INTRODUCTION

For banks and other lending organizations, controlling credit risk is essential in the modern financial sector. Credit risk is the chance that a borrower won't pay back a loan, which might lead to a loss of money. Manually evaluating risk becomes laborious, unreliable, and prone to mistakes when more people and companies request for financing. In order to assess loan applicants more precisely and effectively, there is a great demand for automated, data-driven solutions.

The goal of this project is to create a machine learning model that, using past data, can determine if a loan application is a good or terrible credit risk. To create a prediction model, we investigate several statistical and machine learning methods using the German Credit Dataset. Data exploration, preprocessing, model development with Random Forest, XGBoost, and Logistic Regression, and assessing these models using important performance indicators are all steps in the workflow. Delivering a dependable model that helps financial institutions make better informed, quicker, and consistent loan choices is the ultimate objective.

Dataset Description

The dataset used in this project is the **Statlog (German Credit Data)** dataset. It contains **1,000 records** of loan applicants with **20 features** and a **binary target variable**. Each record represents a person who either was granted a loan or not, with additional information such as employment status, savings account, age, loan purpose, and more.

Feature Type	Count	Description
Categorical	13	Examples: checking_account, purpose, housing, job
Numerical	7	Examples: duration, credit_amount, age
Target Variable	1	target: 0 = good credit risk, 1 = bad credit risk

The **target variable** is originally labeled as:

- **1 = Good Credit Risk**
- **2 = Bad Credit Risk**

We remapped it as:

- **0 = Good Credit Risk**
- **1 = Bad Credit Risk**
  (to match binary classification standards)

Data Preprocessing & Exploratory Data Analysis (EDA)

1. Target Variable Mapping

- The original target values (1 for good, 2 for bad) were **mapped to 0 (good)** and **1 (bad)** to suit machine learning binary classification.

2. Feature Separation

- **Categorical features** were identified using select_dtypes(include='object').

- **Numerical features** were identified using select_dtypes(include='number'), excluding the target.

3. One-Hot Encoding

- All categorical features were encoded using **pd.get_dummies()** with drop_first=True to avoid multicollinearity.

4. Feature Scaling

- Only **numerical features** were standardized using StandardScaler() to ensure fair treatment across algorithms.

5. Train-Test Split

- Data was split into training and testing sets:
  - **80% for training**
  - **20% for testing**
  - **Stratified split** was used to preserve the class distribution.

Exploratory Data Analysis Highlights

- **Target Distribution:** Imbalanced (more good credit cases than bad).
- **Boxplots** showed trends like:
  - Higher credit amounts and age tend to be associated with bad credit.
- **Categorical Distribution:** Some categories like "no checking account" were more frequent among bad credit cases.
- **Stacked Bar Charts:** Helped visualize relationship between categorical features and target.

Why These Models Were Used & What They Do

In this project, we used **Logistic Regression**, **Random Forest**, and **XGBoost** for credit risk prediction. Each of these models brings different strengths, and together they provide a balanced comparison between simplicity, interpretability, and performance.

---

1. Logistic Regression

What It Does:

Logistic Regression is a statistical model used for **binary classification** problems — in this case, predicting whether a customer is a **good or bad credit risk**. It calculates the probability of the target being 1 (bad) using a logistic (sigmoid) function.

Why Use It:

- Simple and interpretable.
- Fast to train and test.
- Provides **probability estimates**.
- Helps understand the relationship between input features and the target.

---

2. Random Forest

What It Does:

Random Forest is an **ensemble model** that builds multiple decision trees and combines their predictions to improve accuracy and reduce overfitting.

Why Use It:

- Handles both categorical and numerical features well.
- Resistant to overfitting compared to single decision trees.
- Can model complex, nonlinear relationships.
- Provides **feature importance** insights.

---

3. XGBoost (Extreme Gradient Boosting)

What It Does:

XGBoost is a **boosting algorithm** that builds trees sequentially, where each new tree tries to correct the mistakes of the previous one. It's optimized for speed and performance.

Why Use It:

- One of the most **powerful and accurate** models for structured/tabular data.
- Excellent performance in **imbalanced classification problems** like credit risk.
- Offers great control through hyperparameter tuning.
- Built-in handling of missing values and regularization.

Why Not Use Other Models?

- **KNN or SVM**: Can struggle with large feature sets and imbalanced data.
- **Neural Networks**: Often overkill for tabular data and harder to interpret.
- **Naive Bayes**: Assumes feature independence, which is not realistic in this case.

