credit_risk_master

August 22, 2024

1 Credit Risk Prediction Machine Learning Model:

1.1 A Logistic Regression Model to Predict whether Customers will or will not Default on Loans

1.1.1 This machine learning model addresses the issue of credit risk prediction. The goal is to predict whether a customer will default on their loan (bad) or not (good), based on various customer and loan-related features. By accurately identifying high-risk customers, financial institutions can better manage risk, make informed lending decisions, and avoid potential losses.

I chose the logistic regression algorithm because it is not only CPU-friendly (running on my local machine with 20 million rows/6 GB of data in this dataset), it is also well-suited for binary classification problems (default vs. no default). Logistic regression is interpretable, stable, and performs well when the relationship between the features and the target is approximately linear. It is computationally efficient, interpretable, and works well with a large number of features. Additionally, logistic regression's coefficients provide insights into how each feature contributes to the likelihood of default, which is crucial for explainability in financial contexts.

We Begin by Importing all necessary Python libraries.

[115]:
```
import csv
import joblib
import lightgbm as lgb
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import plotly.express as px
import polars as pl
import pyarrow.feather as feather
import seaborn as sns
import shap
import xgboost as xgb
from imblearn.over_sampling import SMOTE
from imblearn.pipeline import make_pipeline
from sklearn.cluster import KMeans
from sklearn.compose import ColumnTransformer
from sklearn.ensemble import RandomForestClassifier
```

```python
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report,␣
 ↪confusion_matrix, make_scorer, f1_score, precision_score, recall_score,␣
 ↪roc_auc_score
from sklearn.model_selection import GridSearchCV, StratifiedKFold,␣
 ↪cross_val_score, train_test_split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import MinMaxScaler, OneHotEncoder, StandardScaler
from xgboost import XGBClassifier
```

This is the .py Script I made that creates the initial 20 million row/6.4 GB dataset for our model.

[ ]:
```python
"""import pandas as pd
import numpy as np
from tqdm import tqdm
import time

# Start the timer (data creation purposes only)
start_time = time.time()

# Set a random seed for reproducibility
np.random.seed(0)

# Define N number of samples to simulate the number of financial customers
n_samples = 20_000_000

# Create the customer_id array separately and ensure its length matches␣
 ↪n_samples
customer_id = np.unique(np.random.randint(1000, 198234870, n_samples))

# If the length of customer_id is less than n_samples, regenerate it
while len(customer_id) < n_samples:
    additional_ids = np.unique(np.random.randint(1000, 198234870, n_samples -␣
 ↪len(customer_id)))
    customer_id = np.concatenate((customer_id, additional_ids))

# Generate structured random data
data = {
    'customer_id': customer_id[:n_samples],
    'age': np.random.randint(18, 85, n_samples),
    'gender': np.random.choice(['Male', 'Female', 'Other'], n_samples),
    'marital_status': np.random.choice(['Single', 'Married', 'Divorced',␣
 ↪'Widowed'], n_samples),
    'dependents': np.random.randint(0, 5, n_samples),
    'employment_status': np.random.choice(['Employed', 'Unemployed',␣
 ↪'Self-employed', 'Retired'], n_samples),
```

```python
    'annual_income': np.random.randint(20_000, 1_000_000, n_samples),
    'loan_amount': np.random.randint(5000, 1_000_000, n_samples),
    'loan_term': np.random.randint(1, 30, n_samples),
    'interest_rate': np.round(np.random.uniform(2.5, 20.0, n_samples), 2),
    'loan_purpose': np.random.choice(['Home', 'Car', 'Education', 'Personal',␣
↪'Business'], n_samples),
    'loan_to_value_ratio': np.round(np.random.uniform(0.5, 1.5, n_samples), 2),
    'credit_score': np.random.randint(300, 850, n_samples),
    'debt_to_income_ratio': np.round(np.random.uniform(0.01, 0.5, n_samples),␣
↪2),
    'delinquencies': np.random.randint(0, 10, n_samples),
    'credit_history_length': np.random.randint(0, 40, n_samples),
}

# Introduce a probabilistic relationship between income, loan amount, credit␣
↪score, and default
default_prob = (
    (data['loan_amount'] / data['annual_income']) * 0.5 +
    (700 - data['credit_score']) * 0.001 +
    data['debt_to_income_ratio'] * 2
)
default_prob = np.clip(default_prob, 0, 1)  # Ensure probabilities are between␣
↪0 and 1

# Generate the default column based on the probability
data['default'] = np.random.binomial(1, default_prob, n_samples)

# Generate the default_amount based on whether the customer defaulted
data['default_amount'] = np.where(data['default'] == 1, np.random.randint(0,␣
↪1_000_000, n_samples), 0)

# Add repayment_tenure based on default and loan term
data['repayment_tenure'] = np.where(data['default'] == 1, np.random.randint(1,␣
↪360, n_samples), data['loan_term'] * 12)

# Create DataFrame
credit_risk_data = pd.DataFrame({key: tqdm(value, desc=key) for key, value in␣
↪data.items()})

# Save dataset to HDD
filepath = "D:\\datasets\\github_credit_risk_modeling_data\\credit_risk_data.
↪csv"
credit_risk_data.to_csv(filepath, index=False)

# End the timer (data creation purposes only)
end_time = time.time()
```

```
elapsed_time = end_time - start_time

# Convert elapsed time to hours, minutes, and seconds
hours = int(elapsed_time // 3600) # integer division to get whole hours (not␣
↪'true division')
minutes = int((elapsed_time % 3600) // 60)
seconds = elapsed_time % 60

# Print the time in a readable format
if hours > 0:
    print(f"Data generation and saving completed in {hours} hours, {minutes}␣
↪minutes, and {seconds:.2f} seconds.")
elif minutes > 0:
    print(f"Data generation and saving completed in {minutes} minutes and␣
↪{seconds:.2f} seconds.")
else:
    print(f"Data generation and saving completed in {seconds:.2f} seconds.")

print(f"Data has been generated and saved to {filepath}.")"""
```

1.1.2 Data Dictionary for our Dataset

Column Name	Data Type	Description
`customer_id`	String	Unique identifier for each customer.
`age`	Integer	Age of the customer in years.
`gender`	String	Gender of the customer (e.g., 'Male', 'Female', 'Other').
`marital_status`	String	Marital status of the customer (e.g., 'Single', 'Married', 'Divorced', 'Widowed').
`dependents`	Integer	Number of dependents the customer has.
`employment_status`	String	Employment status of the customer (e.g., 'Employed', 'Unemployed', 'Self-Employed', 'Retired').
`annual_income`	Float	Customer's annual income in dollars.
`loan_amount`	Float	Total amount of loan requested by the customer in dollars.
`loan_term`	Integer	Duration of the loan in months.
`interest_rate`	Float	Interest rate applied to the loan as a percentage.
`loan_purpose`	String	Purpose of the loan (e.g., 'Home Improvement', 'Debt Consolidation', 'Education').
`loan_to_value_ratio`	Float	Ratio of the loan amount to the appraised value of the asset being purchased or financed.
`credit_score`	Integer	Customer's credit score (typically ranges from 300 to 850).
`debt_to_income_ratio`	Float	Ratio of customer's total monthly debt payments to their gross monthly income, expressed as a percentage.
`delinquencies`	Integer	Number of past missed payments or delinquencies on previous loans.
`credit_history_length`	Integer	Length of time in years the customer has held credit.

Column Name	Data Type	Description
`default`	Integer	Indicates whether the customer defaulted on the loan (0 = No, 1 = Yes).
`default_amount`	Float	Amount of money defaulted by the customer in dollars (if applicable).
`repayment_tenure`	Integer	Number of months the customer has been repaying the loan (can be the loan term if repaid fully).
`income_loan_interaction`	Float	Interaction term between the customer's annual income and loan amount, used to capture non-linear effects.
`credit_interest_interaction`	Float	Interaction term between the customer's credit score and loan interest rate.
`log_loan_amount`	Float	Logarithmic transformation of the loan amount to reduce skewness and deal with outliers.
`log_annual_income`	Float	Logarithmic transformation of the annual income to reduce skewness and deal with outliers.
`credit_to_loan_ratio`	Float	Ratio of the customer's credit score to the loan amount.
`credit_to_income_ratio`	Float	Ratio of the customer's credit score to their annual income.
`age_income_interaction`	Float	Interaction term between the customer's age and annual income to capture non-linear effects.
`debt_to_income_ratio_cluster`	Integer	Cluster assignment based on KMeans clustering for debt-to-income ratio, used to group similar customers.
`age_cluster`	Integer	Cluster assignment based on KMeans clustering for age, used to group similar customers.
`annual_income_cluster`	Integer	Cluster assignment based on KMeans clustering for annual income, used to group similar customers.
`credit_score_bucket`	Integer	Binned categories for credit scores (e.g., 'Poor', 'Fair', 'Good', 'Excellent') based on predefined thresholds.
`income_loan_amount_interaction`	Float	Interaction term between annual income and loan amount to capture combined effect.
`credit_score_income_interaction`	Float	Interaction term between credit score and annual income to capture non-linear effects.
`loan_amount_interest_rate_interaction`	Float	Interaction term between loan amount and interest rate to capture non-linear effects.
`log_debt_to_income_ratio`	Float	Logarithmic transformation of the debt-to-income ratio to reduce skewness and deal with outliers.

