Credit Limit Prediction Using Machine Learning Techniques

Mohammad Hassan Heydari

University of Isfahan, Iran

Introduction

In the ever-evolving landscape of financial technology, the ability to accurately predict a user's credit limit has become a cornerstone for personalized banking services. This project, titled **Credit Limit Prediction**, aims to harness the power of machine learning to forecast credit limits with enhanced precision. By meticulously gathering and preprocessing a robust set of user features, we have laid the groundwork for a predictive model that encapsulates the intricacies of financial behavior.

At the heart of our exploration lies the deployment of two distinct yet complementary machine learning models: **Polynomial Regression** and **Random Forest**. These models were meticulously selected to capture the non-linear patterns and decision boundaries inherent in credit limit prediction. A rigorous evaluation process was conducted, wherein the impact of hyperparameter tuning was observed against key performance metrics, notably the **loss function** and the **R^2 score**. This not only provided insights into model sensitivity but also guided us towards optimal model complexity.

Furthermore, the project delved into the realm of computational performance by contrasting model training on **Graphics Processing Units (GPUs)** versus **Central Processing Units (CPUs)**. This comparison not only highlighted the computational advantages but also shed light on the practical aspects of deploying machine learning models in real-world scenarios.

This document presents a comprehensive account of our methodology, findings, and the implications of our work in the broader context of credit limit prediction. Through this endeavor, we aim to contribute to the financial sector's predictive analytics capabilities and pave the way for more informed credit decisions.

Methodology

The methodology of our project, Credit Limit Prediction, is rooted in the application of advanced machine learning techniques and computational frameworks to predict credit limits with high accuracy. Our approach is characterized by the following key steps:

1. Polynomial Regression Implementation:
    - We leveraged the **JAX** framework for its Just-In-Time (JIT) compilation capabilities, which allowed us to efficiently train our model on a **GPU**.
    - The dataset was transformed into polynomial features using **Scikit-Learn**, enhancing the model's ability to capture complex, non-linear relationships.

2. Functional Programming:
    - To manage complexity and enhance code maintainability, we adopted functional programming paradigms throughout our implementation.
3. Performance Comparison:
    - A comparative analysis revealed that training the model for 100 epochs on a **CPU** averaged **2.5 seconds**, while the same process on a **GPU** was reduced to **1.5 seconds** ; Indicating that even on this small dataset, GPU acceleration can be **50% faster** that CPU training
    - Our dataset comprised 9,000 examples with 1,140 polynomial features derived from the original 17 features.
4. Stochastic Gradient Descent with Momentum
    - For better optimization, we used Stochastic Gradient Descent rather than simple gradient descent . gradient descent with momentum can help the model to escape the local minima and converge faster . also we compared the difference of using multiple momentums on the final results
5. Model Evaluation:
    - We observed a significant increase in the R^2 score when utilizing a degree 3 polynomial feature set compared to a degree 2 set, indicating a better fit for the predictive model.
6. Random Forest Model:
    - Subsequently, we applied the polynomial feature dataset to the Random Forest model implemented in scikit-learn.
    - Due to the lack of GPU support in scikit-learn, this phase of the project was conducted exclusively on a CPU.
7. Hyperparameter Tuning:
    - Through hyperparameter optimization, we determined that a degree 2 polynomial feature set with 200 N estimators significantly outperformed the polynomial regression model.
8. Comparative Analysis:
    - Further experimentation with a degree 3 polynomial feature set revealed negligible changes in performance, suggesting that a degree 2 polynomial was sufficient for our purposes.

The consistency of the dataset across both models ensured a fair comparison, with the initial dataset containing 9,000 training examples and 1,160 test examples, each with 17 features. The application of polynomial features expanded the feature space, providing a rich dataset for our predictive models.

