Collaborative Filtering with Alternating Least Squares

Sidd Singal
December 12, 2016

Abstract

Collaborative filtering is a popular machine learning strategy that provides product suggestions to users based on that user's current experience and recorded observations from the past. Applications include suggesting movies to Netflix users, songs to Spotify users, and books to Amazon users. We use alternating least squares to implement collaborative filtering. Collaborative filtering on big data sets is not practical without parallelizing the workload. Therefore, we introduce a distributed matrix library developed in Scala to help us perform all necessary computations. We feed MovieLens data, which supplies us with movie preferences of users, through the filter and experiment with different parameters. We note a linear correlation between execution time and data workload and an inversely proportional relationship between execution time and number of processors.

1 Introduction

Spotify has a useful feature that suggests songs back to someone based on what they currently have in their playlists. It can be hard to discover certain songs, so these recommendations help reveal some that the user might not have been able to find otherwise.

As simple and basic as it may seem as a feature, a large amount of algorithms and machine learning goes behind making it possible. These algorithms are packaged into something called recommender systems, which "recommend" experiences and products to users based on their and others' past observed experiences. Spotify is not alone; Netflix recommends movies, Amazon recommends books and products, YouTube recommends videos, and Facebook recommends friends. In this day and age, recommender systems are practically a requirement for many consumer facing services.

There are three primary approaches to implementing recommender systems[1].

- **Collaborative filtering** - Information is collected on all past user behavior. In order to recommend products to a specific user, other users with similar behaviors are identified and their preferences are reflected into the recommendation.
- **Content-based filtering** - Users are recommended a product based on their preferences and the characteristics of the product. From past products rated by the user, this filtering technique can build a profile on the user to determine the type of products they would like.
- **Hybrid systems** - As the name suggests, hybrid systems combine both collaborative and content-based filtering capabilities.

The goal of this paper is to implement a functional recommender system. One problem many computation engines are facing today is the vast amount of data that needs to be processed. A user will become disinterested if they have to wait even minutes to receive recommendations for products. Therefore, a lot of recommender systems need to allow horizontal scalability so the speed of computations can grow with the number of computing nodes. Because this is such an important aspect of these systems, we will need to look at one that has parallelizable computations.

In section 2, we will explore an implementation of collaborative filtering that uses the Alternat-

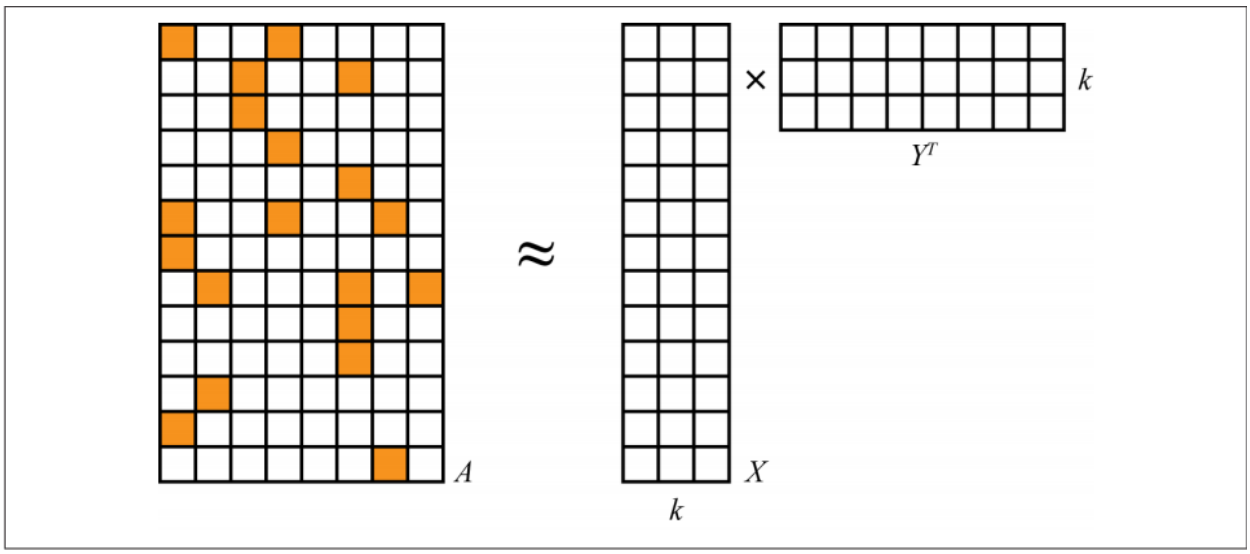

Figure 1: Factorizing a sparse matrix into two smaller dense matrices[3]

ing Least Squares technique to factorize a matrix representing an input set of observations. This factorization is necessary to perform a sort of semantic analysis and compression on the data.

