Credit Card Default Prediction

Mohammed El-Bathy
Deaprtement of Computer Science and Engineering
Oakland Universityn
Rochester, Michigan, USA
elbathy@oakland.edu

***Abstract*— This project is designed to predict credit card defaults by analyzing a variety of financial and demographic data attributes. The dataset undergoes multiple stages, including data preprocessing, feature engineering, and model training, which culminate in the evaluation and comparison of three machine learning models: Decision Tree, Logistic Regression, and Random Forest. Each model is tested through various evaluation metrics to determine its effectiveness, and the results are carefully documented to support model selection**

I. Introduction

Credit card default prediction is a critical issue for financial institutions that aim to mitigate risk and minimize losses. By leveraging customer data such as payment history, credit limits, and demographic attributes, we can build predictive models to estimate the likelihood of default. This project explores the use of decision tree classifiers, focusing on preparing the data, handling class imbalance, and evaluating model performance.

II. Domain Knowledge

In the domain of credit risk, default refers to the failure to meet the legal obligations of debt repayment. Credit card issuers face significant financial risks due to defaults, so it's crucial to develop predictive models to assess the likelihood of default before issuing credit..

The features used in this analysis cover various aspects of customer behavior and demographics, such as credit limit, payment history, bill amounts, and personal attributes (e.g., age, gender, education, and marital status). Understanding these variables is essential to modeling the risk of default

III. Dataset Analysis & Understanding

All the headings in the main body are numbered (automatically).

A. *Data Characteristics*

The dataset contains 30 variables, including both numerical and categorical data. Key variables include:

- LIMIT_BAL: The credit limit assigned to the customer.
- AGE: The age of the customer
- PAY_0 **to** PAY_6: Repayment status in different months.
- BILL_AMT1 to BILL_AMT6: Amount of bill statements from the past six months.
- PAY_AMT1 to PAY_AMT6: Amount paid in the past six months.
- SEX, EDUCATION, MARRIAGE: Categorical features indicating demographic details

The target variable is default payment next month, indicating whether the customer defaulted (1) or not (0)

B. *Feature Analysis & Selection*

- The key financial variables are credit limit (LIMIT_BAL)**,** bill amounts (BILL_AMT1-6), and payment history (PAY_0-6). These features directly reflect the customer's ability to manage credit and meet payment obligations
- Demographic variables such as AGE, SEX, EDUCATION, and MARRIAGE also provide insights into the customer profile and their associated risk levels

IV. DATA PREPROCESSING FUNCTIONS

To ensure a smooth workflow, the project begins by importing the required libraries, which helps avoid any errors from missing dependencies. The import_libraries function accomplishes this initial step, setting the foundation for the project by confirming that all necessary packages are available. In addition we defined global variables.

```
import gradio as gr
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from io import BytesIO
from PIL import Image
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import StratifiedKFold, cross_val_score, GridSearchCV
from sklearn.metrics import confusion_matrix, classification_report, roc_auc_score
from sklearn.preprocessing import StandardScaler
from statsmodels.stats.outliers_influence import variance_inflation_factor
from statsmodels.tools.tools import add_constant
from imblearn.over_sampling import SMOTE
from sklearn import tree
```

```
# Global variables
data = None
encoded_data = None
X_resampled = None
y_resampled = None
dt_model = None
lr_model = None
rf_model = None
lr_results = None
rf_results = None
```

A. *Loading Dataset.*

The next step is loading the dataset through the load_dataset function. This function reads in the credit card client data, providing access to the information necessary for further processing and analysis.

```python
# Function 2: Load the Dataset
def load_dataset():
    global data
    file_path = 'default_of_credit_card_clients.csv'
    data = pd.read_csv(file_path, header=1)
    return data
```

```
load_dataset()
```