Why We Remove the target Column from numeric_cols During EDA

In exploratory data analysis (EDA), we often use histograms to visualize the distribution of numerical features in our dataset. These features are the inputs that help us train a machine learning model. However, even though the target column is technically a numeric column (e.g., values like 0 and 1), we do not consider it a numeric feature in this context. Here's why:

1. target is a Label, Not a Feature

- The target column represents the outcome or label — in this case, whether a person has good (0) or bad (1) credit.
- We are not trying to analyze it as a variable to understand its distribution like age, credit amount, or duration.
- Instead, we use the target to evaluate relationships (e.g., how credit amount varies across different target values).

---

2. target is Categorical in Meaning

- Although it is stored as numbers (0 and 1), its purpose is classification — not continuous measurement.
- A histogram of target would just show how many 0s and 1s, which is better visualized using a countplot or bar chart, not a histogram meant for continuous numeric data.

---

1. Why didn't you use deep learning?

Deep learning models are powerful but often unnecessary for tabular data like the credit dataset. Our data is relatively small (1000 records), and tree-based models like XGBoost perform very well in this context with less complexity.

Deep learning requires large datasets, longer training time, and is harder to interpret — which is critical in banking. In regulated industries like finance, **model explainability** is essential for trust and compliance.

✅ Tree-based models like XGBoost are often preferred for structured/tabular data.

---

2. Can the model be improved?

Yes, definitely. There are several ways to improve the model:

- **Feature Engineering**: Create new features such as income-to-loan ratio, credit usage history, or flag high-risk customers.
- **Advanced Tuning**: Use Bayesian Optimization or RandomizedSearchCV to find better hyperparameters.
- **Ensemble Models**: Combine multiple models (stacking, blending) to boost accuracy.
- **Use More Data**: If we can collect more recent or detailed data (e.g., payment history, income proof), the model could learn better patterns.

Also, techniques like **SMOTE** can help balance the dataset to improve recall on the minority class.

---

3. How often should the model be retrained?

Answer:

Ideally, we should monitor the model continuously and **retrain it periodically**, such as:

- Every **3–6 months**
- Or when we notice **data drift** (customer behavior changes)
- Or if model performance drops below a set threshold (e.g., recall or AUC)

📌 Retraining frequency depends on how fast the financial environment or customer profile evolves.

---

4. How would you monitor model performance in production?

Once deployed, the model should be monitored using:

- **Performance metrics**: Track accuracy, precision, recall, AUC on new data
- **Drift detection**: Monitor changes in input data distribution (data drift) and target changes (concept drift)
- **Feedback loop**: Use actual repayment outcomes to retrain and fine-tune the model
- **Dashboards**: Create dashboards (using tools like Power BI, Grafana) for business stakeholders to monitor credit risk insights

---

Model Evaluation

To assess the performance of the trained machine learning models, a comprehensive evaluation function was developed. This function provides both quantitative metrics and visual insights to understand how well the model predicts credit risk.

The evaluation includes:

- **Confusion Matrix:** This matrix summarizes the number of correct and incorrect predictions, distinguishing between true positives, true negatives, false positives, and false negatives. It helps to identify the types of errors the model makes, which is crucial for credit risk classification.
- **Classification Report:** This report provides detailed metrics including:
    - **Precision:** The accuracy of positive predictions (i.e., how many predicted bad credits are actually bad).
    - **Recall (Sensitivity):** The ability of the model to detect all actual bad credits.
    - **F1-score:** The harmonic mean of precision and recall, balancing both metrics for overall effectiveness.
- **Accuracy Score:** The proportion of total correct predictions among all predictions.
- **ROC-AUC Score:** The Receiver Operating Characteristic - Area Under the Curve score measures the model's ability to distinguish between good and bad credit classes across different threshold levels. A higher AUC indicates better discrimination.
- **ROC Curve Plot:** The function plots the ROC curve to visualize the trade-off between the true positive rate and false positive rate, helping to understand model performance at various classification thresholds.

CODE EXPLANATION

Importing Libraries

import pandas as pd

- pandas is used for data manipulation and analysis. Commonly used to handle tabular data using DataFrames.

import numpy as np

- numpy is a numerical library used for handling arrays and performing mathematical operations.

import seaborn as sns

- seaborn is a statistical data visualization library built on top of Matplotlib. It's used for creating beautiful plots like heatmaps, boxplots, etc.

import matplotlib.pyplot as plt

- matplotlib.pyplot is a plotting library used to create static, animated, and interactive visualizations in Python.

