# logistic_regression

December 18, 2025

## 1 Model Fitting

Now that we have processed the data, we can use the cleaned data to model and predict the chances of someone being approved for a loan.

Based on the EDA, we find that the factor that is most strongly correlated to the loan status would be the credit score of an individual. We also find that credit cards have the highest loan status approval rate among loan types. Among loan use cases, we observe that education and personal use cases generally have higher loan approval ratings, while debt consolidation and business reasons tend to decrease loan approval ratings.

Since the outcome we are trying to model is binary, $y_i \sim \text{Bern}(p_i)$, then a model that is well-suited to fit the data would be the logistic regression, as it is catered to fit data whose output is binary in nature. The output of a logistic regression is a probability value $\hat{\pi}_i$, which is not binary in nature, so to convert it to a binary value, we will implement a cutoff threshold, $k$, we then define $\hat{y}_i = I\{\hat{\pi}_i \geq k\}$, where $I\{x\}$ is the indicator function. To prevent overfitting, we want to use a train/test split and we will quantify the performance of our model by using the classification rate on the test set, where we define the classification rate as $\frac{I\{y_i = \hat{y}_i\}}{n}$. Since the cutoff value is arbitrarily defined, we can implement a grid search approach to find an optimal cutoff threshold that will maximize our classification rate.

## 2 Data Processing and Exploration

[1]:
```
import pandas as pd
import numpy as np
from sklearn.linear_model import LogisticRegression
from loan_tools.fittools import get_classification_rate, best_alpha
```

[2]:
```
df = pd.read_csv("data/cleaned_data.csv")
df.head(5)
```

[2]:
```
   age occupation_status  years_employed  annual_income  credit_score  \
0   40          Employed            17.2          25579           692
1   33          Employed             7.3          43087           627
2   42           Student             1.1          20840           689
3   53           Student             0.5          29147           692
4   32          Employed            12.5          63657           630
```

```
   credit_history_years  savings_assets  current_debt  defaults_on_file  \
0                   5.3             895         10820                 0
1                   3.5             169         16550                 0
2                   8.4              17          7852                 0
3                   9.8            1480         11603                 0
4                   7.2             209         12424                 0

   delinquencies_last_2yrs  derogatory_marks   product_type  \
0                        0                 0    Credit Card
1                        1                 0  Personal Loan
2                        0                 0    Credit Card
3                        1                 0    Credit Card
4                        0                 0  Personal Loan

          loan_intent  loan_amount  interest_rate  debt_to_income_ratio  \
0            Business          600          17.02                 0.423
1    Home Improvement        53300          14.10                 0.384
2  Debt Consolidation         2100          18.33                 0.377
3            Business         2900          18.74                 0.398
4           Education        99600          13.92                 0.195

   loan_to_income_ratio  payment_to_income_ratio  loan_status
0                 0.023                    0.008            1
1                 1.237                    0.412            0
2                 0.101                    0.034            1
3                 0.099                    0.033            1
4                 1.565                    0.522            1
```

[3]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50000 entries, 0 to 49999
Data columns (total 19 columns):
 #   Column                   Non-Null Count  Dtype
---  ------                   --------------  -----
 0   age                      50000 non-null  int64
 1   occupation_status        50000 non-null  object
 2   years_employed           50000 non-null  float64
 3   annual_income            50000 non-null  int64
 4   credit_score             50000 non-null  int64
 5   credit_history_years     50000 non-null  float64
 6   savings_assets           50000 non-null  int64
 7   current_debt             50000 non-null  int64
 8   defaults_on_file         50000 non-null  int64
 9   delinquencies_last_2yrs  50000 non-null  int64
 10  derogatory_marks         50000 non-null  int64
 11  product_type             50000 non-null  object
```