	ID	LIMIT_BAL	SEX	EDUCATION	MARRIAGE	AGE	PAY_0	PAY_2	PAY_3	PAY_4	...	BILL_AMT4	BILL_AMT5	BILL_AMT6	PAY_AMT1	PAY_AMT2	PAY_AMT3	PAY_AMT4	PAY_AMT5	PAY_AMT6
0	1	20000	2	2	1	24	2	2	-1	-1	...	0	0	0	0	689	0	0	0	0
1	2	120000	2	2	2	26	-1	2	0	0	...	3272	3455	3261	0	1000	1000	1000	0	2000
2	3	90000	2	2	2	34	0	0	0	0	...	14331	14948	15549	1518	1500	1000	1000	1000	5000
3	4	50000	2	2	1	37	0	0	0	0	...	28314	28959	29547	2000	2019	1200	1100	1069	1000
4	5	50000	1	2	1	57	-1	0	-1	0	...	20940	19146	19131	2000	36681	10000	9000	689	679
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
29995	29996	220000	1	3	1	39	0	0	0	0	...	88004	31237	15980	8500	20000	5003	3047	5000	1000
29996	29997	150000	1	3	2	43	-1	-1	-1	-1	...	8979	5190	0	1837	3526	8998	129	0	0
29997	29998	30000	1	2	2	37	4	3	2	-1	...	20878	20582	19357	0	0	22000	4200	2000	3100
29998	29999	80000	1	3	1	41	1	-1	0	0	...	52774	11855	48944	85900	3409	1178	1926	52964	1804
29999	30000	50000	1	2	1	46	0	0	0	0	...	36535	32428	15313	2078	1800	1430	1000	1000	1000

B. Statistical Summary.

Once the data is loaded, a statistical summary is generated using display_statistics, which gives insight into each feature's characteristics, such as mean, median, and standard deviation. This summary helps in identifying potential issues, such as outliers, and provides a foundational understanding of the dataset's distributions.

```python
# Function 3: Display statistical summary
def display_statistics():
    global data
    if data is not None:
        return data.describe().T
    else:
        return "Data not loaded."
```

```
display_statistics()
```

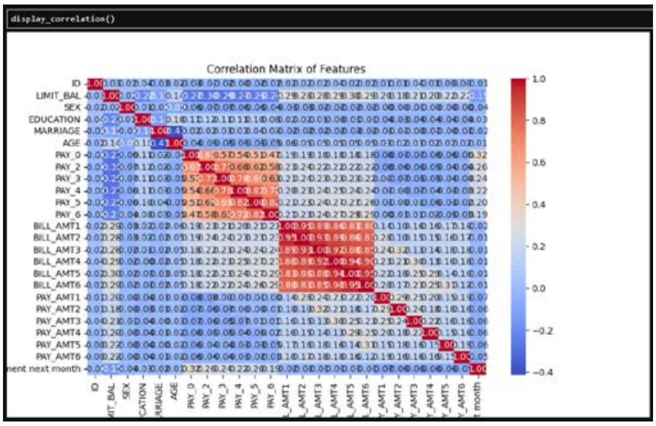
	count	mean	std	min	25%	50%	75%	max
ID	30000.0	15000.500000	8660.398374	1.0	7500.75	15000.5	22500.25	30000.0
LIMIT_BAL	30000.0	167484.322667	129747.661567	10000.0	50000.00	140000.0	240000.00	1000000.0
SEX	30000.0	1.603733	0.489129	1.0	1.00	2.0	2.00	2.0
EDUCATION	30000.0	1.853133	0.790349	0.0	1.00	2.0	2.00	6.0
MARRIAGE	30000.0	1.551867	0.521970	0.0	1.00	2.0	2.00	3.0
AGE	30000.0	35.485500	9.217904	21.0	28.00	34.0	41.00	79.0
PAY_0	30000.0	-0.016700	1.123802	-2.0	-1.00	0.0	0.00	8.0
PAY_2	30000.0	-0.133767	1.197186	-2.0	-1.00	0.0	0.00	8.0
PAY_3	30000.0	-0.166200	1.196868	-2.0	-1.00	0.0	0.00	8.0
PAY_4	30000.0	-0.220667	1.169139	-2.0	-1.00	0.0	0.00	8.0
PAY_5	30000.0	-0.266200	1.133187	-2.0	-1.00	0.0	0.00	8.0
PAY_6	30000.0	-0.291100	1.149988	-2.0	-1.00	0.0	0.00	8.0
BILL_AMT1	30000.0	51223.330900	73635.860576	-165580.0	3558.75	22381.5	67091.00	964511.0
BILL_AMT2	30000.0	49179.075167	71173.768783	-69777.0	2984.75	21200.0	64006.25	983931.0
BILL_AMT3	30000.0	47013.154800	69349.387427	-157264.0	2666.25	20088.5	60164.75	1664089.0
BILL_AMT4	30000.0	43262.948967	64332.856134	-170000.0	2326.75	19052.0	54506.00	891586.0
BILL_AMT5	30000.0	40311.400967	60797.155770	-81334.0	1763.00	18104.5	50190.50	927171.0
BILL_AMT6	30000.0	38871.760400	59554.107537	-339603.0	1256.00	17071.0	49198.25	961664.0
PAY_AMT1	30000.0	5663.580500	16563.280354	0.0	1000.00	2100.0	5006.00	873552.0

