Post-Implementation Report

Solution Summary

This project was developed to address operational inefficiencies in the loan application process at Northern Bank, where manual determination methods resulted in delays, inconsistencies, and wasted resources. The solution involved developing an automated loan eligibility prediction system using a machine learning model that was trained on historical loan data. This model was integrated into an intuitive user-friendly interface designed to streamline the application process and eliminate manual eligibility determination. The system ensures consistent loan eligibility determinations with minimal errors and conserves valuable resources for Northern Bank.

The application offers a comprehensive solution to the operational inefficiencies that Northern Bank faces in its loan application process by utilizing machine learning and an intuitive user interface. The application addresses delays and inconsistencies in loan applications due to manual processing limitations and thorough preprocessing ensures the reliability and consistency of the input data. Incorporating the Random Forest Classifier model, which was trained and fine-tuned on the historical loan dataset, provides accurate and consistent determinations, and mitigates errors associated with manual decision-making.

Data Summary

The dataset used for the development of the loan eligibility prediction system was obtained from Kaggle.com [1].

The initial dataset had a total of 13 columns, as shown here:

```
# List columns and data types for each column
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 614 entries, 0 to 613
Data columns (total 13 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   Loan_ID            614 non-null    object 
 1   Gender             601 non-null    object 
 2   Married            611 non-null    object 
 3   Dependents         599 non-null    object 
 4   Education          614 non-null    object 
 5   Self_Employed      582 non-null    object 
 6   ApplicantIncome    614 non-null    int64  
 7   CoapplicantIncome  614 non-null    float64
 8   LoanAmount         592 non-null    float64
 9   Loan_Amount_Term   600 non-null    float64
 10  Credit_History     564 non-null    float64
 11  Property_Area      614 non-null    object 
 12  Loan_Status        614 non-null    object 
dtypes: float64(4), int64(1), object(8)
memory usage: 62.5+ KB
```

	Loan_ID	Gender	Married	Dependents	Education	Self_Employed	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Term	Credit_History	Property_Area	Loan_Status
0	LP001002	Male	No	0	Graduate	No	5849	0.0	NaN	360.0	1.0	Urban	Y
1	LP001003	Male	Yes	1	Graduate	No	4583	1508.0	128.0	360.0	1.0	Rural	N
2	LP001005	Male	Yes	0	Graduate	Yes	3000	0.0	66.0	360.0	1.0	Urban	Y
3	LP001006	Male	Yes	0	Not Graduate	No	2583	2358.0	120.0	360.0	1.0	Urban	Y
4	LP001008	Male	No	0	Graduate	No	6000	0.0	141.0	360.0	1.0	Urban	Y

Upon initial data analysis, I determined that label encoding would be necessary and missing values were present and needed handling. The following were used for initial data analysis:

```
# List columns and data types for each column
df.info()
```

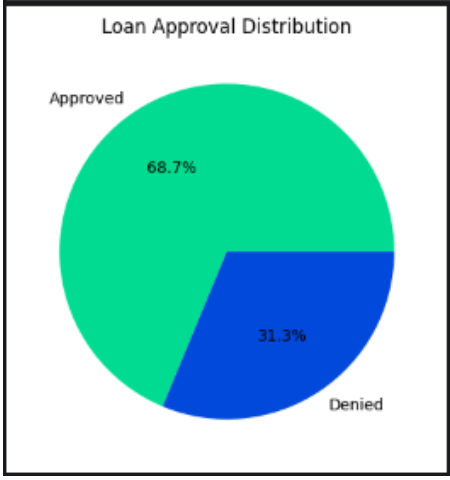
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 614 entries, 0 to 613
Data columns (total 13 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   Loan_ID            614 non-null    object
 1   Gender             601 non-null    object
 2   Married            611 non-null    object
 3   Dependents         599 non-null    object
 4   Education          614 non-null    object
 5   Self_Employed      582 non-null    object
 6   ApplicantIncome    614 non-null    int64
 7   CoapplicantIncome  614 non-null    float64
 8   LoanAmount         592 non-null    float64
 9   Loan_Amount_Term   600 non-null    float64
 10  Credit_History     564 non-null    float64
 11  Property_Area      614 non-null    object
 12  Loan_Status        614 non-null    object
dtypes: float64(4), int64(1), object(8)
memory usage: 62.5+ KB
```

```
# Check the number of missing values
df.isna().sum()
```

```
Loan_ID              0
Gender              13
Married              3
Dependents          15
Education            0
Self_Employed       32
ApplicantIncome      0
CoapplicantIncome    0
LoanAmount          22
Loan_Amount_Term    14
Credit_History      50
Property_Area        0
Loan_Status          0
dtype: int64
```

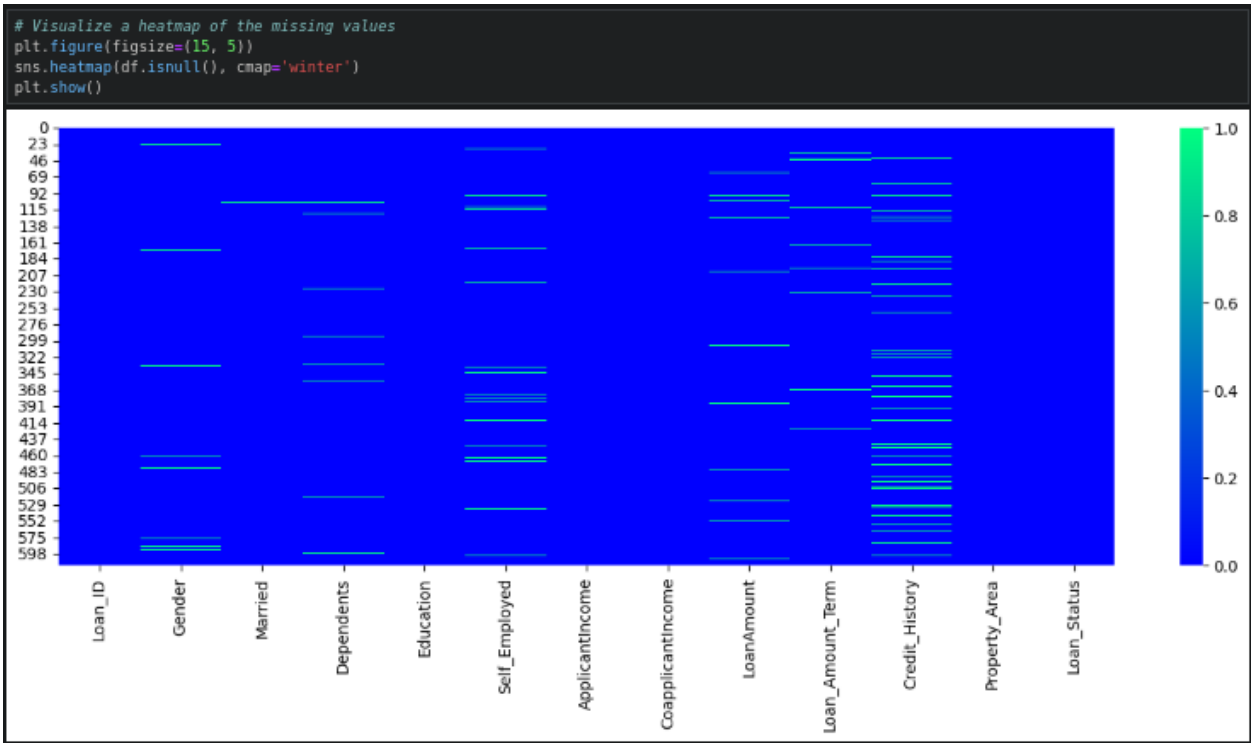
With a pie chart, I visualized the loan approval balance within the dataset and determined there to be an imbalance, as seen here:

