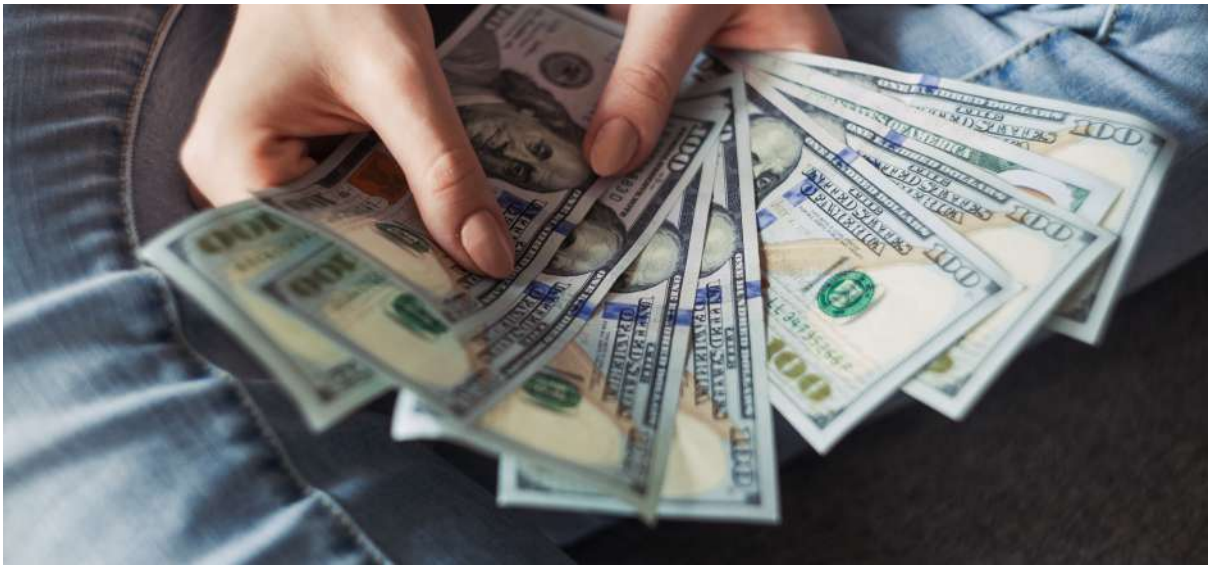

Credit Risk Modelling

Concise Explanation of Code

```
# Import necessary libraries
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score, p
recision_score, recall_score, f1_score
```

Explanation: Here, import the required Python libraries and modules, including pandas for data manipulation, scikit-learn (sklearn) for machine learning tasks, and relevant metrics for model evaluation.

```
# Load the dataset from a CSV file
df = pd.read_csv("lending_club_loan_dataset.csv")
```

https://www.kaggle.com/datasets/qusaybtoush1990/machine-learning/data

Explanation: Load the dataset from a CSV file named "lending_club_loan_dataset.csv" into a pandas DataFrame called `df`.

```
# Display basic information about the dataset
df.info()
```

Explanation: Use `df.info()` to display basic information about the dataset, including the number of non-null entries in each column and data types.

```
# Check the shape of the dataset
df.shape
```

Explanation: Check and display the shape of the dataset, which shows the number of rows and columns.

```
# Check for missing values in the dataset
df.isnull().sum()
```

Explanation: Check for missing values in the dataset and display the count of missing values for each column.

```
# Drop unnecessary columns ('id' and 'last_major_derog_none')
df.drop(["id", "last_major_derog_none"], axis=1, inplace=True)
```

Explanation: Remove the columns 'id' and 'last_major_derog_none' as they are deemed unnecessary for the analysis.

```
# Display summary statistics of the dataset
df.describe()
```

Explanation: Generate summary statistics for the dataset, including measures like mean, standard deviation, and quartiles for numerical columns.

```
# Handle missing values in the 'dti' column by filling with the mean
mean_dti = df["dti"].mean()
df["dti"].fillna(mean_dti, inplace=True)
```

Explanation: Fill missing values in the 'dti' (debt-to-income ratio) column with the mean of that column, which is a common strategy to handle missing data.

```
# Check the distribution of the 'home_ownership' column
df['home_ownership'].value_counts()
```

