CSE 587/4B: Data Intensive Computing
Project Phase – 3

Team Members

Name, UBIT, UB Person Number:

Manish Malepati, manishma, 50541149

Nihal Muhammed Basim, nihalmuh, 50540332

Bhavesh Tharlapally, bhavesht, 50541076

Problem Statement:

The business manager responsible for a consumer credit card portfolio is confronted with a pressing issue of customer attrition. The objective of this assignment is to analyze the available data to identify the underlying factors contributing to customer attrition, and subsequently develop a predictive model to forecast which customers are at risk of dropping off soon. The solution should empower the business to proactively implement retention strategies and reduce customer churn, ultimately safeguarding the portfolio's profitability and long-term sustainability.

Context:

Contributions made till Phase 1 include collecting the raw data, data processing, data cleaning and performing exploratory data analysis. In Phase 2 of the project, we performed machine learning classification tasks using six algorithms namely KNN, Naïve Bayes, Support Vector Machine, Random Forest, Logistic Regression, and Decision tree.

Random forest was our best performing algorithm with the highest accuracy percentage. Now, we go ahead and perform model tuning by optimizing our hyperparameters so as to increase our respective model's performance and its matrix.

Model Tuning:

Logistic Regression:

Tuning the logistic regression model can be done by optimizing hyperparameters inverse of regularization strength, penalty. The code demonstration can be shown as below:

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn import metrics

logisticRegressionModified = LogisticRegression(random_state=25, solver='lbfgs', max_iter=1000)
parameterGrid = {'C': [0.001, 0.01, 0.1, 1, 10, 100],'penalty': ['l2'] }
gridSearch = GridSearchCV(estimator=logisticRegressionModified, param_grid=parameterGrid, scoring='accuracy', cv=5)
gridSearch.fit(x_train, y_train)
parameters_best = gridSearch.best_params_
print("Best Hyperparameters:", parameters_best)
modifiedLogisticRegression = gridSearch.best_estimator_
predictions_modified = modifiedLogisticRegression.predict(x_test)

accuracy_modified = metrics.accuracy_score(y_test, predictions_modified)
print(f"Accuracy Modified: {accuracy_modified}")
accuracy_train = modifiedLogisticRegression.score(x_train, y_train)
print(f"Accuracy Training: {accuracy_train}")
print("Precision Modified: ", metrics.precision_score(y_test, predictions_modified))
print("Recall Modified: ", metrics.recall_score(y_test, predictions_modified))
```

Here, the parameter grid has different values for regularization parameter and the type of regularization “penalty”.

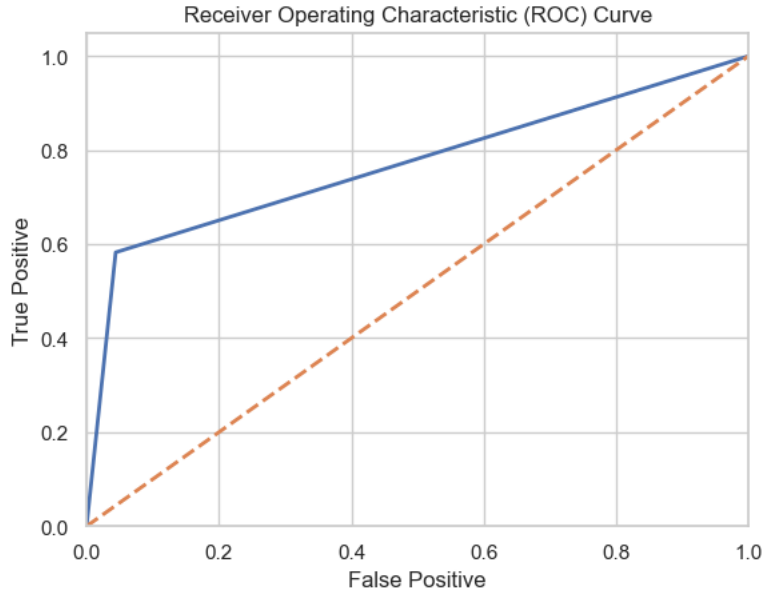
‘C’ here indicates the inverse of the regularization, and its values here include 0.001, 0.01, 0.1, 1, 10, and 100.

Penalty specifies the type of regularization applied. Here it is “l2” which is L2 regularization.

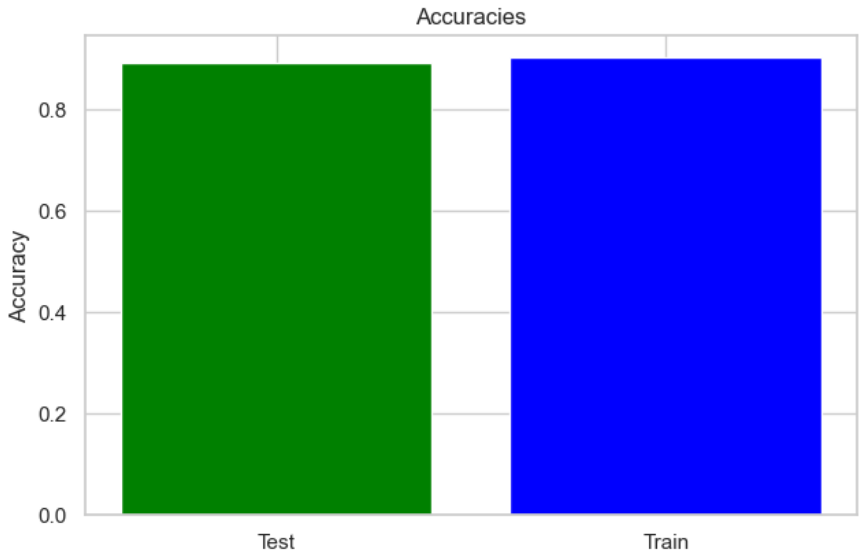
Performance:

```
Best Hyperparameters: {'C': 100, 'penalty': 'l2'}
Accuracy Modified: 0.8934092758340114
Accuracy Training: 0.9026044492674986
Precision Modified:  0.7272727272727273
Recall Modified:  0.5825242718446602
```

Updated ROC Curve:



Updated comparison, accuracy of test and training data:



Decision Tree:

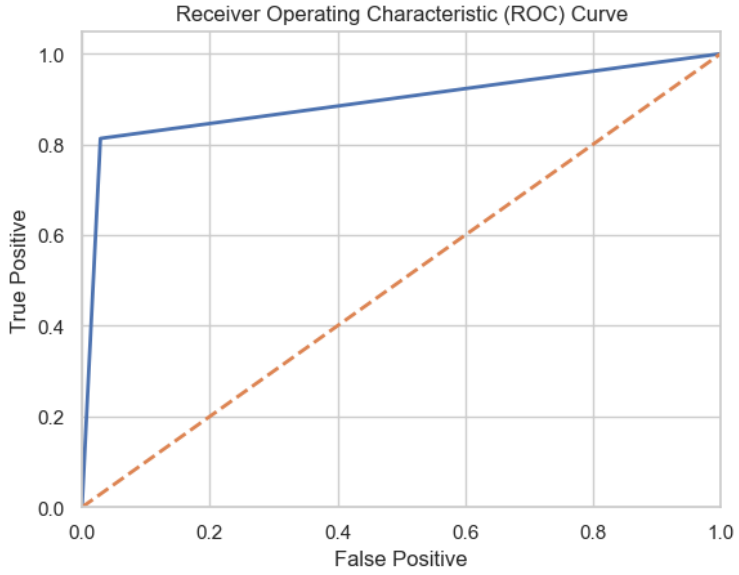
```
from sklearn.model_selection import GridSearchCV
from sklearn.tree import DecisionTreeClassifier
from sklearn import metrics

decTree_Modified = DecisionTreeClassifier(random_state=25)
parameterGrid = {'max_depth': [None, 5, 10, 15], 'min_samples_split': [2, 5, 10], 'min_samples_leaf': [1, 2, 4] }
gridSearch = GridSearchCV(estimator=decTree_Modified, param_grid=parameterGrid, scoring='accuracy', cv=5)
gridSearch.fit(x_train, y_train)
parameters_best = gridSearch.best_params_
print("Best Hyperparameters:", parameters_best)
modified_decision_tree = gridSearch.best_estimator_
modified_predictions = modified_decision_tree.predict(x_test)

accuracy_modified = modified_decision_tree.score(x_test, y_test)
print(f"Accuracy Modified: {accuracy_modified}")
accuracy_training = modified_decision_tree.score(x_train, y_train)
print(f"Accuracy Training: {accuracy_training}")
print("Precision Modified: ", metrics.precision_score(y_test, modified_predictions))
print("Recall Modified: ", metrics.recall_score(y_test, modified_predictions))
```

Here, max_depth controls the maximum depth of the decision tree, here we are exploring values None, 5, 10, 15. min_samples_split sets the minimum number of samples required to split an internal node, here we explore values 2, 5 and 10. min_samples_leaf sets the minimum number of samples required to be at a leaf node. The values we explore here are 1, 2 and 4.

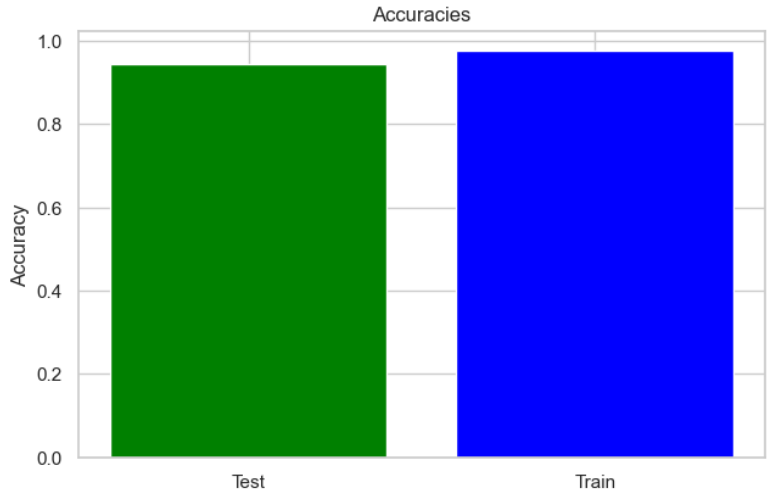
Performance:

```
Best Hyperparameters: {'max_depth': 10, 'min_samples_leaf': 4, 'min_samples_split': 10}
Accuracy Modified: 0.9446704637917006
Accuracy Training: 0.9742268041237113
Precision Modified:  0.850253807106599
Recall Modified:  0.8131067961165048
```

Updated ROC Curve:



Updated accuracies of training and test data:



Support Vector Machine:

Model Tuning:

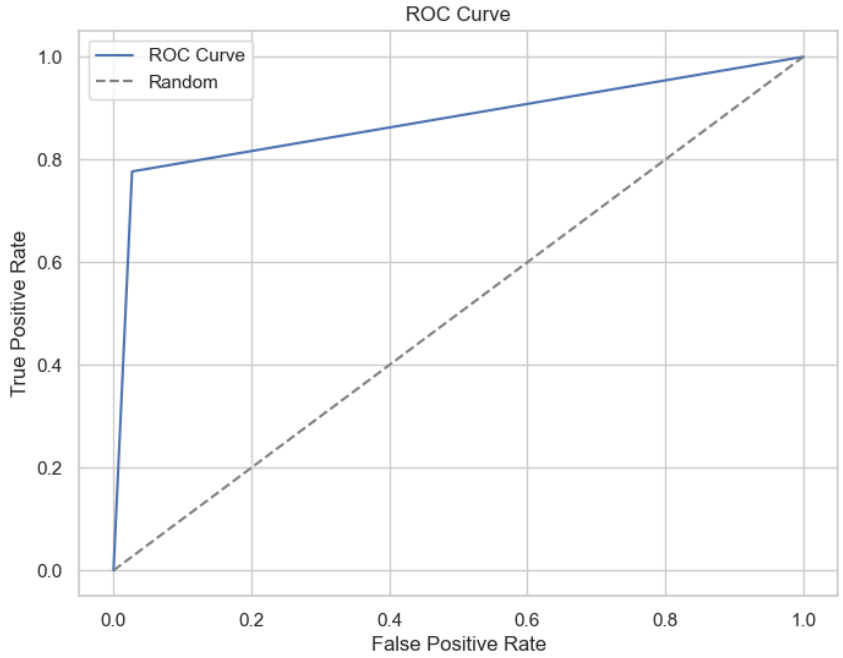
```
# We tune the Support Vector Machine model by optimizing the hyperparameters such as regularization parameters
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix

svm_model_modified = SVC()
parameterGrid = {'C': [0.1, 1, 10], 'kernel': ['linear', 'rbf'], 'gamma': ['scale', 'auto'] }
gridSearch = GridSearchCV(estimator=svm_model_modified, param_grid=parameterGrid, scoring='accuracy', cv=5)
gridSearch.fit(x_train, y_train)

parameters_best = gridSearch.best_params_
print("Best Hyperparameters:", parameters_best)

modified_svm_model = gridSearch.best_estimator_
y_test_predictions_modified = modified_svm_model.predict(x_test)

accuracy_modified = accuracy_score(y_test, y_test_predictions_modified)
precision_modified = precision_score(y_test, y_test_predictions_modified)
recall_modified = recall_score(y_test, y_test_predictions_modified)
f1_modified = f1_score(y_test, y_test_predictions_modified)

print("Testing Accuracy Modified:", accuracy_modified)
print("Precision Modified:", precision_modified)
print("Recall Modified:", recall_modified)
print("F1 Score Modified:", f1_modified)
```

Here C is regularization parameter, values we use are 0.1, 1,10 ‘kernel’ determines the type of kernel function used in the SVM algorithm (linear, rbf), ‘gamma’ defines how far the influence of a single training example reaches (scale, auto).

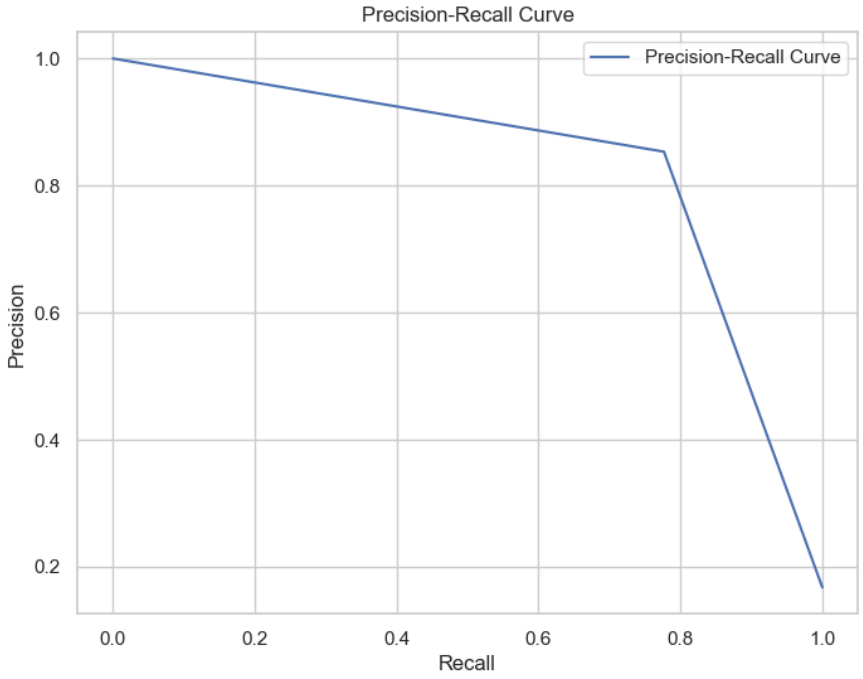
Performance:

```
Best Hyperparameters: {'C': 10, 'gamma': 'auto', 'kernel': 'rbf'}
Testing Accuracy Modified: 0.9401952807160293
Precision Modified: 0.8533333333333334
Recall Modified: 0.7766990291262136
F1 Score Modified: 0.8132147395171537
```

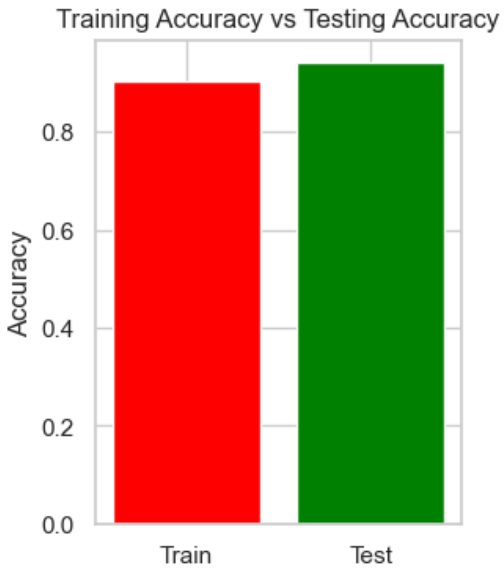
Updated ROC Curve:



Precision-Recall Curve:



Accuracies:



Random Forest Classifier:

Model Tuning:

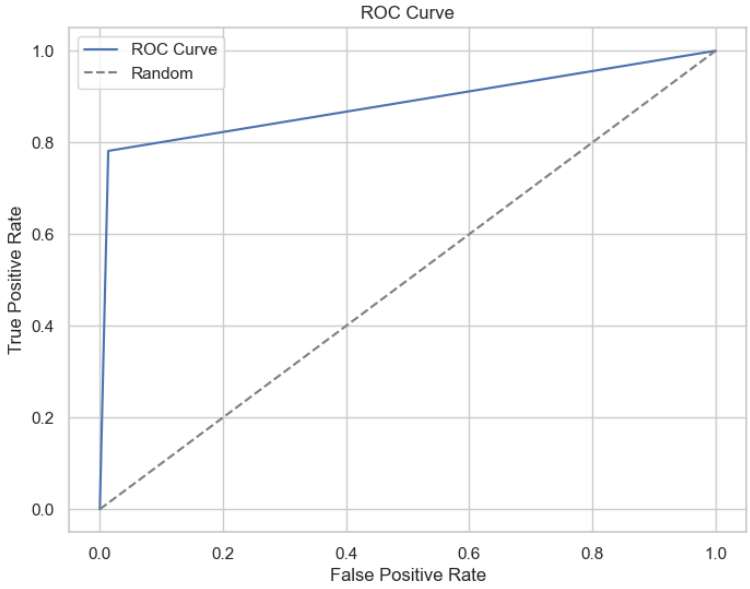
```
# We try to tune the Random Forest Classifier model using random search
# Here we tune the model by limiting our search space, reducing the number of folds and utilizing parallel processing by using n_jobs
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint as sp_randint

rf_model_modified = RandomForestClassifier(random_state=25)
parameter_distribution = {'n_estimators': sp_randint(50, 200), 'max_depth': [None, 10, 20, 30], 'min_samples_split': sp_randint(2, 10), 'min_samples_leaf':
sp_randint(1, 4) }
random_search = RandomizedSearchCV(estimator=rf_model_modified, param_distributions=parameter_distribution, n_iter=10, scoring='accuracy', cv=5,
random_state=25)

random_search.fit(x_train, y_train)

best_parameters = random_search.best_params_
print("Best Hyperparameters:", best_parameters)

modified_rf_model = random_search.best_estimator_
y_test_predictions_modified = modified_rf_model.predict(x_test)

accuracy_modified = accuracy_score(y_test, y_test_predictions_modified)
precision_modified = precision_score(y_test, y_test_predictions_modified)
recall_modified = recall_score(y_test, y_test_predictions_modified)
f1_modified = f1_score(y_test, y_test_predictions_modified)

