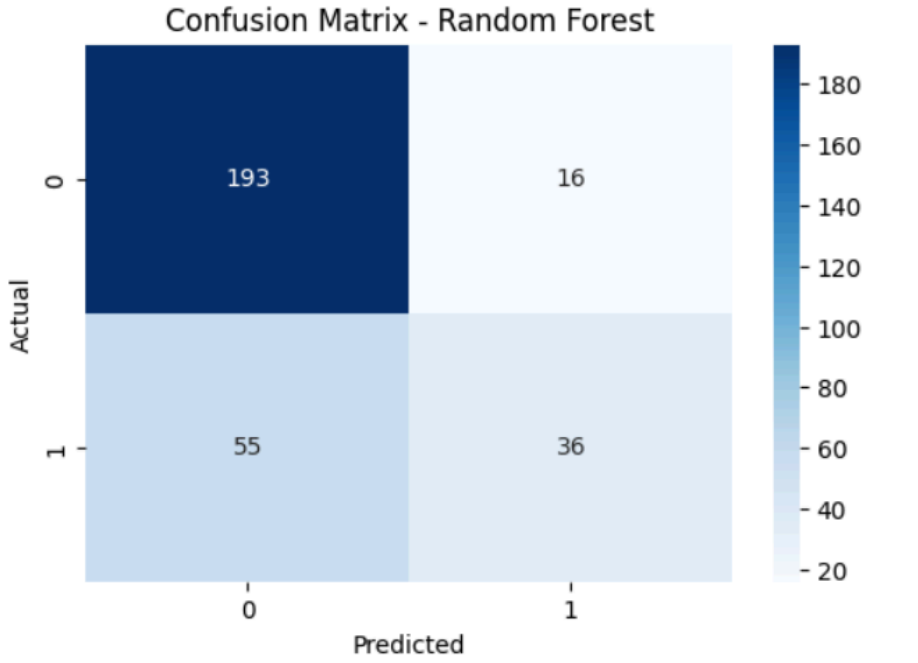
- **Accuracy**: 70%
- **Precision**: 0.5
- **Recall**: 0.45
- **False Positive Rate (FPR)**: 0.19
- **True Positive Rate (TPR)**: 0.45
- **AUC**: 0.6296

Bagging Classifier:



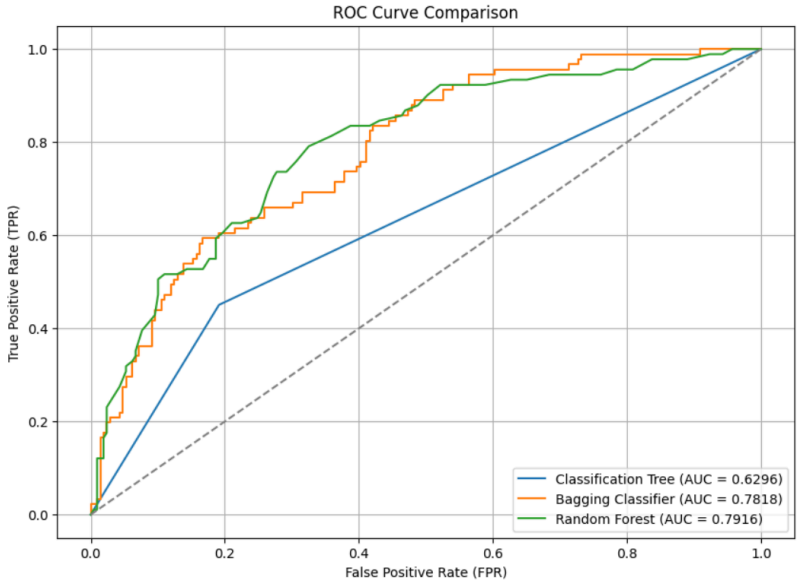
- **Accuracy**: 73.67%
- **Precision**: 0.76
- **Recall:** 0.42
- **FPR**: 0.09
- **TPR**: 0.42
- **AUC**: 0.7818

Random Forest:



	Model	Accuracy	Precision	Recall	FPR	TPR
0	Classification Tree	0.700000	0.506173	0.450549	0.191388	0.450549
1	Bagging Classifier	0.760000	0.666667	0.417582	0.090909	0.417582
2	Random Forest	0.763333	0.692308	0.395604	0.076555	0.395604

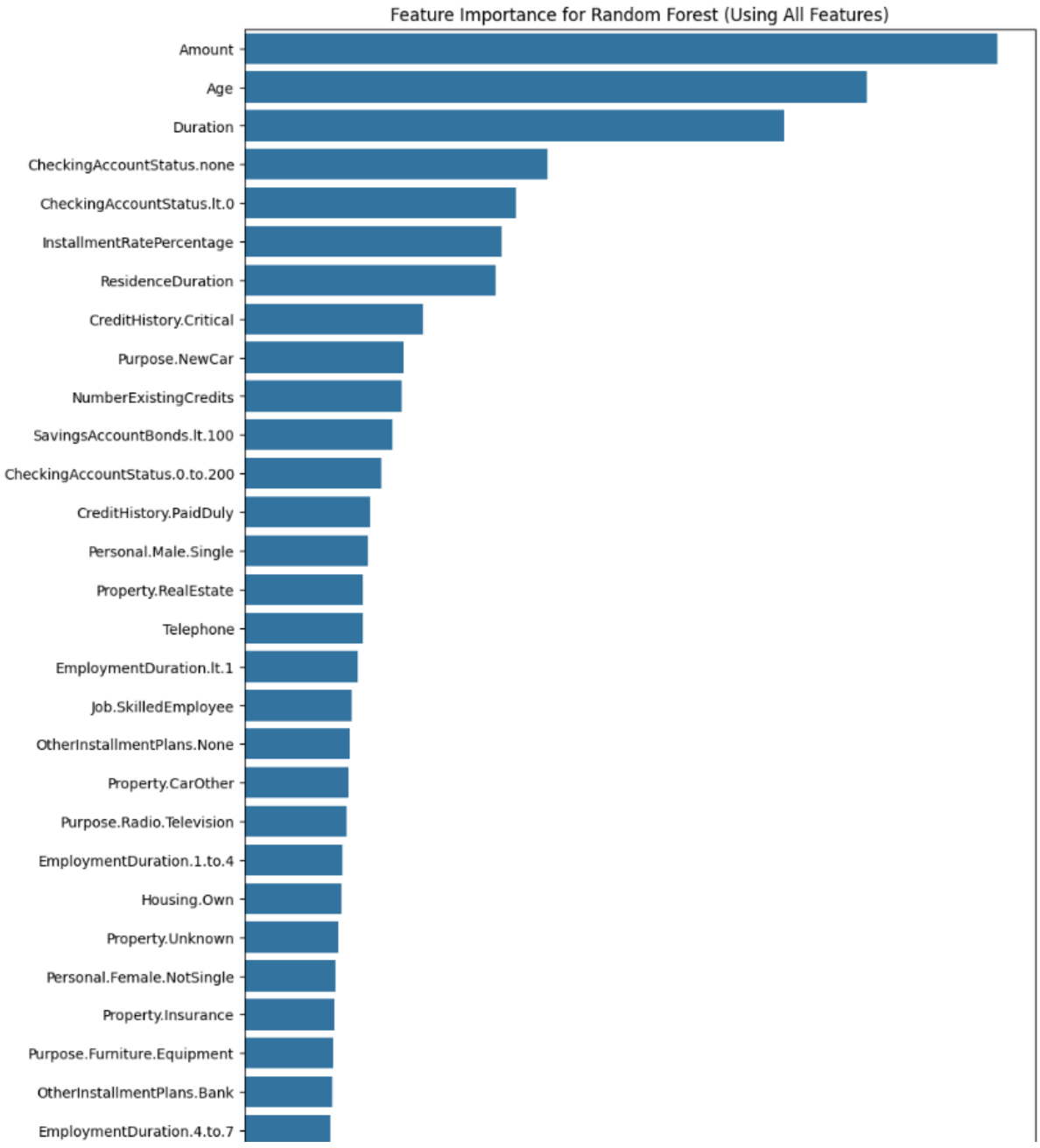
- **Accuracy**: 73.67%
- **Precision**: 0.69
- **Recall:** 0.39
- **FPR**: 0.07
- **TPR**: 0.39
- **AUC**: 0.79