We will briefly look at a Scala implementation of a distributed matrix library in section 3. From the client's perspective, this matrix library hides the fact that all computations are actually done in a distributed fashion across a given number of processors, and so a developer can use the library without having to worry about any parallel computing optimizations. We will also look at the implementation of the collaborative filtering with alternating least squares approach. All code is available for public use at `https://github.com/ssingal05/collaborativefiltering`.

Section 4 will describe the dataset, MovieLens's movie ratings data, used to perform validation and tests. More specifically, in section 5 we will see how execution time is affected by both the workload and number of processors.

We will conclude with insights from this work and potential future goals in section 6.

2 Technical Dive into Collaborative Filtering

As there are multiple approaches for recommender systems, there are also multiple approaches for collaborative filtering. We will be using a technique called Alternating Least Squares. The below subsections will go over why alternating least squares are appropriate for this problem.

2.1 Matrix Factorization

We can generally expect our input matrix to be incredibly sparse. Remember that users will have many possible products that they could potentially try out. However, they will only have tried out a small subset of them. For example, an input matrix for Amazon products could be ratings users give the Amazon products. You could imagine that most of the entries in this matrix will be 0, which means that the user has not considered that product.

A matrix this big could be hard to process, and so we make attempts to factorize it into two smaller matrices as shown in Figure 1. Matrix factorization will also bring to light some latent factors that would not be directly observable before. As a brief example, imagine user 1 enjoying products A, B, and C and user 2 enjoying prod-

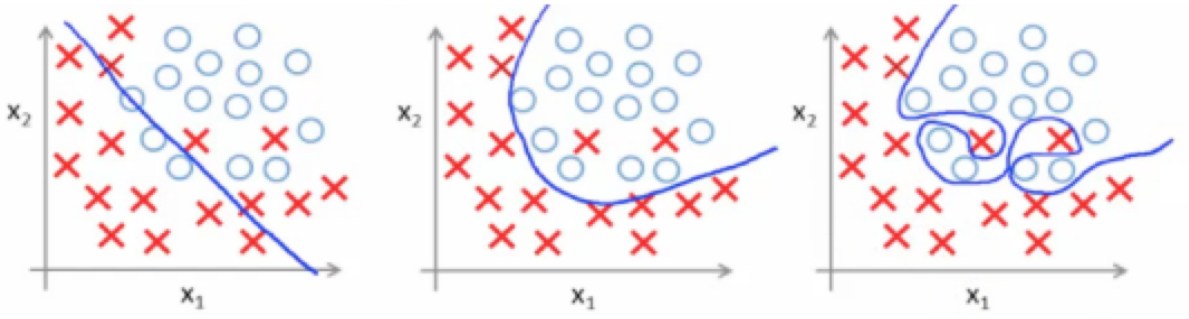

Figure 2: On the left is an example of underfitting, where we do not have enough features to describe the data. On the right is an example of overfitting, where we have features with too much influence over a training set that does not accurately represent the real situation. The middle graph is what we should aim for.[4]

ucts B, C, and D. Our initial input matrix might not realize any association between products A and D, whereas the factorized matrix representation would.[2]

From here on out, we will refer to our sparse input matrix as A. We would like to find matrices X and Y such that

$$A = XY^T.$$

If A is of size $m \times n$, then X will be $m \times k$ and Y will be $n \times k$, where k is the rank of the two factorized matrices. The smaller k is, the more dimensional reduction we will be performing on A. To connect this back to collaborative filtering, m represents the number of total users, and n represents the number of total products. X will be k-dimensional information about each user, and Y will be k-dimensional information about each product.