C. Correlation Matrix

The display_correlation function then visualizes the correlation matrix, which reveals relationships between variables. This matrix assists in detecting multicollinearity among features, which is critical for certain models that can be sensitive to correlated inputs.

```python
# Function 4: Display correlation matrix
def display_correlation():
    global data
    if data is not None:
        fig, ax = plt.subplots(figsize=(10, 6))
        correlation_matrix = data.corr()
        sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt='.2f', ax=ax)
        ax.set_title('Correlation Matrix of Features')

        buf = BytesIO()
        fig.savefig(buf, format="png")
        buf.seek(0)
        plt.close(fig)
        return Image.open(buf)
    else:
        return "Data not loaded."
```



D. Outlier Handling

Next, the outlier_detection function identifies and caps extreme values by setting limits at the 1st and 99th percentiles. This step is essential because outliers can skew the data and negatively impact model performance, particularly in distance-based models.

```python
# Function 5: Outlier Detection and Handling
def outlier_detection():
    global data
    if data is not None:
        numeric_columns = ['LIMIT_BAL', 'AGE', 'BILL_AMT1', 'BILL_AMT2', 'BILL_AMT3',
                           'BILL_AMT4', 'BILL_AMT5', 'BILL_AMT6', 'PAY_AMT1',
                           'PAY_AMT2', 'PAY_AMT3', 'PAY_AMT4', 'PAY_AMT5', 'PAY_AMT6']
        for column in numeric_columns:
            upper_limit = data[column].quantile(0.99)
            lower_limit = data[column].quantile(0.01)
            data[column] = np.clip(data[column], lower_limit, upper_limit)
        return "Outliers capped at 1st and 99th percentiles."
    else:
        return "Data not loaded."
```

```
outlier_detection()

'Outliers capped at 1st and 99th percentiles.'
```

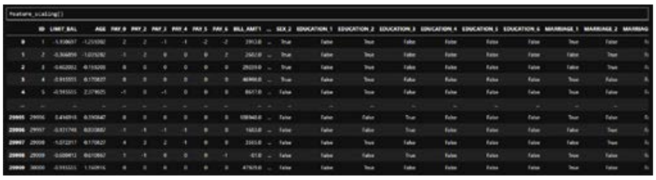
E. Encoding Categorical Variables

The display_correlation function then visualizes the correlation matrix, which reveals relationships between variables. This matrix assists in detecting multicollinearity among features, which is critical for certain models that can be sensitive to correlated inputs.

```python
# Function 6: Encoding Categorical Variables
def encoding_categorical():
    global data, encoded_data
    if data is not None:
        encoded_data = pd.get_dummies(data, columns=['SEX', 'EDUCATION', 'MARRIAGE'],
                                      drop_first=True)
        return encoded_data
    else:
        return "Data not loaded."
```

F. Feature Scaling

```
# Function 7: Feature Scaling
def feature_scaling():
    global encoded_data
    if encoded_data is not None:
        continuous_columns = ['LIMIT_BAL', 'AGE', 'PAY_AMT1', 'PAY_AMT2', 'PAY_AMT3',
                              'PAY_AMT4', 'PAY_AMT5', 'PAY_AMT6']
        scaler = StandardScaler()
        encoded_data[continuous_columns] = scaler.fit_transform(encoded_data[continuous_columns])
        return encoded_data
    else:
        return "Encoded data not available."
```



G. SMOTE

To address class imbalance, which is a common issue in binary classification, apply_smote applies Synthetic Minority Over-sampling Technique (SMOTE) to the dataset. This technique balances the data by oversampling the minority class, thereby improving the model's ability to learn accurate decision boundaries for both classes.

```
# Function 8: Apply SMOTE
def apply_smote():
    global encoded_data, X_resampled, y_resampled
    if encoded_data is not None:
        X = encoded_data.drop(columns=['default payment next month'])
        y = encoded_data['default payment next month']
        smote = SMOTE(random_state=42)
        X_resampled, y_resampled = smote.fit_resample(X, y)
        return f"SMOTE applied: Resampled dataset shape: {X_resampled.shape}"
    else:
        return "Encoded data not available."
```

```
apply_smote()

'SMOTE applied: Resampled dataset shape: (46728, 31)'
```