Load CSV with Polars instead of Pandas for speed and check length to make sure all data transferred

Convert to Pandas DataFrame for analysis and check length of Pandas version to make sure all data transferred

[3]:
```
df = pl.read_csv('D:
↪\\datasets\\github_credit_risk_modeling_data\\credit_risk_data.csv')
```

[4]:
```
print(f"Loaded dataset has {len(df):,.0f} rows.")
```

```
Loaded dataset has 20,000,000 rows.
```

[5]:
```python
df = df.to_pandas()
```

[6]:
```python
print(f"pandas-from-polars dataset has {len(df):,.0f} rows.")
```

```
pandas-from-polars dataset has 20,000,000 rows.
```

[6]:
```python
# Store so you don't have to do this all over again (takes 75 minutes to make␣
↪this dataset; make sure to store!)
# %store df
```

```
Stored 'df' (DataFrame)
```

[7]:
```python
# In a new Jupyter Notebook session
# %store -r df
```

Because this dataset is massive, I have randomly shuffled and split 10% of it for this deployment. How to handle the full dataset (and even larger) is discussed in my full project on my portfolio: http://github.com/nervousblakedown

Optional: Store dataset as .feather and .parquet files for memory and speed

[135]:
```python
# Create feather file from dataset for speed and memory purposes
df.to_feather('D:\\datasets\\github_credit_risk_modeling_data\\credit_risk_data.
↪feather')
```

[136]:
```python
# Create parquet file from dataset for speed and memory purposes
df.to_parquet('D:\\datasets\\github_credit_risk_modeling_data\\credit_risk_data.
↪parquet')
```

Randomly shuffled 10% of full dataset is used in this notebook for CPU-intensive purposes

[7]:
```python
# Use sample of df for feature engineering, EDA, and model training before␣
↪applying to full dataset
df_sample = df.sample(frac=0.1)  # 10% sample
```

[12]:
```python
%store df_sample
# %store -r df_sample
```

```
Stored 'df_sample' (DataFrame)
```

[13]:
```python
# Get the memory usage of the DataFrame in bytes
memory_usage_bytes = df.memory_usage(deep=True).sum()

# Convert to MB and GB
memory_usage_mb = memory_usage_bytes / (1024 ** 2)  # Convert bytes to megabytes
memory_usage_gb = memory_usage_bytes / (1024 ** 3)  # Convert bytes to gigabytes
```

```python
print(f"Memory usage in MB: {memory_usage_mb:.2f} MB")
print(f"Memory usage in GB: {memory_usage_gb:.6f} GB")
```

```
Memory usage in MB: 6559.35 MB
Memory usage in GB: 6.405617 GB
```

Preview dataset before adding features for the model

[9]:
```python
df_sample.head(2)
```

[9]:
```
          customer_id  age  gender marital_status  dependents  \
19803836    158827883   57    Male       Divorced           0
2804079      29220329   23  Female        Widowed           3

         employment_status  annual_income  loan_amount  loan_term  \
19803836     Self-employed         244202       155856         17
2804079           Employed         532337       357391         15

          interest_rate loan_purpose  loan_to_value_ratio  credit_score  \
19803836          11.68     Personal                 0.95           816
2804079           16.74         Home                 1.03           648

          debt_to_income_ratio  delinquencies  credit_history_length  default  \
19803836                  0.46              2                      0        1
2804079                   0.21              2                     19        1

          default_amount  repayment_tenure
19803836          905049                16
2804079           892323               331
```

Add two columns that could be useful for the model: average loan amount each year and average loan amount each month

[10]:
```python
df_sample['avg_loan_year_amount'] = df_sample['loan_amount'] /␣
↪df_sample['loan_term']
```

[11]:
```python
df_sample['avg_loan_monthly_amount'] = df_sample['avg_loan_year_amount'] / 12
```

[12]:
```python
df_sample.describe()
```

[12]:
```
        customer_id           age    dependents  annual_income   loan_amount  \
count  2.000000e+06  2.000000e+06  2.000000e+06   2.000000e+06  2.000000e+06
mean   9.923974e+07  5.099447e+01  1.998763e+00   5.098184e+05  5.024960e+05
std    5.718279e+07  1.933344e+01  1.413668e+00   2.829206e+05  2.871808e+05
min    1.018000e+03  1.800000e+01  0.000000e+00   2.000000e+04  5.000000e+03
25%    4.979671e+07  3.400000e+01  1.000000e+00   2.648668e+05  2.537228e+05
50%    9.926413e+07  5.100000e+01  2.000000e+00   5.094780e+05  5.022610e+05
75%    1.487612e+08  6.800000e+01  3.000000e+00   7.548595e+05  7.513150e+05
max    1.982349e+08  8.400000e+01  4.000000e+00   9.999990e+05  9.999990e+05
```

```
           loan_term  interest_rate  loan_to_value_ratio  credit_score  \
count   2.000000e+06   2.000000e+06         2.000000e+06  2.000000e+06
mean    1.499913e+01   1.125042e+01         9.999502e-01  5.745105e+02
std     8.369187e+00   5.052433e+00         2.887356e-01  1.587843e+02
min     1.000000e+00   2.500000e+00         5.000000e-01  3.000000e+02
25%     8.000000e+00   6.870000e+00         7.500000e-01  4.370000e+02
50%     1.500000e+01   1.125000e+01         1.000000e+00  5.750000e+02
75%     2.200000e+01   1.563000e+01         1.250000e+00  7.120000e+02
max     2.900000e+01   2.000000e+01         1.500000e+00  8.490000e+02

       debt_to_income_ratio  delinquencies  credit_history_length  \
count          2.000000e+06   2.000000e+06           2.000000e+06
mean           2.550668e-01   4.503995e+00           1.949877e+01
std            1.414534e-01   2.871910e+00           1.154346e+01
min            1.000000e-02   0.000000e+00           0.000000e+00
25%            1.300000e-01   2.000000e+00           9.000000e+00
50%            2.600000e-01   5.000000e+00           2.000000e+01
75%            3.800000e-01   7.000000e+00           2.900000e+01
max            5.000000e-01   9.000000e+00           3.900000e+01

             default  default_amount  repayment_tenure  avg_loan_year_amount  \
count   2.000000e+06    2.000000e+06      2.000000e+06          2.000000e+06
mean    8.857290e-01    4.427157e+05      1.800149e+02          6.864992e+04
std     3.181402e-01    3.148222e+05      1.033854e+02          1.178229e+05
min     0.000000e+00    0.000000e+00      1.000000e+00          1.724138e+02
25%     1.000000e+00    1.530920e+05      9.000000e+01          1.692327e+04
50%     1.000000e+00    4.351960e+05      1.800000e+02          3.349964e+04
75%     1.000000e+00    7.176192e+05      2.700000e+02          6.497935e+04
max     1.000000e+00    9.999990e+05      3.590000e+02          9.999990e+05

       avg_loan_monthly_amount
count             2.000000e+06
mean              5.720827e+03
std               9.818576e+03
min               1.436782e+01
25%               1.410273e+03
50%               2.791637e+03
75%               5.414946e+03
max               8.333325e+04
```

[13]:
```
df_sample.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 2000000 entries, 19803836 to 19437445
Data columns (total 21 columns):
 #   Column                  Dtype
---  ------                  -----
```

```
 0   customer_id             int64
 1   age                     int64
 2   gender                  object
 3   marital_status          object
 4   dependents              int64
 5   employment_status       object
 6   annual_income           int64
 7   loan_amount             int64
 8   loan_term               int64
 9   interest_rate           float64
 10  loan_purpose            object
 11  loan_to_value_ratio     float64
 12  credit_score            int64
 13  debt_to_income_ratio    float64
 14  delinquencies           int64
 15  credit_history_length   int64
 16  default                 int64
 17  default_amount          int64
 18  repayment_tenure        int64
 19  avg_loan_year_amount    float64
 20  avg_loan_monthly_amount  float64
dtypes: float64(5), int64(12), object(4)
memory usage: 335.7+ MB
```

[14]:
```python
# Summary of categorical columns
df_sample.describe(include=['object', 'category'])
```

[14]:

	gender	marital_status	employment_status	loan_purpose
count	2000000	2000000	2000000	2000000
unique	3	4	4	5
top	Other	Divorced	Unemployed	Home
freq	667729	500460	500785	400979

