```python
# Task 1: Data Loading & Cleaning

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

```

```python

# Load the Dataset:

file_path = "mini_lending_club.csv"
df = pd.read_csv(file_path)

# Preview the first rows
print("Preview of dataset:")
display(df.head())

print("\nDataset shape:", df.shape)
```

Preview of dataset:

	loan_amnt	term	int_rate	installment	grade	sub_grade	emp_length	home_ownership	annual_inc	purpose	dti	de
0	16795	36 months	23.39%	55.32	G	A5	2 years	RENT	33704	home_improvement	12.11	
1	1860	60 months	29.96%	664.55	F	C1	6 years	OWN	78933	debt_consolidation	1.50	
2	12284	36 months	28.33%	319.87	E	C1	10+ years	OWN	91101	home_improvement	25.18	
3	7265	36 months	21.06%	455.57	E	B3	10+ years	RENT	63856	credit_card	33.34	
4	17850	36 months	15.53%	557.98	D	A1	8 years	OWN	57256	other	9.24	

Dataset shape: (2000, 16)

```python
# Inspect Generic Info About the Dataset
# --------------------------------------

print("\nDataset Info:")
df.info()

# Summary statistics for numerical features
print("\nSummary Statistics:")
display(df.describe())
```

```
Dataset Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2000 entries, 0 to 1999
Data columns (total 16 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   loan_amnt         2000 non-null   int64  
 1   term              2000 non-null   object 
 2   int_rate          1900 non-null   object 
 3   installment       2000 non-null   float64
 4   grade             2000 non-null   object 
 5   sub_grade         2000 non-null   object 
 6   emp_length        1900 non-null   object 
 7   home_ownership    2000 non-null   object 
 8   annual_inc        2000 non-null   int64  
 9   purpose           1900 non-null   object 
 10  dti               2000 non-null   float64
 11  delinq_2yrs       2000 non-null   int64  
 12  revol_util        1900 non-null   object 
 13  total_acc         2000 non-null   int64  
 14  earliest_cr_line  2000 non-null   int64  
 15  loan_status       2000 non-null   object 
dtypes: float64(2), int64(5), object(9)
memory usage: 250.1+ KB

Summary Statistics:
```

	loan_amnt	installment	annual_inc	dti	delinq_2yrs	total_acc	earliest_cr_line
count	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000
mean	18031.749500	413.815520	85211.457000	17.889410	2.070000	40.250000	2006.874000
std	9788.029854	213.172169	37139.980987	10.114553	1.441575	22.706041	5.899367
min	1009.000000	50.180000	20057.000000	0.020000	0.000000	1.000000	1999.000000
25%	9485.500000	230.990000	52672.250000	9.222500	1.000000	20.000000	2001.000000
50%	18330.500000	411.110000	85122.500000	17.765000	2.000000	40.000000	2008.000000
75%	26567.250000	589.030000	117562.500000	26.510000	3.000000	60.000000	2012.000000
max	34997.000000	799.180000	149986.000000	35.000000	4.000000	79.000000	2016.000000

```python
# 3. Check for missing values
# --------------------------------------
print("\nMissing Values:")
display(df.isnull().sum())

# Visualize missing values
plt.figure(figsize=(10,4))
sns.heatmap(df.isnull(), cbar=False)
plt.title("Missing Values Heatmap")
plt.show()

```

```
Missing Values:
```

	0
loan_amnt	0
term	0
int_rate	100
installment	0
grade	0
sub_grade	0
emp_length	100
home_ownership	0
annual_inc	0
purpose	100
dti	0
delinq_2yrs	0
revol_util	100
total_acc	0
earliest_cr_line	0
loan_status	0