Machine Learning Libraries

from sklearn.model_selection import train_test_split, GridSearchCV

- train_test_split: Splits the dataset into training and testing sets.
- GridSearchCV: Performs hyperparameter tuning using cross-validation to find the best model parameters.

from sklearn.preprocessing import StandardScaler

- StandardScaler: Standardizes the features by removing the mean and scaling to unit variance (important for many ML models).

from sklearn.linear_model import LogisticRegression

- Imports the Logistic Regression model — a basic and widely-used classification algorithm.

from sklearn.ensemble import RandomForestClassifier

- Imports the Random Forest Classifier, an ensemble model that uses multiple decision trees for better accuracy.

from xgboost import XGBClassifier

- Imports XGBoost, a powerful and efficient gradient boosting algorithm known for top performance in classification tasks.

Evaluation Metrics

```
from sklearn.metrics import (
    classification_report, confusion_matrix, accuracy_score,
    roc_auc_score, roc_curve
)
```

- classification_report: Shows precision, recall, f1-score, and support.
- confusion_matrix: Table showing actual vs predicted classifications.
- accuracy_score: Calculates the accuracy of the model.
- roc_auc_score: Area Under the Receiver Operating Characteristic Curve — measures the model's ability to distinguish between classes.
- roc_curve: Plots the true positive rate vs false positive rate.

Model Saving

```
import joblib
```

- joblib is used to save and load Python objects (like trained models). It's faster than pickle for large numpy arrays.

In Summary

This script is setting up a complete machine learning workflow:

- Data handling (Pandas, NumPy)
- Visualization (Seaborn, Matplotlib)
- Data preprocessing (StandardScaler)
- Modeling (Logistic Regression, Random Forest, XGBoost)
- Evaluation (Accuracy, Confusion Matrix, ROC, etc.)
- Model tuning (GridSearchCV)
- Model persistence (Joblib)

Define Column Names

```
columns = [
    'checking_account', 'duration', 'credit_history', 'purpose', 'credit_amount',
    'savings_account', 'employment_since', 'installment_rate', 'personal_status_sex',
    'guarantors', 'residence_since', 'property', 'age', 'other_installment_plans',
    'housing', 'existing_credits', 'job', 'num_liable_people', 'telephone', 'foreign_worker', 'target'
]
```

- You're defining the column names based on the **dataset documentation**.
- The original file does **not include headers**, so you're manually assigning them to understand the data better.

Load Dataset

df = pd.read_csv(r"C:\Users\ashid\Downloads\statlog+german+credit+data\german.data", sep=' ', header=None, names=columns)

- Loads the dataset using pandas.read_csv.
- sep=' ' tells Pandas that the data is space-separated.
- header=None means the file has **no header row** — so Pandas won't try to use the first row as column names.
- names=columns assigns the column names you defined earlier.

Map Target Variable

df['target'] = df['target'].map({1: 0, 2: 1})

- The original dataset has 1 = good credit and 2 = bad credit.
- This line **inverts and converts it to binary**:
  - Good credit (1) → 0
  - Bad credit (2) → 1
- This format (0 = no default, 1 = default) is better for binary classification tasks.

Check Dataset Shape and Preview

print(f"Dataset shape: {df.shape}")
print(df.head())

- df.shape shows the number of rows and columns.
- df.head() prints the first 5 rows of the dataset to preview its structure and values.

Show Target Distribution

print("\nTarget distribution:")
print(df['target'].value_counts())

- Displays how many instances belong to each class (0 and 1).
- Important to check **class imbalance**, which can affect model performance.

Summary

This code:

1. **Prepares** the dataset with readable column names.
2. **Loads** it properly from the raw .data file.
3. **Transforms** the target for machine learning (binary classification).
4. **Gives an overview** of the dataset and target label distribution.

