**CREDIT CARD FRAUD DETECTION MODEL**

Here's a complete Python code structure following your specified workflow. Each section corresponds to the steps in your project workflow, including EDA, preprocessing, modeling, and deployment.

**A. 01\_EDA.ipynb**

**Exploratory Data Analysis**

python

CopyEdit

# Import Libraries

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

# Load Dataset

data\_card = pd.read\_csv('credit\_card\_data.csv') # Replace with the correct file path

# Quick Overview

print(data\_card.info())

print(data\_card.describe())

print(data\_card['default.payment.next.month'].value\_counts())

# Visualizations

# Histograms

plt.figure(figsize=(10, 5))

sns.histplot(data\_card['LIMIT\_BAL'], bins=30, kde=True, color='blue')

plt.title('Distribution of Credit Limit')

plt.xlabel('Credit Limit')

plt.ylabel('Frequency')

plt.show()

plt.figure(figsize=(10, 5))

sns.histplot(data\_card['AGE'], bins=30, kde=True, color='green')

plt.title('Distribution of Age')

plt.xlabel('Age')

plt.ylabel('Frequency')

plt.show()

# Scatter Plots

plt.figure(figsize=(8, 6))

sns.scatterplot(data=data\_card, x='LIMIT\_BAL', y='PAY\_AMT1', hue='default.payment.next.month', palette='viridis')

plt.title('Credit Limit vs Payment Amount (PAY\_AMT1)')

plt.show()

# Box Plots

plt.figure(figsize=(10, 6))

sns.boxplot(data=data\_card, x='default.payment.next.month', y='BILL\_AMT1', palette='coolwarm')

plt.title('Bill Amount (BILL\_AMT1) by Default Status')

plt.show()

# Correlation Heatmap

plt.figure(figsize=(12, 10))

correlation\_matrix = data\_card.corr()

sns.heatmap(correlation\_matrix, annot=True, fmt=".2f", cmap="coolwarm", cbar=True, square=True)

plt.title("Correlation Heatmap")

plt.xticks(rotation=45)

plt.show()

**B. 02\_Preprocessing.ipynb**

**Data Preprocessing**

python

CopyEdit

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

from sklearn.preprocessing import StandardScaler, OneHotEncoder

from sklearn.compose import ColumnTransformer

from sklearn.impute import SimpleImputer

from sklearn.pipeline import Pipeline

# Handle Missing Values

data\_card.fillna(0, inplace=True) # Replace NaNs with 0 (modify as needed)

# Separate Features and Target

X = data\_card.drop('default.payment.next.month', axis=1)

y = data\_card['default.payment.next.month']

# Encode Categorical Variables

categorical\_features = ['SEX', 'EDUCATION', 'MARRIAGE']

numerical\_features = [col for col in X.columns if col not in categorical\_features]

preprocessor = ColumnTransformer(

transformers=[

('num', StandardScaler(), numerical\_features),

('cat', OneHotEncoder(), categorical\_features)

]

)

# Split Data

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

# Feature Preprocessing

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

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

**C. 03\_Modeling.ipynb**

**Model Building and Evaluation**

python

CopyEdit

from sklearn.linear\_model import LogisticRegression

from sklearn.ensemble import RandomForestClassifier

from xgboost import XGBClassifier

from sklearn.metrics import classification\_report, confusion\_matrix, roc\_auc\_score, roc\_curve

import joblib

# Baseline Model: Logistic Regression

logistic\_model = LogisticRegression()

logistic\_model.fit(X\_train, y\_train)

# Advanced Model: Random Forest

rf\_model = RandomForestClassifier(random\_state=42)

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

# Advanced Model: XGBoost

xgb\_model = XGBClassifier(use\_label\_encoder=False, eval\_metric='logloss')

xgb\_model.fit(X\_train, y\_train)

# Evaluate Models

for model, name in zip([logistic\_model, rf\_model, xgb\_model], ['Logistic Regression', 'Random Forest', 'XGBoost']):

y\_pred = model.predict(X\_test)

print(f"{name} Classification Report:")