dtype: int64


```
# 4. Clean & preprocess raw string features (SAFE VERSION)
# --------------------------------------------------------

# Ensure string conversion before using .str
df['int_rate'] = df['int_rate'].astype(str).str.replace('%','', regex=False)
df['revol_util'] = df['revol_util'].astype(str).str.replace('%','', regex=False)

# Convert to numeric (coerce errors turns bad values into NaN)
df['int_rate'] = pd.to_numeric(df['int_rate'], errors='coerce')
df['revol_util'] = pd.to_numeric(df['revol_util'], errors='coerce')

# emp_length cleaning
df['emp_length'] = df['emp_length'].astype(str)

df['emp_length'] = (
    df['emp_length']
    .replace({
        '< 1 year': '0',
```

```
        '10+ years': '10'
    })
    .str.extract(r'(\d+)', expand=False)   # extract number
)

df['emp_length'] = pd.to_numeric(df['emp_length'], errors='coerce')

# earliest credit line
df['earliest_cr_line'] = pd.to_numeric(df['earliest_cr_line'], errors='coerce')


# Handle missing values
num_cols = df.select_dtypes(include=['float64','int64']).columns
df[num_cols] = df[num_cols].fillna(df[num_cols].median())

cat_cols = df.select_dtypes(include=['object']).columns
df[cat_cols] = df[cat_cols].fillna(df[cat_cols].mode().iloc[0])

# ---------------------------------------
# 6. Convert loan_status to a binary target
# ---------------------------------------

df['loan_default'] = df['loan_status'].apply(
    lambda x: 1 if x == "Charged Off" else 0
)

print("\nLoan Status Conversion (value counts):")
print(df['loan_default'].value_counts())


```

```
Loan Status Conversion (value counts):
loan_default
0    1518
1     482
Name: count, dtype: int64
```

```
df.to_parquet("df_cleaned.parquet", index=False)

```

```
#Task 2 — FEATURE ENGINEERING
# --------------------------

from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline


# These help the model understand borrower risk better.

df["income_to_loan"] = df["annual_inc"] / df["loan_amnt"]              # Ability to repay
df["credit_util_ratio"] = df["revol_util"] / 100                       # Revolving credit usage
df["dti_income_ratio"] = df["dti"] / (df["annual_inc"] + 1)            # DTI relative to income
df["credit_history_years"] = 2024 - df["earliest_cr_line"]             # Borrower's credit experience
df["installment_ratio"] = df["installment"] / (df["annual_inc"] + 1)   # Payment pressure
```

```
# SEPARATE FEATURES (X) AND TARGET (y)
# ------------------------------------------------

y = df["loan_default"]  # Target is the dependent variable. what I want to predict and it can't be a part of the data that

# Remove columns that should not be used in modeling, it could lead to data-leakage.
drop_cols = ["loan_status","loan_default","earliest_cr_line"]
X = df.drop(columns=drop_cols)

# IDENTIFY NUMERIC & CATEGORICAL COLUMNS
# ------------------------------------------------
numeric_features = X.select_dtypes(include=["float64", "int64"]).columns
categorical_features = X.select_dtypes(include=["object"]).columns

print("Numeric Columns:\n", list(numeric_features))
print("\nCategorical Columns:\n", list(categorical_features))

```

```
y.to_frame().to_parquet("y.parquet", index=False) # save target variable y

```

```
Numeric Columns:
 ['loan_amnt', 'int_rate', 'installment', 'emp_length', 'annual_inc', 'dti', 'delinq_2yrs', 'revol_util', 'total_acc', 'income_t

Categorical Columns:
 ['term', 'grade', 'sub_grade', 'home_ownership', 'purpose']
```

```
df.columns