Explanation: Check the distribution of values in the 'home_ownership' column to understand the different categories and their frequencies.

```
# Fill missing values in the 'home_ownership' column with the mode
mode_ho = df["home_ownership"].mode().iloc[0]
df["home_ownership"].fillna(mode_ho, inplace=True)
```

Explanation: Fill missing values in the 'home_ownership' column with the mode (most frequent value) of that column, which is a common strategy for categorical data.

```
# Check for duplicate rows in the dataset
df.duplicated().sum()
```

Explanation: Check for duplicate rows in the dataset to identify and potentially remove redundant data.

```
# Remove "months" from the "term" column and convert it to an integer
df['term'] = df['term'].str.strip().str.split(" ").str[0].astype(int)
```

Explanation: Process the 'term' column to remove the "months" label and convert the remaining numeric value to an integer. This is done to make the data suitable for analysis.

```
# Use one-hot encoding to convert categorical columns to numerical
df = pd.get_dummies(df, columns=['home_ownership', 'purpose', 'grade'], prefix=['home_
ownership', 'purpose', 'grade'], drop_first=True)
```

Explanation: Perform one-hot encoding on categorical columns ('home_ownership', 'purpose', 'grade') to convert them into numerical format. This process creates binary (0/1) columns for each category, effectively representing them as separate features. The `drop_first=True` argument avoids multicollinearity by dropping one of the encoded columns for each category.

```
# Separate the data into normal and fraud classes for oversampling
normal = df[df.bad_loan == 0]
fraud = df[df.bad_loan == 1]
```

Explanation: Split the dataset into two parts based on the target variable 'bad_loan'. 'normal' contains samples where 'bad_loan' is 0 (not a bad loan), and 'fraud' contains samples where 'bad_loan' is 1 (bad loan).

```
# Calculate the difference in sample sizes between normal and fraud classes
sample_size_difference = len(normal) - len(fraud)
```

Explanation: Calculate the difference in the number of samples between the 'normal' and 'fraud' classes. This difference will be used to generate additional samples for the 'fraud' class to balance the dataset.

```
# Create additional samples from the fraud class
fraud_samples = fraud.sample(n=sample_size_difference, replace=True)
```

Explanation: Randomly select a subset of samples from the 'fraud' class with replacement (allowing the same sample to be picked multiple times) to balance the class distribution.

```
# Concatenate the original normal samples with the additional fraud samples
oversampled_df = pd.concat([normal, fraud_samples], axis=0)
```

Explanation: Combine the original 'normal' samples and the additional 'fraud' samples to create an oversampled dataset where both classes have an equal number of samples.

```
# Shuffle the oversampled data to ensure randomness
oversampled_df = oversampled_df.sample(frac=1, random_state=42)
```

Explanation: Shuffle the order of rows in the oversampled dataset to ensure that the data is random and not biased by the original order.

```
# Split the data into features (X) and target (y)
X = oversampled_df.drop('bad_loan', axis=1)
y = oversampled_df['bad_loan']
```

Explanation: Split the oversampled dataset into features (X) and the target variable (y), where 'X' contains all the columns except 'bad_loan,' and 'y' contains the 'bad_loan' column.

```
# Split the resampled data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=
42, stratify=y)
```

Explanation: Split the oversampled data into training and testing sets using the `train_test_split` function. The `test_size=0.3` argument specifies that 30% of the data will be used for testing, while the remaining 70% will be used for training. The `stratify=y` argument ensures that the class distribution is preserved in both the training and testing sets.

```
# Initialize and train a RandomForestClassifier model on the training data
model = RandomForestClassifier(random_state=42)
model.fit(X_train, y_train)
```

Explanation: Here, Initialize a RandomForestClassifier model with a random seed of 42 for reproducibility and train it using the training data (`X_train` and `y_train`). This step is where your machine learning model learns from the data to make predictions.

```
# Standardize numerical features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Explanation: Standardize the numerical features in the dataset. Standardization ensures that each feature has a mean of 0 and a standard deviation of 1, which can be important for some machine learning algorithms to perform well.

```
# Initialize and train a RandomForestClassifier model on the resampled data
rfc = RandomForestClassifier(random_state=42)
rfc.fit(X_train, y_train)
```

Explanation: Initialize a RandomForestClassifier model and train it on the standardized training data (`X_train` and `y_train`). This step is a reiteration of model training, but now it's based on the standardized data.