H. Variance Inflation Factor

Subsequently, we calculate_the Variance Inflation Factor assesses multicollinearity among the features by calculating the VIF for each feature. High VIF values indicate redundancy, which may degrade model performance, particularly in linear models. After any necessary modifications, recalculate_vif is used to verify that multicollinearity issues have been resolved, thereby ensuring model stability

```
# Function 9: Calculate VIF
def calculate_vif():
    global encoded_data
    if encoded_data is not None:
        numeric_data = encoded_data.select_dtypes(include=[np.number])
        X_vif = add_constant(numeric_data)
        vif_data = pd.DataFrame()
        vif_data["Feature"] = X_vif.columns
        vif_data["VIF"] = [variance_inflation_factor(X_vif.values, i) for i in range(X_vif.shape[1])]
        return vif_data
    else:
        return "Encoded data not available."
```

```
calculate_vif()
```

	Feature	VIF
0	const	5.502704
1	ID	1.012959
2	LIMIT_BAL	1.513392
3	AGE	1.024821
4	PAY_0	2.008695
5	PAY_2	3.212227
6	PAY_3	3.743819
7	PAY_4	4.402964
8	PAY_5	4.838624
9	PAY_6	3.311836
10	BILL_AMT1	13.271621
11	BILL_AMT2	24.583129
12	BILL_AMT3	21.497720
13	BILL_AMT4	21.390585
14	BILL_AMT5	25.037119
15	BILL_AMT6	14.797413
16	PAY_AMT1	1.762232
17	PAY_AMT2	1.885662
18	PAY_AMT3	1.778416
19	PAY_AMT4	1.764788
20	PAY_AMT5	1.799162
21	PAY_AMT6	1.233234
22	default payment next month	1.143918

I. Recalculate Variance Inflation Factor

After any necessary modifications, recalculate vif is used to verify that multicollinearity issues have been resolved, thereby ensuring model stability.

```
# Function 10: Recalculate VIF
def recalculate_vif():
    global encoded_data
    if encoded_data is not None:
        numeric_data = encoded_data.select_dtypes(include=[np.number])
        X_vif_updated = add_constant(numeric_data)
        vif_data_updated = pd.DataFrame()
        vif_data_updated["Feature"] = X_vif_updated.columns
        vif_data_updated["VIF"] = [variance_inflation_factor(X_vif_updated.values, i) for i in range(X_vif_updated.shape[1])]
        return vif_data_updated
    else:
        return "Encoded data not available."
```

```
recalculate_vif()
```

	Feature	VIF
0	const	5.502704
1	ID	1.012959
2	LIMIT_BAL	1.513392
3	AGE	1.024821
4	PAY_0	2.008695
5	PAY_2	3.212227
6	PAY_3	3.743819
7	PAY_4	4.402964
8	PAY_5	4.838624
9	PAY_6	3.311836
10	BILL_AMT1	13.271621
11	BILL_AMT2	24.583129
12	BILL_AMT3	21.497720
13	BILL_AMT4	21.390585
14	BILL_AMT5	25.037119
15	BILL_AMT6	14.797413
16	PAY_AMT1	1.762232
17	PAY_AMT2	1.885662
18	PAY_AMT3	1.778416
19	PAY_AMT4	1.764788
20	PAY_AMT5	1.799162
21	PAY_AMT6	1.233234
22	default payment next month	1.143918

