**END TERM REPORT -Finsearch 2k25**

***Project: Credit Scoring Algorithms***

**Team :B9**

**Credit Score Analysis - Code explanation :**

**1. Importing Libraries and Loading the Dataset**

**Code:**

import pandas as pd

df = pd.read\_csv('../german\_credit.csv')

df.head()

**Explanation :**

* pandas is imported to handle data manipulation.
* The dataset german\_credit.csv is loaded into a DataFrame called df.
* df.head() shows the first five rows to get an initial view of the data.
* This step brings our data into Python so we can start exploring and preparing it for analysis.

**2. Exploring the Dataset**

**Code:**

df.info()

df.describe()

df.isnull().sum()

**Explanation :**

* df.info() gives details like column names, data types, and number of non-null values.
* df.describe() provides summary statistics (mean, std, min, max, etc.) for numeric columns.
* df.isnull().sum() checks for missing values in each column.
* Understanding the structure of the dataset helps us decide what preprocessing steps are needed.

**3. Target Variable Distribution**

**Code:**

import seaborn as sns

import matplotlib.pyplot as plt

sns.countplot(x='target', data=df)

plt.title('Target Variable Distribution')

plt.show()

**Explanation:**

* We use Seaborn and Matplotlib to create a count plot of the target column.
* The target column indicates whether a credit case is “good” or “bad”.
* Knowing how balanced the target variable is helps in model selection and performance evaluation. An imbalanced dataset might need balancing techniques.

**4. Inspect column names**

**Code:**

df.columns

**Explanation:**

This prints the raw column labels. For this dataset the raw file sometimes lacks proper headers or uses coded values (e.g., "A11"). Inspecting df.columns confirmed whether the CSV had been parsed into the correct columns or whether the delimiter/header needed adjustment.

**5. Re-load if parsing was wrong**

**Code:**

df = pd.read\_csv('../german\_credit.csv', delim\_whitespace=True, header=None)

df.head()

**Explanation:**

The notebook re-reads the CSV using delim\_whitespace=True and header=None because the original read resulted in a single column (values joined together). Using whitespace as the delimiter and specifying no header ensures columns are split correctly. After reloading we inspect the first rows again.

**6. Assign column names**

**Code:**

column\_names = [

'checking\_status', 'duration', 'credit\_history', 'purpose', 'credit\_amount',

'savings', 'employment', 'installment\_commitment', 'personal\_status\_sex',

'other\_debtors', 'residence\_since', 'property', 'age', 'other\_installment\_plans',

'housing', 'existing\_credits', 'job', 'num\_dependents', 'own\_telephone',

'foreign\_worker', 'target'

]

df.columns = column\_names

df.head()

**Explanation:**

After parsing the file correctly, we set a descriptive list of column names (there are 20 features plus the target). This makes later steps readable and ensures the model inputs are identifiable. The target column holds the label: good vs bad credit.

### **Next cells — EDA (exploratory data analysis)**

**Actions performed (summary):**

* Re-check df.info() and df.describe() with the proper headers.
* Plot the distribution of the target column (countplot).
* Compute correlations for numeric columns (heatmap).

**Explanation:** EDA is used to discover patterns and potential problems:

* Target balance: we check how many examples are “good” vs “bad” to understand class imbalance.
* Numeric correlation: we compute correlations only on numeric columns to avoid errors. Because many features are categorical, correlation is limited to numeric features (age, duration, credit\_amount, etc.).
* Visual inspection (histograms, bar plots) helped identify features likely to be predictive (e.g., credit amount, duration, account / checking status).

**Preprocessing: encode categorical features**

**Code:**

from sklearn.preprocessing import LabelEncoder

categorical\_cols = df.select\_dtypes(include=['object']).columns

le = LabelEncoder()

for col in categorical\_cols:

df[col] = le.fit\_transform(df[col])

**Explanation:**

Most classifiers require numeric inputs. We identify categorical columns (dtype object) and convert them to numeric labels using LabelEncoder. This is a simple approach that replaces categories with integer codes; it is quick and common for baseline models.

**Split features and target, and train/test split**

**Code:**

X = df.drop('target', axis=1)

y = df['target']

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

**Explanation:**

We separate inputs (X) from labels (y) and split the data into training (80%) and testing (20%) sets. The random\_state=42 ensures reproducibility. The training set is used to fit models; the test set measures performance on unseen data.

**Feature scaling**

**Code:**

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

**Explanation:**

StandardScaler standardizes numeric features to have zero mean and unit variance. Scaling is especially important for models like Logistic Regression (and often beneficial for others). We fit the scaler on the training set and transform both train and test sets to avoid leaking test information.