```
# Make predictions on the test set
y_pred = rfc.predict(X_test)
```

Explanation: Use the trained RandomForestClassifier model to make predictions on the standardized test data (`X_test`) to evaluate how well the model performs on unseen data.

```
# Evaluate the model using appropriate metrics
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
```

Explanation: Calculate various classification metrics to evaluate the performance of your model on the test data. These metrics include accuracy, precision, recall, and F1-score, which provide insights into how well your model classifies positive and negative cases and overall performance.

```
# Print the evaluation metrics
print(f'Accuracy: {accuracy:.2f}')
print(f'Precision: {precision:.2f}')
print(f'Recall: {recall:.2f}')
print(f'F1-Score: {f1:.2f}')
```

Explanation: Print the calculated evaluation metrics to the console, allowing you to assess the model's performance in terms of accuracy, precision, recall, and F1-score.

```
# Print classification report and confusion matrix
print('Classification Report:\\n', classification_report(y_test, y_pred))
print('Confusion Matrix:\\n', confusion_matrix(y_test, y_pred))
```

Explanation: Finally, Print a more detailed classification report and the confusion matrix. The classification report provides a breakdown of precision, recall, F1-score, and support for each class. The confusion matrix shows the number of true positives, true negatives, false positives, and false negatives, offering insights into how the model is making correct and incorrect predictions.

```
Accuracy: 0.91
Precision: 0.89
Recall: 0.90
F1-Score: 0.89
Classification Report:
              precision    recall  f1-score   support
```

```
              0       0.92      0.91      0.92      4800
              1       0.89      0.90      0.89      3600

    accuracy                           0.91      8400
   macro avg       0.91      0.91      0.91      8400
weighted avg       0.91      0.91      0.91      8400

Confusion Matrix:
 [[4391  409]
 [ 369 3231]]
```

let's break down and explain the above results:

Accuracy: 0.91

- Accuracy measures the overall correctness of the model's predictions.
- In this case, the model's accuracy is 0.91, which means it correctly predicted approximately 91% of the total cases.

Precision: 0.89

- Precision is the ratio of true positive predictions to the total positive predictions made by the model.
- A precision of 0.89 indicates that when the model predicts a positive case, it is correct about 89% of the time.

Recall: 0.90

- Recall, also known as sensitivity or true positive rate, is the ratio of true positive predictions to the total actual positive cases.
- A recall of 0.90 means the model correctly identified 90% of all actual positive cases.

F1-Score: 0.89

- The F1-Score is the harmonic mean of precision and recall and provides a balance between the two.
- An F1-Score of 0.89 suggests a good balance between precision and recall.

Classification Report:

- The classification report provides more detailed performance metrics for each class (0 and 1).
- For class 0 (negative cases):

    - Precision is 0.92, indicating that when the model predicts class 0, it is correct about 88% of the time.
    - Recall is 0.91, which means the model correctly identified 62% of actual class 0 cases.
    - The F1-Score for class 0 is 0.92.
- For class 1 (positive cases):
    - Precision is 0.89, implying that when the model predicts class 1, it is correct about 64% of the time.
    - Recall is 0.90, indicating that the model correctly identified 89% of actual class 1 cases.
    - The F1-Score for class 1 is 0.89.

Support:

- Support refers to the number of instances of each class in the test dataset.
- There are 4800 instances of class 0 and 3600 instances of class 1 in the test data.

Confusion Matrix:

- The confusion matrix provides a breakdown of predicted and actual class labels.
- In this case, the confusion matrix is as follows:
    - True negatives (TN): 4391- The number of cases correctly predicted as class 0.
    - False positives (FP): 409- The number of cases incorrectly predicted as class 1.
    - False negatives (FN): 369 - The number of cases incorrectly predicted as class 0.
    - True positives (TP): 3231 - The number of cases correctly predicted as class 1.

In summary, This model has reasonably good accuracy, but it's especially strong in terms of recall for class 1 (positive cases). This indicates that the model is effective at correctly identifying positive cases, which is often important in credit risk modeling to avoid missing high-risk loans. However, there's a trade-off with precision, which is slightly lower, suggesting that there might be some false positives. The F1-Score provides a balanced measure of the model's performance.