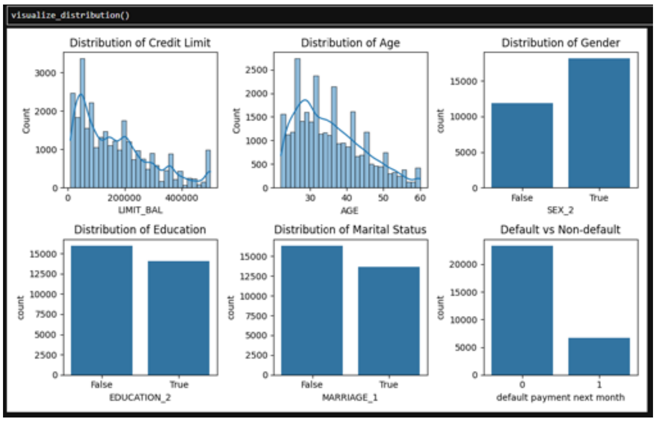
J. *Visualize Distribution*

To further explore the dataset, visualize distribution generates plots that display the distribution of key features. This visualization provides additional insights into data patterns, helping identify skewness or other anomalies that might necessitate transformation or further preprocessing.

```
# Function 11: Visualize Distribution
def visualize_distribution():
    global data, encoded_data
    if data is not None and encoded_data is not None:
        fig, axes = plt.subplots(2, 3, figsize=(10, 6))
        sns.histplot(data['LIMIT_BAL'], bins=30, kde=True, ax=axes[0, 0]).set_title('Distribution of Credit Limit')
        sns.histplot(data['AGE'], bins=30, kde=True, ax=axes[0, 1]).set_title('Distribution of Age')
        sns.countplot(x='SEX_2', data=encoded_data, ax=axes[0, 2]).set_title('Distribution of Gender')
        sns.countplot(x='EDUCATION_2', data=encoded_data, ax=axes[1, 0]).set_title('Distribution of Education')
        sns.countplot(x='MARRIAGE_1', data=encoded_data, ax=axes[1, 1]).set_title('Distribution of Marital Status')
        sns.countplot(x='default payment next month', data=data, ax=axes[1, 2]).set_title('Default vs Non-default')
        plt.tight_layout()

        buf = BytesIO()
        fig.savefig(buf, format="png")
        buf.seek(0)
        plt.close(fig)
        return Image.open(buf)
    else:
        return "Data not loaded."
```

```
visualize_distribution()
```



V. MODEL TRAINING AND EVALUATION FUNCTIONS

A. *Decision Tree Model*

1) The first machine learning model employed in this project is a Decision Tree. The initialize_decision_tree function initializes and trains a Decision Tree classifier on the resampled dataset, establishing a straightforward and interpretable baseline model.

```
# Function 12: Initialize and Train Decision Tree
def initialize_decision_tree():
    global dt_model, X_resampled, y_resampled
    if X_resampled is not None and y_resampled is not None:
        dt_model = DecisionTreeClassifier(random_state=42)
        dt_model.fit(X_resampled, y_resampled)
        return "Decision Tree Classifier initialized and trained."
    else:
        return "SMOTE resampled data not available."
```

```
initialize_decision_tree()

'Decision Tree Classifier initialized and trained.'
```

2) To evaluate the Decision Tree, cross_validation_decision_tree performs cross-validation, calculating accuracy and ROC-AUC scores across folds. Cross-validation reduces the risk of overfitting and provides a reliable estimate of the model's generalization capabilities.

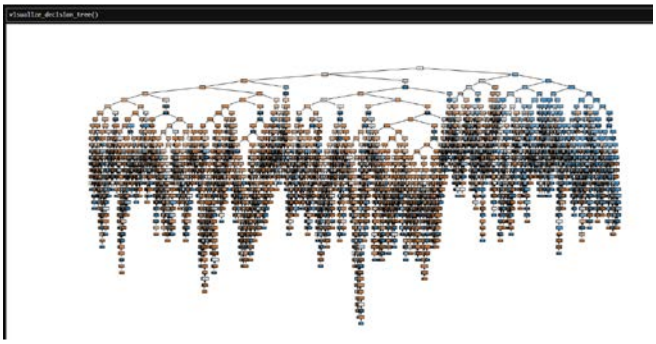
```
# Function 13: K-Fold Cross-Validation for Decision Tree
def cross_validation_decision_tree():
    global dt_model, X_resampled, y_resampled
    if dt_model is not None and X_resampled is not None:
        kf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
        accuracy_scores = cross_val_score(dt_model, X_resampled, y_resampled, cv=kf, scoring='accuracy')
        roc_auc_scores = cross_val_score(dt_model, X_resampled, y_resampled, cv=kf, scoring='roc_auc')
        return f"Mean Accuracy: {accuracy_scores.mean():.4f}, Std Dev: {accuracy_scores.std():.4f}\n" \
               f"Mean ROC-AUC: {roc_auc_scores.mean():.4f}, Std Dev: {roc_auc_scores.std():.4f}"
    else:
        return "Decision Tree or resampled data not available."
```

```
cross_validation_decision_tree()

'Mean Accuracy: 0.8043, Std Dev: 0.0021\nMean ROC-AUC: 0.8043, Std Dev: 0.0021'
```