[15]:
```python
# Check for missing values
df_sample.isnull().sum()
```

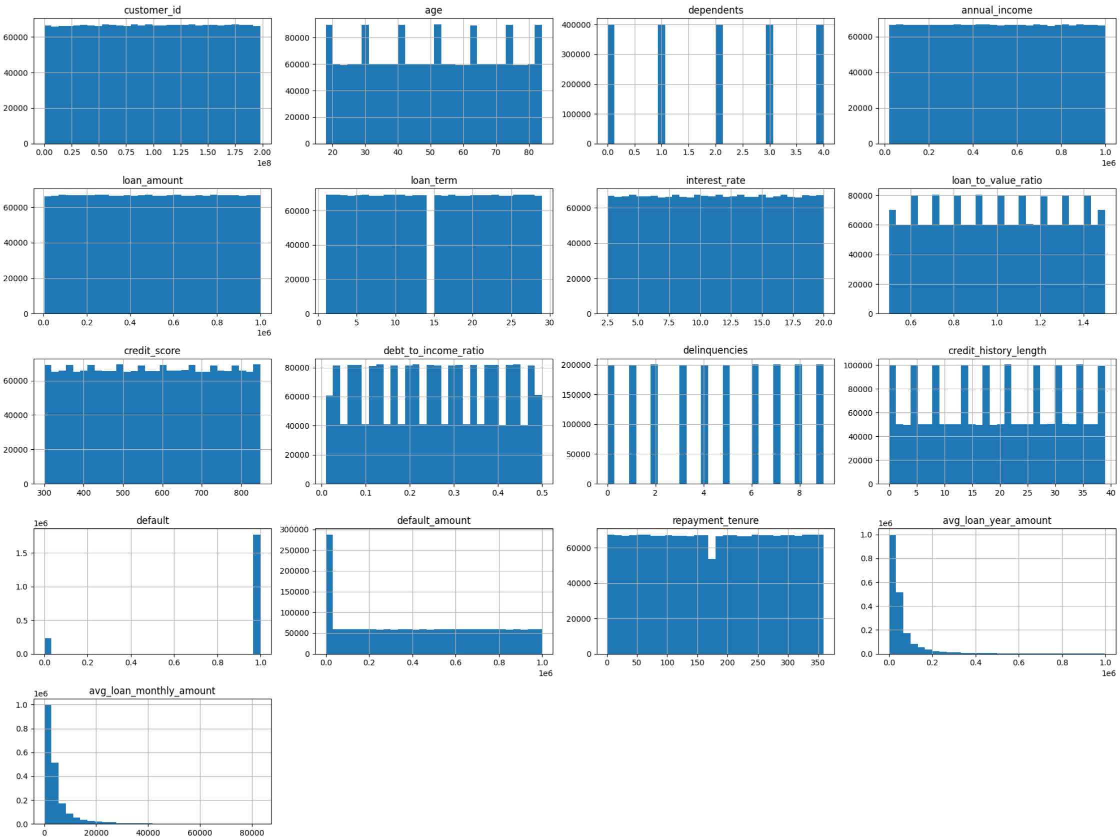
[15]:
```
customer_id            0
age                    0
gender                 0
marital_status         0
dependents             0
employment_status      0
annual_income          0
loan_amount            0
loan_term              0
interest_rate          0
loan_purpose           0
loan_to_value_ratio    0
```

```
credit_score              0
debt_to_income_ratio      0
delinquencies             0
credit_history_length     0
default                   0
default_amount            0
repayment_tenure          0
avg_loan_year_amount      0
avg_loan_monthly_amount   0
dtype: int64
```

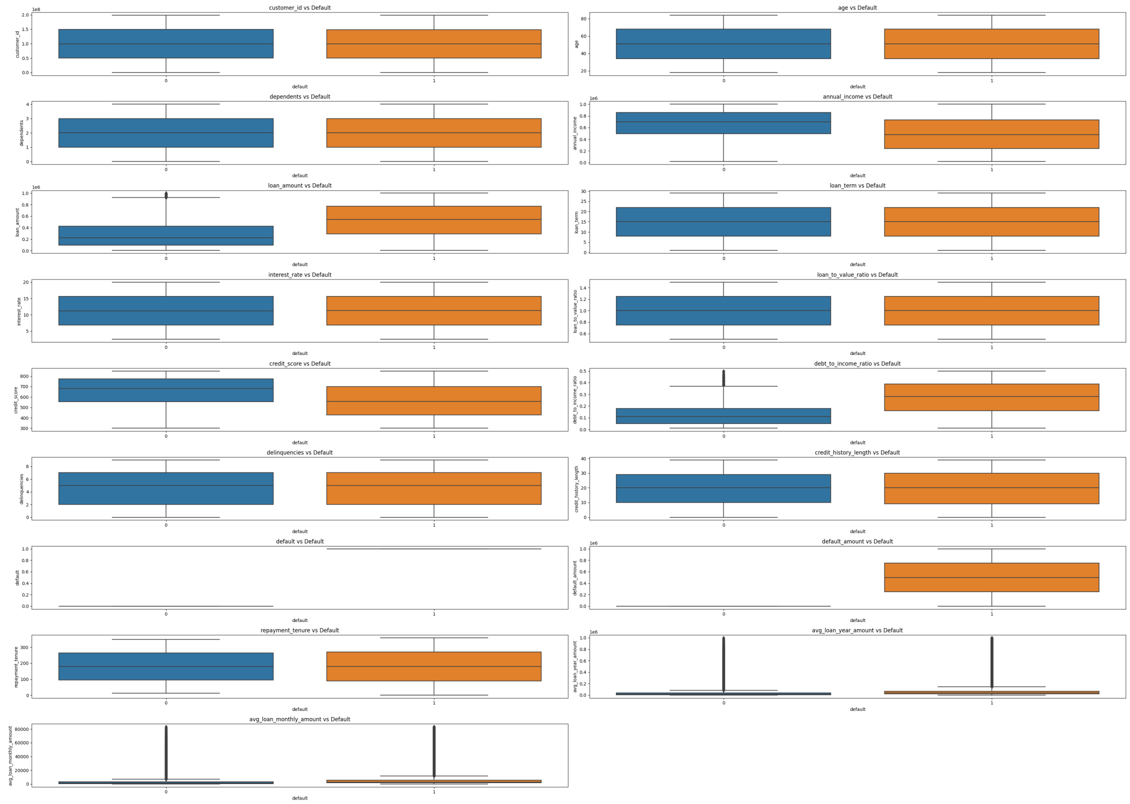
Perform Z-Score test to find outliers in data

[ ]:
```python
from scipy import stats
import numpy as np

# Calculate Z-scores for each column in the dataset
z_scores = np.abs(stats.zscore(df_sample))

# Define a threshold for what you consider an outlier (usually |Z| > 3)
threshold = 3
outliers = (z_scores > threshold)

# Show the number of outliers in each column
outliers_summary = outliers.sum(axis=0)
print(outliers_summary)
```

[16]:
```python
# Plot histograms for numerical features
df_sample.hist(bins=30, figsize=(20, 15))
plt.tight_layout()
plt.show();
```



[20]:
```python
# Box plots to understand the relationship between features and the target variable
plt.figure(figsize=(35, 25))
for i, col in enumerate(df_sample.select_dtypes(include=['float64', 'int64']).columns):
    plt.subplot(len(df_sample.select_dtypes(include=['float64', 'int64']).columns) // 2 + 1, 2, i + 1)
    sns.boxplot(x='default', y=col, data=df_sample)
    plt.title(f'{col} vs Default')
plt.tight_layout()
plt.show()
```



[21]:
```
age_mean = df_sample['age'].mean()
print(f"The mean age of customer in the dataset is {age_mean:.0f}.")
```

The mean age of customer in the dataset is 51.

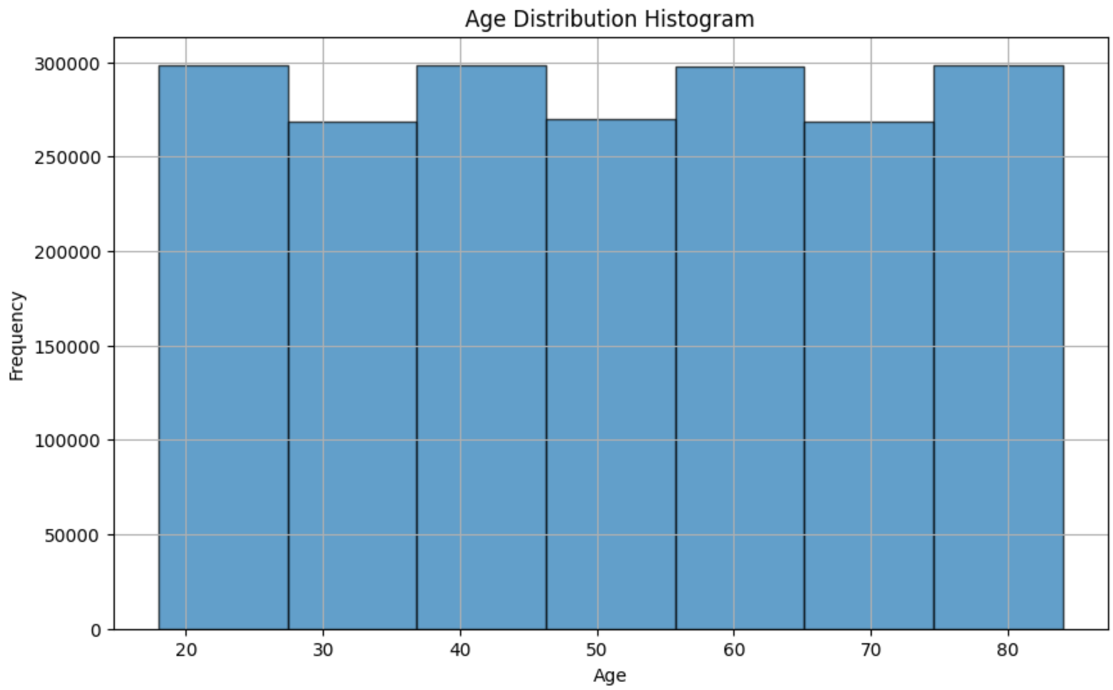
[22]:
```
# Calculate the interquartile range (IQR)
q25, q75 = np.percentile(df_sample['age'], [25, 75])
iqr = q75 - q25

print(iqr, q25, q75)
```