After Comparing the results of these two models, we saw that **Random Forest** model can outperform Polynomial Regression on this task ; Indicating that Random forest can find complex patterns on this dataset better than simple polynomial regression

Implementation

Our Implementation includes three parts :

- Data Preprocessing
- Implementation of Polynomial Regression with JAX
- Implementation of Random Forest Regressor with SciKit Learn

Here we Summarize what we did in these parts :

1. Data Preprocessing : *data_preprocessing.py*

In this part, first we load our dataset *CreditPrediction.csv* and save it to a Pandas data frame , which allows us multiple data processing methods .
We first drop the first column *CLIENTNUM* which is irrelevant to the model features. Also we drop the last column *Unnamed: 19* which is entirely NaN values . Also we move our target column *Credit_Limit* to the last column of the data frame, which allows us to slice the dataset easier . Also we drop the rows with Age bigger Than 75 . because in the dataset , there were some Ages with value over than 200 !

```
import pandas as pd
from sklearn.utils import shuffle

df = pd.read_csv('Data/CreditPrediction.csv')

df = df.drop(['Unnamed: 19'], axis=1)
df = df.drop(['CLIENTNUM'], axis=1)

# Adding the target col as the last col of data frame
credit = df.pop('Credit_Limit')
df['Credit_Limit'] = credit

df = df[df['Customer_Age'] <= 75 ]
```

Then we turn our categorical features into categorical numbers . because of the existance of NaN and ‘Unknown’ Values , we replace these values with the Mean of that specific column . we find the this information using *Describe(), Info() and Mean()* methods .

```
df['Gender'].replace(['M', 'F'], [0, 1], inplace=True)
# Value 0 : Male
df['Gender'].fillna(value=0, inplace=True)

df['Education_Level'].replace(['Uneducated', 'High School',
'College','Graduate', 'Post-Graduate', 'Doctorate', 'Unknown'],
                              [0, 1, 2, 3, 4, 5, 3], inplace=True)

df['Marital_Status'].replace(['Single', 'Married', 'Divorced',
'Unknown'], [0, 1, 2, 1], inplace=True)
# value 1 : Married
df['Marital_Status'].fillna(value=1, inplace=True)

df['Income_Category'].replace(['Less than $40K', '$40K - $60K', '$60K -
$80K', '$80K - $120K', '$120K +', 'Unknown'],
                              [0, 1, 2, 3, 4, 2], inplace=True)

df['Card_Category'].replace(['Blue', 'Silver', 'Gold', 'Platinum'],
                            [0, 1, 2, 3], inplace=True)

# Value 0 : Blue
df['Card_Category'].fillna(value=0, inplace=True)
```

Now for Other numerical columns with NaN values, we replace these NaN values with the Mean of that column . This doesn’t effect other columns because their nan values are already replaced

```
for column in df.columns:
    # Calculate the mean, ignoring NaN values
    mean_value = df[column].mean()
    # Replace NaN values with the mean
    df[column].fillna(mean_value, inplace=True)
```

Then we shuffle the dataset and split it to train and test parts

```
df = shuffle(df)
train_df = df.iloc[:9000, :]
test_df = df.iloc[9000:, :]
```

Then we normalize the remaining columns with MinMax scaling approach

```
max_value = train_df['Customer_Age'].max()
train_df['Customer_Age'] = train_df['Customer_Age'] / max_value
test_df['Customer_Age'] = test_df['Customer_Age'] / max_value

max_value = train_df['Months_on_book'].max()
train_df['Months_on_book'] = train_df['Months_on_book'] / max_value
test_df['Months_on_book'] = test_df['Months_on_book'] / max_value



for column in train_df.columns[11:-1]:
    max_value = train_df[column].max()
    train_df[column] = train_df[column] / max_value
    test_df[column] = test_df[column] / max_value
```

As the last operation of this part, we save this data into *train.csv* and *test.csv* to feed the models

```
train_df.to_csv('Data/train.csv', index=False)
test_df.to_csv('Data/test.csv', index=False)
```

2. Polynomial Regression Model : *regression_model.py*

After the data processing part, we first implement the polynomial regression model with JAX library . JAX is a numerical computation library in python which uses a technique called XLA that can significantly increase the computation power . We will use its JIT compilation and GPU acceleration .