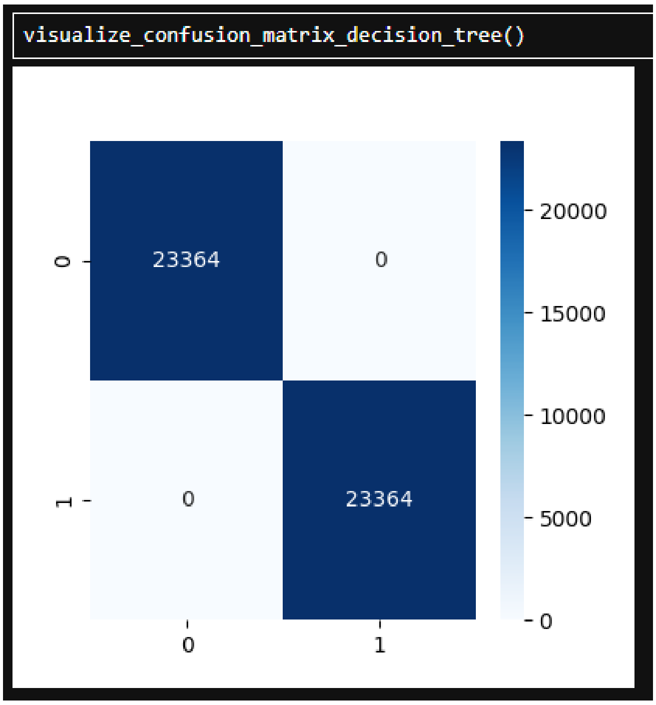
3) To enhance interpretability, visualize_decision_tree produces a graphical representation of the tree structure, displaying the decision-making paths and feature splits.

```
# Function 14: Visualize Decision Tree Structure
def visualize_decision_tree():
    global dt_model, X_resampled
    if dt_model is not None and X_resampled is not None:
        plt.figure(figsize=(20, 10))
        tree.plot_tree(dt_model, feature_names=X_resampled.columns, class_names=['No Default', 'Default'], filled=True)
        buf = BytesIO()
        plt.savefig(buf, format="png")
        buf.seek(0)
        plt.close()
        return Image.open(buf)
    else:
        return "Decision Tree model not trained."
```



4) The confusion matrix for the Decision Tree, generated by visualize_confusion_matrix_decision_tree, offers a detailed view of true versus predicted classifications. This matrix highlights the model's accuracy on each class, helping to identify any biases in prediction.

```
# Function 15: Confusion Matrix Visualization for Decision Tree
def visualize_confusion_matrix_decision_tree():
    global dt_model, X_resampled, y_resampled
    if dt_model is not None and X_resampled is not None:
        plt.figure(figsize=(4, 4))
        sns.heatmap(confusion_matrix(y_resampled, dt_model.predict(X_resampled)), annot=True, fmt='d', cmap='Blues')
        buf = BytesIO()
        plt.savefig(buf, format="png")
        buf.seek(0)
        plt.close()
        return Image.open(buf)
    else:
        return "Decision Tree or resampled data not available."
```



5) Finally, classification_report_dt provides a classification report with precision, recall, and F1-score metrics for the Decision Tree. This report allows for a comprehensive evaluation of the model's performance across each class, especially in imbalanced datasets.

```
# Function 16: Classification Report for Decision Tree
def classification_report_dt():
    global dt_model, X_resampled, y_resampled
    if dt_model is not None and X_resampled is not None:
        y_pred = dt_model.predict(X_resampled)
        report = classification_report(y_resampled, y_pred)
        return report
    else:
        return "Decision Tree or resampled data not available."
```

```
classification_report_dt()

'              precision    recall  f1-score   support\n\n           0       1.00      1.00      1.00     23364\n           1       1.00      1.00      1.00     23364\n\n    accuracy                           1.00     46728\n   macro avg       1.00      1.00      1.00     46728\nweighted avg       1.00      1.00      1.00     46728\n'
```

	precision	recall	f1-score	support
0	1.00	1.00	1.00	23364
1	1.00	1.00	1.00	23364
accuracy			1.00	46728
macro avg	1.00	1.00	1.00	46728
weighted avg	1.00	1.00	1.00	46728

B. *Logistic Regression Model*

1) Logistic Regression is the second model used in this analysis. The initialize_logistic_regression function initializes and trains the Logistic Regression model on the resampled data, providing a robust, interpretable model that serves as a benchmark.

```
# Function 17: Initialize and Train Logistic Regression
def initialize_logistic_regression():
    global lr_model, X_resampled, y_resampled
    if X_resampled is not None and y_resampled is not None:
        lr_model = LogisticRegression(max_iter=1000, random_state=42)
        lr_model.fit(X_resampled, y_resampled)
        return "Logistic Regression model initialized and trained."
    else:
        return "SMOTE resampled data not available."
```

