CS 6375 Project 3
Collaborative Filtering, SVM, and KNN

Khoi Nguyen

Part 1: Collaborative Filtering

I implemented memory-based collaborative filtering for a fraction of the Netflix dataset involving 30,000 users and 2,000 movies. Among all possible user-movie pairs, the training set contains ratings for 6% of them (~3M ratings). The test set asks for ratings of 100K user-movie pairs.

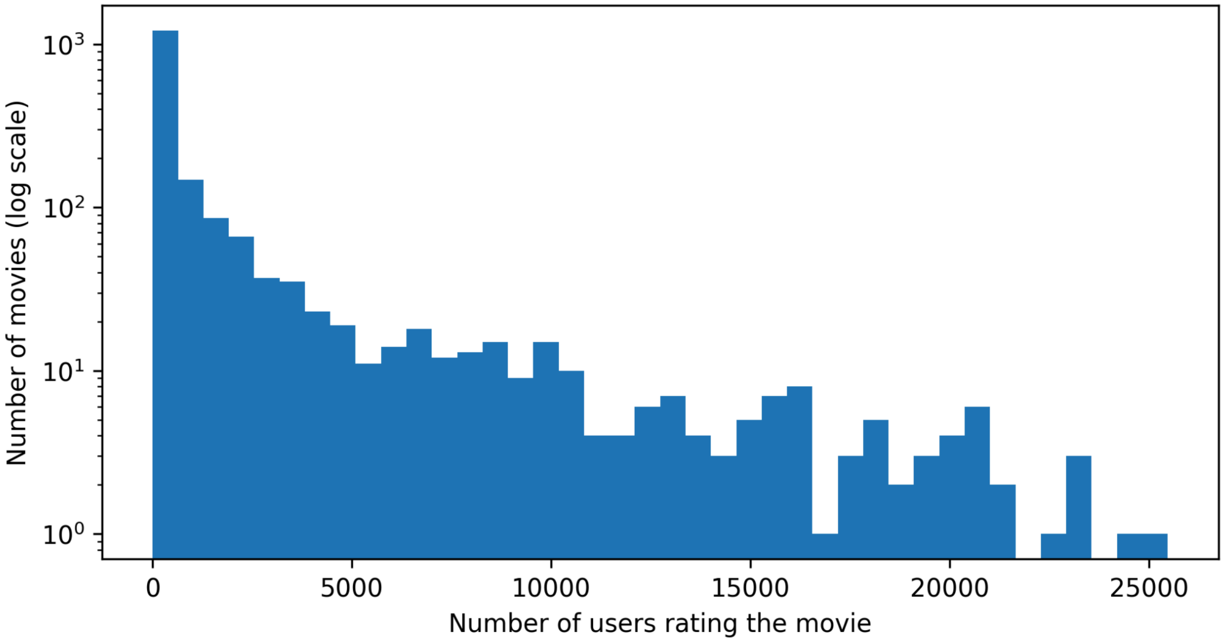

Figure 1: The histogram of movies by the number of users rating them. Looks like it is an exponential distribution with a long tail -- while most movies were rated by <5K users, some outstanding movies are rated by >20K users.

Method

Let `M=1821` and `U=28,978` be the number of movies and users (in the training set). The most non-trivial and expensive step in the program is to calculate matrix `W`, which stores all the pairwise weights between users[1]. The full algorithm for computing `W` is as follows:

- Read `3,255,352` ratings, store it in a matrix `V` of size `MxU=52,768,938`.

[1] The evaluation code actually ran for longer than the W computation. However, it is more trivial and can be easily understood by reading my code.

  - *To map ratings into matrix, I firsted compressed all the user IDs and movie IDs into a condensed list of integers `[1;U]` and `[1; M]`, respectively*
  - *Note that this matrix is sparse with 6% density. I relied on the internal efficiency of `numpy` to handle this big sparse matrix.*
- Let `X` be `V` subtracted by the average rating for each user. Make sure missing entries in `V` are still valued as zeros in `X`.
- Calculate the variance in ratings of each user: `S = np.sqrt(np.multiply(X, X).sum(axis=1, keepdims=True))`
  - *This matrix S has shape (U, 1), which will help calculate the denominator in Pearson correlation*
- Calculate `W`:
  - `denom = S @ S.T`
  - `denom[denom == 0] = 1` (to avoid division by zero later)
  - `W = X @ X.T / denom`

For the detailed implementation, please refer to the attached code.