34.0 34.0 68.0

[23]:
```
# Calculate the bin width using the Freedman-Diaconis rule
bin_width = 2 * iqr * len(df_sample['age']) ** (-1/3)

# Calculate the number of bins
num_bins = int(np.ceil((df_sample['age'].max() - df_sample['age'].min()) /
 ↪bin_width))

print(f"Optimal number of bins according to the Freedman-Diaconis rule:
 ↪{num_bins}")
```

```
Optimal number of bins according to the Freedman-Diaconis rule: 123
```

[27]:
```python
# Plot the histogram
plt.figure(figsize=(10, 6))
plt.hist(df_sample['age'], bins=7, edgecolor='black', alpha=0.7)
plt.title('Age Distribution Histogram')
plt.xlabel('Age')
plt.ylabel('Frequency')
plt.grid(True)
plt.show()
```



[1]:
```python
# recall pandas df from notebook I
%store -r df

# recall sampled 10% of dataset for speed, memory, and everything else for this␣
↪project
%store -r df_sample
```

Make sure the target column in sampled shuffled 10% of the full data is proportional to full dataset before proceeding

[28]:
```python
print("Class distribution in full dataset:")
print(df['default'].value_counts(normalize=True))
print("\nClass distribution in sample:")
print(df_sample['default'].value_counts(normalize=True))
```

```
Class distribution in full dataset:
default
1    0.885802
0    0.114198
Name: proportion, dtype: float64

Class distribution in sample:
default
1    0.885729
0    0.114271
Name: proportion, dtype: float64
```

[137]:

```
print('full dataset target value counts:')
print(df['default'].value_counts())
print()
print('sample dataset target value counts:')
print(df_sample['default'].value_counts())
```

```
full dataset target value counts:
default
1    17716047
0     2283953
Name: count, dtype: int64

sample dataset target value counts:
default
0    1771458
1     228542
Name: count, dtype: int64
```

Check if there's missing data

[30]:

```
total_missing_full = df.isnull().sum().sum()
print(f"Total missing values in full df: {total_missing_full}")
print()
total_missing = df_sample.isnull().sum().sum()
print(f"Total missing values in sample df: {total_missing}")
```

```
Total missing values in full df: 0

Total missing values in sample df: 0
```

Reassign randomly-generated gender values for binary classification purposes

[33]:

```
def reassign_gender(gender):
    if gender == 'Other':
        return np.random.choice(['Male', 'Female'])
    else:
        return gender
```

```python
df_sample['gender'] = df_sample['gender'].apply(reassign_gender)
```

[34]:

```python
df_sample.tail()
```

[34]:

```
          customer_id  age  gender marital_status  dependents  \
15650807    163077750   80  Female        Married           0
509656        5314893   79    Male       Divorced           4
3214049      33490497   27  Female        Married           3
19848664    167967854   71  Female         Single           0
19437445     84347939   36  Female         Single           2

         employment_status  annual_income  loan_amount  loan_term  \
15650807     Self-employed         206521       205633         24
509656       Self-employed         284852       918192         19
3214049         Unemployed         806231       692592         10
19848664           Retired         358793       159182         21
19437445           Retired         190502       679387         26

          interest_rate  … loan_to_value_ratio  credit_score  \
15650807          15.98  …                0.88           581
509656             2.95  …                1.12           838
3214049           16.84  …                0.63           304
19848664          11.37  …                1.19           751
19437445           5.76  …                1.43           793

          debt_to_income_ratio  delinquencies  credit_history_length  default  \
15650807                  0.49              3                     29        1
509656                    0.05              8                     19        1
3214049                   0.26              9                     19        1
19848664                  0.42              3                     26        1
19437445                  0.49              4                     10        1

          default_amount  repayment_tenure  avg_loan_year_amount  \
15650807          197336                16           8568.041667
509656            236555                57          48325.894737
3214049           306525               255          69259.200000
19848664          266536                67           7580.095238
19437445          336048               170          26130.269231

          avg_loan_monthly_amount
15650807               714.003472
509656                4027.157895
3214049               5771.600000
19848664               631.674603
19437445              2177.522436

[5 rows x 21 columns]
```

Add features for modeling based on statistical and financial domain knowledge

We find out which features are best and worst for the model later in this notebook.

[35]:
```
# Feature Engineering on sample df (before scaling and encoding)
df_sample['income_loan_interaction'] = df_sample['annual_income'] *␣
  ↪df_sample['loan_amount']
df_sample['credit_interest_interaction'] = df_sample['credit_score'] *␣
  ↪df_sample['interest_rate']
df_sample['log_loan_amount'] = np.log1p(df_sample['loan_amount'])
df_sample['log_annual_income'] = np.log1p(df_sample['annual_income'])
df_sample['debt_to_income_ratio'] = df_sample['loan_amount'] /␣
  ↪df_sample['annual_income']
df_sample['credit_to_loan_ratio'] = df_sample['credit_score'] /␣
  ↪df_sample['loan_amount']
df_sample['credit_to_income_ratio'] = df_sample['loan_amount'] /␣
  ↪df_sample['annual_income']
df_sample['age_income_interaction'] = df_sample['age'] *␣
  ↪df_sample['annual_income']
```

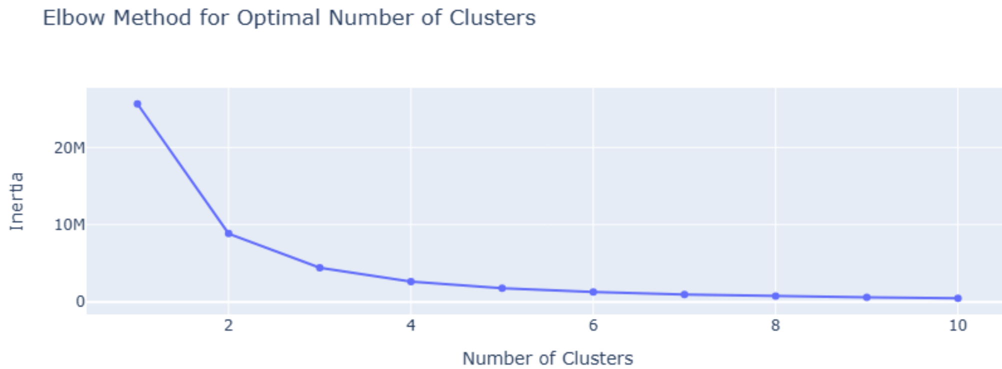
[36]:
```
df_sample.head()
```

[36]:
```
          customer_id  age  gender marital_status  dependents  \
19803836    158827883   57    Male       Divorced           0
2804079      29220329   23  Female        Widowed           3
3230531      33662454   58    Male        Widowed           4
18200485    189653651   32  Female       Divorced           4
4627376      48225708   66    Male         Single           3

         employment_status  annual_income  loan_amount  loan_term  \
19803836     Self-employed         244202       155856         17
2804079           Employed         532337       357391         15
3230531         Unemployed          38255       714284          9
18200485     Self-employed         897128       760022         22
4627376      Self-employed         572895       502473         26

          interest_rate  … repayment_tenure  avg_loan_year_amount  \
19803836          11.68  …               16           9168.000000
2804079           16.74  …              331          23826.066667
3230531            3.57  …              347          79364.888889
18200485           3.20  …              103          34546.454545
4627376           14.44  …              147          19325.884615

          avg_loan_monthly_amount  income_loan_interaction  \
19803836               764.000000              38060346912
2804079               1985.505556             190252452767
3230531               6613.740741              27324934420
18200485              2878.871212             681837016816
```

```
4627376              1610.490385          287864269335

          credit_interest_interaction  log_loan_amount  log_annual_income  \
19803836                      9530.88        11.956694          12.405755
2804079                      10847.52        12.786588          13.185034
3230531                       2806.02        13.479037          10.552056
18200485                      2412.80        13.541104          13.706955
4627376                       8245.24        13.127299          13.258459

          credit_to_loan_ratio  credit_to_income_ratio  age_income_interaction
19803836              0.005236                0.638226                13919514
2804079               0.001813                0.671362                12243751
3230531               0.001100               18.671651                 2218790
18200485              0.000992                0.847172                28708096
4627376               0.001136                0.877077                37811070

[5 rows x 28 columns]
```

To see if clusters on age, debt-income ratio, and other features, I perform K-Means Clustering to find the most meaningful number of clusters.

[39]:
```python
# k-means clustering for debt to income ratio (other columns as well; change␣
↪values here)
inertia = []

# Step 1: Fit KMeans with different cluster numbers and store inertia
for k in range(1, 11):  # Test for 1 to 10 clusters
    kmeans = KMeans(n_clusters=k, random_state=42, n_init = 'auto')
    kmeans.fit(df_sample[['debt_to_income_ratio']])
    inertia.append(kmeans.inertia_)

# Step 2: Plot the Elbow Curve
import plotly.graph_objects as go

# Create the plot
fig = go.Figure()

fig.add_trace(go.Scatter(x=list(range(1, 11)), y=inertia, mode='lines+markers'))

# Add titles and labels
fig.update_layout(
    title="Elbow Method for Optimal Number of Clusters",
    xaxis_title="Number of Clusters",
    yaxis_title="Inertia"
)

# Show the plot
fig.show()
```



With the Elbow Method, the best number of clusters is at the "elbow" of the line. Because 3 seems too small for this size of data, I have picked 4 as the number for my categories.