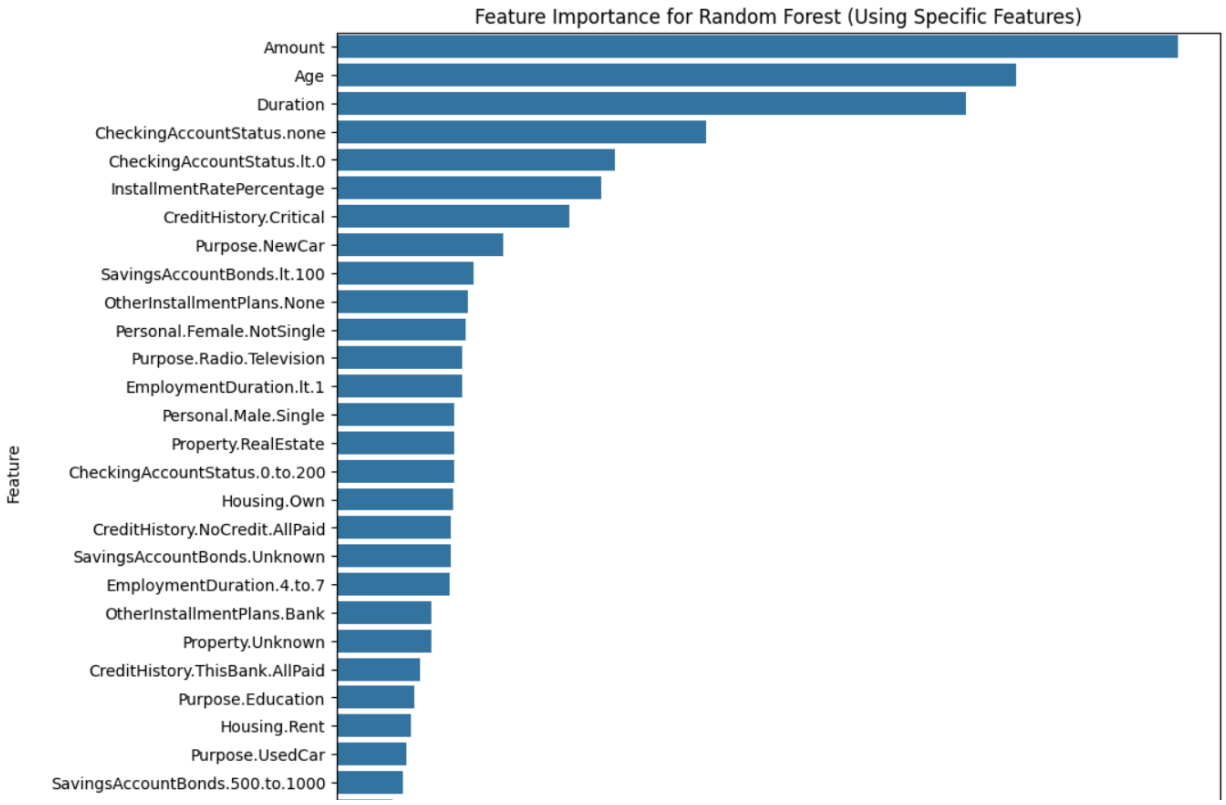



```
Classification Tree AUC: 0.6296
Bagging Classifier AUC: 0.7818
Random Forest AUC: 0.7916
```

2(c) Feature Importance for Random Forest

```
#######################################
# 2. (c) Feature Importance for Random Forest
#######################################

# Get feature importances from the Random Forest model
importances = rf_clf.feature_importances_
feature_names = X_train.columns
feature_importance_df = pd.DataFrame({'Feature': feature_names, 'Importance': importances})

# Sort the feature importances
feature_importance_df = feature_importance_df.sort_values(by='Importance', ascending=False)
print(feature_importance_df)
# Plot feature importances
plt.figure(figsize=(10, 30))
sns.barplot(x='Importance', y='Feature', data=feature_importance_df)
plt.title('Feature Importance for Random Forest (Using All Features)')
plt.show()
```


Feature Importance for Random Forest (Using All Features)
Amount
Age
Duration
CheckingAccountStatus.none
CheckingAccountStatus.lt.0
InstallmentRatePercentage
ResidenceDuration
CreditHistory.Critical
Purpose.NewCar
NumberExistingCredits
SavingsAccountBonds.lt.100
CheckingAccountStatus.0.to.200
CreditHistory.PaidDuly
Personal.Male.Single
Property.RealEstate
Telephone
EmploymentDuration.lt.1
Job.SkilledEmployee
OtherInstallmentPlans.None
Property.CarOther
Purpose.Radio.Television
EmploymentDuration.1.to.4
Housing.Own
Property.Unknown
Personal.Female.NotSingle
Property.Insurance
Purpose.Furniture.Equipment
OtherInstallmentPlans.Bank
EmploymentDuration.4.to.7



The Random Forest model was used to rank the predictors based on their importance. The top predictors were:

1. **Duration:** The most important feature.
2. **Amount:** Significant influence on the classification outcome.
3. **Age**: Strongly impacted the results.
4. **Job Type:** Moderately important.

Visualization
A bar chart showing the relative importance of the top features such as **Duration** and **Amount** was plotted, showing their significance in the prediction process.

2(d) Comparison with Logistic Regression

In Question 1, we evaluated a logistic regression model using different probability thresholds. The best model at a threshold of 0.35 had the following performance:

Logistic Regression Model (Threshold = 0.35):

-Confusion Matrix:
  - True Positives (TP): 62
  - True Negatives (TN): 172
  - False Positives (FP): 37
  - False Negatives (FN): 29

- Accuracy: 78%
- **Precision:** 0.63
- Recall: 0.68
- FPR: 0.18
- TPR: 0.68
- AUC: 0.7104

Comparison:

Logistic Regression: The logistic regression model had a better recall (0.68) but lower precision than Random Forest. It also had a lower AUC (0.7104).
- **Random Forest:** While the Random Forest had a slightly lower recall, its AUC of 0.7962 was the highest, and its precision and accuracy metrics were superior, making it the better choice for this problem.

ROC Curve Comparison
The **ROC** curves for the models were compared, showing the following **AUC** values:
- **Classification Tree**: AUC = 0.6296
-**Bagging Classifier:** AUC = 0.7818
- **Random Forest:** AUC = 0.7962
- **Logistic Regression:** AUC = 0.7104

Analysis
The **Random Forest** outperformed all models, including the logistic regression, based on AUC and general performance metrics. Thus, it is the recommended model for predicting credit risk.

Conclusion:
- The Random Forest model achieved the best performance with the highest AUC (0.79) and the most balanced performance metrics across accuracy,

precision, and recall. The Random Forest outperforms the Classification Tree and Bagging models in terms of both accuracy and robustness.
- However, fitting models on the entire dataset overestimates their performance. To provide more reliable model performance estimates, we evaluated the models using a train-test split.
*******in question 2,we also tried the different loss function for the trees like standard loss function and gini loss and tried different max loss values also to try to control height of the tree.nut since no of data points are under 1k,model was not giving good values for max loss threshold value of even 0.01.***

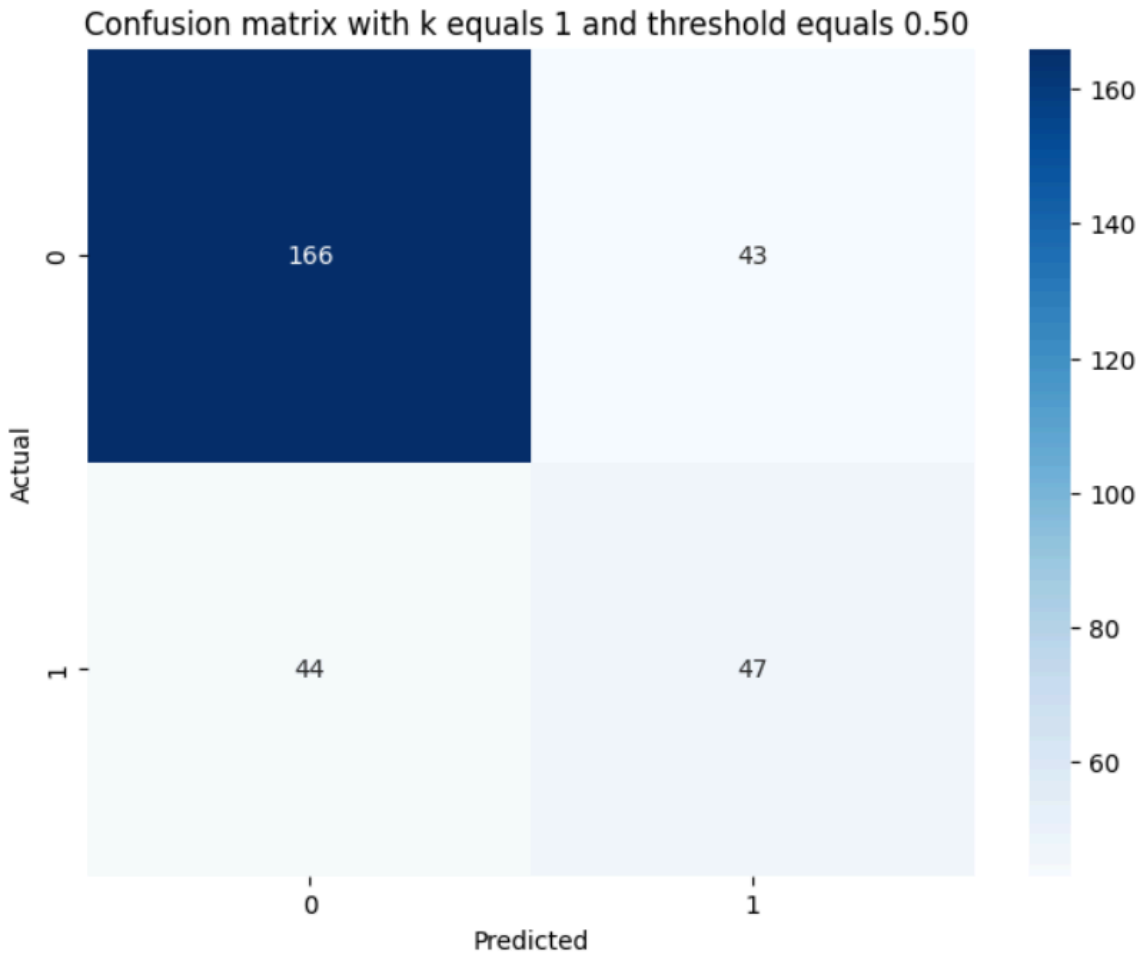
3. (a) Standardize your predictors and fit KNN classifier with K equal to 1, 3, 5, 10, respectively. Evaluate the performance of these models on the test set.

Standardizing Predictors and Fitting KNN Classifier

We standardized the predictors and fit KNN classifiers with different values of K (1, 3, 5, 10). The performance of these models was evaluated on the test set.