First we import necessary libraries for computation and polynomial feature mapping

```
import pandas as pd
import jax.numpy as jnp
import jax
from jax import grad, jit
from jax import random
from sklearn.preprocessing import PolynomialFeatures
from sklearn.metrics import mean_squared_error, mean_absolute_error,
r2_score
import numpy as np
import time
```

For loading the data and converting them to a useable format, we need to create tow functions to load and map the data into polynomial features

```
def get_data():
    dataset_train = pd.read_csv('Data/train.csv')
    dataset_train = dataset_train.to_numpy()
    y_train = dataset_train[:, -1]
    X_train = dataset_train[:, :-1]


    dataset_test = pd.read_csv('Data/test.csv')
    dataset_test = dataset_test.to_numpy()
    y_test = dataset_test[:, -1]
    X_test = dataset_test[:, :-1]

    return (X_train, y_train), (X_test, y_test)

def polynomial_features(X_train, X_test, degree):
    poly = PolynomialFeatures(degree)
    X_train_poly = poly.fit_transform(X_train)
    X_test_poly = poly.transform(X_test)
    return X_train_poly, X_test_poly
```

We load the data using these two function from above like this :

```
#Loading Data
(X_train, y_train), (X_test, y_test) = get_data()

degree = 3
X_train_poly, X_test_poly = polynomial_features(X_train, X_test,
degree=degree)
```

Now, we initialize the parameters and velocity for stochastic gradient descent . We set the parameters uniformly random and the velocity zeros, with shape of the parameters

```
# Initialize parameters
key = random.PRNGKey(0)
params = random.uniform(key, shape= (X_train_poly.shape[1],))
velocity = jnp.zeros_like(params)  # Initialize velocity for momentum
```

We create the model outputs and loss function as follows , the loss function is set to be Mean Squared Error for this regression task

```
# Model
@jit
def predict(params, X):
    return jnp.dot(X, params)




# Loss function (Mean Squared Error)
@jit
def mse_loss(params, X, y):
    preds = predict(params, X)
    return jnp.mean((preds - y) ** 2)
```

Our model parameters will be updated with the Stochastic gradient descent approach , with momentum . This method as discussed , can help the model to converge faster

```
  # SGD with Momentum update
@jit
def update(params, velocity, X, y, lr=0.01, momentum=0.9):
    grads = grad(mse_loss)(params, X, y)
    velocity = momentum * velocity - lr * grads
    params = params + velocity
    return params, velocity
```

The main function of the model, Train, is implemented as follows . We track the loss and training time every 100 epoch ; This would help us to better understand the model performance :

```
# Training loop
def train(params, velocity, X_train, y_train, X_test, y_test,
epochs=1000, lr=0.01, momentum=0.9):

    training_times = []
    start_epoch = time.time()
    for epoch in range(epochs):
        params, velocity = update(params, velocity, X_train, y_train,
lr, momentum)


        if epoch % 100 == 0:
            epoch_time = time.time() - start_epoch
            training_times.append(epoch_time)
            train_loss = mse_loss(params, X_train, y_train)
            test_loss = mse_loss(params, X_test, y_test)
            print(f'Epoch {epoch} | {str(epoch_time)[:5]} | Train Loss:
{train_loss} | Test Loss: {test_loss}')
            start_epoch = time.time()

    return params, training_times
```

We set the hyperparameters and train the model with this part of code . the model will rune the gradient descent algorithm on the dataset over 20000 epochs :

```
# Train the model
learning_rate = 0.00001
momentum = 0.99
epochs = 20000
params, training_times = train(params, velocity, X_train_poly, y_train,
X_test_poly, y_test, epochs=epochs, lr=learning_rate,
momentum=momentum)
```

We compute the final predictions and losses and R2 scores with this part of code :

```
# Predictions
y_train_pred = np.array(predict(params, X_train_poly))
y_test_pred = np.array(predict(params, X_test_poly))

# Compute metrics
train_mse = mean_squared_error(y_train, y_train_pred)
test_mse = mean_squared_error(y_test, y_test_pred)

train_mae = mean_absolute_error(y_train, y_train_pred)
test_mae = mean_absolute_error(y_test, y_test_pred)

train_r2 = r2_score(y_train, y_train_pred)
test_r2 = r2_score(y_test, y_test_pred)
```

Finally, we show the final results for evaluating the results and performance of the model with this part of python code :

```
# Print metrics and hyperparameteres
print(f'Poly Degree : {degree} - LR : {learning_rate} - Momentum :
{momentum} - Epochs : {epochs}')
print(f'Average training time per 100 epochs :
{sum(training_times)/len(training_times)}')
print(f'Training Device : {jax.devices()}')
print(f'Final Train MSE: {train_mse}')
print(f'Final Test MSE: {test_mse}')

print(f'Final Train MAE: {train_mae}')
print(f'Final Test MAE: {test_mae}')

print(f'Final Train R2: {train_r2}')
print(f'Final Test R2: {test_r2}')
```