**Model building — Logistic Regression, Decision Tree, Random Forest**

**Code:**

from sklearn.linear\_model import LogisticRegression

from sklearn.tree import DecisionTreeClassifier

from sklearn.ensemble import RandomForestClassifier

lr = LogisticRegression(random\_state=42, max\_iter=1000)

lr.fit(X\_train, y\_train

dt = DecisionTreeClassifier(random\_state=42)

dt.fit(X\_train, y\_train)

rf = RandomForestClassifier(random\_state=42)

rf.fit(X\_train, y\_train)

**Explanation:**

We instantiate and train three models:

* **Logistic Regression**: a linear, interpretable classifier appropriate for baseline credit-scoring tasks.
* **Decision Tree**: a non-linear model that captures interactions and splits data by conditions.
* **Random Forest**: an ensemble of trees that often improves performance and reduces overfitting.

All models are trained on the scaled/preprocessed training data.

**Predictions and evaluation**

**Code:**

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix

y\_pred\_lr = lr.predict(X\_test)

y\_pred\_dt = dt.predict(X\_test)

y\_pred\_rf = rf.predict(X\_test)

accuracy\_score(y\_test, y\_pred\_rf)

print(classification\_report(y\_test, y\_pred\_rf))

**Explanation:**

We produce predictions on the test set with each model and compute evaluation metrics:

* Accuracy: fraction of correct predictions (simple, easy-to-interpret).
* Classification report: provides precision, recall, and F1-score per class (good and bad credit).
* Confusion matrix: reveals actual vs predicted counts and highlights false positives and false negatives.

These metrics help us determine which model generalizes best.

**Utility: evaluation function + confusion matrix plotting**

**Code:**

def evaluate\_model(y\_test, y\_pred, model\_name):

print(f"--- {model\_name} ---")

print("Accuracy:", accuracy\_score(y\_test, y\_pred))

print("Classification Report:\n", classification\_report(y\_test, y\_pred))

cm = confusion\_matrix(y\_test, y\_pred)

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')

plt.show()

**Explanation:**

A helper function consolidates printing metrics and plotting confusion matrices, so we can run consistent evaluation for each model quickly.

**Feature importance (Random Forest)**

**Code:**

importances = rf.feature\_importances\_

feat\_df = pd.DataFrame({'Feature': X.columns, 'Importance': importances})

feat\_df = feat\_df.sort\_values(by='Importance', ascending=False)

# plot

sns.barplot(x='Importance', y='Feature', data=feat\_df)

plt.title("Feature Importance - Random Forest")

plt.show()

**Explanation:**

Random Forest exposes a numeric importance score per feature. We extract these scores, create a sorted DataFrame and plot a bar chart. This highlights which variables most strongly influence the model’s predictions (examples: credit amount, duration, account status). This interpretability step is useful for explaining the model to stakeholders.

Source of Data: Kaggle

<https://docs.google.com/spreadsheets/d/1OZX62GcEoxXLNRTgPaxnfB9rhjFogtkhfJWKZxY16iU/edit?usp=sharing>

Column wise data set-

* **Existing amount in account-**Status of existing checking account i.e. range of money in existing checking account (if present)

| Value | Description |
| --- | --- |
| A11 | .. < 0 DM |
| A12 | 0 <= ... < 200 DM |
| A13 | ... >= 200 DM / salary assignments for at least 1 year |
| A14 | no checking account |

DM stands for Deutsche Marks.

* **Month Duration (Numerical)**Duration in month of the asked loan
* **Purpose (Qualitative)**Purpose of the requested loan

| Value | Description |
| --- | --- |
| A40 | car (new) |
| A41 | car (used) |
| A42 | furniture/equipment |
| A43 | radio/television |
| A44 | domestic appliances |
| A45 | repairs |
| A46 | education |
| A47 | vacation |
| A48 | retraining |
| A49 | business |
| A410 | others |

* **Credit Amount (Numerical)**Amount of money requested for the credit
* **Personal Status and Sex (Qualitative)**

| Value | Description |
| --- | --- |
| A91 | male : divorced/separated |
| A92 | female : divorced/separated/married |
| A93 | male : single |
| A94 | male : married/widowed |
| A95 | female : single |

* **Property (Qualitative)**Type of property owned by the applicant

| Value | Description |
| --- | --- |
| A121 | real estate |
| A122 | if not A121 : building society savings agreement/ life insurance |
| A123 | if not A121/A122 : car or other, not in attribute 6 |
| A124 | unknown / no property |

* **Age (Numerical)**Age in years
* **Housing (Qualitative)**

| Value | Description |
| --- | --- |
| A151 | rent |
| A152 | own |
| A153 | for free |

* **Existing credit number (Numerical)**Number of existing credit at this bank
* **Result (Response)**

| Value | Description |
| --- | --- |
| 1 | good (applicant eligible for credit) |
| 2 | bad (applicant not eligible for credit) |