[43]:
```
# Clustering
n_clusters = 4 # based off Elbow Method, possibly silhouette score in the future

kmeans_dti = KMeans(n_clusters=n_clusters, random_state=42, n_init = 'auto')
df_sample['debt_to_income_ratio_cluster'] = kmeans_dti.
↪fit_predict(df_sample[['debt_to_income_ratio']])

kmeans_age = KMeans(n_clusters=n_clusters, random_state=42, n_init = 'auto')
df_sample['age_cluster'] = kmeans_age.fit_predict(df_sample[['age']])

kmeans_income = KMeans(n_clusters=n_clusters, random_state=42, n_init = 'auto')
df_sample['annual_income_cluster'] = kmeans_income.
↪fit_predict(df_sample[['annual_income']])
```

[44]:
```
# Further interaction terms and bucketing
df_sample['credit_score_bucket'] = pd.cut(df_sample['credit_score'],
                                          bins=[300, 579, 669, 739, 799, 850],
                                          labels=['Poor', 'Fair', 'Good', 'Very␣
↪Good', 'Excellent'],
                                          right=False)
df_sample['income_loan_amount_interaction'] = df_sample['annual_income'] *␣
↪df_sample['loan_amount']
df_sample['credit_score_income_interaction'] = df_sample['credit_score'] *␣
↪df_sample['annual_income']
df_sample['loan_amount_interest_rate_interaction'] = df_sample['loan_amount'] *␣
↪df_sample['interest_rate']
df_sample['log_debt_to_income_ratio'] = np.
↪log1p(df_sample['debt_to_income_ratio'])
```

Preview data with the several new columns we have added with financial domain knowledge and our K-Means Clusters

[45]:
```
df_sample.head(2)
```

[45]:
```
          customer_id  age  gender marital_status  dependents  \
19803836    158827883   57    Male       Divorced           0
2804079      29220329   23  Female        Widowed           3

         employment_status  annual_income  loan_amount  loan_term  \
19803836     Self-employed         244202       155856         17
2804079           Employed         532337       357391         15

          interest_rate  … credit_to_income_ratio  age_income_interaction  \
19803836          11.68  …               0.638226                13919514
2804079           16.74  …               0.671362                12243751

          debt_to_income_ratio_cluster  age_cluster  annual_income_cluster  \
19803836                             0            1                      2
2804079                              0            0                      3

          credit_score_bucket  income_loan_amount_interaction  \
19803836            Excellent                     38060346912
2804079                  Fair                    190252452767

          credit_score_income_interaction  \
19803836                        199268832
2804079                         344954376

          loan_amount_interest_rate_interaction  log_debt_to_income_ratio
19803836                             1820398.08                  0.493614
2804079                              5982725.34                  0.513639

[2 rows x 36 columns]
```

Begin one-hot encoding and feature scaling for transformations in model

[46]:
```
# Define the columns for the transformer
categorical_columns = ['marital_status', 'employment_status', 'loan_purpose',␣
 ↪'credit_score_bucket']
numerical_columns = ['age', 'annual_income', 'loan_amount', 'loan_term',␣
 ↪'interest_rate',
                     'loan_to_value_ratio', 'credit_score',␣
 ↪'debt_to_income_ratio',
                     'delinquencies', 'credit_history_length', 'default_amount',
                     'repayment_tenure', 'income_loan_interaction',␣
 ↪'credit_interest_interaction',
                     'log_loan_amount', 'log_annual_income',␣
 ↪'credit_to_loan_ratio',
```

```
                         'credit_to_income_ratio', 'age_income_interaction',␣
↪'income_loan_amount_interaction',
                         'credit_score_income_interaction',␣
↪'loan_amount_interest_rate_interaction',
                         'log_debt_to_income_ratio']
```

[47]:
```
# Column Transformer for processing the columns
preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), numerical_columns),
        ('cat', OneHotEncoder(drop='first'), categorical_columns)
    ]
)
```

Make copy of dataset that applies the transformations for clarity

[48]:
```
df_sample_feat_transformed = preprocessor.fit_transform(df_sample)
```

[49]:
```
# Get column names for the transformed data
encoded_columns = preprocessor.named_transformers_['cat'].
 ↪get_feature_names_out(categorical_columns)
all_columns = numerical_columns + list(encoded_columns)
```

[50]:
```
# Convert to a DataFrame
df_sample_feat_transformed = pd.DataFrame(df_sample_feat_transformed,␣
 ↪columns=all_columns)
```

[51]:
```
# Display the transformed DataFrame
df_sample_feat_transformed.head()
```

[51]:
```
        age  annual_income  loan_amount  loan_term  interest_rate  \
0  0.310629      -0.938837    -1.207045   0.239076       0.085024
1 -1.447982       0.079593    -0.505274   0.000104       1.086522
2  0.362353      -1.666769     0.737473  -0.716811      -1.520144
3 -0.982467       1.368969     0.896738   0.836506      -1.593376
4  0.776144       0.222948    -0.000080   1.314450       0.631295

   loan_to_value_ratio  credit_score  debt_to_income_ratio  delinquencies  \
0            -0.172996      1.520866             -0.383040      -0.871892
1             0.104074      0.462826             -0.373785      -0.871892
2             0.450411      1.331930              4.653632       1.217311
3             0.104074      1.130399             -0.324682       1.565511
4            -0.865672     -0.022109             -0.316330      -1.220093

   credit_history_length  …  employment_status_Self-employed  \
0              -1.689162  …                              1.0
1              -0.043208  …                              0.0
2               0.389938  …                              0.0
```

```
3            -0.303096  …                           1.0
4             1.689375  …                           1.0

   employment_status_Unemployed  loan_purpose_Car  loan_purpose_Education  \
0                           0.0               0.0                     0.0
1                           0.0               0.0                     0.0
2                           1.0               1.0                     0.0
3                           0.0               0.0                     1.0
4                           0.0               0.0                     0.0

   loan_purpose_Home  loan_purpose_Personal  credit_score_bucket_Fair  \
0                0.0                    1.0                      0.0
1                1.0                    0.0                      1.0
2                0.0                    0.0                      0.0
3                0.0                    0.0                      0.0
4                0.0                    0.0                      0.0

   credit_score_bucket_Good  credit_score_bucket_Poor  \
0                      0.0                       0.0
1                      0.0                       0.0
2                      0.0                       0.0
3                      0.0                       0.0
4                      0.0                       1.0

   credit_score_bucket_Very Good
0                            0.0
1                            0.0
2                            1.0
3                            1.0
4                            0.0

[5 rows x 37 columns]
```

[18]:
```
# Store the transformed DataFrame for the next part of the pipeline
# %store df_sample_feat_transformed
# %store df_sample
```

```
Stored 'df_sample_feat_transformed' (DataFrame)
Stored 'df_sample' (DataFrame)
```

[19]:
```
# %store all_columns
```

```
Stored 'all_columns' (list)
```

[53]:
```
# Save the preprocessor for future use
preprocessor = joblib.dump(preprocessor, "D:
↪\\datasets\\github_credit_risk_modeling_data\\preprocessor.pkl")
```

[21]:
```
# %store preprocessor
```

```
Stored 'preprocessor' (list)
```

[ ]:
```
# save df's as csv's
df_sample.to_csv("D:
↪\\datasets\\github_credit_risk_modeling_data\\credit_risk_data_sample.csv")
df_sample_feat_transformed.to_csv("D:
↪\\datasets\\github_credit_risk_modeling_data\\credit_risk_data_sample_feat_transformed.
↪csv")
```

1.1.3 We now go to the model training portion of this project.

Switching the 0's and 1's to reflect real-world settings (people do not default on loans majority of time)

[58]:
```
# y is target variable ('default') in our data
# Switch the labels so that 1 represents "defaulted" and 0 represents "did not␣
↪default"
y_switched = df_sample['default'].map({0: 1, 1: 0})
```

[59]:
```
# Assign the switched labels back to the DataFrame if needed
df_sample['default'] = y_switched
```

[60]:
```
# Verify the switch by checking the distribution of labels
print(df_sample['default'].value_counts())
```

```
default
0    1771458
1     228542
Name: count, dtype: int64
```

[61]:
```
# Get the percentage of each unique value in 'default' column
default_percentages = df_sample['default'].value_counts(normalize = True) * 100

# Optional: round the percentages to a decimal place
default_percentages = default_percentages.round()

# Print the rounded percentages
print(default_percentages)
```

```
default
0    89.0
1    11.0
Name: proportion, dtype: float64
```

1.1.4 1 = defaulted on loan (bad), 0 = did not default on loan (good)

1.1.5 For a binary classification problem such as this, the Logistic Regression algorithm is a great choice for predicting 0's and 1's in your data.

Note: I also used the XGBoost algorithm for its ability to capture more complex relationships, but because it is computation heavy on a laptop such as mine, I have removed those results from this notebook and instead saved them to my portfolio found here: http://github.com/nervousblakedown.