As mentioned above , we want to determine the result of changing the hyperparameters on the final result , also we want to compare the model training time on CPU and GPU ; So we run this script multiple times with different setups . we will show these results in the ***Results*** section of this document

3. **Random Forest Regressor** : *random_forest_model.py*

The part of loading data in this file is same as the *regression_model.py ,* The very thing we have to add is to load the *RandomForestRegressor()* model from sk-learn

```
from sklearn.ensemble import RandomForestRegressor
```

We create the model and train it with these two simple lines of code , also we set the verbose to 1 to see the progress time and execution

```
n_estimators = 200
model = RandomForestRegressor(n_estimators= n_estimators,
random_state=42, verbose= 1)


model.fit(X_train_poly, y_train)
```

We compute the final predictions and losses and R2 scores with this part of code :

```
y_train_pred = model.predict(X_train_poly)
y_test_pred = model.predict(X_test_poly)


train_mse = mean_squared_error(y_train, y_train_pred)
test_mse = mean_squared_error(y_test, y_test_pred)

train_mae = mean_absolute_error(y_train, y_train_pred)
test_mae = mean_absolute_error(y_test, y_test_pred)

train_r2 = r2_score(y_train, y_train_pred)
test_r2 = r2_score(y_test, y_test_pred)
```

Finally, we show the final results for evaluating the results and performance of the model with this part of python code :

```
# Print metrics and hyperparameters
print(f'Poly Degree : {degree} - N Estimators : {n_estimators}')
print(f'Final Train MSE: {train_mse}')
print(f'Final Test MSE: {test_mse}')

print(f'Final Train MAE: {train_mae}')
print(f'Final Test MAE: {test_mae}')

print(f'Final Train R2: {train_r2}')
print(f'Final Test R2: {test_r2}')
```

As well as the evaluating the model performance on different hyperparameters in polynomial regression, we will show that the degree 3 polynomial won't change the scores of the model very effective, so the degree 2 is fine . we will show these results in the ***Results*** part

Results

In Overall, the results of Random Forest Regressor were better than the Polynomial Regression with momentum . we will discuss the details in this section .

The original dataset has 10150 examples with 17 features . after mapping them to degree 2 polynomial, we have 171 features and after using degree 3 polynomial , the number of features will become 1140 . we split out dataset into Train dataset with 9000 examples and Test dataset with 1150 examples (shuffled) . Our two model behave differently on each of these polynomial degrees .

Figure(1) shows that the final MSE on test set after 20000 epochs on degree 2 polynomial dataset is around ***22600000*** and its R2 Score is ***0.712*** .

Figure(1) :

```
Poly Degree : 2 - LR : 1e-05 - Momentum : 0.99 - Epochs : 20000
Average training time per 100 epochs : 0.25234646201133726
Training Device : [CpuDevice(id=0)]
Final Train MSE: 24908608.060052402
Final Test MSE: 22601086.283243
Final Train MAE: 3341.984395632596
Final Test MAE: 3200.039956038765
Final Train R2: 0.7000386944701595
Final Test R2: 0.7120588104610416
```

We map our dataset into 3 degree polynomial , which increases the R2 score and decreases MSE significantly . But we see that the training time has become 9.2X slower than the dataset with 2 degree polynomial . Figure(2) shows that the final MSE on test set after 20000 epochs on degree 3 polynomial dataset is around ***16180000*** and its R2 Score is ***0.793*** .

Figure(2) :

```
Poly Degree : 3 - LR : 1e-05 - Momentum : 0.99 - Epochs : 20000
Average training time per 100 epochs : 2.3121647596359254
Training Device : [CpuDevice(id=0)]
Final Train MSE: 17248600.272620548
Final Test MSE: 16187331.470618844
Final Train MAE: 2694.951260723877
Final Test MAE: 2687.986155551081
Final Train R2: 0.7922841515726697
Final Test R2: 0.7937709975220445
```

We saw that the CPU performance decreases when the size of the input dataset increases .we determine the performance of GPU on the degree 3 polynomial dataset, with shape of *(9000, 1140)* . Figure(3) shows that GPU training time is ***1.56X faster*** than CPU training time on the same size of data . The final results are the same as the CPU results, but much faster