```
# Visualize the balance of loan approvals to see if we will need to fix any imbalance
plt.pie(df['Loan_Status'].value_counts().values, labels=['Approved', 'Denied'], colors=['#00db92', '#0049db'], autopct='%1.1f%%')
plt.title('Loan Approval Distribution')
plt.show()
```



Then, I determined how to handle the missing values by visualizing the distribution of missing values. I determined the data to be missing at random (MAR) with this heat map:

```
# Visualize a heatmap of the missing values
plt.figure(figsize=(15, 5))
sns.heatmap(df.isnull(), cmap='winter')
plt.show()
```



Credit history was the most prevalent feature with missing values, and due to the high correlation observed between credit history and loan eligibility, those values were dropped instead of assumed and filled. Loan ID was also dropped due to its irrelevance. All other missing values were handled by filling them with the most common value (mode) and the average for loan amount, seen below:

```
# Drop the Loan ID as it's not relevant
df = df.drop(['Loan_ID'], axis = 1)

# Remove missing credit history entries from the dataset
df = df[df['Credit_History'].notnull()]

# Loan amount should be filled with the average loan amount
df['LoanAmount'] = df['LoanAmount'].fillna(df.LoanAmount.mean())

# Fill missing data using the most common variable
df['Gender'] = df['Gender'].fillna(df.Gender.mode()[0])
df['Married'] = df['Married'].fillna(df.Married.mode()[0])
df['Dependents'] = df['Dependents'].fillna(df.Dependents.mode()[0])
df['Self_Employed'] = df['Self_Employed'].fillna(df.Self_Employed.mode()[0])
df['Loan_Amount_Term'] = df['Loan_Amount_Term'].fillna(df.Loan_Amount_Term.mode()[0])
```

Once all missing values were filled, I continued with label encoding of categorical columns and printed them to confirm the changes with the following code:

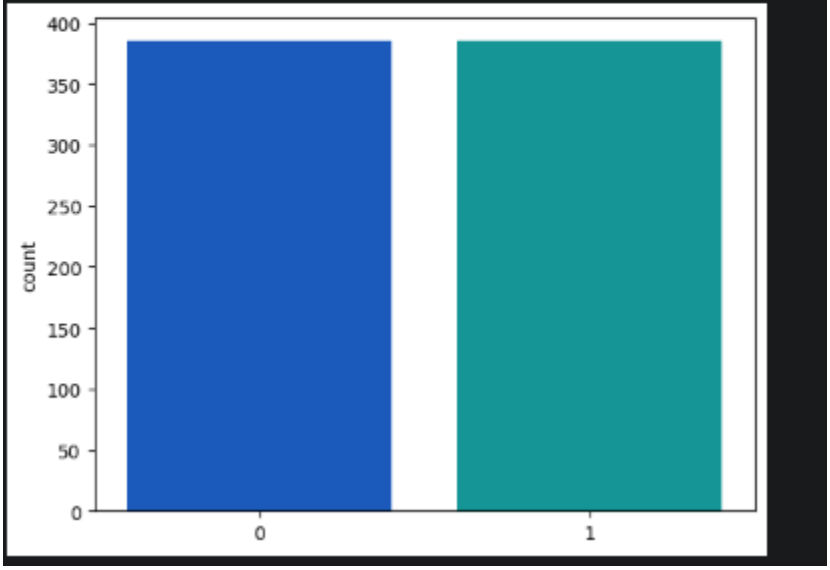
```
# Label encoding, where categorical columns are encoded to be numerical.
le = LabelEncoder()
for col in categorical_cols:
    df[col] = le.fit_transform(df[col])
    print(df[col].value_counts(), "\n")
```

Then, I defined two datasets, where the independent features in X are a one-to-one mapping of the dependent variables (targets) in y, before balancing and scaling the testing dataset.

```
# Set up the feature and target datasets
X = df.drop(['Loan_Status'], axis=1).values
y = df['Loan_Status'].values
```

To balance the dataset, I used RandomOverSampler to over-sample the minority classes and ensured balance with a bar graph of target classes:

```
# Balance the data by oversampling. SMOTE had lesser results
ros = RandomOverSampler(random_state=0)
X, y = ros.fit_resample(X, y)

# Visualize the new balance
sns.countplot(x=y, hue=y, legend=False, palette='winter')
plt.show()
```



After the dataset was balanced, it was split into testing and training datasets that were used to train and evaluate the model's performance and the feature datasets (X_train and X_test) were then scaled with StandardScaler:

```
# Split the data into two sets: training (75%) and testing (25%)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=0)