2.2 Cost Function

It might be hard, or even impossible, to find X and Y such that $A = XY^T$. If it is not impossible, then it might require more computational resources than available. Therefore, we must find approximations for X and Y. Rather than $A = XY^T$, this yields

$$\hat{A} = XY^T.$$

The closer $\hat{A}$ is to A, the better. We call the "difference" between the matrices the cost. Cost functions (J) are tools to help minimize the cost associated with a certain solution. As an example, the cost function of this problem could be represented by

$$J = \sum_{i,j} |A_{i,j} - \hat{A}_{i,j}|.$$

More traditionally, data scientists like to use the sum of squared deviations to give a higher penalty to deviations that are too high. Therefore, we would model our cost function as

$$J = \sum_{i,j} (A_{i,j} - \hat{A}_{i,j})^2.$$

Like with many other machine learning models, we also need to worry about overfitting.[4] Figure 2 depicts the problem when overfitting (and underfitting). We underfit the data when the model does not have enough features to describe the model well enough. We overfit the data when the model has too many features and can only describe the training set well. In order to avoid overfitting the data, we add a regularization term to our cost function in order to penalize features with too much influence on the model.[4] Our new and final cost function now is

$$J = \sum_{i,j} (A_{i,j} - X_i Y_j^T)^2 + \lambda(||X_i||^2 + ||Y_j||^2).$$

λ is a data dependent parameter that can only be deduced from cross-validation.

2.3 Alternating Least Squares

Now that we have our cost function, we need a strategy for finding X and Y such that J is minimal. One approach is to use stochastic gradient descent. For this, we would guess values for X and Y and then, for each iteration, find the derivative of J and move along the gradient to approximate better values for X and Y until a local minimum is reached. [5]

It turns out that stochastic gradient descent is too computationally expensive, especially in comparison to Alternating Least Squares. For this approach, we solve for both X and Y. [5]

$$X = ((Y^T Y + \lambda I)^{-1} Y^T A)^T$$
$$Y = ((X^T X + \lambda I)^{-1} X^T A^T)^T$$

where I is a $k \times k$ identity matrix.

We can initially make X randomized. From there, we can calculate Y using our random X and A. Once we have Y, we can recalculate X using Y and A. We can keep repeating this alternating process back and forth until either enough iterations have occurred or the cost associated with a solution has changed a small enough amount from the previous cost. We can expect X and Y to converge at some local minimum, eventually, using this technique, and it will much more efficient than stochastic gradient descent.

2.4 Making Recommendations

After developing an Alternating Least Squares model, we need to be able to use them to make recommendations to new users. Our model only needs to hold on to Y, k, and λ.

Users that need recommendations are associated with a vector $\boldsymbol{p}$ of n dimensions that represents their rating of each product they have tried. We can also think of $\boldsymbol{p}$ as a $1 \times n$ matrix P. We can compute

$$X' = (Y^T Y + \lambda I)^{-1} Y^T P^T.$$

We can use the resulting matrix X' of dimensions $k \times 1$ and compute cosine similarities with rows in Y to find the highest rated recommendations for that user. Recommendation scores (the higher the better) are calculated as $Y_j X'$.

2.5 Opportunity for Parallelization

As mentioned earlier, we wanted to introduce an algorithm that could be parallelized across a number of processors. From the previous equations solving for X and Y, we notice the following matrix operations:

- matrix addition
- matrix multiplication
- scalar multiplication
- inverse

Matrix addition and multiplication and scalar multiplication are clearly parallelizable because each cell can be computed independently of other cells. Given p processors, for matrix addition and scalar multiplication on matrices of dimensions $m \times n$, we can achieve a $O(\frac{mn}{p})$ time complexity. For matrix multiplication of matrices of dimensions $m \times k$ and $k \times n$, we can achieve a $O(\frac{mnk}{p})$ time complexity.

Inverse is a little bit trickier. First instincts might dictate using Gaussian Elimination to finding the inverse, but that does not allow for trivial parallelization. We will only be finding the inverses of positive definite symmetric matrices (because $A^T A + \lambda I$ will always be symmetric). Given this, we can use the Cholesky Decomposition, instead.