A)With all features

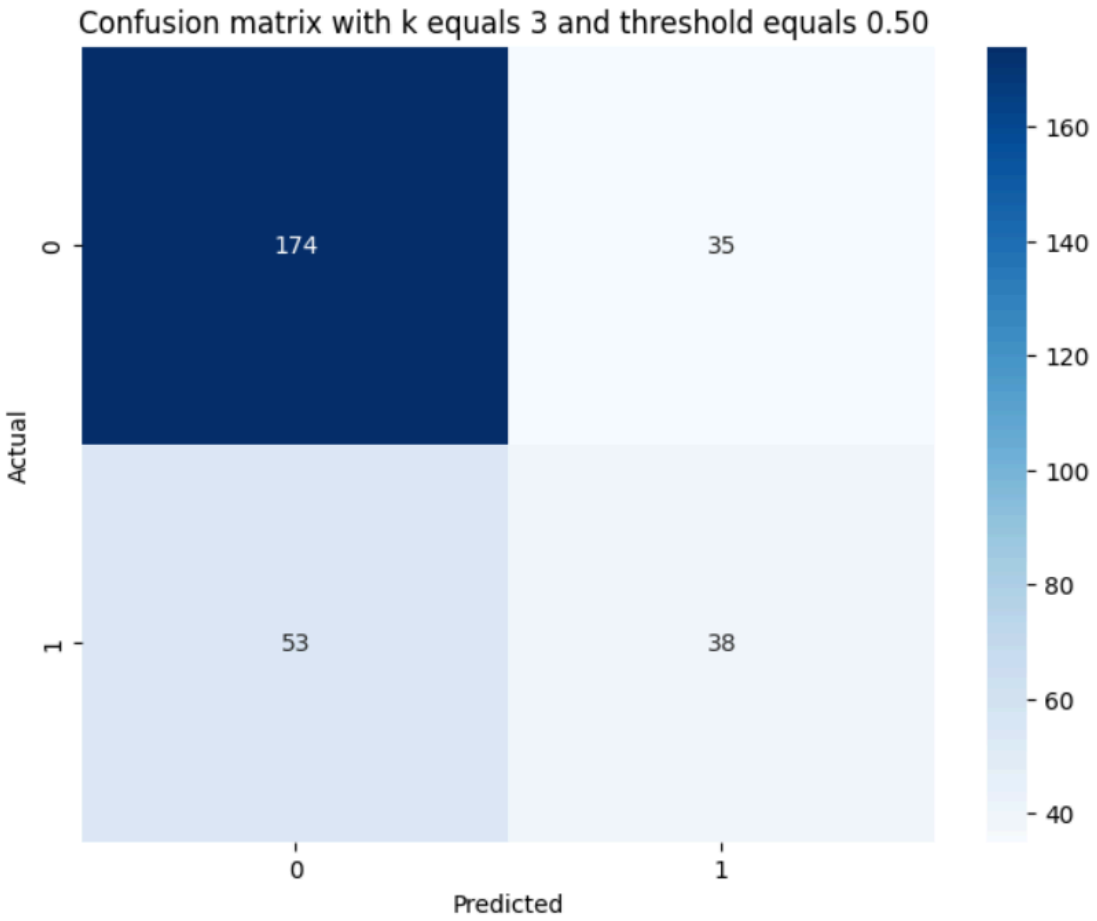
KNN Classifier with K = 1



- **Confusion Matrix:**
  - **True Positives (TP):** 47
  - **True Negatives (TN):** 166
  - **False Positives (FP):** 43
  - **False Negatives (FN):** 44

- **Accuracy:** 71.00%
- **AUC:** 0.655

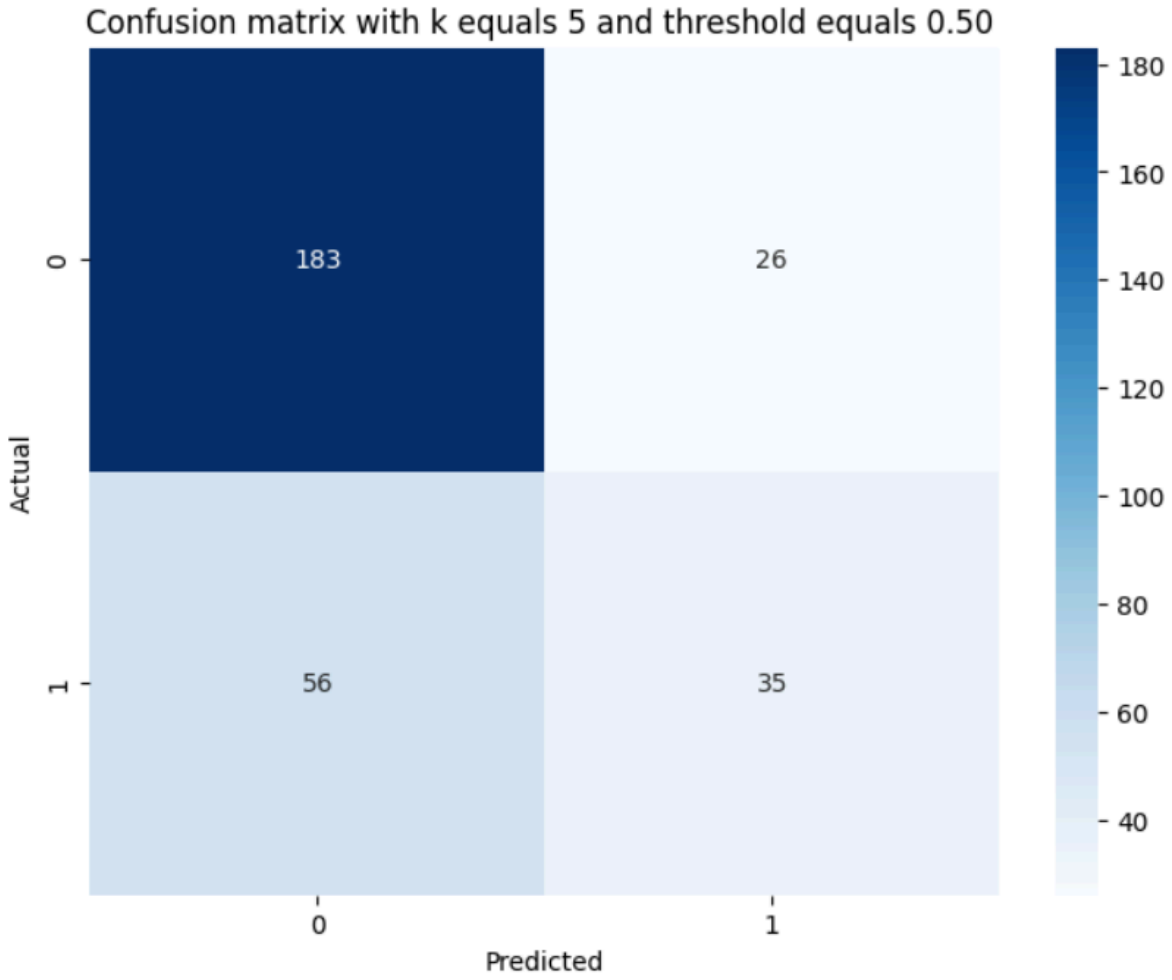
KNN Classifier with K = 3



- Confusion Matrix:
  - True Positives (TP): 38
  - True Negatives (TN): 174
  - False Positives (FP): 35
  - False Negatives (FN): 53

- Accuracy: 70.88%
- AUC: 0.687

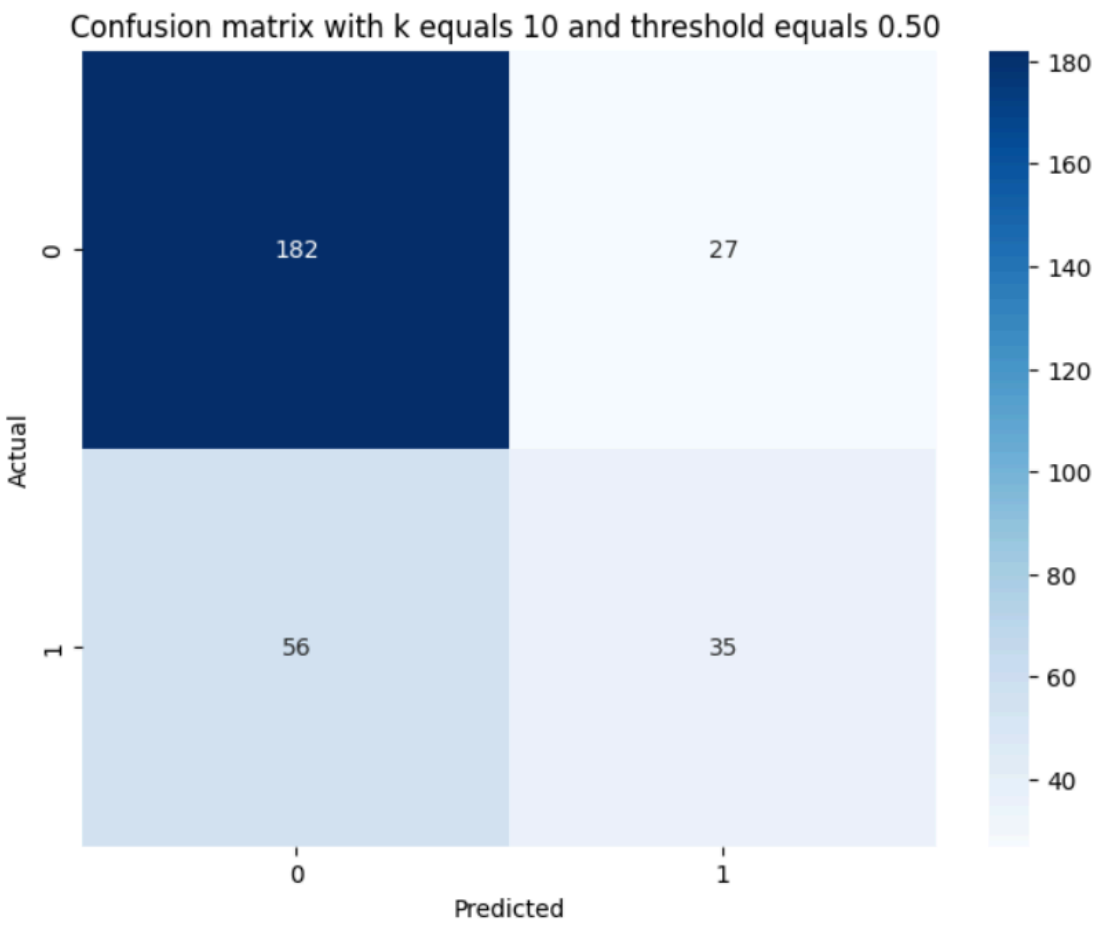
KNN Classifier with K = 5



- Confusion Matrix:
  - True Positives (TP): 35
  - True Negatives (TN): 183
  - False Positives (FP): 26
  - False Negatives (FN): 56