# Normalize the data with scaling
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.fit_transform(X_test)

print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
```

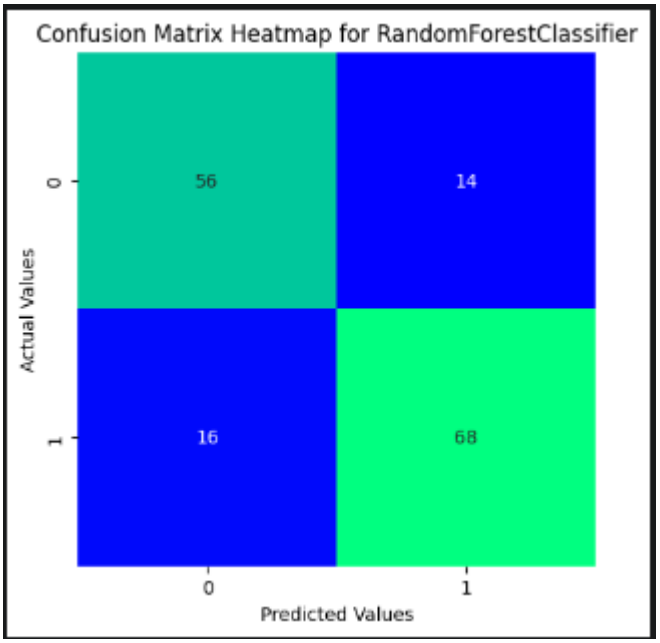
```
(616, 11) (154, 11) (616,) (154,)
```

The datasets at this point have now been normalized and are ready to be used to fit and evaluate each model's performance. This was done through a utility function I wrote, fit_and_eval, that was designed to fit and evaluate each model with the same process, providing consistent evaluation results across all models. This function fits the model, makes predictions on test data, captures metrics such as accuracy, recall, precision, f1 score, and ROC AUC score, performs cross-validation, and outputs the evaluation in text form as well as a confusion matrix heat map.

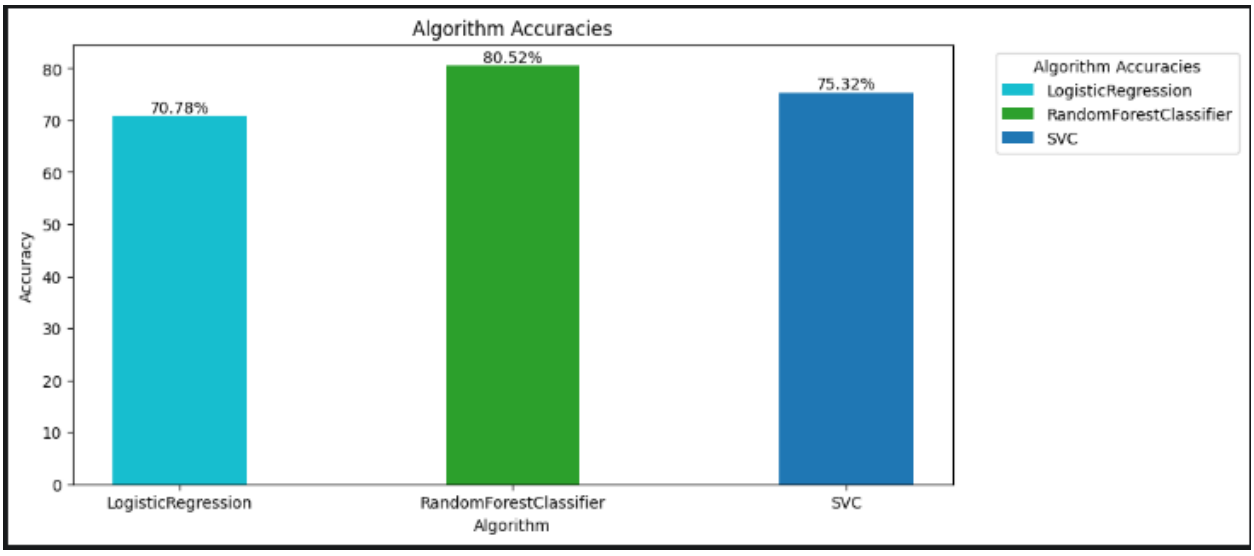
```
# Define the algorithms that are to be evaluated
algorithms = {
    'lr':  LogisticRegression(random_state=0, max_iter=500),
    'rfc': RandomForestClassifier(random_state=0),
    'svc': SVC(random_state=0, kernel='rbf')
}

# Define dictionaries for classifiers and accuracies
clfs = {}
accuracies = {}

for algo in algorithms:
    # Fit and evaluate classifiers and accuracies
    clfs[algo], accuracies[algo] = fit_and_eval(algorithms[algo])

    # Adds a space between algorithm outputs
    if algo != 'svc':
        print()
```

A sample output for Random Forest Classifier can be seen here, with the other two algorithms omitted:

```
Metrics for RandomForestClassifier:
Accuracy: 0.8051948051948052
Precision: 0.8292682926829268
Recall: 0.8095238095238095
F1 Score: 0.8192771084337348
ROC AUC Score: 0.8047619047619049
Mean CV Accuracy: 0.9025974025974026

Classification Report:
              precision    recall  f1-score   support

           0       0.78      0.80      0.79        70
           1       0.83      0.81      0.82        84

    accuracy                           0.81       154
   macro avg       0.80      0.80      0.80       154
weighted avg       0.81      0.81      0.81       154
```



Since more than one algorithm was evaluated, I compare the accuracy results for each algorithm and pick one to proceed with hyperparameter tuning:



Random Forest Classifier results in the highest accuracy, so I set up a GridSearchCV parameter grid to begin hyperparameter tuning for best results:

```
# Grid search can be very resource intensive when a lot of params are evaluated, so a limited number of params were specified.

# Define a dictionary of hyperparameters for the grid search
param_grid = {
    'criterion': ['entropy', 'gini'],
    'max_depth': [2, 5, 10, 20],
    'max_features': ['sqrt', 'log2'],
    'n_estimators': [10, 25, 50, 100, 500]
}

# Define a default Random Forest model
estimator = RandomForestClassifier(random_state=0)

# Define the GridSearchCV model
grid_search = GridSearchCV(estimator=estimator, param_grid=param_grid, cv=5, verbose=True)

# Fit and evaluate the grid search model
clf = fit_and_eval(grid_search)[0]

# Output the best parameters and score found
print(grid_search.best_params_)
print(grid_search.best_score_)
```

The fine-tuned model resulted in negligible changes in accuracy, but I used the best parameters suggested by the GridSearchCV model and ran fit_and_eval on the fine-tuned Random Forest Classifier model to provide a final model to use for predictions.