print("Testing Accuracy Modified:", accuracy_modified)
print("Precision Modified:", precision_modified)
print("Recall Modified:", recall_modified)
print("F1 Score Modified:", f1_modified)
```

Here, n_estimators parameter represents the number of trees in the Random Forest, we explore values 50 and 200. The max_depth controls the maximum depth of each decision tree in the Random Forest, we explore values 10, 20, 30. min_samples_split sets the minimum number of samples required to split an internal node, we explore values 2 and 10. Min_samples_leaf sets the minimum number of samples required to be at a leaf node we explore values 1 and 4.

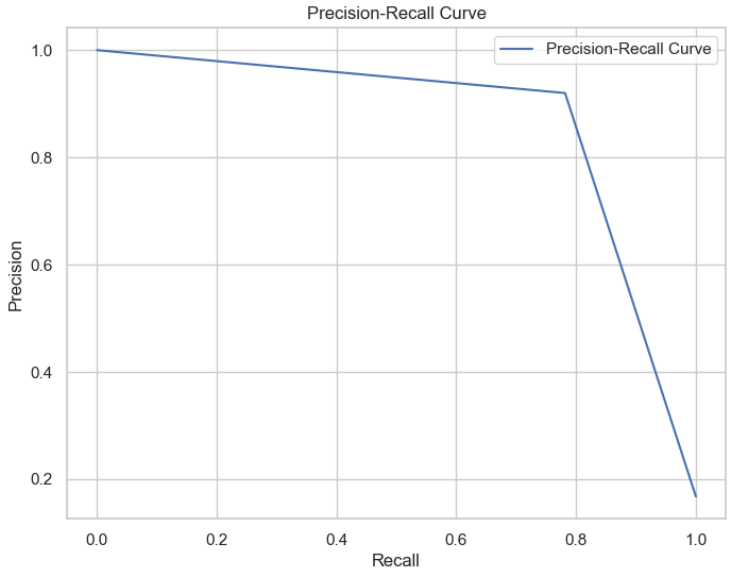
Performance:

```
Best Hyperparameters: {'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 3, 'n_estimators': 179}
Testing Accuracy Modified: 0.951993490642799
Precision Modified: 0.92
Recall Modified: 0.7815533980582524
F1 Score Modified: 0.8451443569553806
```

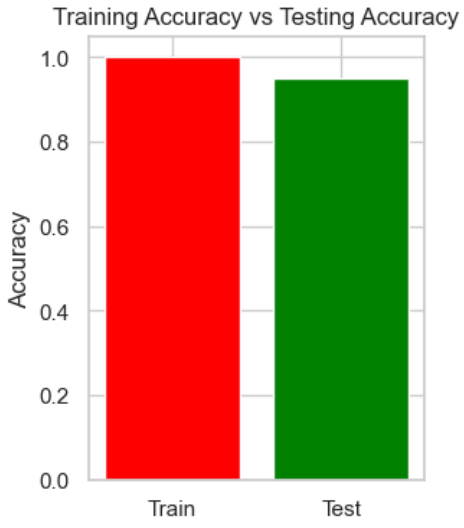
Updated ROC Curve:



Precision Recall Curve:



Accuracies:



KNN Classifier:

Model Tuning:

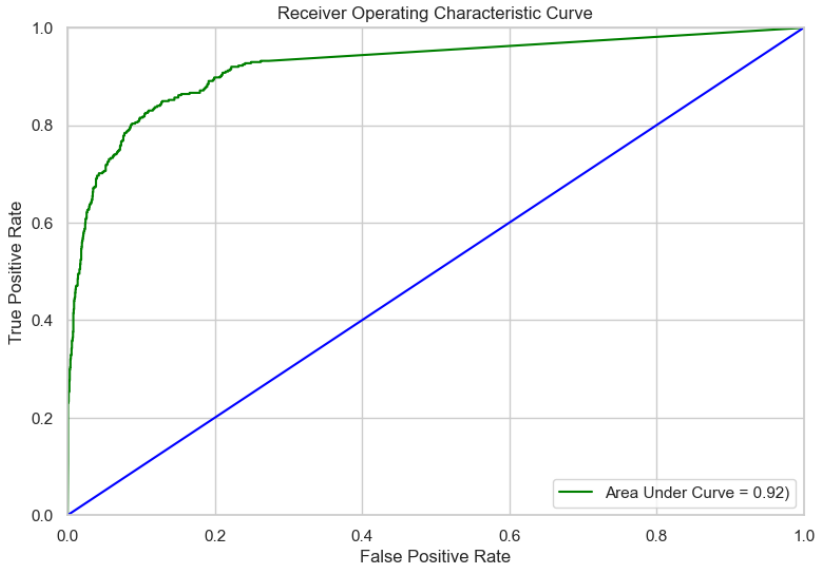
```
#Tuning the hyperparameters of K-Nearest Neighbors using grid search

from sklearn.model_selection import GridSearchCV

# Defining the parameter grid
parameterGrid = {'n_neighbors': [3, 5, 7, 9], 'weights': ['uniform', 'distance'], 'metric': ['euclidean', 'manhattan'] }

kNNClassifier = KNeighborsClassifier()
gridSearch = GridSearchCV(estimator=kNNClassifier, param_grid=parameterGrid, scoring='accuracy', cv=5)
gridSearch.fit(x_train, y_train)