Common Plot Types & When to Use Them

Goal	Data Type	Recommended Plot	Example
Distribution of a single numeric variable	Numerical	Histogram	Age distribution
Distribution of a single categorical variable	Categorical	Bar Chart / Count Plot	Loan Purpose counts
Compare numeric variable between groups	Categorical + Numerical	Box Plot / Violin Plot	Income vs Gender
Explore relationship between two numeric variables	Numerical + Numerical	Scatter Plot	Credit Amount vs Age
Trend over time	Time series	Line Plot	Loan volume by month
Compare proportion of categories	Categorical	Pie Chart / Stacked Bar *(if few categories)*	Marital Status Breakdown
Relationship between categorical feature & target	Categorical + Target	Stacked Bar Plot (Crosstab)	Job vs Credit Risk
Show correlation between features	Multiple numerical	Heatmap	Correlation matrix
Model evaluation	Classification Models	Confusion Matrix, ROC Curve, Precision-Recall Curve	Performance reports

1. Separates the target variable (y) from the features (X)

```
X = df.drop('target', axis=1)
y = df['target']
```

- X: Contains all the input features (independent variables)
- y: Contains the target column (target) which is what you're predicting (0 = good credit, 1 = bad credit)

2. Identifies the column types

```
categorical_cols = X.select_dtypes(include='object').columns.tolist()
numeric_cols = X.select_dtypes(include='number').columns.tolist()
```

- categorical_cols: List of columns that contain categorical (non-numeric) values like job, housing, etc.
- numeric_cols: List of columns that are numeric like age, credit_amount, duration, etc.

Why this is important:

Before building a machine learning model, you need to:

- Treat **categorical and numerical data differently**
- Apply **encoding** for categorical features (e.g., OneHotEncoder or LabelEncoder)

- Normalize or scale numerical features if needed

Encoding Categorical Features

To convert non-numeric categorical variables into a machine-readable format, we applied **One-Hot Encoding** using the pandas.get_dummies() function.

```
X_encoded = pd.get_dummies(X, columns=categorical_cols, drop_first=True)
print(f"Encoded feature set shape: {X_encoded.shape}")
```

- drop_first=True: Drops the first category of each variable to prevent **multicollinearity** (also known as the dummy variable trap).
- This transformation increases the number of columns, depending on how many unique values each categorical column had.

Example:
A column like checking_account with 4 categories becomes 3 new binary columns:

- checking_account_<category1>
- checking_account_<category2>
- checking_account_<category3>

Outcome

- After encoding, the shape of the feature set changed to reflect the newly created dummy variables.
- All input features are now **numerical**, which is essential for training machine learning models.

Feature Scaling

Machine learning models are often sensitive to the scale of features—especially those that rely on distance-based calculations (like logistic regression or KNN). To ensure all numerical features contribute equally, we applied **Standardization** using StandardScaler from scikit-learn.

```
from sklearn.preprocessing import StandardScaler

# Initialize the scaler
scaler = StandardScaler()

# Apply scaling to only the numeric columns
X_encoded[numeric_cols] = scaler.fit_transform(X_encoded[numeric_cols])
```

Explanation:

- StandardScaler transforms the data so that each numeric feature has:
    - A **mean of 0**
    - A **standard deviation of 1**

- This avoids features with larger ranges (e.g., credit amount vs. age) from dominating the model training process.

Outcome:

- All numerical features are now standardized and on the same scale.
- This improves **model convergence speed** and ensures better **model performance** and **interpretability**.

Train-Test Split

To evaluate model performance fairly, we split our dataset into a **training set** and a **test set** using train_test_split from scikit-learn.

```
from sklearn.model_selection import train_test_split

# Final features and target
X = X_encoded
y = df['target']

# Split into training and testing sets (80/20) with stratified sampling
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, stratify=y, random_state=42
)

print(f"Training set shape: {X_train.shape}")
print(f"Test set shape: {X_test.shape}")
```

Explanation:

- **80%** of the data was used for training, and **20%** was reserved for testing.
- **Stratified sampling** was applied to maintain the same class distribution in both sets, which is important due to slight class imbalance.
- A **random seed (random_state=42)** ensures reproducibility of results.

Outcome:

- The training and testing sets are properly separated.
- Class distributions are preserved, which supports **fair model evaluation**.

Model Evaluation Function

To evaluate each machine learning model consistently, we defined a function evaluate_model, that calculates and displays:

- Confusion Matrix
- Classification Report (precision, recall, F1-score)
- Accuracy score

- ROC-AUC score
- ROC Curve plot

Explanation:

- confusion_matrix helps identify types of errors (false positives, false negatives).
- classification_report provides precision, recall, and F1-score for each class.
- accuracy_score gives the overall correct prediction rate.
- roc_auc_score measures the model's ability to discriminate between classes.
- The **ROC curve** visually presents the trade-off between true positive rate and false positive rate.