```
# Create a new Random Forest model with the fine-tuned hyperparameters
rfc = RandomForestClassifier(random_state=0, criterion='gini', max_depth=10, max_features='sqrt', n_estimators=500)

# Fit and evaluate the Random Forest model
clf = fit_and_eval(rfc)[0]
```

Machine Learning

Three machine learning algorithms were trained and evaluated in the development of this model: Logistic Regression, Random Forest Classifier, and Support Vector Classifier.

Logistic Regression is a supervised machine learning algorithm primary used for binary classification problems and uses probability to predict the target class. The method was developed by fitting the sigmoid function to the training data and optimizing coefficients through gradient descent. Logistic Regression was chosen as a possible algorithm due to its simplicity and effectiveness in binary classification problems.

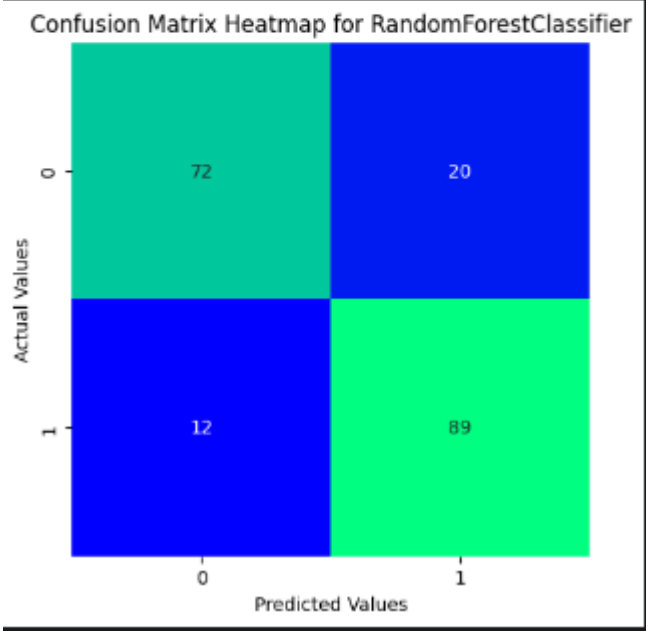
Random Forest Classifier is a machine learning algorithm that is an extension of the bagging ensemble learning method. It builds a Random Forest of decision trees with random subsets of data during training and outputs the mode of the classes for classification. The method was developed by combining various decision trees that were each trained on different subsets of data and features. The Random Forest Classifier was chosen for its ability to handle complex relationships in the dataset, reduce overfitting, and provide feature importance metrics.

Support Vector Classifier is a supervised learning algorithm that performs classification by finding the hyperplane that best separates the feature classes. The method was developed by selecting the ‘linear’ kernel function, which is responsible for transforming the input data into the required form. The hyperplane is then optimized to minimize the classification error and determine optimal coefficients for the support vectors. The Support Vector Classifier was selected for its ability to generalize new, unseen data, and for its effectiveness in finding decision boundaries in complex relationships.

Validation

The accuracy of all three models was determined using k-fold cross-validation to provide a multitude of metrics for each fold, such as accuracy, precision, recall, and ROC AUC score. The average results of these metrics provided me with insights into each model's overall performance and its effectiveness in correctly predicting the correct class. An evaluation report for each model was generated after fitting the model and was the basis for selecting a final model for fine-tuning. The final model's evaluation report, coming in at 83.41% accuracy and 86.75% cross-validation mean accuracy, can be seen below:

```
Metrics for RandomForestClassifier:
Accuracy: 0.8341968911917098
Precision: 0.8165137614678899
Recall: 0.8811881188118812
F1 Score: 0.8476190476190476
ROC AUC Score: 0.8318984072320276
Mean CV Accuracy: 0.8675324675324674

Classification Report:
              precision    recall  f1-score   support

           0       0.86      0.78      0.82        92
           1       0.82      0.88      0.85       101