2.5.1 Cholesky Decomposition

Cholesky Decomposition involves decomposing a matrix into two triangular matrices that are transposes of each other. [6] In other words, we must find some triangular matrix L for a symmetric matrix A such that

$$A = LL^T.$$

This decomposition cannot be parallelized as with addition and multiplication because of the dependencies the cells have in the decomposition. More specifically, each cell is calculated as fol-

1	0	0	0	0	0	0	0
2	3	0	0	0	0	0	0
3	4	5	0	0	0	0	0
4	5	6	7	0	0	0	0
5	6	7	8	9	0	0	0
6	7	8	9	10	11	0	0
7	8	9	10	11	12	13	0
8	9	10	11	12	13	14	15

Figure 3: Illustrates the order in which the output of the Cholesky Decomposition would be computed. Cells with 0 in at will have a value of 0. Each diagonal (from bottom-left to top-right) can be parallelized.

lows:

$$L_{k,k} = \sqrt{A_{k,k} - \sum_{j=1}^{k-1} L_{k,j}^2}$$

$$L_{i,k} = \frac{1}{L_{k,k}} \left(A_{i,k} - \sum_{j=1}^{k-1} L_{i,j} L_{k,j} \right) \quad \text{for } i > k$$

$$L_{i,k} = 0 \quad \text{for } i < k$$

To introduce parallelization, we must compute each diagonal sequentially because each diagonal depends on the previous diagonal. However, cells within a diagonal do not depend on each other and therefore can be parallelized. Figure 3 shows the order in which cells would be calculated.

In practice, this matrix will have dimensions $k \times k$. Therefore, we need to calculate nearly $k^2/2$ cells, and each cell will take up to j computations. Therefore, we can bound the complexity as $O(k^3)$. If we assume that $k \gg p$, introducing parallelization would reduce our complexity down to $O(\frac{k^3}{p})$.

2.5.2 Finding the Inverse

Using our decomposed L matrix, we can compute

$$A^{-1} = (L^{-1})^T (L^{-1})$$

Clearly, there is some circular logic here because we need to be able to compute the inverse of a matrix in order to compute the inverse of a different matrix. Fortunately, we know that L is a lower triangular matrix. Finding the inverse of such a matrix is much easier through forward substitution [7] because

$$LL^{-1} = I$$

and we can solve for L^{-1} one row at a time. Each column of L^{-1} can be computed independently of each other, and so we can parallelize this process as well! Each cell of L^{-1} takes approximately $O(k)$ computations. Therefore, computing L^{-1} has a time complexity of $O(\frac{k^3}{p})$. Overall, finding the inverse of any symmetric matrix of dimensions $k \times k$ will have a time complexity of $O(\frac{k^3}{p})$.

2.5.3 Overall time complexity

Let review the iterative equations we need to constantly solve:

$$X = (Y^T Y + \lambda I)^{-1} Y^T A^T$$
$$Y = (X^T X + \lambda I)^{-1} X^T A$$

To solve for X once, we need to perform:

1. Transpose on Y $(O(\frac{kn}{p}))$
2. Multiplication with Y $(O(\frac{k^2 n}{p}))$
3. Scalar multiplication of λI $(O(\frac{k^2}{p}))$
4. Matrix addition $(O(\frac{k^2}{p}))$
5. Matrix inverse $(O(\frac{k^3}{p}))$
6. Transpose on Y $(O(\frac{kn}{p}))$
7. Multiplication with Y^T $(O(\frac{k^2 n}{p}))$
8. Transpose on A $(O(\frac{mn}{p}))$
9. Multiplication with A^T $(O(\frac{mnk}{p}))$

Summing those all up, assuming $n > k$ and $m > k$, and considering we have to perform q iterations, we end up with an overall time complexity of $O(\frac{mnkq}{p})$.

3 Implementation

As mentioned, we had to build a linear algebra library that could handle all the computations mentioned in section 2. In this section, we will go through the technical details of the library. We will then look at the collaborative filtering specific code. All code is available at `https://github.com/ssingal05/collaborativefiltering`.

3.1 Tools

Below is a list of all the software tools used to build this project.