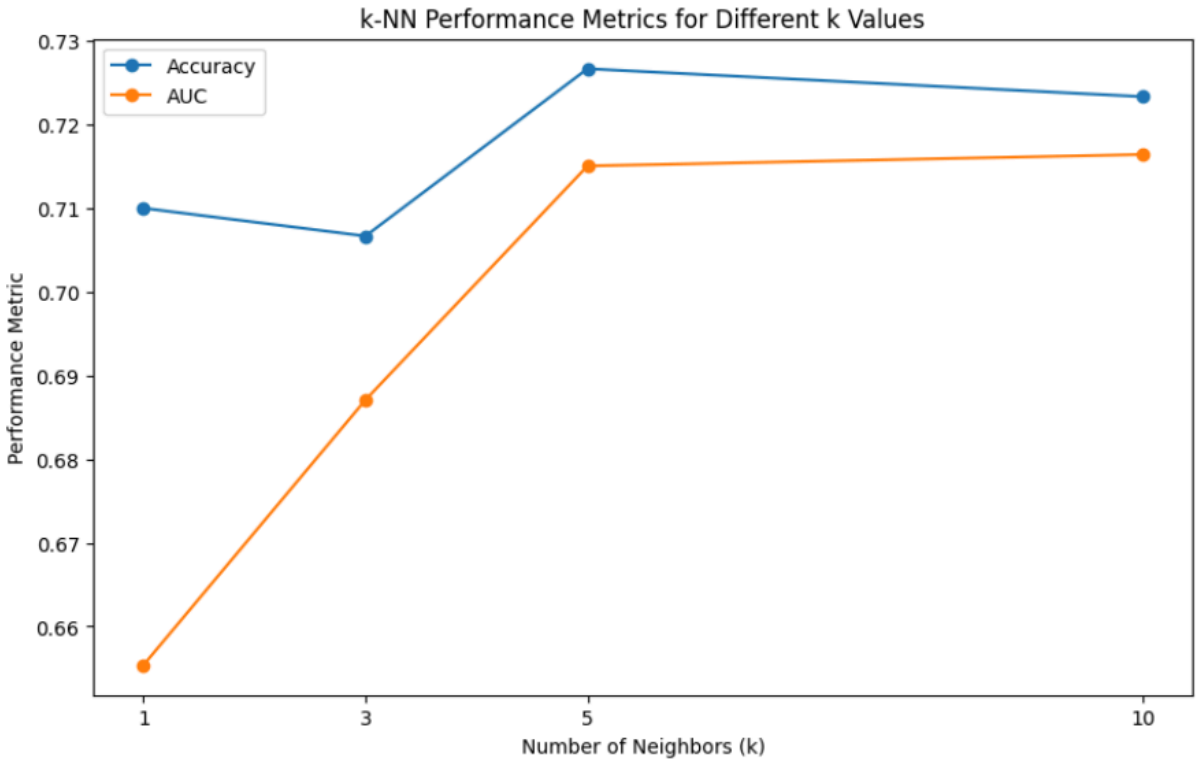
- Accuracy: 72.66%
- AUC: 0.715

KNN Classifier with K = 10



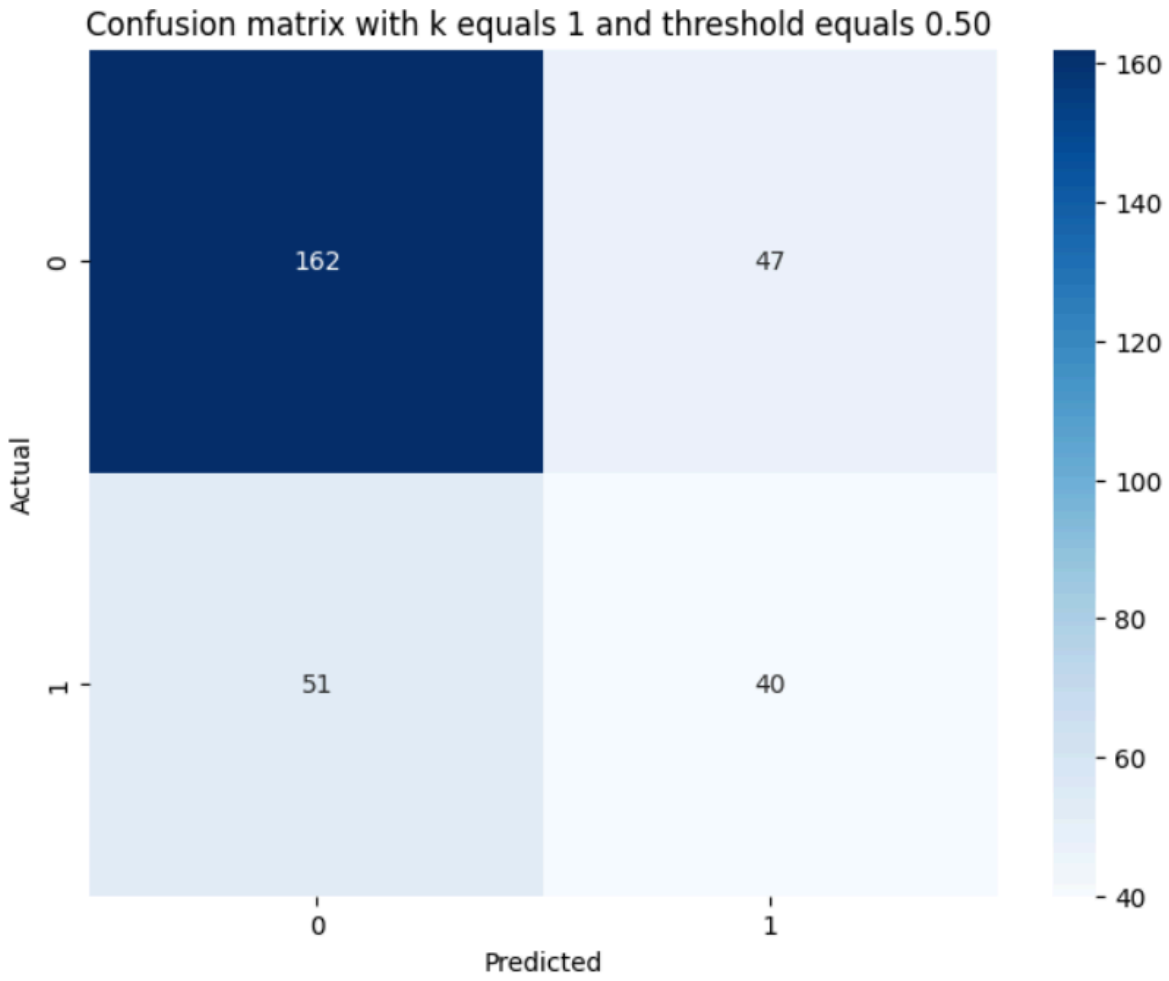
- **Confusion Matrix:**
  - **True Positives (TP):** 35
  - **True Negatives (TN):** 182
  - **False Positives (FP):** 27
  - **False Negatives (FN):** 56

- **Accuracy:** 72.33%
- **AUC:** 0.716



B)With specific features.