    accuracy                           0.83       193
   macro avg       0.84      0.83      0.83       193
weighted avg       0.84      0.83      0.83       193
```



Visualizations

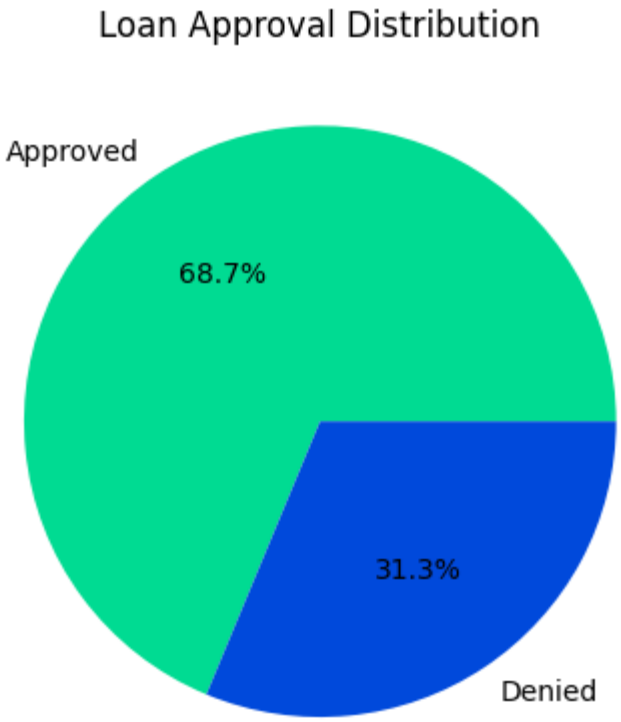


Figure 1: Loan approval distribution

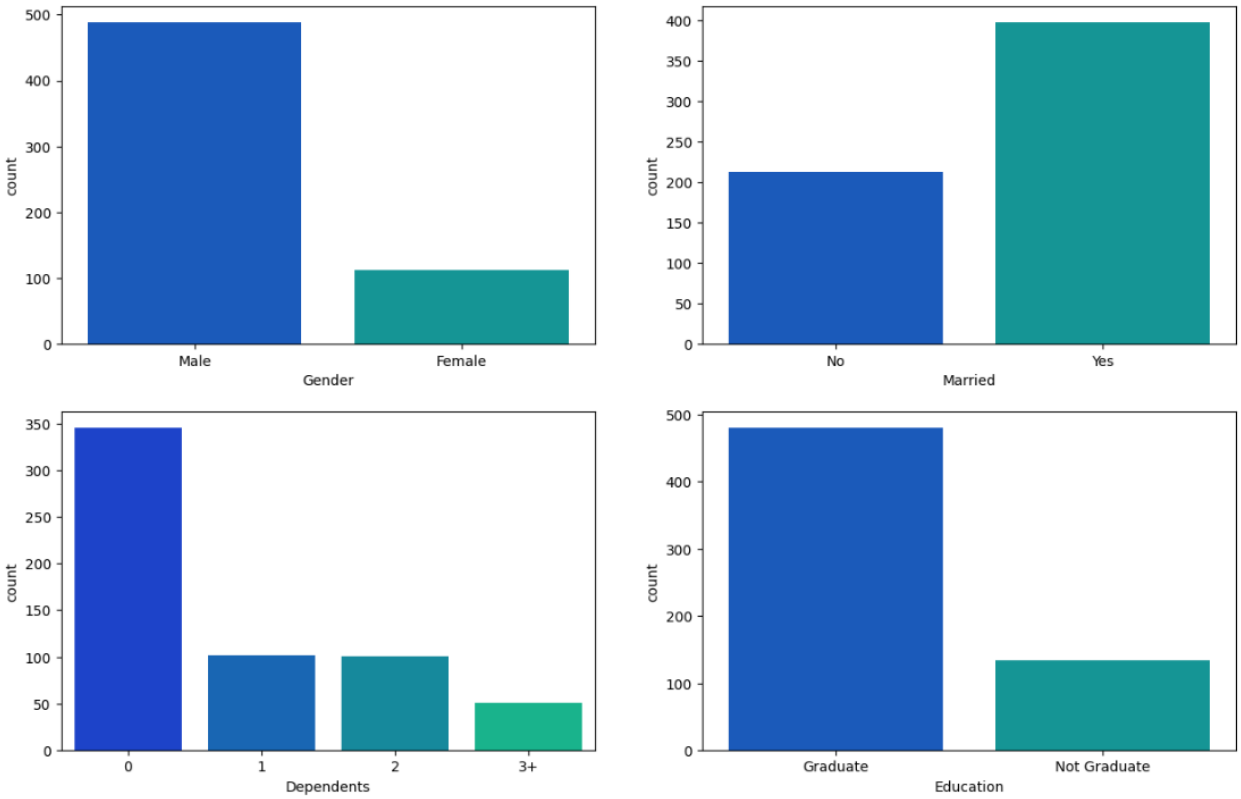


Figure 2.1: Distribution of independent features

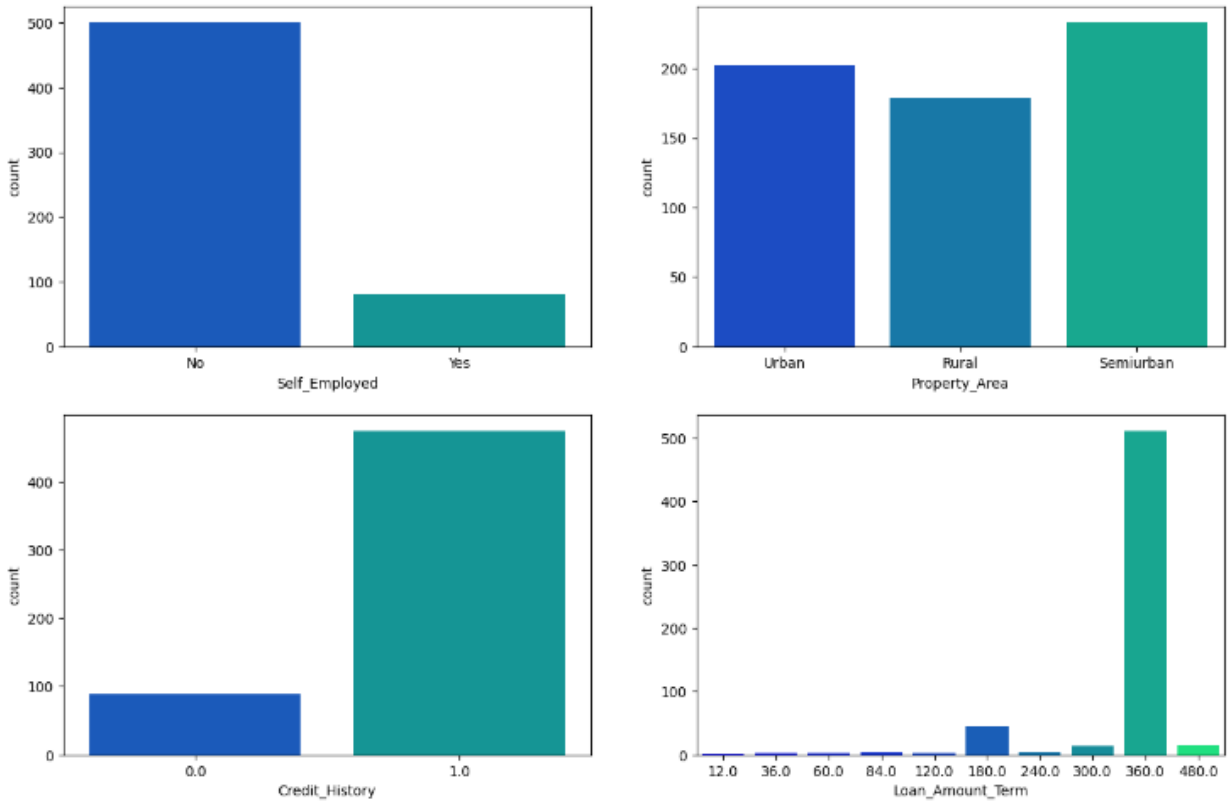


Figure 2.2: Distribution of independent features (cont.)

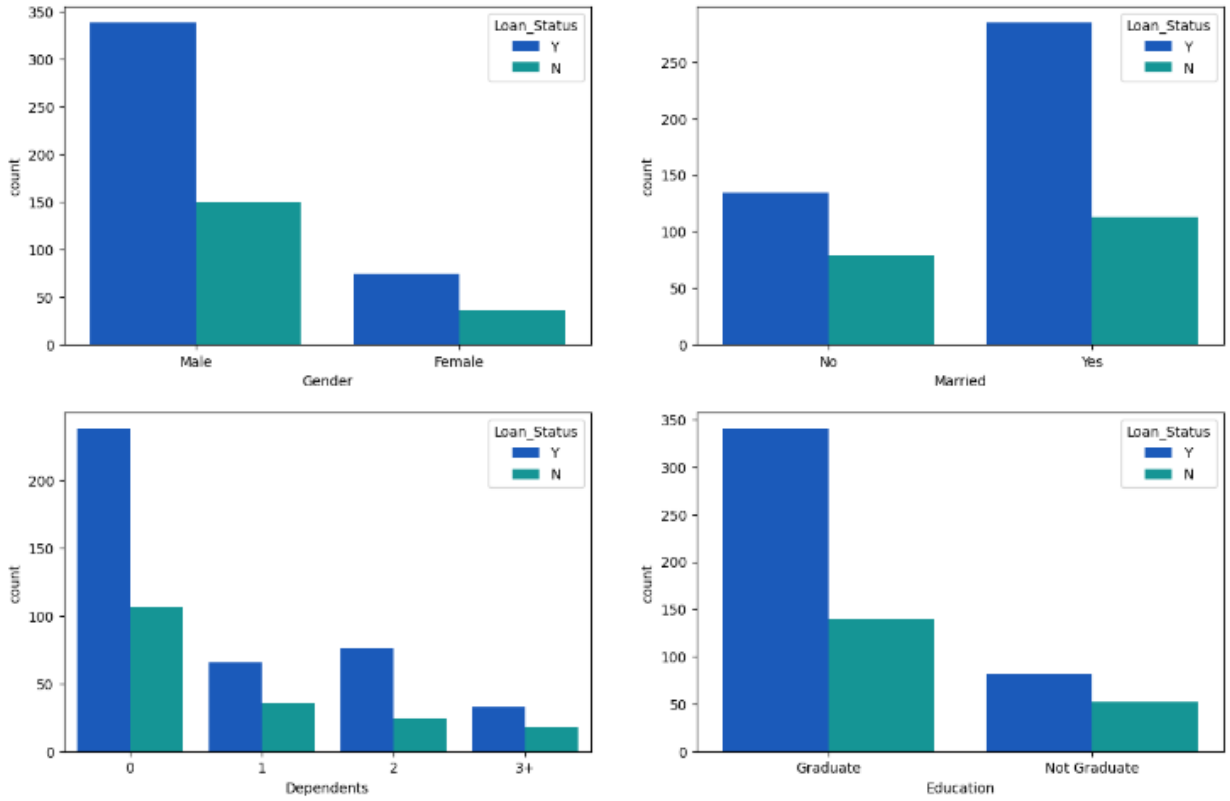