Result

I got a **Root Mean Squared Error of 0.88** and a **Mean Absolute Error of 0.69**. For a quick comparison, a dummy baseline algorithm which constantly predicts 3 as the rating (the middle point of the 1-5 range) should achieve roughly 2.0 on both errors. Therefore, my algorithm is an obvious improvement over the baseline, showcasing the effectiveness of memory-based collaborative filtering.

The entire pipeline, from computing `W` to evaluation, took 95 seconds on a machine with AMD Ryzen Threadripper 3960X 24-Core Processor.[2]

Discussion

Earlier, I also tried to compute W using for loops “efficiently” by exploiting the sparsity of the matrix. For each movie, I looped through all pairs of users watching that movie and updated the statistics for the W of those two users. After 7.5 hours, it only went through 75% of all the movies[3]. Switching to matrix-based implementation, it took *less than a minute* to complete that step. To gain a deep understanding of this difference, I will analyze their time complexities.

For loop-based implementation, its complexity is $O\left(\sum_{i=0}^{M-1} u_i^2\right)$, where u_i is the number of users watching the i-th movie. Based on the training set, this number is 32,434,835,022. However, we are assuming the data structure for saving the W matrix is a static array of size

[2] On a Macbook Air M2 with 8 CPU cores, it took 270 seconds.
[3] At that point, I just halted the program. Programming is only fun when the program is fast.

$M \times U \approx 53 \times 10^6$. This is a huge array, which requires a large constant factor C for reading/writing. Combining all of this, the time complexity is $32C \times 10^9$. Being done sequentially in Python, it is understandable why it takes ~10 hours to finish. Assuming $100M$ calculation/second speed, this running time means C is around 100.

For matrix-based implementation, its complexity is due to `X @ X.T`. Matrix `X` has shape (U, M), there for this multiplication requires $O\left(U^2M\right)$ time. That is around the order of *1.5 trillion* operations. This is a lot and should take at least 30 minutes to complete. Fortunately, thanks to numpy, this operation is parallelized across different processor cores and is implemented in C, which makes it run in 30 seconds.

As such, despite the sparsity of the matrix, using matrix-multiplication in numpy can result in a 1200x speed up over a smart loop-based approach. Note that this is purely in CPU. GPU may speed it up even further. This once again highlights the importance of hardware advances in today's computing.

Part 2: KNN and SVM

I tuned the hyperparameters of SVM and KNN for the MNIST problem.

Method

For SVM, I used the following param grid (16 combinations)

```
    'C': [0.1, 1, 10, 100]
    'kernel': ['linear', 'poly', 'rbf', 'sigmoid']
```

For KKN, I used the following param grid (12 combinations)

```
    'n_neighbors': [1, 3, 5, 7, 9, 11]
    'weights': ['uniform', 'distance']
```

For each parameter set, I trained each model on the training set and measured its accuracy on the test set.

Result

weights	n_neighbors	accuracy	seconds
uniform	1	0.9691	1.9
uniform	3	0.9705	2
uniform	5	0.9688	2
uniform	7	0.9694	1.9
uniform	9	0.9659	2
uniform	11	0.9668	1.9
distance	1	0.9691	1.7
distance	**3**	**0.9717**	**1.7**
distance	5	0.9691	1.7
distance	7	0.97	1.8
distance	9	0.9673	1.8
distance	11	0.9678	1.7

Table 1: KNN results

kernel	C	accuracy	seconds
linear	0.1	0.9472	140.8
linear	1	0.9404	183.6
linear	10	0.931	285.7
linear	100	0.9258	1135
poly	0.1	0.9572	394.2
poly	1	0.9771	181.8
poly	10	0.9786	130.6
poly	100	0.9787	126.8
rbf	0.1	0.9595	425
rbf	1	0.9792	199.1
rbf	**10**	**0.9837**	**188.1**
rbf	100	0.9833	188.5
sigmoid	0.1	0.842	481
sigmoid	1	0.7759	307
sigmoid	10	0.7672	256.1
sigmoid	100	0.7637	255.7

Table 2: SVM results

SVM’s best accuracy is 0.9837 at C=10 and ‘rbf’ kernel (Table 1). Meanwhile, KNN’s best accuracy is 0.9717 at ‘distance’ weights and N=3 (Table 2). Therefore, SVM outperforms KNN by about 1% of accuracy.