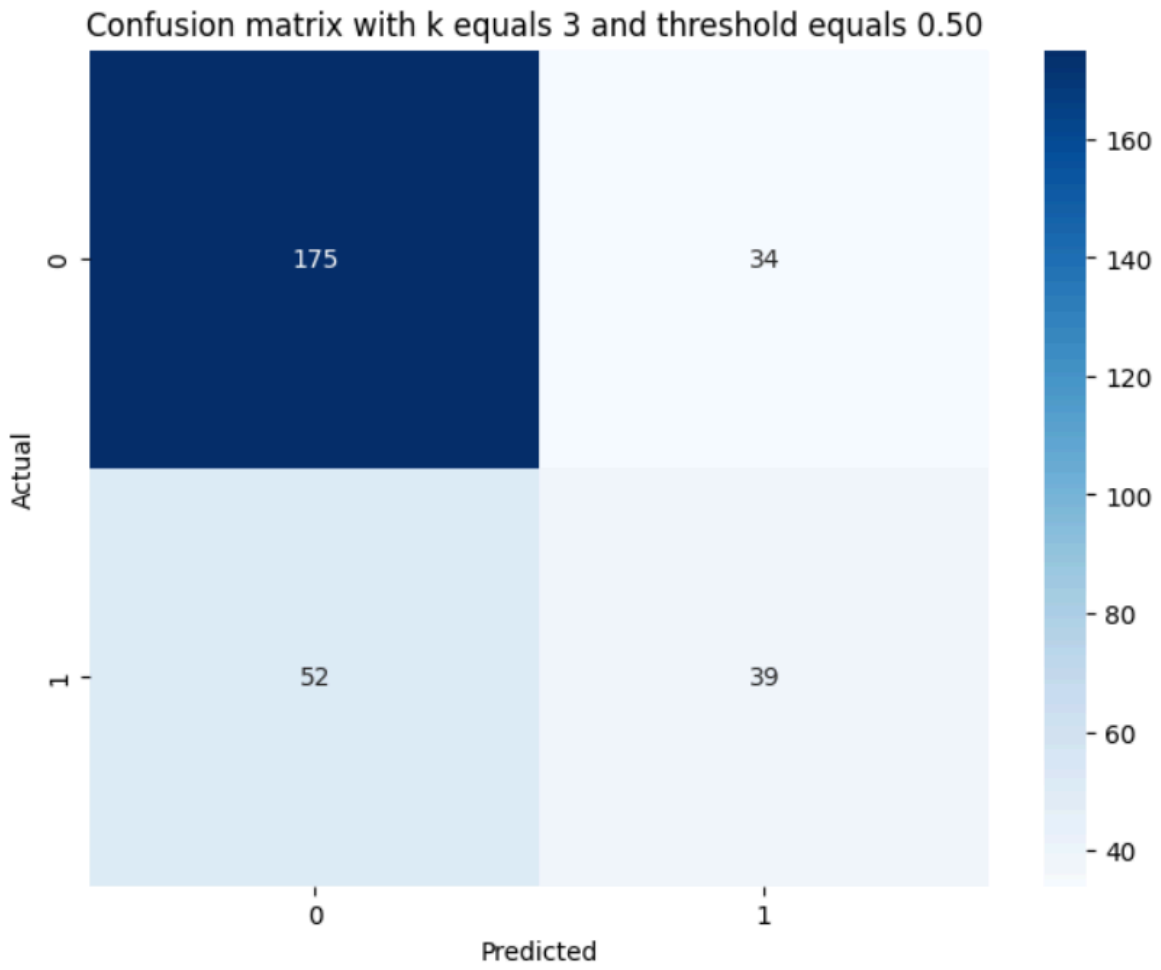
KNN Classifier with K = 1



- Confusion Matrix:
  - True Positives (TP): 40
  - True Negatives (TN): 162
  - False Positives (FP): 47
  - False Negatives (FN): 51

- Accuracy: 67.33%
- AUC: 0.607

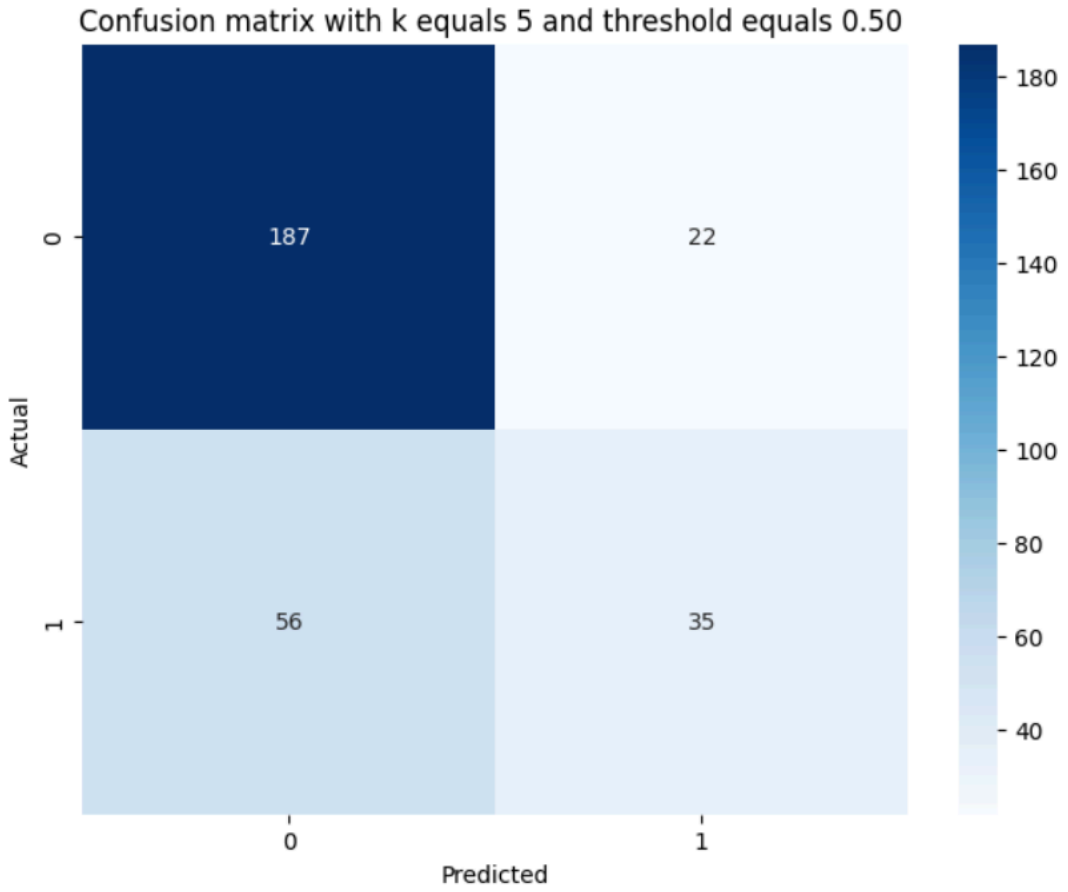
KNN Classifier with K = 3



- Confusion Matrix:
  - True Positives (TP): 39
  - True Negatives (TN): 175
  - False Positives (FP): 34
  - False Negatives (FN): 52