Explanation of the evaluate_model function

```
def evaluate_model(model, X_test, y_test):
  y_pred = model.predict(X_test)
  y_prob = model.predict_proba(X_test)[:, 1]
```

- **Inputs:** The function takes three inputs:
  - model: The trained machine learning model.
  - X_test: The test dataset features.
  - y_test: The true labels for the test dataset.
- **Predictions:**
  - y_pred = model.predict(X_test): Predicts the class labels (e.g., 0 or 1) for the test set.
  - y_prob = model.predict_proba(X_test)[:, 1]: Predicts the probabilities that each sample belongs to the positive class (class 1). We select the second column ([:, 1]) because it's the probability of the positive class, which is needed for ROC and AUC calculations.

```
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred))
print("Accuracy:", accuracy_score(y_test, y_pred))
print("ROC-AUC:", roc_auc_score(y_test, y_prob))
```

- **Evaluation metrics printed:**
  - **Confusion Matrix:** Shows the count of true positives, true negatives, false positives, and false negatives, helping you understand types of prediction errors.
  - **Classification Report:** Includes precision, recall, and F1-score for each class (0 and 1), which provide deeper insight into model performance beyond accuracy.
  - **Accuracy:** The proportion of total correct predictions over all predictions.
  - **ROC-AUC Score:** Measures how well the model can distinguish between the positive and negative classes, with 1 being perfect and 0.5 being random guessing.

```
fpr, tpr, _ = roc_curve(y_test, y_prob)
plt.figure(figsize=(6, 4))
plt.plot(fpr, tpr, label=f"{model.__class__.__name__} (AUC = {roc_auc_score(y_test, y_prob):.2f})")
plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
```

```
plt.title("ROC Curve")
plt.legend()
plt.show()
```

- **ROC Curve Plot:**
  - roc_curve(y_test, y_prob) computes the False Positive Rate (FPR) and True Positive Rate (TPR) at different classification thresholds.
  - plt.plot(fpr, tpr, ...) plots the ROC curve, showing the trade-off between sensitivity (TPR) and fall-out (FPR).
  - plt.plot([0,1], [0,1], 'k--') adds a diagonal line that represents random guessing (baseline).
  - Labels, title, and legend improve readability and indicate the model name and its AUC score on the plot.

Model Training and Evaluation: Logistic Regression

The Logistic Regression model was trained using the scaled and encoded training data (X_train, y_train). To address the class imbalance in the credit risk dataset, the class_weight parameter was set to 'balanced'. This helps the model pay appropriate attention to both classes (good and bad credit) during training.

Key parameters used:

- max_iter=1000: Increased the maximum number of iterations to ensure convergence.
- class_weight='balanced': Automatically adjusts weights inversely proportional to class frequencies.
- random_state=42: Ensures reproducibility of results.

After training, the model was evaluated on the unseen test set (X_test, y_test) using the previously defined evaluate_model function. This function provided detailed metrics including confusion matrix, precision, recall, F1-score, accuracy, ROC-AUC, and the ROC curve plot.

This approach allowed us to objectively measure how well Logistic Regression predicts credit risk on new data, balancing both sensitivity to the minority class and overall accuracy.

Model Training and Evaluation: Random Forest Classifier

The Random Forest Classifier was trained using the training data (X_train, y_train) with the following key settings:

- n_estimators=100: The model uses 100 decision trees to form the forest, which helps improve robustness and reduce overfitting.
- class_weight='balanced': This parameter addresses the class imbalance by adjusting weights inversely proportional to class frequencies.
- random_state=42: Ensures that the results are reproducible.

After training, the model was evaluated on the test set using the evaluate_model function, which reports detailed performance metrics such as the confusion matrix, precision, recall, F1-score, accuracy, ROC-AUC score, and displays the ROC curve.

Random Forest is a powerful ensemble method that combines multiple decision trees to improve prediction accuracy and control overfitting, making it well-suited for credit risk classification.

Model Training and Evaluation: XGBoost Classifier

The XGBoost classifier was trained on the training dataset with the following parameters:

- eval_metric='logloss': This evaluation metric optimizes the logarithmic loss during training, suitable for binary classification.
- scale_pos_weight=3: This parameter helps to handle the imbalanced dataset by assigning more weight to the minority class (bad credit).
- random_state=42: Ensures reproducibility of results.

After training, the model was evaluated on the test set using the evaluate_model function, which reports metrics including confusion matrix, precision, recall, F1-score, accuracy, ROC-AUC, and displays the ROC curve.