Figure 3.1: Loan approval distribution for independent features

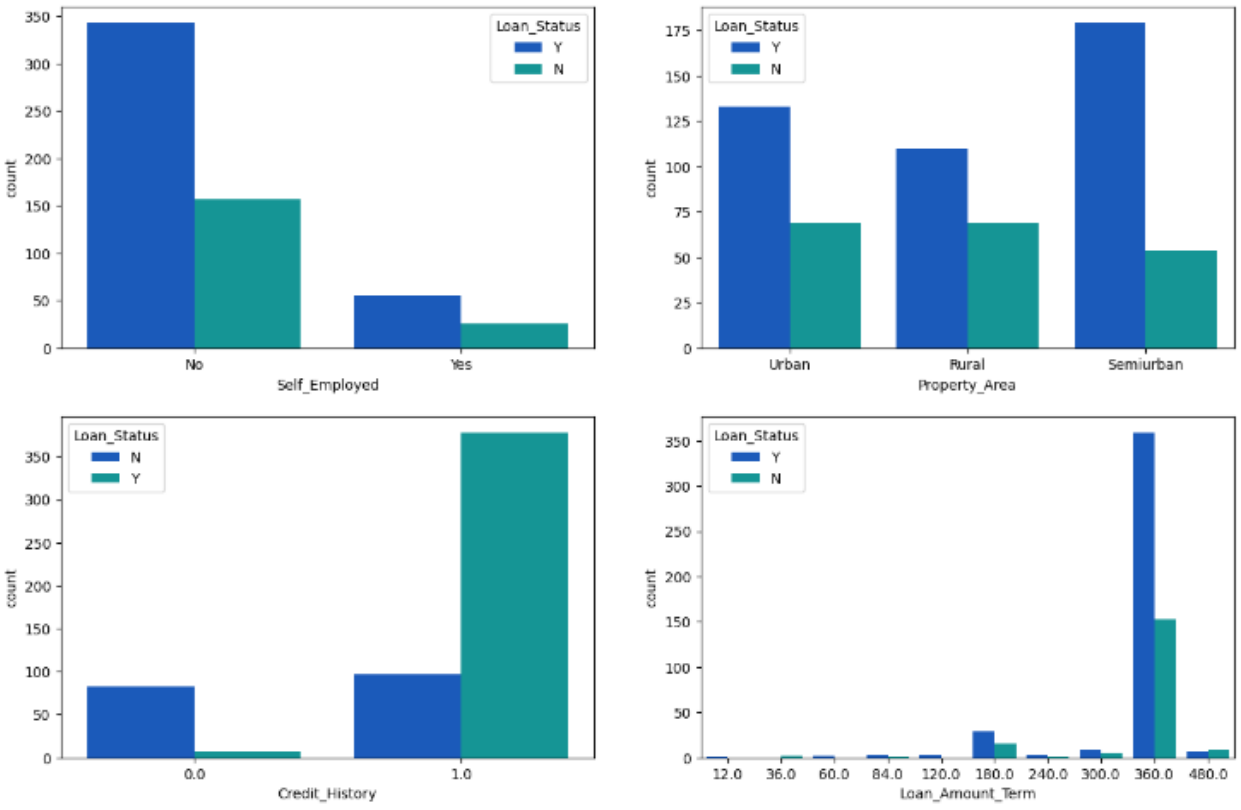


Figure 3.2: Loan approval distribution for independent features (cont.)

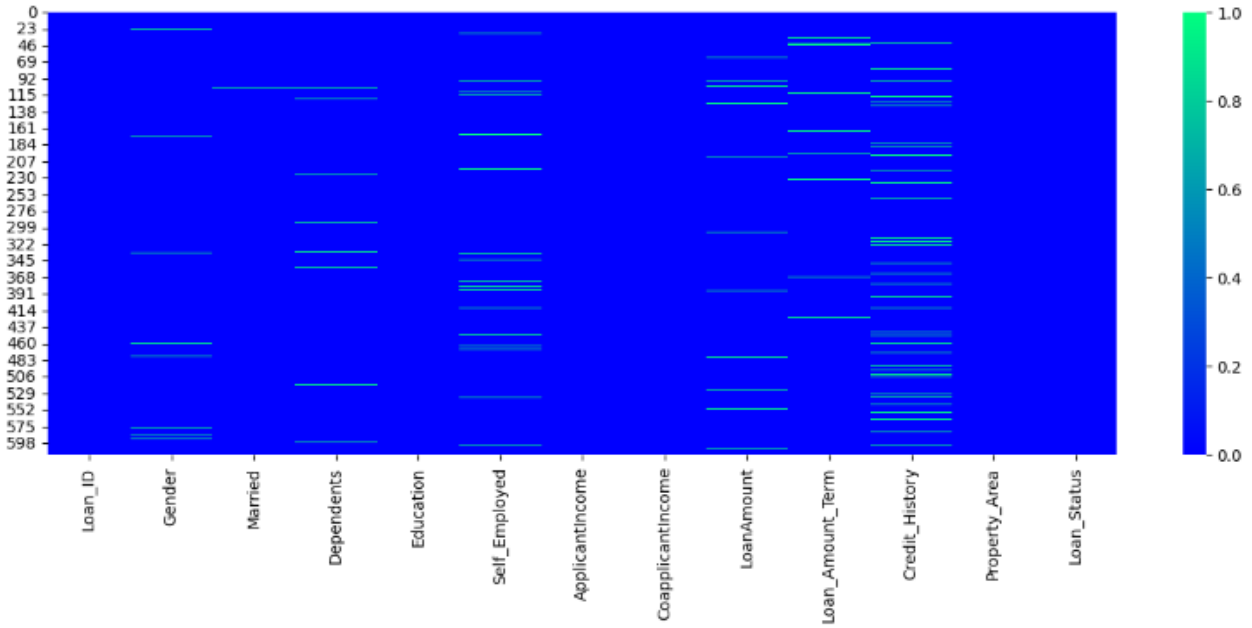


Figure 4: Matrix heat map for missing values

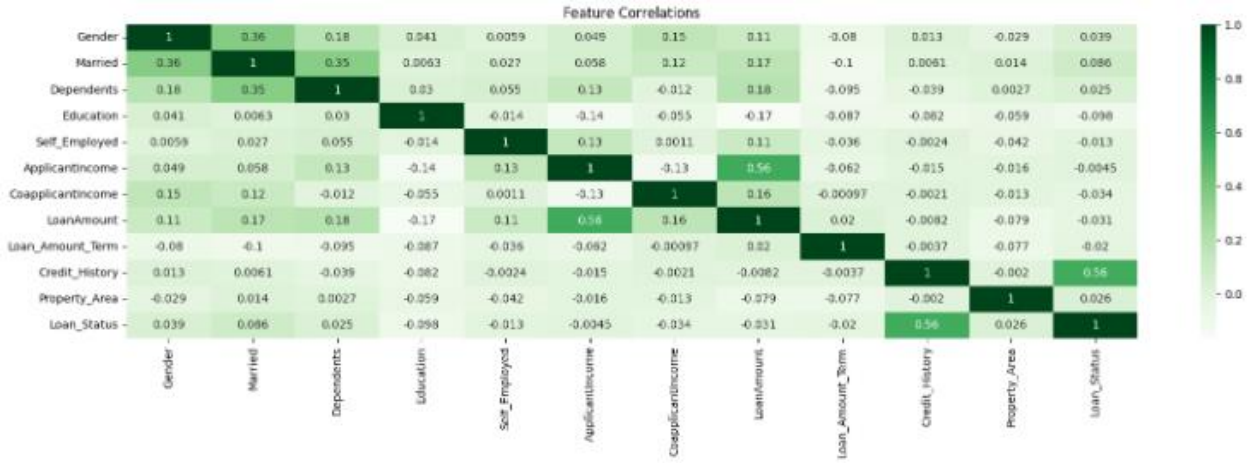


Figure 5: Feature correlation matrix heat map

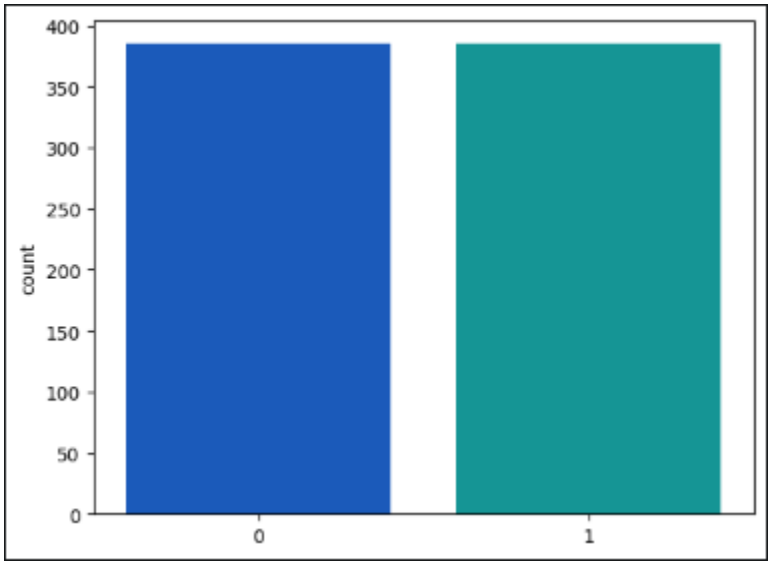


Figure 6: Dependent variable distribution after oversampling

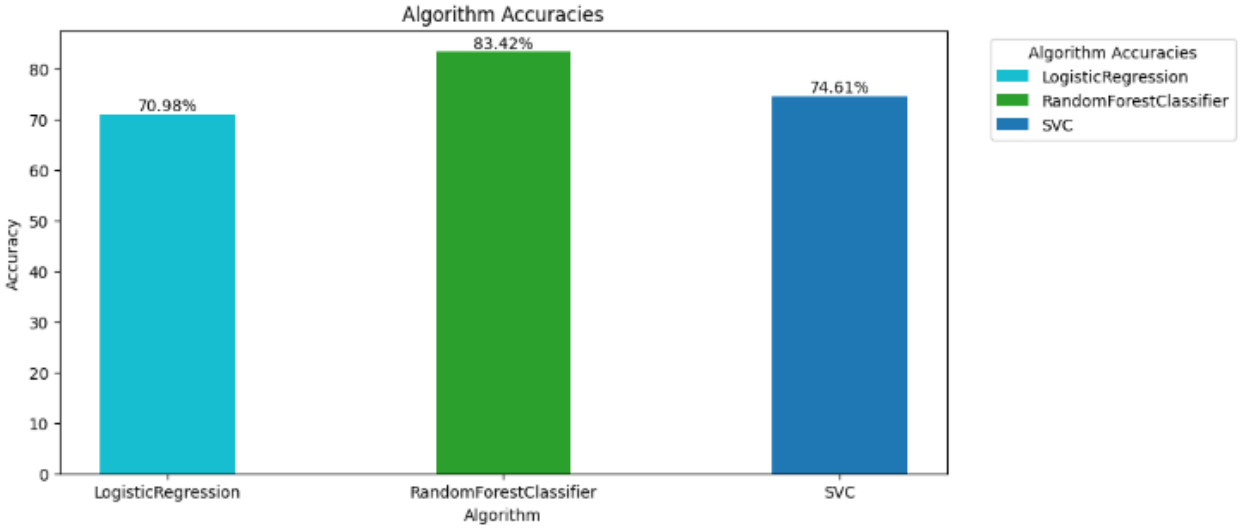


Figure 7: Algorithm accuracy comparison for trained models