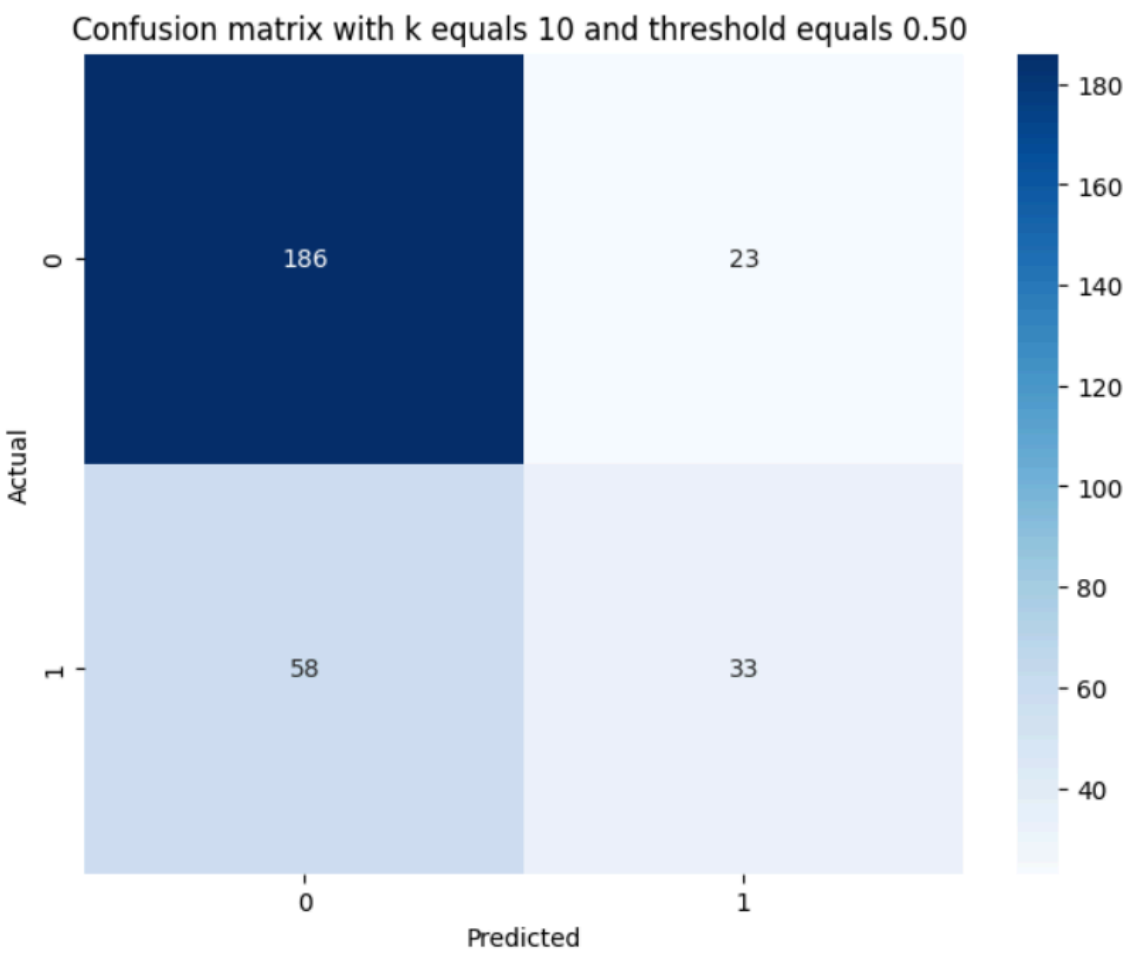
- Accuracy: 73.33%
- AUC: 0.726

KNN Classifier with K = 5



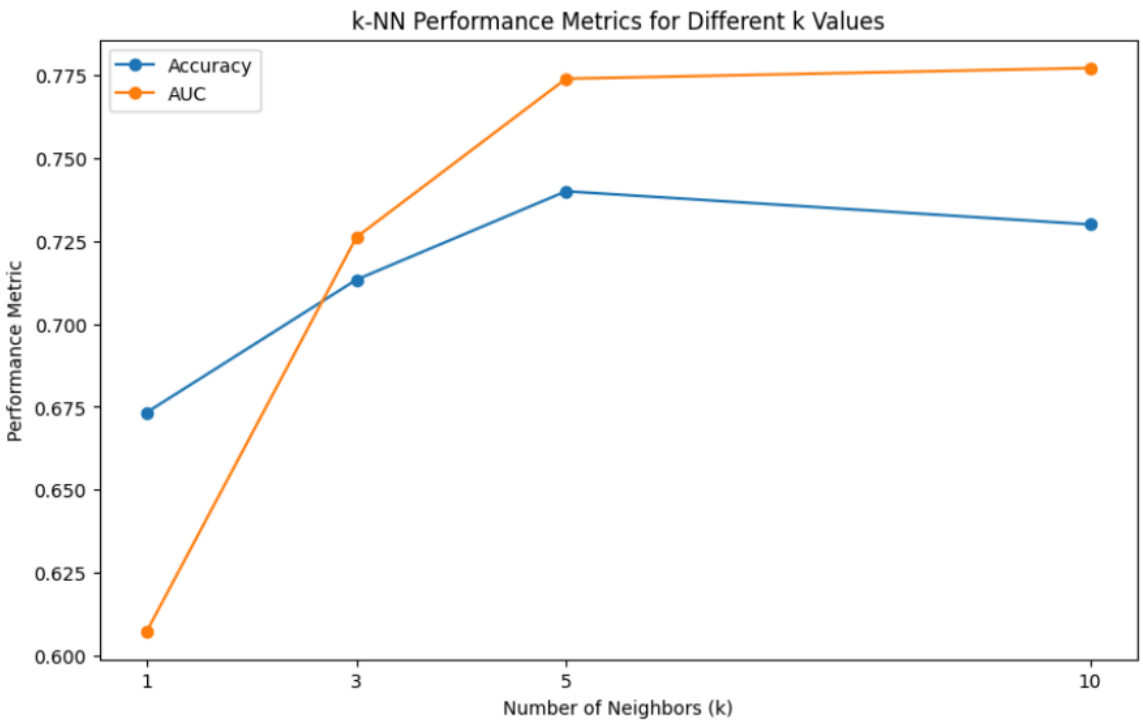
- Confusion Matrix:
  - True Positives (TP): 35
  - True Negatives (TN): 187
  - False Positives (FP): 22
  - False Negatives (FN): 56
- Accuracy: 74.00%
-AUC: 0.774

KNN Classifier with K = 10



- **Confusion Matrix:**
- **True Positives (TP):** 33
- **True Negatives (TN):** 186
- **False Positives (FP):** 23
- **False Negatives (FN):** 58

- **Accuracy:** 73.00%
- **AUC:** 0.777



	k	Accuracy	AUC
0	1	0.673333	0.607340
1	3	0.713333	0.726142
2	5	0.740000	0.773937
3	10	0.730000	0.777170

	k	Threshold	Accuracy	AUC	Precision	Recall	FPR	TPR
0	1	0.54	0.673333	0.607340	0.459770	0.439560	0.224880	0.439560
1	1	0.52	0.673333	0.607340	0.459770	0.439560	0.224880	0.439560
2	1	0.50	0.673333	0.607340	0.459770	0.439560	0.224880	0.439560
3	1	0.48	0.673333	0.607340	0.459770	0.439560	0.224880	0.439560
4	1	0.46	0.673333	0.607340	0.459770	0.439560	0.224880	0.439560
5	1	0.42	0.673333	0.607340	0.459770	0.439560	0.224880	0.439560
6	1	0.40	0.673333	0.607340	0.459770	0.439560	0.224880	0.439560
7	3	0.54	0.713333	0.726142	0.534247	0.428571	0.162679	0.428571
8	3	0.52	0.713333	0.726142	0.534247	0.428571	0.162679	0.428571
9	3	0.50	0.713333	0.726142	0.534247	0.428571	0.162679	0.428571
10	3	0.48	0.713333	0.726142	0.534247	0.428571	0.162679	0.428571
11	3	0.46	0.713333	0.726142	0.534247	0.428571	0.162679	0.428571
12	3	0.42	0.713333	0.726142	0.534247	0.428571	0.162679	0.428571
13	3	0.40	0.713333	0.726142	0.534247	0.428571	0.162679	0.428571
14	5	0.54	0.740000	0.773937	0.614035	0.384615	0.105263	0.384615
15	5	0.52	0.740000	0.773937	0.614035	0.384615	0.105263	0.384615
16	5	0.50	0.740000	0.773937	0.614035	0.384615	0.105263	0.384615
17	5	0.48	0.740000	0.773937	0.614035	0.384615	0.105263	0.384615
18	5	0.46	0.740000	0.773937	0.614035	0.384615	0.105263	0.384615
19	5	0.42	0.740000	0.773937	0.614035	0.384615	0.105263	0.384615
20	5	0.40	0.723333	0.773937	0.532787	0.714286	0.272727	0.714286
21	10	0.54	0.706667	0.777170	0.560000	0.153846	0.052632	0.153846
22	10	0.52	0.706667	0.777170	0.560000	0.153846	0.052632	0.153846
23	10	0.50	0.730000	0.777170	0.589286	0.362637	0.110048	0.362637
24	10	0.48	0.730000	0.777170	0.589286	0.362637	0.110048	0.362637
25	10	0.46	0.730000	0.777170	0.589286	0.362637	0.110048	0.362637
26	10	0.42	0.730000	0.777170	0.589286	0.362637	0.110048	0.362637
27	10	0.40	0.723333	0.777170	0.540000	0.593407	0.220096	0.593407

Analysis:

- The best performance in terms of AUC is achieved with **K = 10**, with an **AUC of 0.777**and an accuracy of **73%**.
- As the value of K increases, the AUC generally improves, indicating better model performance at higher values of K.

3(b) Comparison with Tree-Based Models and Logistic Model

- KNN Classifiers

- The best KNN model (K = 10) achieved an AUC of 0.777 and an accuracy of 73%.

- Tree-Based Models (Question 2):

- Random Forest achieved the highest AUC at **0.7962**with an accuracy of **76.33%.** The tree-based models, especially Random Forest, outperform the KNN classifiers in terms of both AUC and accuracy. This indicates that Random Forest generalizes better to the test set compared to KNN.

-Logistic Regression (Question 1):

- The logistic regression model had an AUC of 0.7104 with an accuracy of**78%** at a threshold of **0.35.** While the logistic regression model had better recall, the KNN models performed comparably in terms of overall accuracy and precision.

Conclusion:

- While the KNN classifier (with K = 10) shows reasonable performance, it is not as robust as the **Random Forest** from the tree-based models in question 2**.** The Random Forest model remains the top performer overall**.**
- KNN models offer simpler interpretability and work well, but for this dataset, tree-based models like Random Forest offer better predictive power.

Q4. (a) Fit at least one other binary classifier (e.g., a linear probability model or a Support Vector Machine classifier) to the dataset. Describe its performance relative to the classifiers highlighted above.

Fitting the SVM Classifier:

We split the data into training (70%) and testing set (30%).

How to find the hyperparameter values for training the SVM model?

Unlike model parameters (like weights in a neural network), hyperparameters must be manually set and optimized using methods like grid search. **Grid search** tries every combination of the hyperparameters provided in the "grid," systematically evaluating each one to find the best performing combination. Key hyperparameters to set include the **C** parameter (controls the trade-off between maximizing margin and minimizing classification error), the **kernel** type (linear, polynomial, radial basis function), and the **gamma** parameter (controls the influence of individual data points).

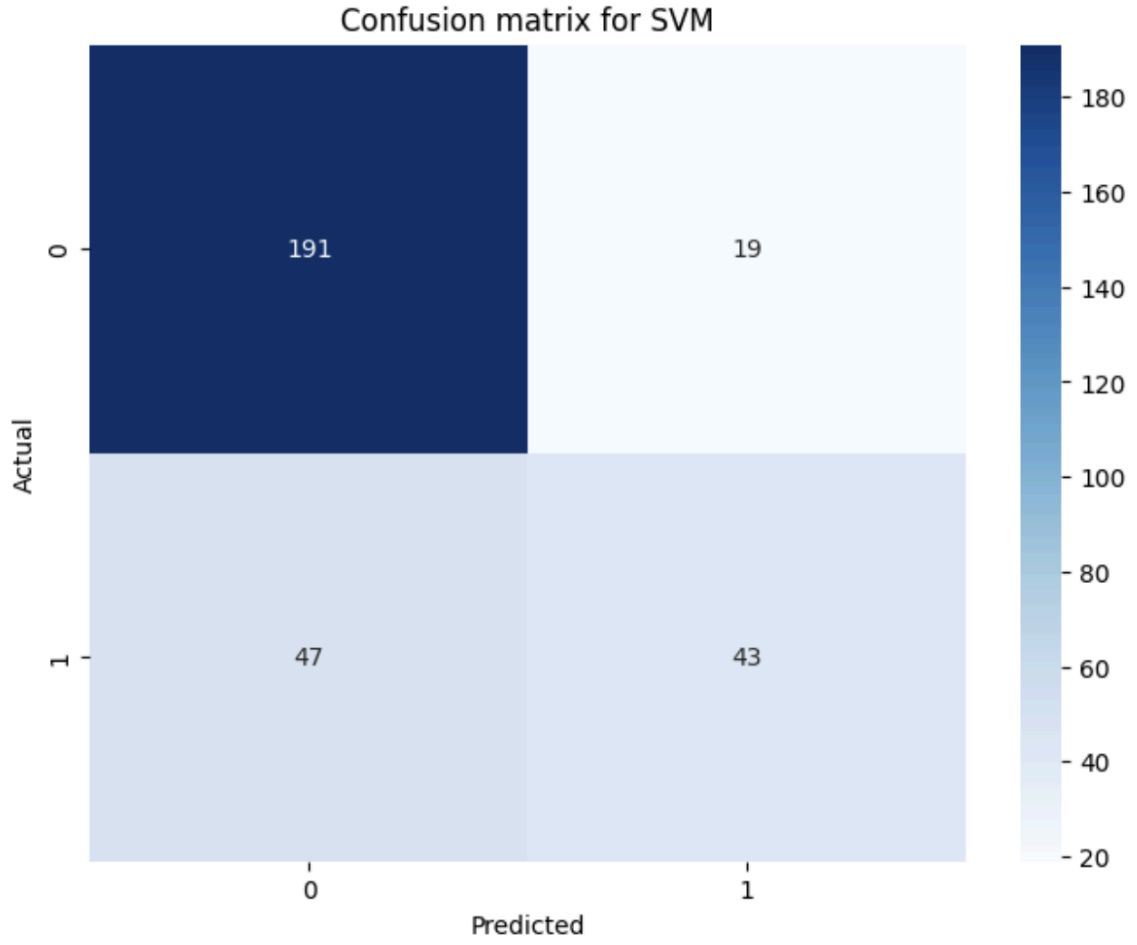
```python
# Define the parameter grid
param_grid = {
    'C': [0.1, 1, 10, 100],
    'gamma': [1, 0.1, 0.01, 0.001],
    'kernel': ['rbf', 'linear', 'poly', 'sigmoid']
}

# Initialize the SVM model
svc = SVC()

# Perform grid search
grid = GridSearchCV(svc, param_grid, refit=True, verbose=3)
grid.fit(X_train, y_train)

# Print the best parameters
print(grid.best_params_)
```