```

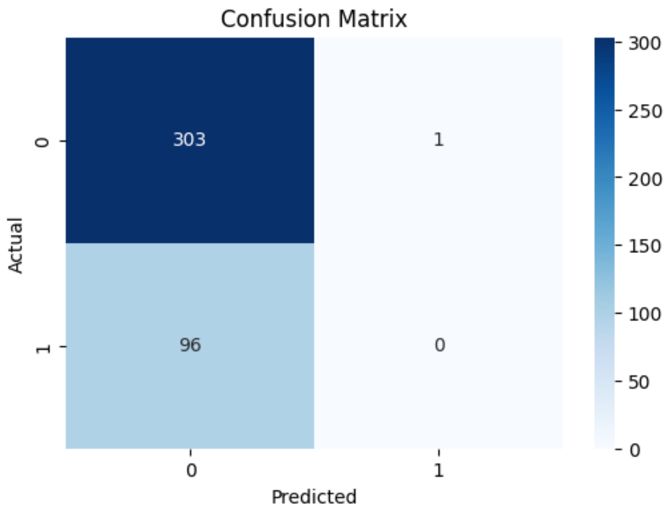
```
Index(['loan_amnt', 'term', 'int_rate', 'installment', 'grade', 'sub_grade',
       'emp_length', 'home_ownership', 'annual_inc', 'purpose', 'dti',
       'delinq_2yrs', 'revol_util', 'total_acc', 'earliest_cr_line',
       'loan_status', 'income_to_loan', 'credit_util_ratio',
       'dti_income_ratio', 'credit_history_years', 'installment_ratio',
       'loan_default'],
      dtype='object')
```

```
# SET UP TRANSFORMERS
# ------------------------------------------------

# StandardScaler:
# → Makes numeric features comparable by converting them to mean=0, variance=1.

numeric_transformer = StandardScaler()

# OneHotEncoder:
# → Converts categories (e.g., "RENT", "OWN") into 0/1 columns the model can understand.
categorical_transformer = OneHotEncoder(handle_unknown='ignore')

# ColumnTransformer:
# → Applies transformations to different column groups:
#      - scale numeric features
#      - one-hot encode categorical features
# → Combines everything into ONE clean matrix
# In plain sense I am using the in-built libr ColumnTransf to consolidate both col groups (numericla and Category into one

preprocessor = ColumnTransformer(
    transformers=[
        ("num", numeric_transformer, numeric_features),  # scale numerics
        ("cat", categorical_transformer, categorical_features)  # encode categoricals
    ]
)

# ------------------------------------------------
#  APPLY TRANSFORMATIONS
# ------------------------------------------------
# fit_transform:
# → Learns how to scale from data
# → Creates a transformed feature matrix ready for modelling

X_prepared = preprocessor.fit_transform(X)

# ================================
# FIX FEATURE NAMES (IMPORTANT FOR SHAP)
# ================================

# 1. Numeric feature names
numeric_feature_names = numeric_features.tolist()

# 2. One-hot encoded categorical feature names
categorical_feature_names = (
    preprocessor.named_transformers_['cat']
    .get_feature_names_out(categorical_features)
    .tolist()
)

# 3. Combine them into one final list
all_feature_names = numeric_feature_names + categorical_feature_names

print("Number of feature names:", len(all_feature_names)) # the len indicate the number of features

import scipy.sparse # for saving the file jsut i case it crashes
scipy.sparse.save_npz("X_prepared.npz", X_prepared)
```

```
Number of feature names: 66
```