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

print(confusion\_matrix(y\_test, y\_pred))

print(f"AUC-ROC Score: {roc\_auc\_score(y\_test, model.predict\_proba(X\_test)[:, 1]):.4f}\n")

**Updated Code: Hyperparameter Tuning and Cross-Validation**

**Import Necessary Libraries**

Add these imports to your code if they are not already included:

python

CopyEdit

from sklearn.model\_selection import GridSearchCV, cross\_val\_score

from sklearn.metrics import classification\_report, confusion\_matrix, roc\_auc\_score, roc\_curve

**Hyperparameter Tuning**

Integrate hyperparameter tuning for each model using GridSearchCV:

python

CopyEdit

# Logistic Regression Hyperparameter Tuning

logistic\_params = {'C': [0.01, 0.1, 1, 10], 'penalty': ['l2']}

logistic\_grid = GridSearchCV(LogisticRegression(max\_iter=1000), logistic\_params, cv=5, scoring='roc\_auc', n\_jobs=-1)

logistic\_grid.fit(X\_train, y\_train)

# Random Forest Hyperparameter Tuning

rf\_params = {

'n\_estimators': [50, 100, 200],

'max\_depth': [None, 10, 20],

'min\_samples\_split': [2, 5, 10]

}

rf\_grid = GridSearchCV(RandomForestClassifier(random\_state=42), rf\_params, cv=5, scoring='roc\_auc', n\_jobs=-1)

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

# XGBoost Hyperparameter Tuning

xgb\_params = {

'n\_estimators': [50, 100, 200],

'learning\_rate': [0.01, 0.1, 0.2],

'max\_depth': [3, 5, 7]

}

xgb\_grid = GridSearchCV(XGBClassifier(use\_label\_encoder=False, eval\_metric='logloss'), xgb\_params, cv=5, scoring='roc\_auc', n\_jobs=-1)

xgb\_grid.fit(X\_train, y\_train)

**Cross-Validation**

Add cross-validation to evaluate model performance:

# Evaluate with Cross-Validation

def evaluate\_with\_cross\_validation(model, X\_train, y\_train):

scores = cross\_val\_score(model, X\_train, y\_train, cv=5, scoring='roc\_auc')

print(f"Cross-Validation AUC-ROC Scores: {scores}")

print(f"Mean AUC-ROC Score: {scores.mean():.4f}\n")

# Cross-validate each tuned model

print("Logistic Regression Cross-Validation:")

evaluate\_with\_cross\_validation(logistic\_grid.best\_estimator\_, X\_train, y\_train)

print("Random Forest Cross-Validation:")

evaluate\_with\_cross\_validation(rf\_grid.best\_estimator\_, X\_train, y\_train)

print("XGBoost Cross-Validation:")

evaluate\_with\_cross\_validation(xgb\_grid.best\_estimator\_, X\_train, y\_train)

**Final Evaluation**

Use the best-tuned models to predict and evaluate performance on the test set:

python

CopyEdit

# Evaluate Best Models on Test Set

for model, name in zip(

[logistic\_grid.best\_estimator\_, rf\_grid.best\_estimator\_, xgb\_grid.best\_estimator\_],

['Logistic Regression (Tuned)', 'Random Forest (Tuned)', 'XGBoost (Tuned)']

):

y\_pred = model.predict(X\_test)

y\_proba = model.predict\_proba(X\_test)[:, 1]

print(f"{name} Classification Report:")

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

print("Confusion Matrix:")

print(confusion\_matrix(y\_test, y\_pred))

print(f"AUC-ROC Score: {roc\_auc\_score(y\_test, y\_proba):.4f}\n")

**Recommendations:**

1. **Feature Engineering:** Investigate the feature set further to identify additional predictors of default. For example, interaction terms or non-linear transformations might capture hidden patterns.
2. **Data Balancing:** Use techniques like oversampling (SMOTE) or undersampling to address class imbalance, which could improve the recall for default cases.
3. **Model Optimization:**
   * Perform hyperparameter tuning (e.g., grid search or Bayesian optimization) for Random Forest and XGBoost to improve their performance further.
   * Adjust the classification threshold to optimize for recall on the default class, as false negatives are more critical in this context.
4. **Deploy Ensemble Models:** Combine the strengths of multiple models (e.g., stacking or voting classifiers) to improve recall for defaults while maintaining overall accuracy.
5. **Business Strategy:** Focus on high-risk groups flagged by the models for targeted interventions, such as credit counseling or revised credit terms.