X variable: our transformed/scaled dataset

Y variable: our 'default' column, telling us which customers did or did not default on their loan we gave them

[62]:
```
# Step 1: Prepare the data
X = df_sample_feat_transformed  # Transformed features (without the 'default'␣
↪column)
y = df_sample['default']  # Target variable from the original/sampled 10%␣
↪dataset
```

We will randomly split our data into two chunks: 80% of it being used as the training set for the model, and the remaining 20% as the test set against the training set, which will determine how excellent or poor our model performs.

[66]:
```
# Step 2: Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)
```

[67]:
```
# Step 3: Train the Logistic Regression model
model = LogisticRegression(max_iter=1000, random_state=42)
model.fit(X_train, y_train)
```

[67]:
```
LogisticRegression(max_iter=1000, random_state=42)
```

[68]:
```
# Step 4: Make predictions on the test set
y_pred = model.predict(X_test)
```

[69]:
```
# Step 5: Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, pos_label=1)  # Defaults as␣
↪positive class
recall = recall_score(y_test, y_pred, pos_label=1)  # Defaults as positive class
conf_matrix = confusion_matrix(y_test, y_pred)
```

Print model results after first iteration

[70]:
```
# Print the results
print(f"Model Accuracy: {accuracy:.2f}")
print(f"Model Precision: {precision:.2f}")
print(f"Model Recall: {recall:.2f}")
```

```
print("Confusion Matrix:")
print(conf_matrix)
```

```
Model Accuracy: 1.00
Model Precision: 0.97
Model Recall: 1.00
Confusion Matrix:
[[352629   1581]
 [     1  45789]]
```

The model did really well on its first run, which most likely means overfitting (basically the training data getting used to itself) occurred. To make the model better, we will apply cross-validation, akin to making a validation set after a training set and test set have been made.

[71]:
```
# Perform 5-fold cross-validation on the Logistic Regression model
cv_scores = cross_val_score(model, X, y, cv=5, scoring='accuracy')

# Print cross-validation results
print(f'Cross-Validation Accuracy Scores: {cv_scores}')
print(f'Mean Accuracy from Cross-Validation: {cv_scores.mean():.2f}')
```

```
Cross-Validation Accuracy Scores: [0.9959825 0.9960225 0.99591   0.99599
0.9960875]
Mean Accuracy from Cross-Validation: 1.00
```

With the mean score being 100% on all five runs, overfitting definitely occurred which means our model is performing well under false pretenses. To improve authenticity in the model, we will apply the SMOTE technique to make sure the 0's and 1's have the same number of occurrences, making it more difficult (a 50/50 chance, to be exact) for the model to make correct predictions.

Since the dataset is imbalanced (there are more non-defaults than defaults since that is typical behavior with loans), I used SMOTE (Synthetic Minority Over-sampling Technique) to balance the classes. This helped to ensure that the model doesn't become biased toward the majority class (customers who did not default on their loans).

[72]:
```
# Apply SMOTE to the training set to balance the classes
smote = SMOTE(random_state=42)
X_train_balanced, y_train_balanced = smote.fit_resample(X_train, y_train)
```

[73]:
```
# Print the distribution of the new balanced dataset
print("Balanced Class Distribution:", y_train_balanced.value_counts())
```

```
Balanced Class Distribution: default
0    1417248
1    1417248
Name: count, dtype: int64
```

Our Target column is now evenly balanced between 0's and 1's (customer did or did not default on their loan), making it much harder for the model to make the correct prediction on the data, which is what we want.

[74]:
```
# Train the Logistic Regression model on the balanced data
model_balanced = LogisticRegression(max_iter=1000, random_state=42,␣
↪class_weight = 'balanced')
model_balanced.fit(X_train_balanced, y_train_balanced)

# Evaluate the model on the test set
y_pred_balanced = model_balanced.predict(X_test)

# Evaluate the model
accuracy_balanced = accuracy_score(y_test, y_pred_balanced)
precision_balanced = precision_score(y_test, y_pred_balanced)
recall_balanced = recall_score(y_test, y_pred_balanced)
conf_matrix_balanced = confusion_matrix(y_test, y_pred_balanced)

# Print the results for the balanced dataset
print(f"Model Accuracy (Balanced): {accuracy_balanced:.2f}")
print(f"Model Precision (Balanced): {precision_balanced:.2f}")
print(f"Model Recall (Balanced): {recall_balanced:.2f}")
print("Confusion Matrix (Balanced):")
print(conf_matrix_balanced)
```

```
Model Accuracy (Balanced): 0.99
Model Precision (Balanced): 0.96
Model Recall (Balanced): 1.00
Confusion Matrix (Balanced):
[[352083   2127]
 [     0  45790]]
```

We will apply cross-validation again now that the SMOTE technique has been applied to the model.

[75]:
```
# Step 1: Create a pipeline with SMOTE and Logistic Regression
pipeline = make_pipeline(SMOTE(random_state=42),␣
↪LogisticRegression(max_iter=1000, random_state=42))
```

[76]:
```
# Step 2: Perform Stratified K-Fold Cross-Validation
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
```

[77]:
```
# Step 3: Evaluate with Cross-Validation ('precision' parameter to reduce false␣
↪positives)
cv_scores = cross_val_score(pipeline, X, y, cv=cv, scoring='precision')
```

[78]:
```
# Step 4: Print the cross-validation results
print(f"Cross-Validation Precision Scores: {cv_scores}")
print(f"Mean Precision from Cross-Validation: {cv_scores.mean():.2f}")
```

```
Cross-Validation Precision Scores: [0.95663458 0.95543478 0.9560543  0.95637528
 0.95523605]
Mean Precision from Cross-Validation: 0.96
```

```
[ ]: """# Optional: visualize Confusion Matrix instead of solely printing it
def plot_confusion_matrix(y_true, y_pred):
    cm = confusion_matrix(y_true, y_pred)
    sns.heatmap(cm, annot=True, fmt='g', cmap='Blues', xticklabels=['No␣
 ↪Default', 'Default'], yticklabels=['No Default', 'Default'])
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.title('Confusion Matrix')
    plt.show()

plot_confusion_matrix(y_test, y_pred_balanced)  # Assuming y_test and␣
 ↪y_pred_balanced are your test labels and predictions"""
```

96% accuracy is much more realistic than 100%. We now aim to reduce the 2,127 false positives in our model, as denying 2,127 customers a loan that would most likely not default on it is a huge issue in the finance industry.

```
[79]: # Get predicted probabilities for the positive class (default)
y_pred_proba = model_balanced.predict_proba(X_test)[:, 1] # Get probabilities␣
 ↪for class 1 (default)
```

```
[81]: # Define a list of thresholds to evaluate
thresholds = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]

for threshold in thresholds:
    # Apply the threshold to convert probabilities to binary outcomes
    y_pred_adjusted = np.where(y_pred_proba >= threshold, 1, 0)

    # Calculate confusion matrix
    conf_matrix_adjusted = confusion_matrix(y_test, y_pred_adjusted)

    # Calculate precision and recall
    precision_adjusted = precision_score(y_test, y_pred_adjusted)
    recall_adjusted = recall_score(y_test, y_pred_adjusted)

    # Print the results for this threshold
    print(f"\nThreshold: {threshold}")
    print(f"Confusion Matrix:\n{conf_matrix_adjusted}")
    print(f"Precision: {precision_adjusted:.2f}")
    print(f"Recall: {recall_adjusted:.2f}")
    print(f"False Positives: {conf_matrix_adjusted[0, 1]}")
```

```
Threshold: 0.1
Confusion Matrix:
```

```
[[350859   3351]
 [     0  45790]]
Precision: 0.93
Recall: 1.00
False Positives: 3351

Threshold: 0.2
Confusion Matrix:
[[351314   2896]
 [     0  45790]]
Precision: 0.94
Recall: 1.00
False Positives: 2896

Threshold: 0.3
Confusion Matrix:
[[351617   2593]
 [     0  45790]]
Precision: 0.95
Recall: 1.00
False Positives: 2593

Threshold: 0.4
Confusion Matrix:
[[351855   2355]
 [     0  45790]]
Precision: 0.95
Recall: 1.00
False Positives: 2355

Threshold: 0.5
Confusion Matrix:
[[352083   2127]
 [     0  45790]]
Precision: 0.96
Recall: 1.00
False Positives: 2127

Threshold: 0.6
Confusion Matrix:
[[352306   1904]
 [     0  45790]]
Precision: 0.96
Recall: 1.00
False Positives: 1904

Threshold: 0.7
Confusion Matrix:
```

```
[[352545   1665]
 [     1  45789]]
Precision: 0.96
Recall: 1.00
False Positives: 1665

Threshold: 0.8
Confusion Matrix:
[[352810   1400]
 [     3  45787]]
Precision: 0.97
Recall: 1.00
False Positives: 1400

Threshold: 0.9
Confusion Matrix:
[[353196   1014]
 [    67  45723]]
Precision: 0.98
Recall: 1.00
False Positives: 1014
```

Precision, Recall, and Accuracy are 3 different metrics to show our model's success.

