NYU

Predicting Probability of Credit Card Default

Jennifer Rodriguez-Trujillo
Joseph Schuman
Khevna Parikh
Kristin Mullaney
Rodrigo Kreis de Paula
Sarvesh Patki

05.03.2023

Problem Statement

Predict if customers will default on their credit cards

Background:

- Credit default prediction is central to managing risk in a consumer lending business and key for a healthy business environment
- A successful model creates a better customer experience for cardholders by making it easier to be approved for a credit card

Effectively analyze extensive & complex data quickly:

- Optimized code can quickly process the enormous quantity of credit card data to identify future defaults
- Credit Default prevention could save businesses and consumers a substantial amount of money, creating a more efficient business environment

Data & Model Overview:

Dataset

- Source: American Express Default Prediction (Kaggle)
- 5,531,451 records. Each is a credit card statement
- 190 variables, which track the customers' profile by observing an 18-month performance window
- Binary target variable is Default Status. Default = Y if the customer does not pay the credit card statement in 120 days

Model

- The objective is to implement a model that predicts the probability of a default
- We first implemented a simple yet computationally intensive Logistic Regression Model (baseline)
- We then implemented various techniques to improve performance

Optimization Techniques:

We improved the efficiency of our model and code by implementing the following:

- Line Profiling
- Minimizing repetitive for-loops
- Vectorization through NumPy
- Python Jax
- Parallel Processing

NumPy

Numba

Optimization Techniques:

Parquet files and Line Profiler

- Implemented line profiling to identify the time and resources usage of each line of our code
- Our training dataset (in CSV) is quite large, at 16.3 GB in size.
- As a result, reading in the file takes several minutes.
- To speed up the process, we first compressed the dataset into Parquet files, resulting in a smaller size of 3 GB.
- The baseline model took 6,368 seconds to run on the entire training dataset, achieving an accuracy of 0.87.

```
File: /var/folders/9c/9y0zmgk55297pv9nj1zpn02c0000gn/T/ipykernel_27817/1098087167.py
Function: train at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def train(X, y, bs, epochs, lr):
     2         1       2000.0   2000.0      0.0      m, n = X.shape
     3         1      11000.0  11000.0      0.0      w = np.zeros((n,1))
     4         1          0.0      0.0      0.0      b = 0
     5         1 2906651000.0 2906651000.0     24.9      x = normalize(X)
     6         1       1000.0   1000.0      0.0      losses = []
     7      1000     728000.0    728.0      0.0      for epoch in range(epochs):
     8    100000   38132000.0    381.3      0.3          for i in range((m-1)//bs + 1):
     9    100000   34305000.0    343.1      0.3              start_i = i*bs
    10    100000   36200000.0    362.0      0.3              end_i = start_i + bs
    11    100000   79885000.0    798.9      0.7              xb = x[start_i:end_i]
    12    100000   49688000.0    496.9      0.4              yb = y[start_i:end_i]
    13    100000 3413203000.0  34132.0     29.2              y_hat = sigmoid(np.dot(xb, w) + b)
    14    100000 3666929000.0  36669.3     31.4              dw, db = gradients(xb, yb, y_hat)
    15    100000  270609000.0   2706.1      2.3              w -= lr*dw
    16    100000   50683000.0    506.8      0.4              b -= lr*db
    17      1000 1136103000.0 1136103.0      9.7          l = loss(w, x, y)
    18      1000    2054000.0   2054.0      0.0          losses.append(l)
    19         1          0.0      0.0      0.0      return w, b, losses
```

Optimization Techniques:

Numpy & Vectorization

- Through line profiling, discovered that nested for loops were causing the majority of the processing time
- Minimized the use of for-loops by vectorizing techniques where possible.
- Utilized the NumPy library for performing efficient mathematical operations like np.dot
    - Simplified our code by taking advantage of other NumPy functions, like np.where
- As a result of these optimizations, we were able to reduce our code's processing time to 3,000 seconds (half the original time!).

Optimization Techniques:

Python Jax

- Jax provides the functionality of automatic differentiation.
  - For example, we used ‘grad’ function to compute the derivative of the loss function automatically
- We used Jax composable functions ‘jit’ and ‘vmap’ to vectorize functions where applicable
- Jax took 216 seconds to run our code with 100,000 records, whereas NumPy took 164 records.
- In conclusion, Jax is better for certain operations like gradient calculations, while NumPy is better for others.

Optimization Techniques:

Parallel Processing

- MPI (Messaging Passing Interface), a parallel programming implementation, enables communication between processes. With MPI, we were able to coordinate computation via shared data and messaging
- We used distributed computing to allocate calculations across multiple nodes and multiple cores
- We did not experience efficiency improvements on the subset of 100,000 records

Results

Full Dataset (5mm records)	Runtime (seconds)	Improvement
Baseline	10,368	N/A
NumPy	3,012	3.44x

Subset (100k records)	Runtime (seconds)	Improvement
Baseline	408.83	N/A
NumPy	103.86	3.94x
NumPy + Jax	183.63	2.22x
NumPy + MPI	456.2	0.89x

Thank You