By implementing these steps, the models' ability to identify defaulting customers can be enhanced, aligning better with the project's goal of mitigating credit risk effectively.

**D. 04\_Deployment.ipynb**

**Model Deployment**

1. **Save the Best Model**:

python

CopyEdit

joblib.dump(rf\_model, 'credit\_default\_model.pkl') # Save the Random Forest model

1. **Create a Prediction API**:

python

CopyEdit

from flask import Flask, request, jsonify

app = Flask(\_\_name\_\_)

model = joblib.load('credit\_default\_model.pkl')

@app.route('/predict', methods=['POST'])

def predict():

input\_data = request.json['data']

prediction = model.predict([input\_data])

return jsonify({'prediction': int(prediction[0])})

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

**STEPS**

**Solution 1: Run Flask in a Script (Recommended)**

Save your Flask code to a Python script file (e.g., app.py) and run it from the terminal or command prompt.

1. **Save the Flask App**: Create a file named app.py with the following content:

python

CopyEdit

from flask import Flask, request, jsonify

import joblib

app = Flask(\_\_name\_\_)

model = joblib.load('credit\_default\_model.pkl')

@app.route('/predict', methods=['POST'])

def predict():

input\_data = request.json['data']

prediction = model.predict([input\_data])

return jsonify({'prediction': int(prediction[0])})

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

1. **Run the Script**: Open a terminal or command prompt, navigate to the folder containing app.py, and execute:

bash

CopyEdit

python app.py

1. **Access the API**: The app will run on http://127.0.0.1:5000 by default. Use tools like Postman, curl, or a Python script to send POST requests.

**Solution 2: Run Flask in Jupyter Notebook Using flask.run**

If you want to run the app in Jupyter, you'll need to disable Flask's reloader (which conflicts with Jupyter).

1. **Modify Your Flask Code**: Update the if \_\_name\_\_ == '\_\_main\_\_' section:

python

CopyEdit

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=False, use\_reloader=False)

1. **Run the Cell**: After making this change, re-run the cell. Flask should start successfully.

**Debugging Tips**

* **Check Model Path:** Ensure 'credit\_default\_model.pkl' exists in the current directory.
* **IPython Conflict:** If the issue persists, consider using %tb in Jupyter to view the full traceback for additional insights.

If you're testing an API from Jupyter, you can send requests like this:

import requests

url = 'http://127.0.0.1:5000/predict'

data = {'data': [1, 23, 5000, 0, 2, 1, 1200]} # Example data; replace with actual input format

response = requests.post(url, json=data)

print(response.json())

1. **Optional Streamlit Dashboard**:

python

CopyEdit

import streamlit as st

import numpy as np

st.title("Credit Default Prediction")

st.write("Provide customer details to predict default status:")

limit\_bal = st.number\_input("Credit Limit (LIMIT\_BAL):", min\_value=0.0)

age = st.number\_input("Age:", min\_value=18)

# Add other input fields as necessary

if st.button("Predict"):

input\_data = np.array([limit\_bal, age, ...]) # Replace '...' with other inputs

prediction = model.predict([input\_data])

st.write("Prediction:", "Default" if prediction[0] == 1 else "Non-Default")

**Final Notes**

1. Replace credit\_card\_data.csv with the actual dataset path.
2. Update preprocessing and model parameters based on the dataset’s specifics.
3. Test the API and dashboard locally before deploying them to a cloud environment (e.g., AWS, Heroku).