Confusion Matrix Numbers Mean:

Top Left - True Positives (what the model predicted 'true' that was correct)

Top Right - False Positives (in this context, rejecting loans to customers but they would not have defaulted)

Bottom Left - False Negatives (approving loans to customers that will most likely reject them)

Bottom Right - True Negatives (what the model predicted 'false' that was correct)

In the next few cells, I tune the threshold amount (which is 50% or .5 by default) to lower the number of false positives and false negatives, which in turn increases the true positives and true negatives.

[82]:
```
thresholds = [0.60, 0.65, 0.70, 0.75]

for threshold in thresholds:
    # Apply the threshold to convert probabilities to binary outcomes
    y_pred_adjusted = np.where(y_pred_proba >= threshold, 1, 0)

    # Calculate confusion matrix
    conf_matrix_adjusted = confusion_matrix(y_test, y_pred_adjusted)
```

```
    # Calculate precision and recall
    precision_adjusted = precision_score(y_test, y_pred_adjusted)
    recall_adjusted = recall_score(y_test, y_pred_adjusted)

    # Print the results for this threshold
    print(f"\nThreshold: {threshold}")
    print(f"Confusion Matrix:\n{conf_matrix_adjusted}")
    print(f"Precision: {precision_adjusted:.2f}")
    print(f"Recall: {recall_adjusted:.2f}")
    print(f"False Positives: {conf_matrix_adjusted[0, 1]}")
```

```
Threshold: 0.6
Confusion Matrix:
[[352306   1904]
 [     0  45790]]
Precision: 0.96
Recall: 1.00
False Positives: 1904

Threshold: 0.65
Confusion Matrix:
[[352419   1791]
 [     0  45790]]
Precision: 0.96
Recall: 1.00
False Positives: 1791

Threshold: 0.7
Confusion Matrix:
[[352545   1665]
 [     1  45789]]
Precision: 0.96
Recall: 1.00
False Positives: 1665

Threshold: 0.75
Confusion Matrix:
[[352672   1538]
 [     1  45789]]
Precision: 0.97
Recall: 1.00
False Positives: 1538
```

[83]:
```
# round 2 of threshold tuning
thresholds = [0.65, 0.66, 0.67, 0.68, 0.69, 0.70, 0.71]
```

```python
for threshold in thresholds:
    # Apply the threshold to convert probabilities to binary outcomes
    y_pred_adjusted = np.where(y_pred_proba >= threshold, 1, 0)

    # Calculate confusion matrix
    conf_matrix_adjusted = confusion_matrix(y_test, y_pred_adjusted)

    # Calculate precision and recall
    precision_adjusted = precision_score(y_test, y_pred_adjusted)
    recall_adjusted = recall_score(y_test, y_pred_adjusted)

    # Print the results for this threshold
    print(f"\nThreshold: {threshold}")
    print(f"Confusion Matrix:\n{conf_matrix_adjusted}")
    print(f"Precision: {precision_adjusted:.2f}")
    print(f"Recall: {recall_adjusted:.2f}")
    print(f"False Positives: {conf_matrix_adjusted[0, 1]}")
```

```
Threshold: 0.65
Confusion Matrix:
[[352419   1791]
 [     0  45790]]
Precision: 0.96
Recall: 1.00
False Positives: 1791

Threshold: 0.66
Confusion Matrix:
[[352444   1766]
 [     0  45790]]
Precision: 0.96
Recall: 1.00
False Positives: 1766

Threshold: 0.67
Confusion Matrix:
[[352469   1741]
 [     0  45790]]
Precision: 0.96
Recall: 1.00
False Positives: 1741

Threshold: 0.68
Confusion Matrix:
[[352501   1709]
 [     0  45790]]
```

```
Precision: 0.96
Recall: 1.00
False Positives: 1709

Threshold: 0.69
Confusion Matrix:
[[352519   1691]
 [     0  45790]]
Precision: 0.96
Recall: 1.00
False Positives: 1691

Threshold: 0.7
Confusion Matrix:
[[352545   1665]
 [     1  45789]]
Precision: 0.96
Recall: 1.00
False Positives: 1665

Threshold: 0.71
Confusion Matrix:
[[352568   1642]
 [     1  45789]]
Precision: 0.97
Recall: 1.00
False Positives: 1642
```

[116]:

```python
# round 3 of threshold tuning
thresholds = [0.691, 0.692, 0.693, 0.694, 0.695, 0.696, 0.697, 0.698, 0.699]

for threshold in thresholds:
    # Apply the threshold to convert probabilities to binary outcomes
    y_pred_adjusted = np.where(y_pred_proba >= threshold, 1, 0)

    # Calculate confusion matrix
    conf_matrix_adjusted = confusion_matrix(y_test, y_pred_adjusted)

    # Calculate precision, recall, and F1-score
    precision_adjusted = precision_score(y_test, y_pred_adjusted)
    recall_adjusted = recall_score(y_test, y_pred_adjusted)
    f1_adjusted = f1_score(y_test, y_pred_adjusted)

    # Print the results for this threshold
    print(f"\nThreshold: {threshold}")
    print(f"Confusion Matrix:\n{conf_matrix_adjusted}")
    print(f"Precision: {precision_adjusted:.2f}")
```

```python
    print(f"Recall: {recall_adjusted:.2f}")
    print(f"F1-Score: {f1_adjusted:.2f}")
    print(f"False Positives: {conf_matrix_adjusted[0, 1]}")
```

```
Threshold: 0.691
Confusion Matrix:
[[352521   1689]
 [     0  45790]]
Precision: 0.96
Recall: 1.00
F1-Score: 0.98
False Positives: 1689

Threshold: 0.692
Confusion Matrix:
[[352524   1686]
 [     0  45790]]
Precision: 0.96
Recall: 1.00
F1-Score: 0.98
False Positives: 1686

Threshold: 0.693
Confusion Matrix:
[[352527   1683]
 [     0  45790]]
Precision: 0.96
Recall: 1.00
F1-Score: 0.98
False Positives: 1683

Threshold: 0.694
Confusion Matrix:
[[352529   1681]
 [     0  45790]]
Precision: 0.96
Recall: 1.00
F1-Score: 0.98
False Positives: 1681

Threshold: 0.695
Confusion Matrix:
[[352530   1680]
 [     1  45789]]
Precision: 0.96
Recall: 1.00
F1-Score: 0.98
```

```
False Positives: 1680

Threshold: 0.696
Confusion Matrix:
[[352533   1677]
 [     1  45789]]
Precision: 0.96
Recall: 1.00
F1-Score: 0.98
False Positives: 1677

Threshold: 0.697
Confusion Matrix:
[[352534   1676]
 [     1  45789]]
Precision: 0.96
Recall: 1.00
F1-Score: 0.98
False Positives: 1676

Threshold: 0.698
Confusion Matrix:
[[352541   1669]
 [     1  45789]]
Precision: 0.96
Recall: 1.00
F1-Score: 0.98
False Positives: 1669

Threshold: 0.699
Confusion Matrix:
[[352543   1667]
 [     1  45789]]
Precision: 0.96
Recall: 1.00
F1-Score: 0.98
False Positives: 1667
```

We found some winners! Machine learning focuses on trade-offs, so domain knowledge and business context are important for figuring out which numbers are most important to you. For this model, I have made the executive decision that the false negatives are our top priority, as a financial institution giving money to a customer that ends up defaulting on their loan is very bad for business.

Since tuning the threshold improved the model's results, we now apply the threshold to our model and pipeline for future deployment and iterations.

[117]:
```python
# Apply the threshold of 0.694 to convert probabilities to binary outcomes
threshold = 0.694
y_pred_final = np.where(y_pred_proba >= threshold, 1, 0)  # If probability >= 0.
↪694, predict default (1)
```

[118]:
```python
# Evaluate the model's performance with the new threshold
accuracy_final = accuracy_score(y_test, y_pred_final)
precision_final = precision_score(y_test, y_pred_final)
recall_final = recall_score(y_test, y_pred_final)
f1_final = f1_score(y_test, y_pred_final)
conf_matrix_final = confusion_matrix(y_test, y_pred_final)

# Print the results for the 0.694 threshold
print(f"Model Accuracy with {threshold} threshold: {accuracy_final:.2f}")
print(f"Model Precision with {threshold} threshold: {precision_final:.2f}")
print(f"Model Recall with {threshold} threshold: {recall_final:.2f}")
print(f"Model F1-Score with {threshold} threshold: {f1_final:.2f}")
print(f"Confusion Matrix with {threshold} threshold:\n{conf_matrix_final}")
```

```
Model Accuracy with 0.694 threshold: 1.00
Model Precision with 0.694 threshold: 0.96
Model Recall with 0.694 threshold: 1.00
Model F1-Score with 0.694 threshold: 0.98
Confusion Matrix with 0.694 threshold:
[[352529   1681]
 [     0  45790]]
```

We now make a pipeline function for the threshold for future iterations and deployment of this model.