XGBoost is an efficient gradient boosting algorithm known for high performance and speed, especially in structured/tabular data like credit scoring.

Hyperparameter Tuning for XGBoost Classifier

To improve the performance of the XGBoost classifier, we performed hyperparameter tuning using Grid Search with 5-fold cross-validation. The goal was to find the best combination of parameters that maximize the F1 score, which balances precision and recall—important for imbalanced classification tasks such as credit risk prediction.

Parameters Tuned:

- n_estimators: Number of boosting rounds (tested 100 and 200)
- max_depth: Maximum depth of each tree (tested 3, 5, and 7)
- learning_rate: Step size shrinkage to prevent overfitting (tested 0.01, 0.1, 0.2)
- subsample: Fraction of samples used for fitting individual trees (tested 0.8 and 1.0)

Process:

- Initialized a base XGBoost classifier with class imbalance adjustment (scale_pos_weight=3) and fixed evaluation metric.
- Applied GridSearchCV to systematically try all combinations of the parameters above.
- Used 5-fold cross-validation to evaluate performance on different subsets of the training data.

- Selected the best model based on the highest mean F1 score.

Results:

- The best hyperparameters found are:
  n_estimators = ... (fill from output)
  max_depth = ...
  learning_rate = ...
  subsample = ...
- The best cross-validated F1 score achieved was: ... (fill from output)

Final Evaluation:

- The best tuned model was then evaluated on the test set.
- Evaluation metrics, including confusion matrix, classification report, accuracy, and ROC-AUC, were obtained.
- The ROC curve was plotted to visualize the tradeoff between sensitivity and specificity.

Feature Importance from XGBoost Model

Understanding which features have the most influence on the model's predictions is crucial for interpretability and trust in the results. We extracted the feature importances from the best-tuned XGBoost model to identify the top predictors of credit risk.

Steps:

- Retrieved the importance scores for each feature from the trained XGBoost model.
- Created a DataFrame mapping feature names to their importance scores.
- Sorted the features by their importance in descending order.
- Visualized the top 10 features using a horizontal bar chart for clear comparison.

Insights:

- The plot highlights the most impactful features driving the model's decisions.
- Features with higher importance values contribute more significantly to predicting good or bad credit.
- This analysis helps prioritize which variables are most critical for credit risk assessment and can guide further business or domain-specific investigations.

Saving the Final Model

To preserve the trained and optimized model for future use (e.g., deployment, evaluation, or integration into applications), we saved the best-performing XGBoost model using the joblib library.

Code Used:

```
import joblib
joblib.dump(best_xgb, 'final_credit_risk_xgb_model.pkl')
```

Purpose:

- This creates a .pkl file containing the trained model object.
- It allows reloading the model later without needing to retrain.
- Useful for deploying the model in production or running predictions in a web or batch application.

Next Steps :

To use the saved model later:

```
loaded_model = joblib.load('final_credit_risk_xgb_model.pkl')
predictions = loaded_model.predict(X_new)
```

Final Dataset Preparation for Export

After preprocessing, feature encoding, and model training, we prepared the final version of the dataset for export or further analysis. This included:

Steps Taken:

1. **Combine Features and Target**:
   Merged the feature matrix X and target vector y back into a single DataFrame.
2. df_encoded = X.copy()
3. df_encoded['target'] = y
4. **Add Human-Readable Labels**:
   Converted the binary target values into more interpretable labels:
   - 0 → Good (indicating good credit risk)
   - 1 → Bad (indicating bad credit risk)
5. df_encoded['target'] = df_encoded['target'].map({0: 'Good', 1: 'Bad'})

This format makes it easier for stakeholders (non-technical teams, business analysts) to understand the prediction outputs.

```
df_encoded.to_csv(r"C:\Users\ashid\Downloads\statlog+german+credit+data\final_credit_risk_data.csv", index=False)
```

Did the following:

- Saved your final DataFrame (df_encoded) — which includes all the encoded features and the human-readable "Good" / "Bad" target labels — as a .csv file.

- The file was saved to:
- C:\Users\ashid\Downloads\statlog+german+credit+data\
- index=False ensures that row indices are **not** included in the CSV.

Final Output

The CSV file final_credit_risk_data.csv now includes:

- All preprocessed and scaled features.
- The final target column with "Good" and "Bad" labels.
- It's ready for sharing with your team, analysis in Excel, or for uploading to your GitHub project.