This structure provides a robust end-to-end solution for your project. Let me know if you need further assistance!

**NOTE**:

Create a separate file for each section: Creating separate .ipynb files for each section (e.g., EDA, Preprocessing, Modeling, Deployment) is a recommended approach. This modular structure keeps your project organized and easy to navigate, especially when sharing or maintaining the code. Here's why it's beneficial and how to structure your project:

**Benefits of Separate Files**

1. **Modularity**: Each file focuses on a specific aspect of the project, making it easier to understand and debug.
2. **Reusability**: You can reuse the preprocessing or modeling code in other projects without extracting it from a monolithic notebook.
3. **Collaboration**: Easier for team members to work on different sections without conflicts.
4. **Clarity**: Separates exploratory steps (EDA) from the production-ready workflow (Modeling and Deployment).

**Suggested File Structure**

Organize your project into the following files and directories:

bash

CopyEdit

project-folder/

│

├── data/

│ └── credit\_card\_data.csv # Dataset

│

├── notebooks/

│ ├── 01\_EDA.ipynb # Exploratory Data Analysis

│ ├── 02\_Preprocessing.ipynb # Data Preprocessing

│ ├── 03\_Modeling.ipynb # Model Building and Evaluation

│ └── 04\_Deployment.ipynb # Model Deployment

│

├── models/

│ └── credit\_default\_model.pkl # Trained model file

│

├── scripts/

│ └── api.py # Flask or FastAPI script

│ └── dashboard.py # Streamlit Dashboard script

│

└── README.md # Project documentation

**Guidelines for Each Notebook**

1. **01\_EDA.ipynb**:
   * Import libraries.
   * Load and explore the dataset.
   * Include visualizations (histograms, scatter plots, box plots, heatmaps).
   * Summarize insights.
2. **02\_Preprocessing.ipynb**:
   * Handle missing values.
   * Encode categorical features.
   * Engineer features if needed.
   * Scale/normalize numerical data.
   * Save preprocessed datasets if necessary.
3. **03\_Modeling.ipynb**:
   * Split the dataset into training and testing sets.
   * Train baseline models (Logistic Regression).
   * Train advanced models (Random Forest, XGBoost).
   * Evaluate models and save the best-performing one.
4. **04\_Deployment.ipynb**:
   * Load the saved model.
   * Create prediction examples.
   * Demonstrate API usage (if applicable).
   * Discuss integration with dashboar

**Optional Enhancements**

* **README.md**: Include project objectives, dataset description, and usage instructions.
* **Version Control**: Use GitHub to track changes and collaborate.
* **Environment Setup**: Add a requirements.txt file with Python libraries used in the project.

This approach ensures your project is professional, maintainable, and ready for deployment or sharing. Let me know if you need help with any specific section!

To load and reuse the model you saved using joblib.dump, you can use joblib.load. Here's how you can do it:

**STEPS TO LOAD AND REUSE THE SAVED MODEL**

Lets assume we save a model with the code

joblib.dump(xgb\_model, 'credit\_default\_model.pkl') # Save the Random Forest model,

1. **Import joblib**: Ensure you import the joblib library in your Jupyter Notebook.
2. **Load the Model**: Use joblib.load to load the saved model.
3. **Use the Loaded Model**: The loaded model can be used for predictions or further analysis.

**Example Code:**

python

CopyEdit

import joblib

# Load the saved model

rf\_model\_loaded = joblib.load('credit\_default\_model.pkl')

# Use the loaded model for predictions

# Assuming X\_test is already prepared

y\_pred = rf\_model\_loaded.predict(X\_test)

# Evaluate the loaded model (example: accuracy)

from sklearn.metrics import accuracy\_score

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

print(f"Accuracy of the loaded model: {accuracy:.2f}")

**Notes:**

* Ensure that the test data (X\_test) used for predictions has the same structure and preprocessing as the data used during training.
* If you used feature scaling during training, apply the same scaling to the test data before making predictions.