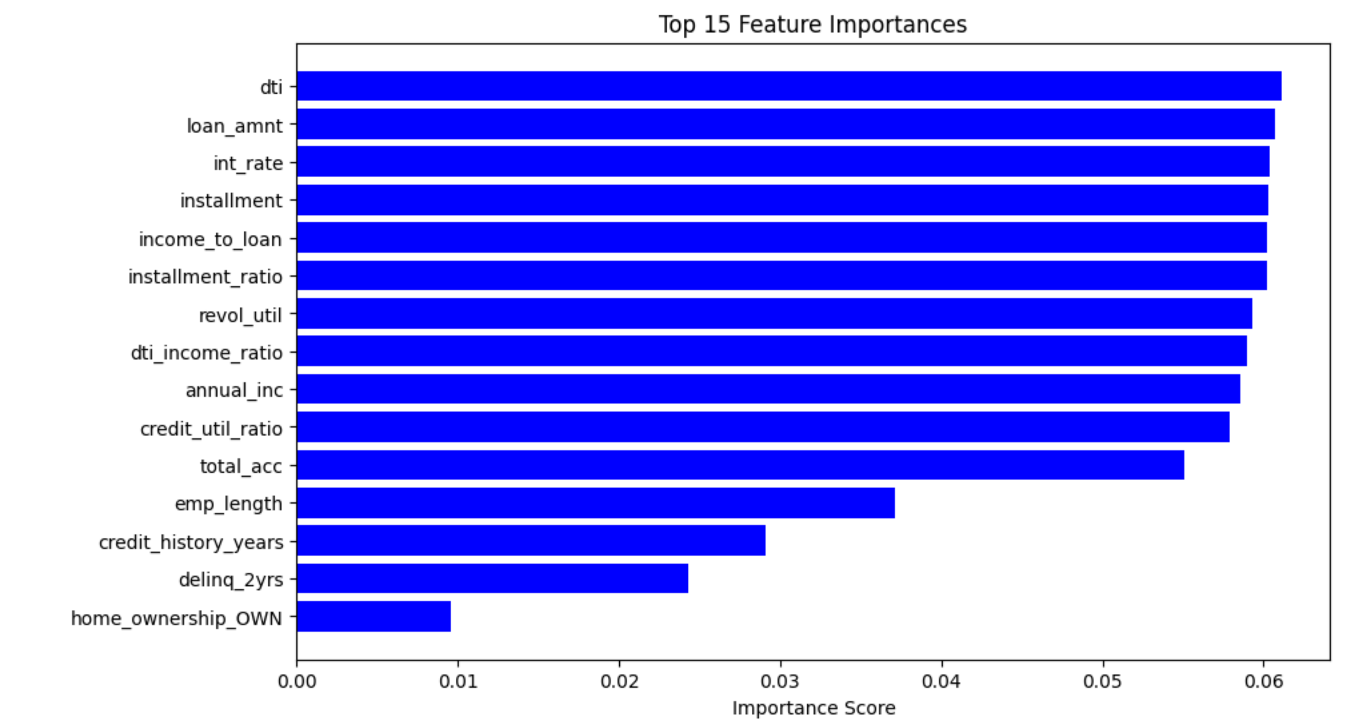
```
# Task 3: MODEL TRAINING & EVALUATION
# ======================================

# Import Suitable Libraries

from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

# 1. TRAIN-TEST SPLIT
# -----------------------------------------

# X_prepared = transformed feature matrix from Task 2 (pipeline output)
# y = target variable from Task 2

X_train, X_test, y_train, y_test = train_test_split(
    X_prepared, # this replaces the X value for training
    y, # this represents the target value
    test_size=0.2,
    random_state=42,
    stratify=y  # ensures balanced target distribution
)


# TRAIN RANDOM FOREST MODEL
# -----------------------------------------

# Model Selection: I am using RandomForest to determine the prediction for the loan_default prediction

random_forex = RandomForestClassifier(
    n_estimators=200,
    max_depth=None,
    random_state=42
)

random_forex .fit(X_train, y_train)

# 3. MAKE PREDICTIONS
# -----------------------------------------

y_pred = random_forex.predict(X_test) # - this is the prediction of the model


# 4. MODEL EVALUATION - this is the process of evaluating that the model has a higher accuracy rate or not.
# -----------------------------------------

print("Accuracy:", accuracy_score(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred))

# Confusion Matrix
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(6,4))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues")
plt.title("Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
```

```
Accuracy: 0.7575

Classification Report:
              precision    recall  f1-score   support

           0       0.76      1.00      0.86       304
           1       0.00      0.00      0.00        96

    accuracy                           0.76       400
   macro avg       0.38      0.50      0.43       400
weighted avg       0.58      0.76      0.66       400
```




```
y.to_frame().to_parquet("y.parquet", index=False)