Results of Grid Search:

Hyperparameter	Value from Grid Search
C	0.1
Gamma	1
Kernel	Linear

```python
from sklearn.metrics import confusion_matrix, classification_report

# Fit a Support Vector Machine classifier
svm_clf = SVC(kernel='linear', C=0.1, gamma=1, random_state=42)
svm_clf.fit(X_train, y_train)

# Make predictions
y_pred = svm_clf.predict(X_test)

conf_matrix = confusion_matrix(y_test, y_pred)
# Plotting the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues')
plt.title("Confusion matrix for SVM")
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()

#To calculate the TPR and FPR
TN, FP, FN, TP = conf_matrix.ravel()

# Calculating TPR and FPR
TPR = TP / (TP + FN)
FPR = FP / (FP + TN)

# Display the results
print("True Positive Rate (TPR):", TPR)
print("False Positive Rate (FPR):", FPR)
```

Performance of SVM model:




```
True Positive Rate (TPR): 0.4777777777777778
False Positive Rate (FPR): 0.09047619047619047
```

The confusion matrix and metrics in the image provide the following information about the SVM model's performance:

1. **True Positives (TP)**: 43 (Predicted 1, Actual 1)
2. **True Negatives (TN)**: 191 (Predicted 0, Actual 0)
3. **False Positives (FP)**: 19 (Predicted 1, Actual 0)
4. **False Negatives (FN)**: 47 (Predicted 0, Actual 1)

Model Metrics:

- **True Positive Rate (TPR)**: 0.477
  This indicates that the model correctly classifies 47.7% of the actual positive cases.

- **False Positive Rate (FPR)**: 0.090
  This shows the model misclassifies 9% of the actual negatives as positives.

Overall:

- The SVM has a low true positive rate (sensitivity) of around 47.7%, indicating it misses a significant number of positive cases (47 false negatives).
- However, it performs well in predicting negative cases, with a false positive rate of only 9%.

This suggests that the model is better at identifying negative cases but struggles with identifying positive ones, potentially due to **class imbalance** or **suboptimal hyperparameters.**

Comparison with other model
1. Logistic Regression (Best Model with 0.35 Threshold)

- **Accuracy: 78%**
- **Precision: 0.63**
- **Recall (TPR): 0.68**
- **False Positive Rate (FPR): 0.18**
- **AUC: 0.7104**
- **Comment:** The Logistic Regression model provides a balanced performance in terms of precision and recall. However, it performs relatively worse than tree-based models and the SVM in terms of AUC, indicating slightly inferior discrimination between the two classes.

2. Classification Tree (Best Model)

- **Accuracy: 70.00%**
- **Precision: 0.50**
- **Recall (TPR): 0.45**
- **False Positive Rate (FPR): 0.19**
- **AUC: 0.6296**
- **Comment:** The classification tree performs reasonably well but has lower precision and recall. It overfits to the dataset more easily compared to bagging and random forest models, and its AUC is relatively low.

3. Bagging Classifier (Best Model)

- **Accuracy: 73.67%**
- **Precision: 0.76**
- **Recall (TPR): 0.42**
- **False Positive Rate (FPR): 0.09**
- **AUC: 0.7818**
- **Comment:** Bagging improves the overall performance of the base classification tree, reducing false positives and boosting precision. It also has a significantly higher AUC compared to the classification tree, indicating better generalization.

4. Random Forest (Best Model)

- **Accuracy: 76.33%**
- **Precision: 0.69**
- **Recall (TPR): 0.39**
- **False Positive Rate (FPR): 0.07**
- **AUC: 0.7962**
- **Comment:** The Random Forest outperforms all models in terms of accuracy and AUC. It also provides a good balance between precision and recall, making it the top-performing model for this dataset.

5. K-Nearest Neighbors (K = 10, Specific Features)

- **Accuracy: 73.00%**
- **Precision: 0.77**
- **Recall (TPR): 0.42**
- **AUC: 0.777**
- **Comment:** The KNN model with K = 10 performs similarly to the bagging classifier in terms of accuracy and AUC. However, it has a slightly lower precision compared to Random Forest and Logistic Regression. It is a good choice for a simpler, distance-based model but is not as powerful as tree-based methods.

6. Support Vector Machine (SVM)

- **Accuracy: 79.17%**
- **Precision: 0.69**

- **Recall (TPR): 0.477**
- **False Positive Rate (FPR): 0.090**
- **AUC: Not provided in detail, but based on confusion matrix, the model performs better in handling class imbalance with lower false positives.**
- **Comment:** The SVM classifier performs better than Logistic Regression and KNN in terms of accuracy but has a relatively low recall. It is highly sensitive to the class imbalance and misses a significant number of positive cases (i.e., bad credit customers). However, it shows potential for tuning and optimization through hyperparameters like kernel, C, and gamma.

Overall Comparison

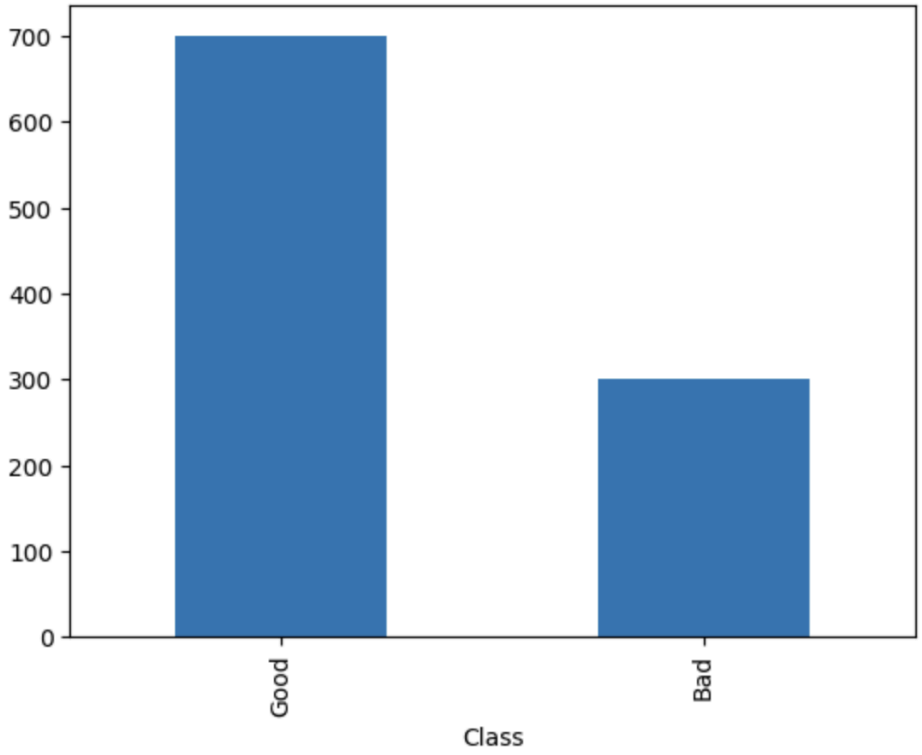
Best Overall Model: Random Forest with an AUC of 0.7962 and high accuracy of 76.33% emerges as the most reliable model for predicting credit risk. It offers the best trade-off between accuracy, precision, recall, and AUC.

SVM: The SVM classifier, despite good accuracy, struggles with recall due to class imbalance. Techniques like SMOTE or class weight adjustments would improve its performance in imbalanced datasets.
KNN: Performs reasonably well, with an AUC of 0.777 (using specific features), but falls short compared to Random Forest and SVM.

(b) Is your training dataset balanced? Comment on the drawbacks of fitting a Statistical Learning technique on an unbalanced dataset. Can confusion matrix be a useful performance metric for this problem? Can you think of / identify a technique to address this concern? If so, why do you think that the method(s) could work?
Based on the data distribution, it is clear that the training dataset is **not balanced**.
Good class (majority): Has significantly more instances ~70%
Bad class (minority): Has fewer instances ~30%

```
# Distribution of the target variable
data['Class'].value_counts().plot(kind='bar')
```

```
<Axes: xlabel='Class'>
```



Drawbacks of fitting a Statistical Learning technique on an unbalanced dataset

Biased Predictions: Models trained on imbalanced datasets often become biased toward the majority class because the cost of misclassifying the minority class is underrepresented during training.

Reduced Sensitivity (Recall): As seen in your results, the true positive rate (TPR) is only 47.7%. The model struggles to identify the minority class (1 here), leading to high false negatives, which can be critical depending on the problem context (e.g., in fraud detection, medical diagnosis).