2) Evaluation of the Logistic Regression model is conducted through evaluate_logistic_regression, which generates a classification report and calculates the ROC-AUC score. These metrics offer a thorough view of the model's ability to discriminate between classes and provide a basis for comparing it to other models.

```
# Function 18: Logistic Regression Evaluation
def evaluate_logistic_regression():
    global lr_model, X_resampled, y_resampled
    if lr_model is not None and X_resampled is not None:
        y_pred = lr_model.predict(X_resampled)
        report = classification_report(y_resampled, y_pred)
        auc = roc_auc_score(y_resampled, lr_model.predict_proba(X_resampled)[:, 1])
        return f"Classification Report:\n{report}\nROC-AUC Score: {auc:.4f}"
    else:
        return "Logistic Regression model or resampled data not available."
```

```
Classification Report:
              precision    recall  f1-score   support

           0       0.77      0.75      0.76     23364
           1       0.76      0.77      0.76     23364

    accuracy                           0.76     46728
   macro avg       0.76      0.76      0.76     46728
weighted avg       0.76      0.76      0.76     46728

ROC-AUC Score: 0.8431
```

C. *Random Forest Model*

1) The third and final model in this project is a Random Forest, which is both flexible and powerful for handling complex data patterns. The initialize_random_forest function initializes and tunes the Random Forest model using GridSearchCV, optimizing hyperparameters based on ROC-AUC scores. This tuning process enhances the model's performance by selecting the best combination of parameters.

```
# Function 19: Initialize and Train Random Forest with Hyperparameter Tuning
def initialize_random_forest():
    global rf_model, X_resampled, y_resampled, rf_results
    if X_resampled is not None and y_resampled is not None:
        rf_model = RandomForestClassifier(random_state=42)
        param_grid = {
            'n_estimators': [50, 100, 200],
            'max_depth': [None, 10, 20, 30],
            'min_samples_split': [2, 5, 10]
        }
        grid_search = GridSearchCV(rf_model, param_grid, cv=3, scoring='roc_auc', n_jobs=-1)
        grid_search.fit(X_resampled, y_resampled)

        rf_model = grid_search.best_estimator_
        rf_results = {
            "Best Parameters": grid_search.best_params_,
            "Best ROC-AUC": grid_search.best_score_
        }
        return f"Random Forest model initialized and trained with tuning. Best ROC-AUC: {rf_results['Best ROC-AUC']:.4f}"
    else:
        return "SMOTE resampled data not available."
```

output

```
Random Forest model initialized and trained with tuning. Best ROC-AUC: 0.5598
```

2) The evaluation function evaluate_random_forest then generates a classification report and ROC-AUC score, which reveal the model's performance in distinguishing between classes. This assessment is critical for comparing the Random Forest's predictive power against the other models.

```
# Function 20: Random Forest Evaluation
def evaluate_random_forest():
    global rf_model, X_resampled, y_resampled
    if rf_model is not None and X_resampled is not None:
        y_pred = rf_model.predict(X_resampled)
        report = classification_report(y_resampled, y_pred)
        auc = roc_auc_score(y_resampled, rf_model.predict_proba(X_resampled)[:, 1])
        return f"Classification Report:\n{report}\nROC-AUC Score: {auc:.4f}"
    else:
        return "Random Forest model or resampled data not available."
```

output

```
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     23364
           1       1.00      1.00      1.00     23364

    accuracy                           1.00     46728
   macro avg       1.00      1.00      1.00     46728
weighted avg       1.00      1.00      1.00     46728

ROC-AUC Score: 1.0000
```