- **Scala** (`scala-lang.org`)
- **Maven** (`maven.apache.org`)
- **IntelliJ** (`jetbrains.com/idea/`)

3.2 Matrix Implementation

It is important that matrix operations are developed as efficiently as possible. Some operations, such as multiplication and decompositions, can be incredibly expensive. Another non-trivial aspect of matrices is its partitioning mechanism. We need to optimize the number and size of partitions in order to minimize execution time. Lastly, we need to have support for both dense matrices and sparse matrices. Sparse matrices are good when the matrix primarily consists of zeros as we only store the non-zero values. Because there is more overhead with storing each element, we would want to use a dense matrix when we have a full or nearly full matrix. Below is the class hierarchy of the matrix library.

- `Matrix`: abstract class that provides all the basic matrix operations.
  - `DistributedMatrix`: abstract class that adds distributed capabilities to the `Matrix` class.
    - `DistributedDenseMatrix`: Implements all inherited methods and stores all data in a two-dimensional array.
    - `DistributedSparseMatrix`: Implements all inherited methods and stores all data in a hash map, with the indices as the key and the entry in the matrix as the value.

3.2.1 Matrix Operations

Below is a list of all functions that must be implemented if superclassing the `Matrix` class. They are basic operations one would expect from a small linear algebra library:

- Constructor: create an a empty constructor initialized with a given number of rows and columns
- `get(row,column)`: for a matrix A, return the element located at $A_{\text{row,column}}$
- `transpose`: returns a transposed version of the matrix
- `+`: if given a matrix, return the sum of the two matrices. If given a number, return the matrix with the number added to each element
- $*$: if given a matrix, return the product of the two matrices. If given a number, return the matrix with the number multiplied to each element
- `inverse`: return the inverse of a square symmetric matrix

3.3 Alternating Least Squares Library

There are two main functions which are important regarding our collaborative filtering library.

- `train(inputMatrix,k,`λ`, iterations,numProcessors)`: creates an ALS model with the given input matrix of observations and parameters
- `predict(inputVector)`: returns a list of recommendation scores for each available movie.

The `train` code is an implementation of the equations from Section 2.3, the only difference being that X and Y are recomputed `iterations` times. `predict` is an implementation of the equation from Section 2.4.

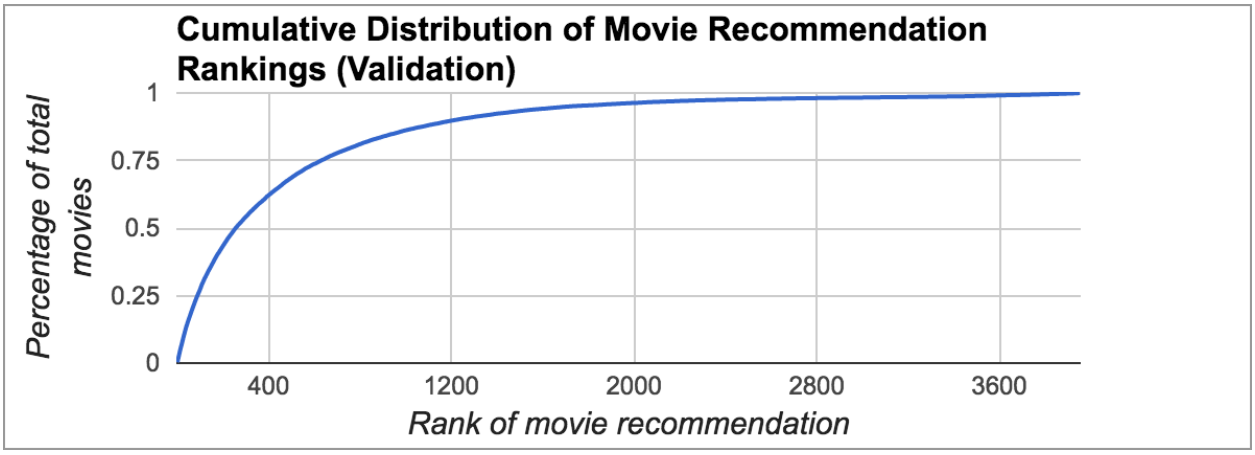

Figure 4: Cumulative distribution where Y-axis represents the percentage of missing movies accounted for within the first-X recommendations.