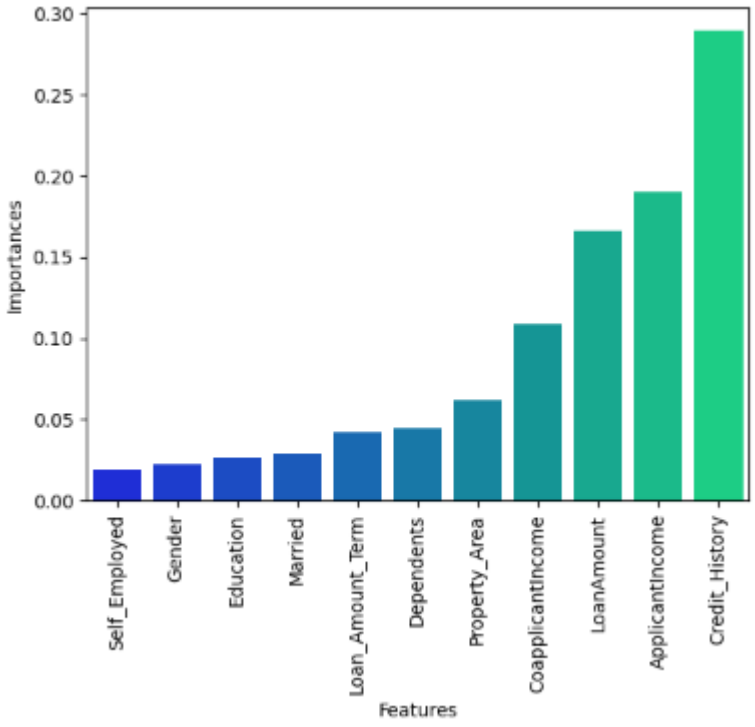
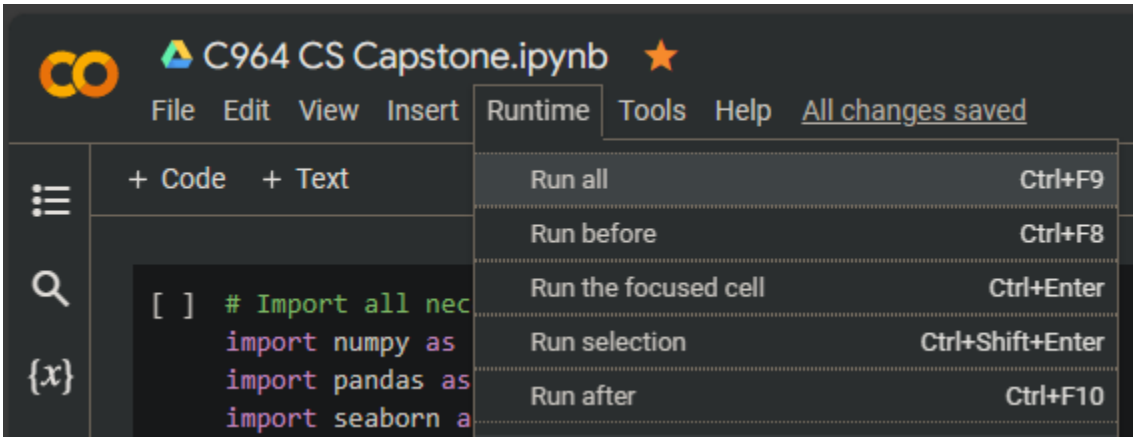


Figure 8: Feature importances for fine-tuned model

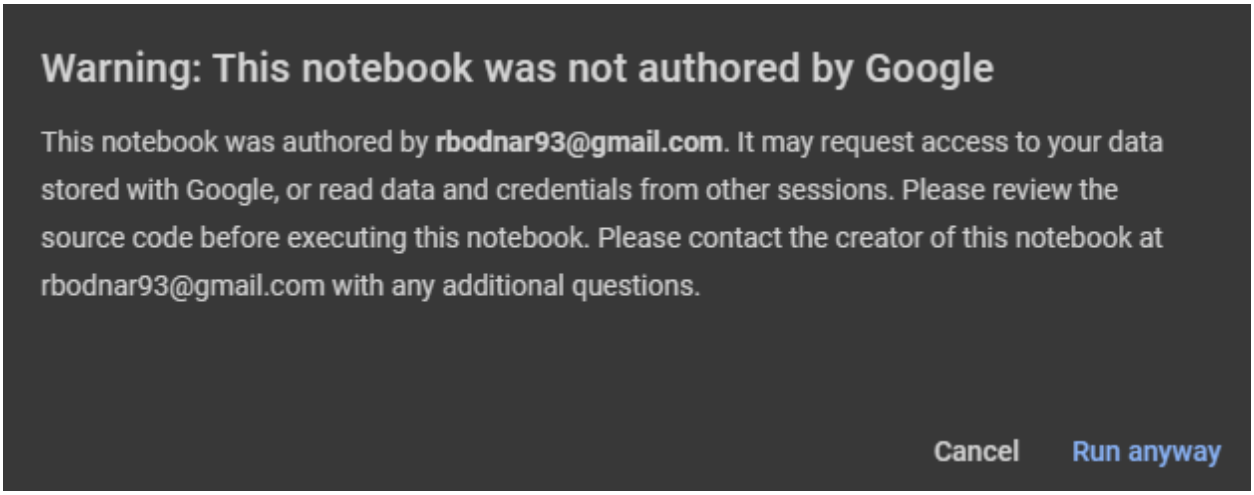
User Guide

1. Open a web browser and navigate to the Google Colaboratory online notebook link [2].

2. In the menu at the top, select **Runtime** and then **Run all**, as seen below:



3. If you receive a popup stating this notebook was not authored by Google, click **Run anyway**:



5. Scroll down to the bottom of the page and wait for the user interface to begin. This will take some time, please be patient.

You should see the following under the last cell at the bottom of the page when it completes:

6. Answer the prompt questions to receive a loan eligibility prediction.

References

[1] Devzohaib. (2022). Eligibility Prediction for Loan. Retrieved 12/18/2023 from https://www.kaggle.com/datasets/devzohaib/eligibility-prediction-for-loan.

[2] Google. (n.d.). Google Colaboratory. https://colab.research.google.com/drive/1idhWC1-8uCMqcpl1CTYpKDxWe-0ljfqB?usp=sharing