```
 12  loan_intent              50000 non-null  object
 13  loan_amount              50000 non-null  int64
 14  interest_rate            50000 non-null  float64
 15  debt_to_income_ratio     50000 non-null  float64
 16  loan_to_income_ratio     50000 non-null  float64
 17  payment_to_income_ratio  50000 non-null  float64
 18  loan_status              50000 non-null  int64
dtypes: float64(6), int64(10), object(3)
memory usage: 7.2+ MB
```

### 2.1 One Hot Encoding and Train/Test Splitting

[4]:
```
df = pd.get_dummies(df, columns = ['loan_intent', 'product_type',␣
↪'occupation_status'], drop_first = True)
```

[5]:
```
df.columns
```

[5]:
```
Index(['age', 'years_employed', 'annual_income', 'credit_score',
       'credit_history_years', 'savings_assets', 'current_debt',
       'defaults_on_file', 'delinquencies_last_2yrs', 'derogatory_marks',
       'loan_amount', 'interest_rate', 'debt_to_income_ratio',
       'loan_to_income_ratio', 'payment_to_income_ratio', 'loan_status',
       'loan_intent_Debt Consolidation', 'loan_intent_Education',
       'loan_intent_Home Improvement', 'loan_intent_Medical',
       'loan_intent_Personal', 'product_type_Line of Credit',
       'product_type_Personal Loan', 'occupation_status_Self-Employed',
       'occupation_status_Student'],
      dtype='object')
```

[6]:
```
train, test = df[:-10000], df[-10000:]
```

### 2.2 Model Fitting

Using the information from the preface, we can naively fit a model using the loan intent, product type, and credit score to see how our model performs.

[7]:
```
y_train = train['loan_status']
X_cols_naive = ['loan_intent_Debt Consolidation', 'loan_intent_Education',
                'loan_intent_Home Improvement', 'loan_intent_Medical',
                'loan_intent_Personal', 'product_type_Line of Credit',
                'product_type_Personal Loan', 'credit_score']
X_train_naive = train[X_cols_naive].astype(float)

model_naive = LogisticRegression(C=1e10, solver='lbfgs', max_iter=1000,␣
↪random_state=42)
model_naive.fit(X_train_naive, y_train)
```

```
classification_rate = get_classification_rate(model = model_naive, X_cols = X_cols_naive, df = test)
print(f"Using our naive model, we achieve a classification performance of {classification_rate[0]}%")
```

```
Using our naive model, we achieve a classification performance of 76.22%
```

We find that our naive approach produces a model that is correct about 76% of the time, however, we can definitely make a model that performs better.

### 2.3 Regularized Model Fitting

Regularization is a common technique used to impose a penalty, $\alpha$, on our regressor coefficients, $\beta$, this penalty will help find regressors that are stronger for prediction. As $\alpha \to \infty$, our $\beta \to 0$, helping reduce our model complexity and finding the strongest regressors. One issue that would occur is how do we find the optimal $\alpha$ such that we get the highest classification rate? To solve this, we can simply do a grid search to find the model that maximizes our classification rate. Since statsmodels only supports Tikhonov regularization, we will only test Tikhonov regularization.

[8]:
```
best_model_l1, best_classification_l1, best_alpha_l1 = best_alpha(np.logspace(0, 5, num=6), train, test, lp = 'l1')
print("At an regularization penalty of " + str(best_alpha_l1) + ", we achieve the best classification performance with a performance of " + str(best_classification_l1[0]) + "%")
```

```
At an regularization penalty of 1.0, we achieve the best classification
performance with a performance of 86.78%
```

By implementing Tikhonov Regularization with a penalty of $\alpha = 1$, we managed to improve our classification performance from 76% to 86%, which is a significant improvement. We can then do a finer grid search to further optimize our model.

[9]:
```
best_model_l1_fine, best_classification_l1_fine, best_alpha_l1_fine = best_alpha(np.linspace(0.01, 1, num = 100), train, test, lp = 'l1')
print("At an regularization penalty of " + str(best_alpha_l1_fine) + ", we achieve the best classification performance with a performance of " + str(best_classification_l1_fine[0]) + "%")
```

```
At an regularization penalty of 0.01, we achieve the best classification
performance with a performance of 86.78%
```

Based on our finer grid search, we find that the optimal $\alpha = 0.01$ does not improve our model performance.

[ ]:
```

```