Accuracy Paradox: On an imbalanced dataset, the model could achieve a high accuracy by merely predicting the majority class. For example, if 90% of the data belongs to class 0, the model could predict class 0 for every input and still have 90% accuracy, even though it's not correctly identifying the minority class.

Usefulness of the Confusion Matrix:

Yes, the confusion matrix is an essential performance metric for imbalanced classification problems because it gives a detailed insights into class-level performance. It helps us assess how well the model is handling the minority class and whether key metrics like precision, recall, and F1-score are in line with the objectives. This way, we can make informed decisions about whether the model is sufficiently sensitive to the minority class or needs further tuning.

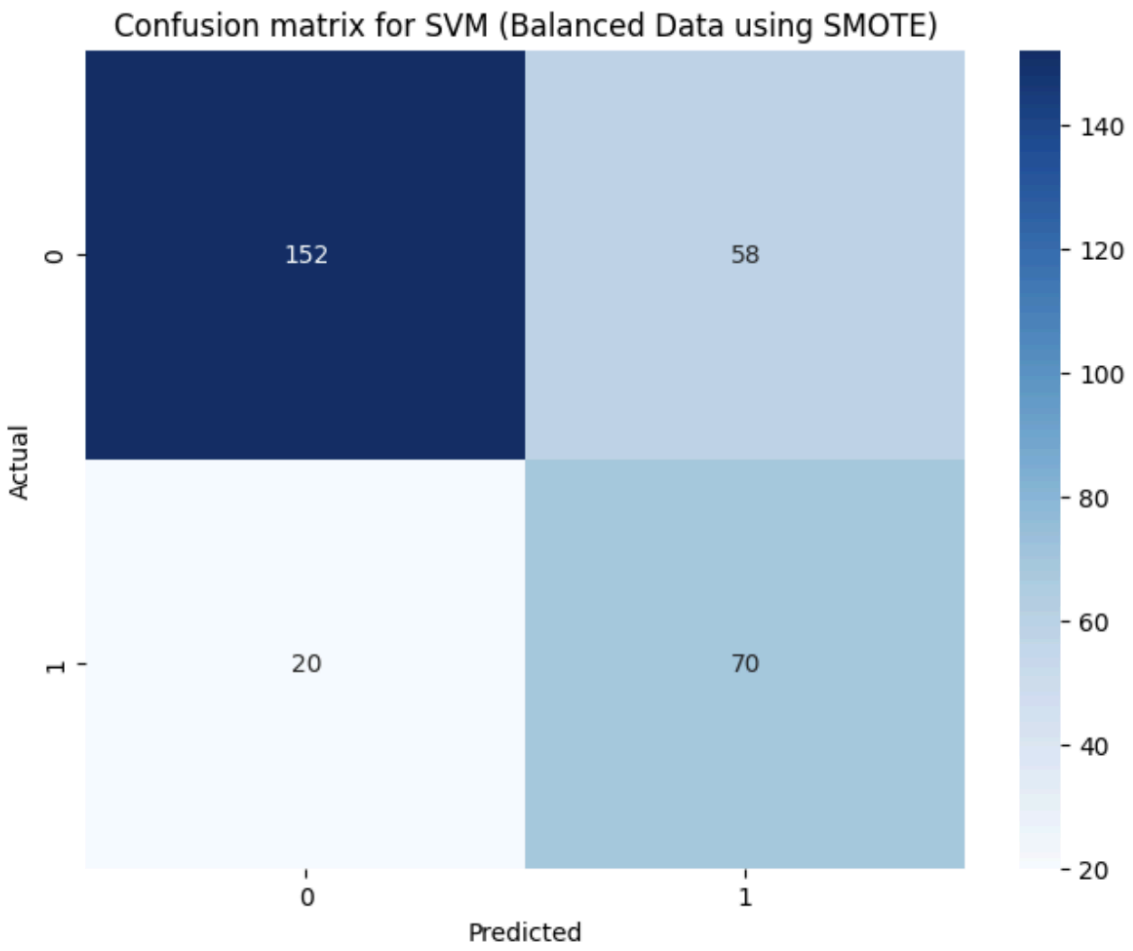
Technique to handle the unbalanced data:
There are various methods used to handle the imbalance in the data like- **SMOTE (Synthetic Minority Over-sampling Technique)** and **Class Weight Adjustment.**

Using SMOTE :
SMOTE tackles class imbalance by generating synthetic samples for the minority class rather than simply duplicating the existing minority class instances.
Steps involved :
1. For each instance in the minority class, SMOTE finds its nearest neighbors from within the minority class using a distance metric like Euclidean distance.
2. Synthetic samples are created by selecting one of the nearest neighbors of the minority class instance and generating a new sample between the two instances. The new sample is generated along the line segment connecting the two instances in the feature space.
3. This process is repeated multiple times to create enough synthetic samples, balancing the dataset.

```
[28] from imblearn.over_sampling import SMOTE
```

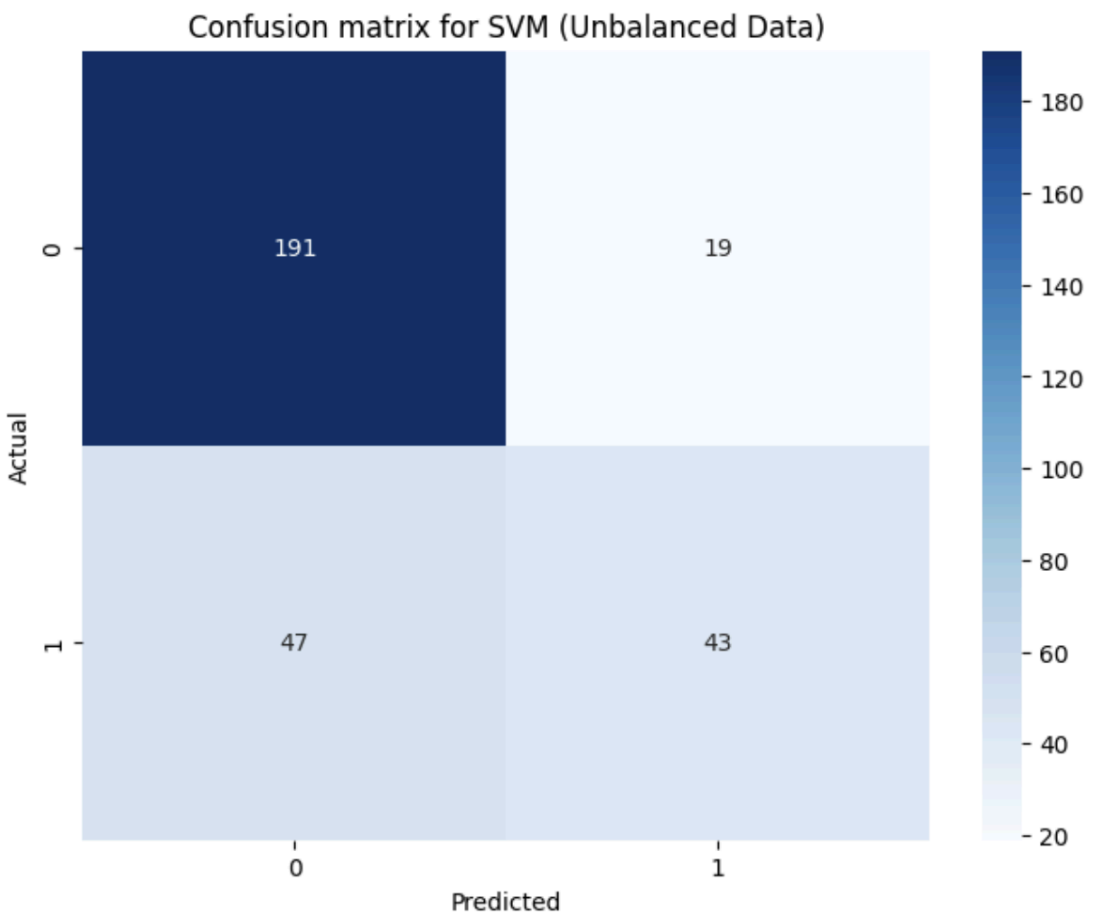
```
[29] # Apply SMOTE to the training data
     smote = SMOTE(random_state=42)
     X_train_resampled, y_train_resampled = smote.fit_resample(X_train, y_train)
```

Result of SMOTE data on SVM :




```
True Positive Rate (TPR): 0.7777777777777778
False Positive Rate (FPR): 0.2761904761904762
Recall: 0.7777777777777778
Precision: 0.546875
Accuracy: 0.74
```

Comparison with the unbalanced data for SVM :




```
True Positive Rate (TPR): 0.4777777777777778
False Positive Rate (FPR): 0.09047619047619047
Recall: 0.4777777777777778
Precision: 0.6935483870967742
Accuracy: 0.78
```

SVM with SMOTE (Balanced Data):

- **Recall**: 77.7% (High)
- **False Negatives**: 20 (Fewer missed bad credit customers)
- **Trade-off**: Higher false positives (27.6%) and slightly lower precision (54.7%).

SVM without SMOTE (Unbalanced Data):

- **Recall**: 47.7% (Low)
- **False Negatives**: 47 (More missed bad credit customers)
- **Trade-off**: Lower false positives (9%) and higher precision (69.4%).

Using SMOTE: This model is much better at identifying bad credit customers (label 1), which aligns with our goal. Even though there are more false positives, the trade-off is worth it in this context since our main concern is ensuring that as few bad credit customers as possible are missed.

WORK DISTRIBUTION

Sabari S(208170829):	Q1 code
Deeksha Rawat(210303):	Data pre processing and Q4(code+report)
Shorya Agarwal(221023):	Q2(code+report) and Q3 report
Kumar Aditya(220559):	Q3 code
Harsh Nirmal(220431):	Q1 report