Figure(3):

```
Poly Degree : 3 - LR : 1e-05 - Momentum : 0.99 - Epochs : 20000
Average training time per 100 epochs : 1.4853972482681275
Training Device : [cuda(id=0)]
Final Train MSE: 17248599.710889664
Final Test MSE: 16187337.213886414
Final Train MAE: 2694.950810111491
Final Test MAE: 2687.9865298276154
Final Train R2: 0.7922841583373001
Final Test R2: 0.7937709243519643
```

Figure(4) and Figure(5) showing that decreasing Momentum will cause the MSE increase and R2 Score decrease .

Figure(4) :

```
Poly Degree : 3 - LR : 1e-05 - Momentum : 0.9 - Epochs : 20000
Average training time per 100 epochs : 1.4754125475883484
Training Device : [cuda(id=0)]
Final Train MSE: 22883051.479332566
Final Test MSE: 21257202.127595726
Final Train MAE: 3159.4579722507056
Final Test MAE: 3077.537389223845
Final Train R2: 0.7244314102298053
Final Test R2: 0.7291800938157509
```

Figure(5) :

```
Poly Degree : 3 - LR : 1e-05 - Momentum : 0.8 - Epochs : 20000
Average training time per 100 epochs : 1.556397886276245
Training Device : [cuda(id=0)]
Final Train MSE: 24576697.06731072
Final Test MSE: 22695162.83539159
Final Train MAE: 3293.1820682013613
Final Test MAE: 3186.331758746603
Final Train R2: 0.7040357245114396
Final Test R2: 0.7108602612411543
```

Figure(6) indicates that Random Forest Regressor can have significantly better results that polynomial results over the degree 2 Polynomial dataset .

Also this shows that random forest regressor with only 2 degree polynomial can outperform degree 3 polynomial regression .

And Figure(7) shows that using degree 3 polynomial will not change the final results , it only increases the training time and complexity .

Figure(6) :

```
[Parallel(n_jobs=1)]: Done  49 tasks      | elapsed:   23.2s
[Parallel(n_jobs=1)]: Done 199 tasks      | elapsed:  1.6min
[Parallel(n_jobs=1)]: Done  49 tasks      | elapsed:    0.1s
[Parallel(n_jobs=1)]: Done 199 tasks      | elapsed:    0.2s
[Parallel(n_jobs=1)]: Done  49 tasks      | elapsed:    0.0s
[Parallel(n_jobs=1)]: Done 199 tasks      | elapsed:    0.0s
Poly Degree: 2 - N Estimators: 200
Final Train MSE: 1391723.8941939964
Final Test MSE: 8288295.94052456
Final Train MAE: 465.9663952777776
Final Test MAE: 1131.7600060869568
Final Train R2: 0.9832401989210703
Final Test R2: 0.8944058810954143
```

Figure(7) :

```
[Parallel(n_jobs=1)]: Done  49 tasks      | elapsed:  2.7min
[Parallel(n_jobs=1)]: Done 199 tasks      | elapsed: 11.5min
[Parallel(n_jobs=1)]: Done  49 tasks      | elapsed:    0.1s
[Parallel(n_jobs=1)]: Done 199 tasks      | elapsed:    0.5s
[Parallel(n_jobs=1)]: Done  49 tasks      | elapsed:    0.0s
[Parallel(n_jobs=1)]: Done 199 tasks      | elapsed:    0.1s
Poly Degree: 3 - N Estimators: 200
Final Train MSE: 1385150.250949435
Final Test MSE: 8137970.286336614
Final Train MAE: 474.9081348333332
Final Test MAE: 1147.106470869565
Final Train R2: 0.9833193618596404
Final Test R2: 0.8963210522134146
```

Refrences

- Ruder, Sebastian. "An overview of gradient descent optimization algorithms." *arXiv preprint arXiv:1609.04747* (2016).
- Louppe, Gilles. "Understanding random forests: From theory to practice." *arXiv preprint arXiv:1407.7502* (2014).
- Stuart Russell, Peter Norvig. “Artificial Intelligence : A Modern Approach” , fourth edition
- Christopher M. Bishop . “ Deep Learning, Foundations and Concepts” (2024).
- JAX: High Performance Array Computing