```

```
# 5. FEATURE IMPORTANCE VISUALIZATION
# ----------------------------------------

# Grab feature names from the preprocessor
# (numerical + encoded categorical names)
num_features = numeric_features
cat_features = list(preprocessor.named_transformers_['cat'].get_feature_names_out(categorical_features))

all_feature_names = list(num_features) + cat_features

importances = random_forex.feature_importances_

# Sort for visualization
indices = np.argsort(importances)[-15:]  # top 15 features

plt.figure(figsize=(10,6))
plt.barh(range(len(indices)), importances[indices], color='blue')
plt.yticks(range(len(indices)), [all_feature_names[i] for i in indices])
plt.title("Top 15 Feature Importances")
plt.xlabel("Importance Score")
plt.show()
```




```
# ================================
# TASK 4 — SHAP & FAIRNESS ANALYSIS
# ================================
import shap
shap.initjs()

# -----------------------------------------
# 1. PREP SHAP INPUT (convert sparse → dense)
# I had an issue with this section and request the assistance of Google Gemini
# -----------------------------------------

X_test_sample = X_test[:300]   # smaller sample for speed

# Convert sparse matrix to dense
if hasattr(X_test_sample, "toarray"):
    X_test_sample_dense = X_test_sample.toarray()
else:
    X_test_sample_dense = np.array(X_test_sample)

# -----------------------------------------
# 2. SHAP EXPLAINER FOR RANDOM FOREST
# -----------------------------------------

# The error indicates a mismatch in the number of features between shap_values
# and X_test_sample_dense. This often happens if the explainer is not
# perfectly aligned with the data it's explaining, or if the model's
# internal feature representation differs.
# To try and align them, we explicitly pass the data to the explainer.

explainer = shap.TreeExplainer(random_forex, data=X_test_sample_dense)

# RandomForest produces 2 classes → index 1 is “default”

shap_values = explainer.shap_values(X_test_sample_dense)

# Diagnostic: Print shapes before plotting to confirm the issue
print(f"Shape of X_test_sample_dense: {X_test_sample_dense.shape}")

# shap_values can be a list of arrays or a 3D array; handle both

if isinstance(shap_values, list):
    print(f"Shape of shap_values (list of arrays): {[s.shape for s in shap_values]}")
    if len(shap_values) > 1:
        print(f"Shape of shap_values[1]: {shap_values[1].shape}")
else: # assuming it's a numpy array
    print(f"Shape of shap_values (3D array): {shap_values.shape}")
```

```
    # The previous line shap_values[1] was incorrect for a 3D array; it picked a sample, not a class.
    # The correct way to get class 1's shap values for all samples is shap_values[:, :, 1]
    print(f"Shape of shap_values[:, :, 1]: {shap_values[:, :, 1].shape}")
print(f"Length of all_feature_names: {len(all_feature_names)}")


# ----------------------------------------
# 3. GLOBAL SHAP SUMMARY PLOTS
# ----------------------------------------

# Beeswarm plot — feature impact
shap.summary_plot(
    shap_values[:, :, 1],          # Correctly select SHAP values for class 1 (default) across all samples
    X_test_sample_dense,
    feature_names=all_feature_names
)

# Clean bar plot — easy for reports
shap.summary_plot(
    shap_values[:, :, 1],
    X_test_sample_dense,
    feature_names=all_feature_names,
    plot_type="bar"
)

# ----------------------------------------
# 4. LOCAL EXPLANATION (single borrower)
# ----------------------------------------

idx = 0
shap.force_plot(
    explainer.expected_value[1],
    shap_values[idx, :, 1], # Correctly select SHAP values for a single sample (idx) and class 1
    X_test_sample_dense[idx],
    feature_names=all_feature_names
)

# ----------------------------------------
# 5. FAIRNESS ANALYSIS — GROUP ACCURACY
# ----------------------------------------

# Recreate original X_test (non-transformed) for grouping
from sklearn.model_selection import train_test_split

X_train_df, X_test_df, y_train_df, y_test_df = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

# Attach predictions to original test data
fairness_df = X_test_df.copy()
fairness_df["y_true"] = y_test_df.values
fairness_df["y_pred"] = random_forex.predict(X_test)

# Choose grouping variable
group_col = "home_ownership"

print("Groups:", fairness_df[group_col].unique())

group_metrics = {}
for group in fairness_df[group_col].unique():
    sub = fairness_df[fairness_df[group_col] == group]
    acc = (sub["y_true"] == sub["y_pred"]).mean()
    group_metrics[group] = acc
    print(f"Accuracy for {group}: {acc:.3f}")

# Bar chart for fairness visualization
plt.figure(figsize=(6,4))
plt.bar(group_metrics.keys(), group_metrics.values(), color="steelblue")
plt.title("Model Accuracy by Home Ownership Group")
plt.xlabel("Group")
plt.ylabel("Accuracy")
plt.ylim(0, 1)
plt.show()
```