VI. GARDIO USER INTERFACE

```
# Gradio Interface Setup
with gr.Blocks() as credit_default_interface:
    with gr.Tab("Import Libraries"):
        gr.Interface(fn=import_libraries, inputs=[], outputs="text").render()
    with gr.Tab("Load Dataset"):
        gr.Interface(fn=load_dataset, inputs=[], outputs="dataframe").render()
    with gr.Tab("Display Statistics"):
        gr.Interface(fn=display_statistics, inputs=[], outputs="dataframe").render()
    with gr.Tab("Display Correlation"):
        gr.Interface(fn=display_correlation, inputs=[], outputs=gr.Image(type="pil")).render()
    with gr.Tab("Outlier Detection"):
        gr.Interface(fn=outlier_detection, inputs=[], outputs="text").render()
    with gr.Tab("Encoding Categorical Variables"):
        gr.Interface(fn=encoding_categorical, inputs=[], outputs="dataframe").render()
    with gr.Tab("Feature Scaling"):
        gr.Interface(fn=feature_scaling, inputs=[], outputs="dataframe").render()
    with gr.Tab("Apply SMOTE"):
        gr.Interface(fn=apply_smote, inputs=[], outputs="text").render()
    with gr.Tab("Calculate VIF"):
        gr.Interface(fn=calculate_vif, inputs=[], outputs="dataframe").render()
    with gr.Tab("Recalculate VIF"):
        gr.Interface(fn=recalculate_vif, inputs=[], outputs="dataframe").render()
    with gr.Tab("Visualize Distribution"):
        gr.Interface(fn=visualize_distribution, inputs=[], outputs=gr.Image(type="pil")).render()
    with gr.Tab("Initialize and Train Decision Tree"):
        gr.Interface(fn=initialize_decision_tree, inputs=[], outputs="text").render()
    with gr.Tab("Decision Tree Cross-Validation"):
        gr.Interface(fn=cross_validation_decision_tree, inputs=[], outputs="text").render()
    with gr.Tab("Visualize Decision Tree"):
        gr.Interface(fn=visualize_decision_tree, inputs=[], outputs=gr.Image(type="pil")).render()
    with gr.Tab("Decision Tree Classification Report"):
        gr.Interface(fn=classification_report_dt, inputs=[], outputs="text").render()
    with gr.Tab("Initialize and Train Logistic Regression"):
        gr.Interface(fn=initialize_logistic_regression, inputs=[], outputs="text").render()
    with gr.Tab("Evaluate Logistic Regression"):
        gr.Interface(fn=evaluate_logistic_regression, inputs=[], outputs="text").render()
    with gr.Tab("Initialize and Train Random Forest"):
        gr.Interface(fn=initialize_random_forest, inputs=[], outputs="text").render()
    with gr.Tab("Evaluate Random Forest"):
        gr.Interface(fn=evaluate_random_forest, inputs=[], outputs="text").render()

# Launch interface
credit_default_interface.launch(share=True)
```

VII. CONCLUSION

This project successfully built a **Credit Card Default Prediction** model using decision trees, providing valuable insights into the factors that contribute to credit default risk. The preprocessing steps, including outlier handling, one-hot encoding, feature scaling, and SMOTE, ensured that the model was well-prepared to handle the data complexities. While the model performed moderately well, future improvements could include hyperparameter tuning and trying other models like **Random Forest** or **Gradient Boosting** for potentially better accuracy.

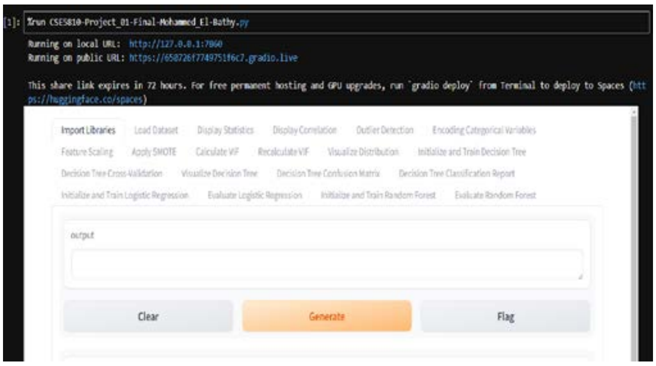
By analyzing the **confusion matrix** and the **classification report**, we can identify areas where the model's performance could be improved, particularly in reducing false positives and false negatives.

This solution is highly interpretable, making it a useful tool for financial institutions to assess and mitigate credit risk.

APPENDIX A

The Jupyter Notebook, the Python file and this report with the data file are uploaded to **GITHUB** and accessed using the URL **https://github.com/MohammedEl-Bathy/CreditCardDefaultPrediction**

STEPS EXECUTION USING GRADIO



APPENDIX B

The execution steps of the application using Jupiter Lap are illustrated after the references

REFERENCES

[1] Breiman, L. (2001). Random forests. Machine learning, 45(1), 5-32.

[2] Chawla, N. V., et al. (2002). SMOTE: Synthetic Minority Over-sampling Technique. Journal of Artificial Intelligence Research..

[3] Handling Imbalanced Datasets. Towards Data Science. Retrieved from https://github.com/MohammedEl-Bathy/CreditCardDefaultPrediction/tree/main.