[119]:
```python
# Function to predict using a specific threshold
def predict_with_threshold(model, X, threshold=0.694):
    y_pred_proba = model.predict_proba(X)[:, 1]
    y_pred = np.where(y_pred_proba >= threshold, 1, 0)
    return y_pred
```

[120]:
```python
# Example usage on test data
y_pred_final = predict_with_threshold(model_balanced, X_test, threshold=0.694)
```

[121]:
```python
# Evaluate the final model predictions
accuracy_final = accuracy_score(y_test, y_pred_final)
precision_final = precision_score(y_test, y_pred_final)
recall_final = recall_score(y_test, y_pred_final)
f1_final = f1_score(y_test, y_pred_final)
conf_matrix_final = confusion_matrix(y_test, y_pred_final)

# Print the final results on the test set
print(f"Test Set Accuracy: {accuracy_final:.2f}")
print(f"Test Set Precision: {precision_final:.2f}")
```

```python
print(f"Test Set Recall: {recall_final:.2f}")
print(f"Test Set F1-Score: {f1_final:.2f}")
print("Test Set Confusion Matrix:\n", conf_matrix_final)
```

```
Test Set Accuracy: 1.00
Test Set Precision: 0.96
Test Set Recall: 1.00
Test Set F1-Score: 0.98
Test Set Confusion Matrix:
 [[352529   1681]
 [     0  45790]]
```

We now do more tuning to make sure the logistic regression is not overfitting.

[125]:
```python
# Set the best threshold you identified
best_threshold = 0.694
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
recall_scores = []

# Start stratified k-fold cross-validation
for train_idx, val_idx in skf.split(X_train_balanced, y_train_balanced):
    # Train on the training indices
    model_balanced.fit(X_train_balanced.iloc[train_idx], y_train_balanced.
 ↪iloc[train_idx])

    # Predict probabilities on the validation indices
    y_pred_cv = model_balanced.predict_proba(X_train_balanced.iloc[val_idx])[:,␣
 ↪1]

    # Apply the fixed threshold to convert probabilities to binary predictions
    y_pred_cv_adjusted = np.where(y_pred_cv >= best_threshold, 1, 0)

    # Calculate recall for this fold
    recall = recall_score(y_train_balanced.iloc[val_idx], y_pred_cv_adjusted)
    recall_scores.append(recall)

# Calculate the average recall across all folds
avg_recall = np.mean(recall_scores)

# Output the results
print(f"Threshold: {best_threshold}")
print(f"Average Recall across all folds: {avg_recall:.2f}")
```

```
Threshold: 0.694
Average Recall across all folds: 1.00
```

Cross-validation to ensure test accuracy

[129]:
```python
# cross-validation
from sklearn.model_selection import cross_val_predict

# Perform cross-validation and predict with a threshold of our choosing
y_pred_cv = cross_val_predict(model_balanced, X_train_balanced,␣
 ↪y_train_balanced, cv=5, method='predict_proba')[:, 1]

# Apply the threshold to make final predictions for cross-validation
y_pred_cv_final = np.where(y_pred_cv >= 0.63, 1, 0)

# >= 0.64: 7735, 1
# >= 0.63: 7829, 0

# Evaluate the cross-validated predictions
accuracy_cv = accuracy_score(y_train_balanced, y_pred_cv_final)
precision_cv = precision_score(y_train_balanced, y_pred_cv_final)
recall_cv = recall_score(y_train_balanced, y_pred_cv_final)
conf_matrix_cv = confusion_matrix(y_train_balanced, y_pred_cv_final)

# Print the cross-validation results
print(f"Cross-Validation Accuracy: {accuracy_cv:.2f}")
print(f"Cross-Validation Precision: {precision_cv:.2f}")
print(f"Cross-Validation Recall: {recall_cv:.2f}")
print()
print("Cross-Validation Confusion Matrix:")
print(conf_matrix_cv)
```

```
Cross-Validation Accuracy: 1.00
Cross-Validation Precision: 0.99
Cross-Validation Recall: 1.00

Cross-Validation Confusion Matrix:
[[1409419    7829]
 [      0 1417248]]
```

1.1.6 The confusion matrix above means the following:

1,409,419 true positives

7,829 false positives (denied loans to customers but they would have not defaulted)

0 false negatives (our most imporant metric in this model: 0 loans given to customers that would end up defaulting)

1,417,248 true negatives

Test model on test set (20% of data) for validation

```
[131]: # Assuming X_test and y_test are your original test set (20% split from the
       ↪beginning)
       y_pred_test = predict_with_threshold(model_balanced, X_test, threshold=0.63)

       # Evaluate the model on the test set
       accuracy_test = accuracy_score(y_test, y_pred_test)
       precision_test = precision_score(y_test, y_pred_test)
       recall_test = recall_score(y_test, y_pred_test)
       conf_matrix_test = confusion_matrix(y_test, y_pred_test)

       # Print the performance on the test set
       print(f"Test Set Accuracy: {accuracy_test:.2f}")
       print(f"Test Set Precision: {precision_test:.2f}")
       print(f"Test Set Recall: {recall_test:.2f}")
       print()
       print("Test Set Confusion Matrix:")
       print(conf_matrix_test)
```

```
Test Set Accuracy: 1.00
Test Set Precision: 0.96
Test Set Recall: 1.00

Test Set Confusion Matrix:
[[352244   1966]
 [     0  45790]]
```

The confusion matrix on the test shows correlated results to the confusion matrix on our training set, confirming our model's accuracy.

Which Features were most Important in our Model?

```
[132]: coefficients = model_balanced.coef_[0]  # LogisticRegression returns an array;
       ↪we get the first (and only) set of coefficients

       # Assuming X_train is a pandas DataFrame
       feature_importance = pd.DataFrame({
           'Feature': X_train_balanced.columns,
           'Coefficient': coefficients
       })

       # Sort by absolute value of coefficients to get feature importance ranking
       feature_importance['Importance'] = feature_importance['Coefficient'].abs()
       feature_importance.sort_values(by='Importance', ascending=False, inplace=True)

       print(feature_importance)
```

```
                               Feature  Coefficient  Importance
10                      default_amount  -139.464461  139.464461
17                credit_to_income_ratio    -9.335405    9.335405
```

7	debt_to_income_ratio	-9.335405	9.335405
22	log_debt_to_income_ratio	2.174407	2.174407
6	credit_score	1.193774	1.193774
20	credit_score_income_interaction	-0.491873	0.491873
15	log_annual_income	0.338369	0.338369
35	credit_score_bucket_Poor	0.324406	0.324406
33	credit_score_bucket_Fair	0.208909	0.208909
31	loan_purpose_Home	-0.130314	0.130314
32	loan_purpose_Personal	-0.128159	0.128159
30	loan_purpose_Education	-0.121051	0.121051
12	income_loan_interaction	0.117047	0.117047
19	income_loan_amount_interaction	0.117047	0.117047
29	loan_purpose_Car	-0.108713	0.108713
25	marital_status_Widowed	-0.104644	0.104644
34	credit_score_bucket_Good	0.093654	0.093654
28	employment_status_Unemployed	-0.091175	0.091175
27	employment_status_Self-employed	-0.089296	0.089296
23	marital_status_Married	-0.087125	0.087125
36	credit_score_bucket_Very Good	0.071916	0.071916
21	loan_amount_interest_rate_interaction	0.070027	0.070027
24	marital_status_Single	-0.053838	0.053838
26	employment_status_Retired	-0.050055	0.050055
2	loan_amount	-0.049465	0.049465
13	credit_interest_interaction	-0.045968	0.045968
16	credit_to_loan_ratio	-0.041759	0.041759
14	log_loan_amount	0.030628	0.030628
1	annual_income	-0.023061	0.023061
11	repayment_tenure	0.020317	0.020317
3	loan_term	-0.016965	0.016965
18	age_income_interaction	0.009270	0.009270
4	interest_rate	-0.007829	0.007829
9	credit_history_length	-0.005500	0.005500
0	age	-0.004945	0.004945
8	delinquencies	0.004786	0.004786
5	loan_to_value_ratio	0.000783	0.000783

As expected, the amount left on the defaulted loan was our most significant feature, while the ratios we calculated with credit, debt, and logarithms were close behind.

Now that the model is trained, tested, and validated, it can be saved and deployed in a production environment in the future.

I recommend saving the model, feature engineering transformations, and any other operations for machine learning as .pkl or .json files so they can be replicated when scaling a model on more or new data as needed.

[133]:
```
import joblib
```

```
# Save the trained model
joblib.dump(model_balanced, 'D:
 ↪\\datasets\\github_credit_risk_modeling_data\\logistic_regression_model.pkl')

# To load the model later
# model_loaded = joblib.load('logistic_regression_model.pkl')
```

[133]: ['D:\\datasets\\github_credit_risk_modeling_data\\logistic_regression_model.pkl'
]

[134]:
```
# Save the SMOTE object if necessary
joblib.dump(smote, 'D:\\datasets\\github_credit_risk_modeling_data\\smote.pkl')

# Save any scalers or encoders (if used)
# joblib.dump(scaler, 'D:\\datasets\\github_credit_risk_modeling_data\\scaler.
 ↪pkl')  # Example for a scaler
```

[134]: ['D:\\datasets\\github_credit_risk_modeling_data\\smote.pkl']

We have now created a logistic regression model with 97% accuracy on two million rows of data that can be scaled to twenty millions rows of data or more that tells financial institutions if a customer or potential customer will default on a loan. Scoring zero false negatives means our model does not give a single loan to any customer that will end up defaulting during that loan's repayment tenure.

This full project with extended explanation is available on my portfolio at http://github.com/nervousblakedown. For employment inquiries, please send an email to blakecalhoun@tuta.io. Thank you for reading.