However, regarding running time, SVM runs for much longer than KNN -- about 100x on average for each setting. In total, the entire SVM grid search ran for 80 minutes, while the KNN’s search ran for only 22 seconds![4]

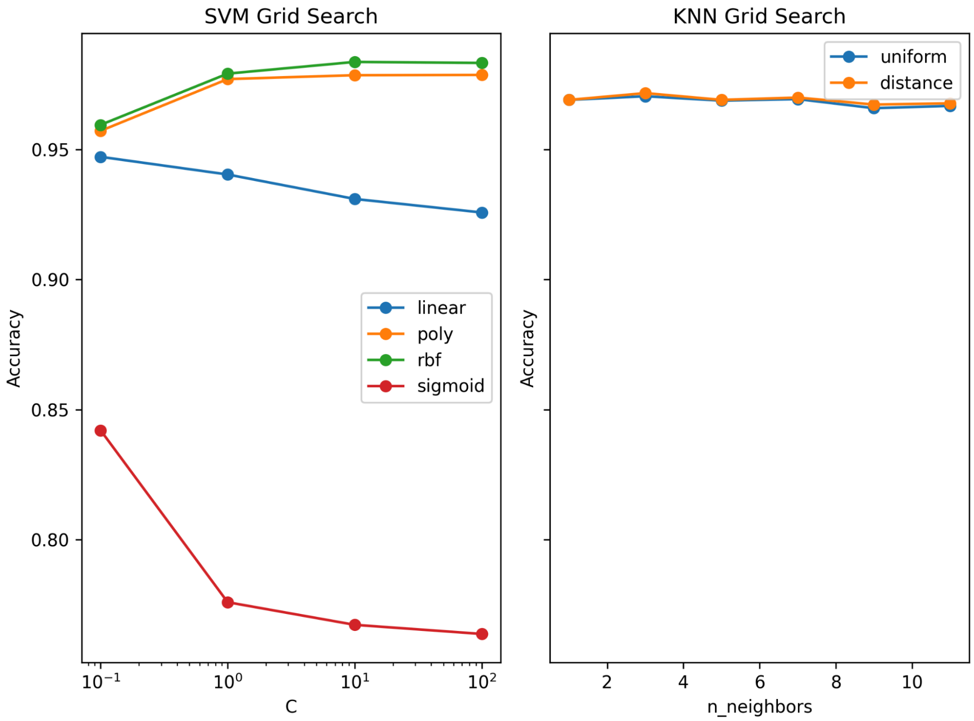

Figure 2: Accuracy of SVM and KNN during grid search.

To better illustrate the difference in behaviors between the two models and among their parameters, Figure 2 put their performances side by side. It is now apparent that KNN is much more robust to hyperparameter choice. While KNN’s accuracy is always above 95%, SVM’s accuracy is only that high when ‘rbf’ and ‘poly’ kernels are used.

Within SVM, there is a clear accuracy distinction between the kernels, which is rbf > poly > linear > sigmoid. It is surprising to me that linear > sigmoid, since linear is the simplest kernel, equivalent to logistic classifier, while sigmoid should be equivalent to much higher feature dimensions. Regarding the effect of C (regularization constant), the accuracy increases with C for rbf and poly kernels, while decreasing with linear and sigmoid kernels.

[4] Note that KKN makes use of multiple cores of my machine, while SVM does not. Perhaps SVM’s internal quadratic optimizer needs to be run sequentially.

Within KNN, distance weights are slightly better than uniform weights. Recall that distance weights take into account the distance to the nearest neighbors during prediction, while uniform weights treat all neighbors the same. Given that distance weights give richer information to the predictor, it is intuitive while this metric performs better.