4 Dataset

The data set we will be using in this looking at in this paper is provided by MovieLens at `http://grouplens.org/datasets/movielens/`. From this dataset, we can extract movie ratings that different users have given to some movies. They have datasets of varying sizes, but we will be using an older dataset published in February 2003 with 1,000,000 ratings from 6000 users on 4000 movies. It is a big enough size that it is interesting to process, but it is also small enough that a single computer can handle it in a reasonable amount of time. This entire dataset consists of three files.

- **Ratings**
  `UserID::MovieID::Rating::Timestamp`
  For each user, rating given to each movie seen by that user. They must have seen at least 20 movies to be included in this data
- **Users**
  `UserID::Gender::Age::Occupation::Zip-code`
  Characteristics of every user in the database
- **Movies**
  `MovieID::Title::Genres`
  Characteristics of every movie in the database

It should be clear that the only file we will need to build a collaborative filter is the ratings file. User and movie characteristics could be useful if we wanted to build a content-based recommender system.

Ratings are on a five-star scale (only whole stars can be given). A rating of 0 indicates that the user has not watched the movie.

5 Results

Before we can analyze how tweaking different parameters of our distributed system affects our total processing time, we need to make sure our collaborative filter algorithm is correct.

5.1 Data Validation

In order to validate the data, it must be split up into a training set and a test set (which is a process that should be familiar with all machine learning experts). Because we have very few users, the training set will be comprised of a randomly selected 90% of the data while the test set will be the remaining 10%. The training set will then be used to build our collaborative filtering model, which is made up of two decomposed matrices.

We will also modify the test set to remove only a couple of rated movies from each user. For our tests, we removed 25% of movie preferences from the users in the test set, which equates to about 24,000 movies in our case. We can then feed our