parameters_best = gridSearch.best_params_
print("Best Hyperparameters:", parameters_best)

kNNClassifier_modified = gridSearch.best_estimator_
y_predections_modified = kNNClassifier_modified.predict(x_test)

accuracyValue_modified = accuracy_score(y_test, y_predections_modified)
precisionValue_modified = precision_score(y_test, y_predections_modified)
recallValue_modified = recall_score(y_test, y_predections_modified)
f1Score_modified = f1_score(y_test, y_predections_modified)

print(f"Accuracy Modified: {accuracyValue_modified:.2f}")
print(f"Precision Modified: {precisionValue_modified:.2f}")
print(f"Recall Modified: {recallValue_modified:.2f}")
print(f"F1-Score Modified: {f1Score_modified:.2f}")
```

n_neighbors parameter determines the number of nearest neighbors to consider when making predictions for a new data point. Here we explore value 7. weights parameter specifies how the contributions of neighbors are weighted when making predictions. Metric is for distance, euclidean or Manhattan.

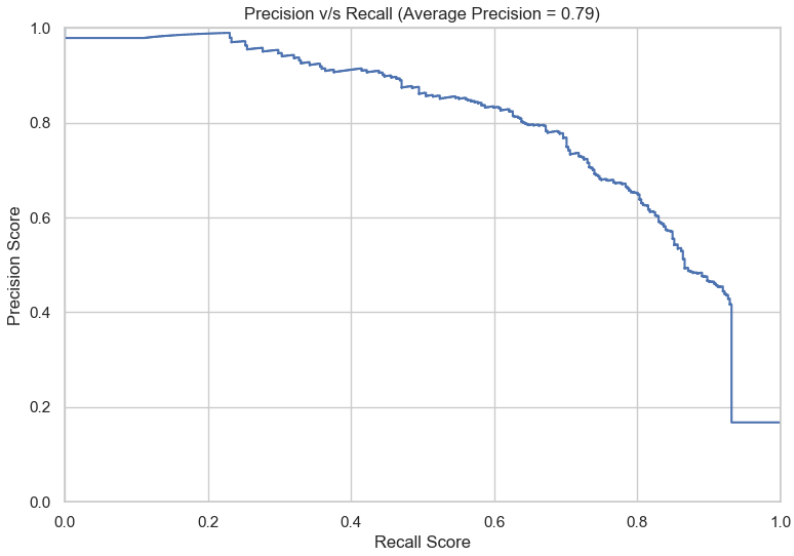
Performance:

```
Best Hyperparameters: {'metric': 'manhattan', 'n_neighbors': 7, 'weights': 'distance'}
Accuracy Modified: 0.91
Precision Modified: 0.85
Recall Modified: 0.57
F1-Score Modified: 0.68
```

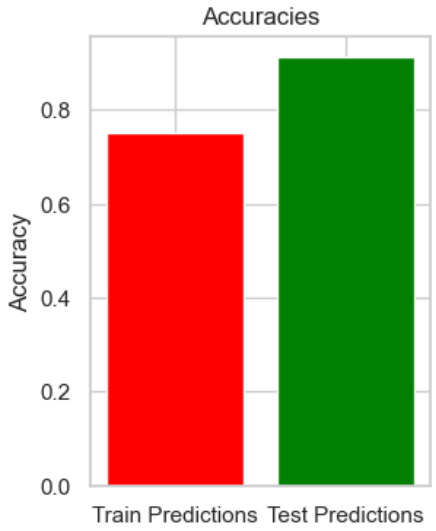
Updated ROC:



Updated Precision-Recall:



Accuracies:



Naïve Bayes:

We try to tune the Naive Bayes model using GridSearchCV. As we know, Gaussian Naive Bayes works well for classification tasks when we have limited amount of data and few features. But, as we are working with huge datasets, fine tuning the Naive Bayes algorithm might not lead to substantial performance improvements.

Here we do not have any explicit hyperparameters to tune unlike kNNClassifier as it relies on feature independence.

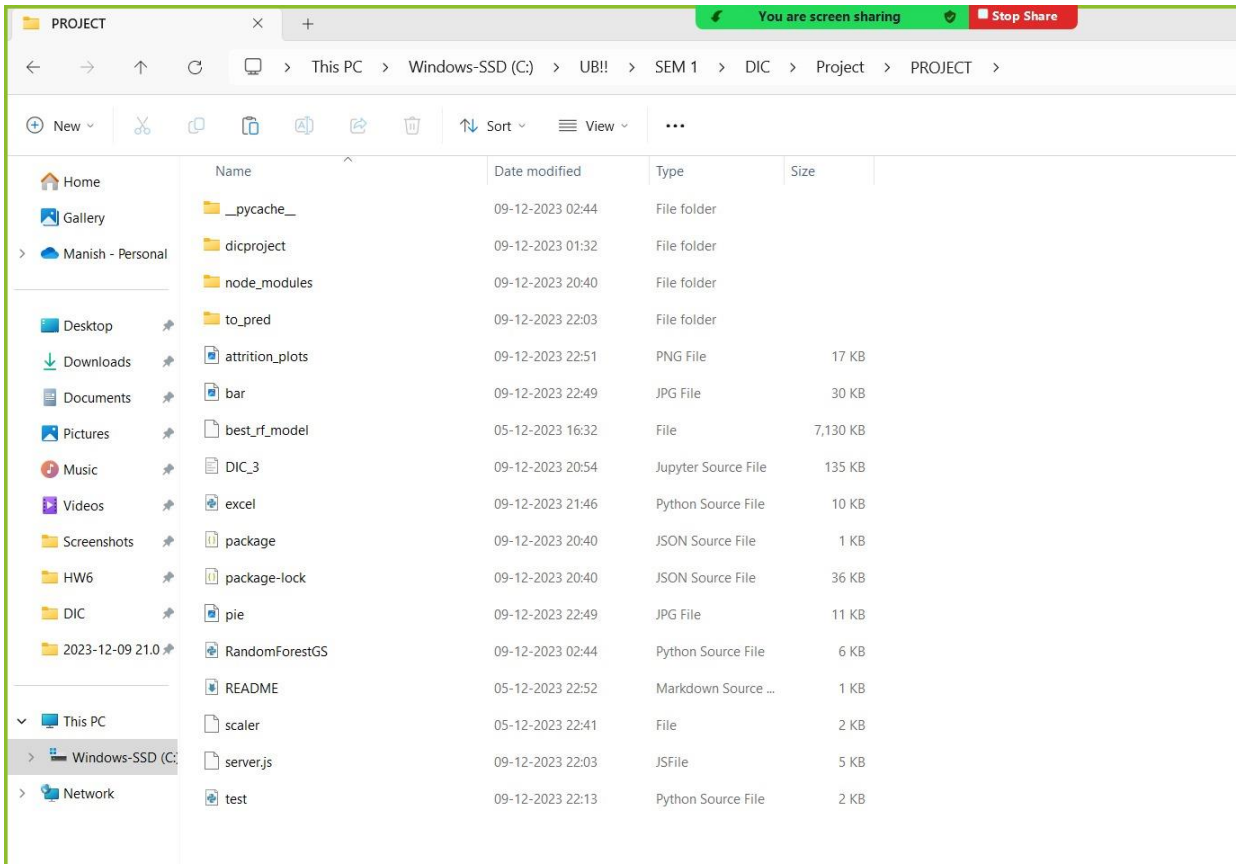
Some pre-processing steps which can be considered for model tuning are: Feature Selection, Feature Engineering, handling imbalanced data etc.

Post model tuning process, the model which performed most efficiently is Random Forest Classifier with a testing accuracy of **95.19%**. Therefore, for our phase 3 implementation of our project, we move ahead with the **Random Forest Classifier** algorithmic implementation.

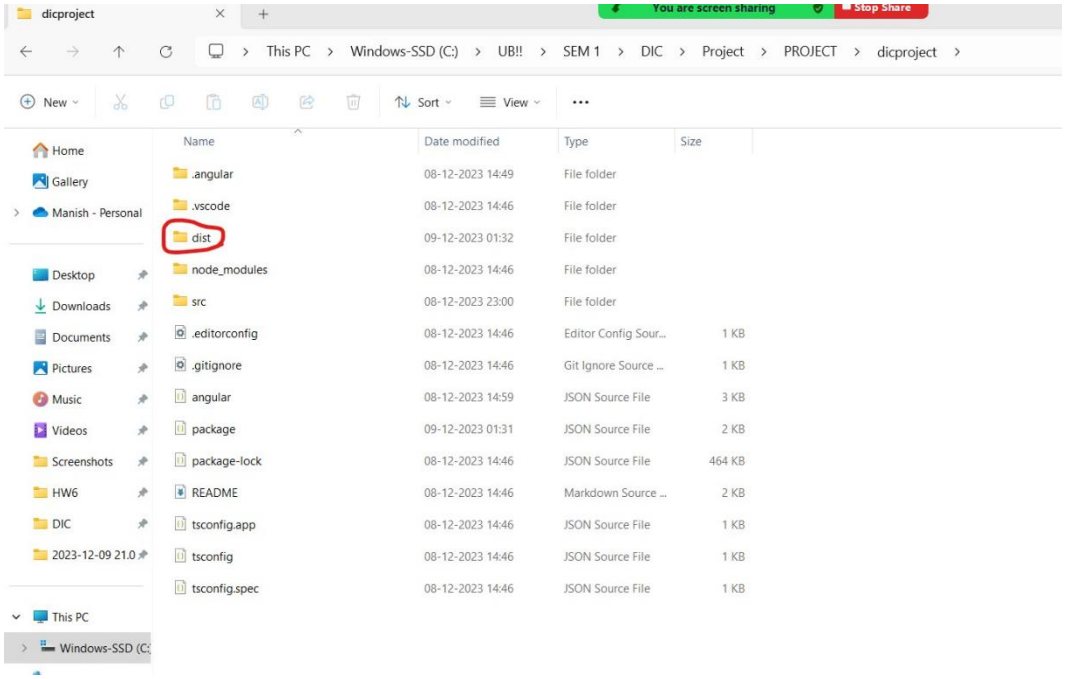
Working instructions of our demo:

- Unzip the “PROJECT” file under the Phase 3 folder in src.

The folder structure should be as follows:



The dist folder should be present inside the dicproject folder.

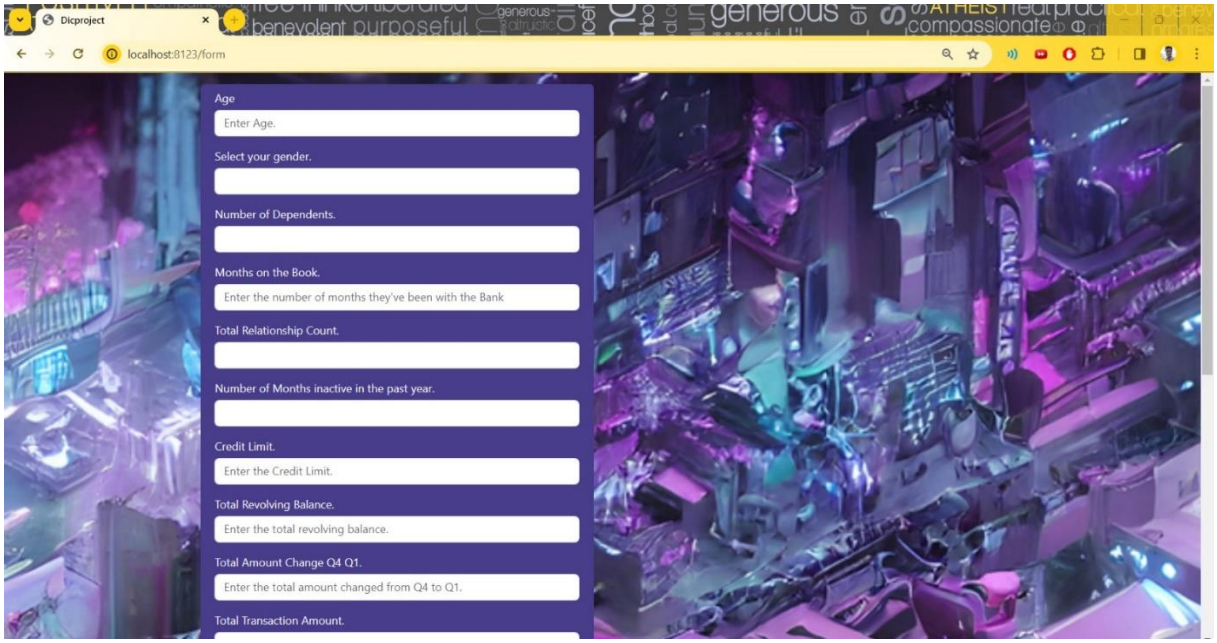


Run the below bash command to start the backend:

```
mrmal@SKYLOO MINGW64 /c/UB!!/SEM 1/DIC/Project/PROJECT
$ node server.js
Server Started!
```

Open localhost:8123/ to see if the website is up.
Go to the directory localhost:8123/form to be able to access our product.
Access localhost:8123/test to check if the python integration is working properly.

Our web application:



Recommendations for our project:

The predictive model we designed can be used to segment customers based on their likelihood to churn. This information is crucial for our business manager to take proactive measures and keep our customers satisfied.

Using these data, we have developed an interactive dashboard that visualizes key metrics, insights, and predictions from the model. This enables the business manager to explore the data, understand trends, and make informed decisions on retention strategies. The User-friendly interfaces empower non-technical users to leverage the power of the model for day-to-day decision-making.

We've also devised a plan that covers two key aspects to effectively manage customer retention.

Firstly, for customers who might be thinking about leaving, we can now sort them into groups based on how likely they are to go. This helps us tailor specific plans for each group, addressing the unique reasons that might be making them consider leaving. We can setup a system that keeps a constant eye out for customers with a high risk of leaving, allowing us to act quickly and prevent it. This approach ensures that we use our resources efficiently while maintaining a positive customer base.

Secondly, we want to ensure we don't lose customers who are already happy and likely to stay. By using our predictive model, we can identify these content customers and create plans to make their experience even better. This involves suggesting additional products or services they might find appealing, making special offers, and finding ways to keep them engaged with our services. Through these efforts, we aim to build strong, long-lasting relationships with these satisfied customers.

We also have few recommendations which can be integrated on top of this model in the future.

The model can be turned into a real-time monitoring system that continuously assesses customer churn risk. Automated alerts can be set up for high-risk customers, enabling the business manager to take immediate action and deploy timely retention tactics. Also the model can be integrated with customer feedback mechanisms to incorporate more data into the analysis to gain a comprehensive understanding of the reasons behind attrition.

Ultimately, our comprehensive plan addresses both the challenge of retaining customers who might be at risk of leaving and the opportunity to further engage and satisfy those who are happy and likely to stay. With help of this model a detailed roadmap can be designed that guides our business manager in making informed decisions, ultimately helping our credit card business not only navigate challenges but thrive in a competitive market.