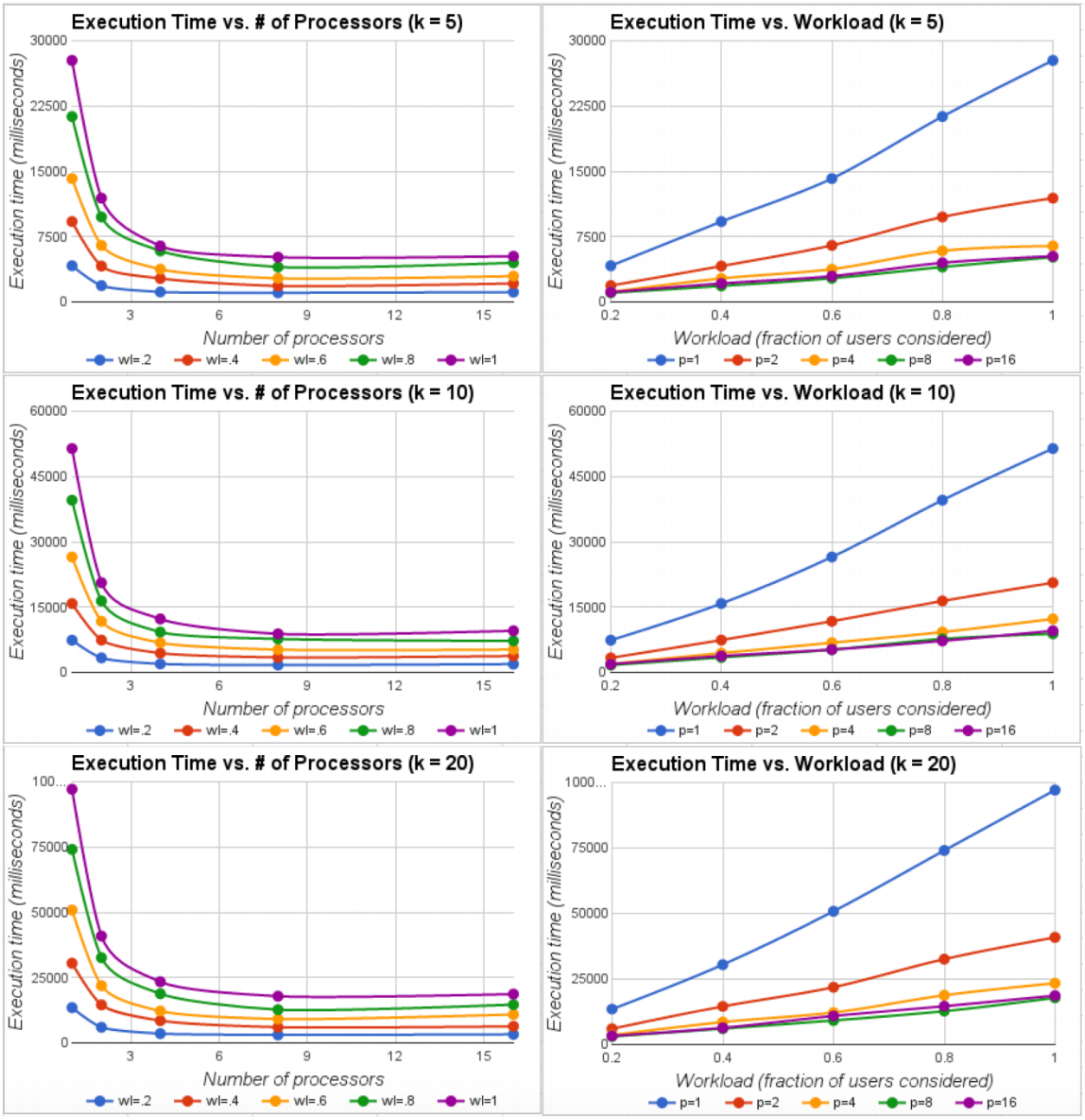

Figure 5: *Left*: Comparing execution time vs the number of processors with k values of 5, 10, and 20 and different workload amounts. *Right*: Comparing execution time vs the fraction of total workload (users) with k values of 5, 10, and 20 and different numbers of processors.

test set data with the remaining movies through our model and recommend movies to each user. The assumption is that our model is correct if the movies that we removed from each user are highly recommended to that user. In Figure 4, we graph a cumulative distribution showing what ranks the removed movies were recommended to the test set users. Our model appears to be correct as more than half of the recommendations fall within the first 300 recommended movies for each user.

5.2 MovieLens Results

After performing data validation, we want to see how our data set performs against our algorithm. The point of the algorithm is to prove that parallelization can speed of the computations. It would also be interesting to observe how the execution time correlates with the size of the workload. For each test, we only time the amount of time it takes for the program to train the model.

The following results were produced on a computer with an i7 processor, 16 GB of RAM, and up to 8 physical cores.

5.2.1 Number of processors

The graphs on the left side of Figure 5 compare execution time to the number of processors used. k values of 5, 10, and 20 were used for the tests. Each graph also has five lines, each corresponding to a different amount of workload (wl). For example, a $wl = .2$ represents tests where only 20% of users were considered.

We can see that there is an inversely proportional relationship between the execution time and number of processors. This is what we expect because the time complexity of our program is $O(\frac{mnkq}{p})$. Using t to represent execution time, both the time complexity and our results support

$$t \propto \frac{1}{p}.$$

Note that execution times slightly increased when $p = 16$. This is because only 8 cores exist on the testing machine, and adding more processors could add overhead which would negatively affect the efficiency.

5.2.2 Size of Workload

The graphs on the right side of Figure 5 compare execution time to the number of processors used. Again, k values of 5, 10, and 20 were used for the tests. Each graph also has five lines, each corresponding to the number of processors used (p). For example, a $p = 4$ represents tests where tasks were parallelized over 4 processors.

We can see that there is an linear correlation between the execution time and number of processors. This is what we expect because the time complexity of our program is $O(\frac{mnkq}{p})$. Using t to represent execution time, both the time complexity and our results support

$$t \propto m.$$

6 Conclusion

Building a collaborative filter from scratch seems to have been successful for analyzing this small MovieLens dataset. We were able to validate our equations and running tests described in Section 5.1. We were able to time our program by varying the number of processors available to the program and the fraction of total workload to consume. We found that $t \propto \frac{1}{p}$ and $t \propto m$. Presumably, the matrix library should also be able to handle other datasets of similar size.

The software created for this project was meant more as an exploration into the broad topic of parallel computing as opposed to a product to be used for practical purposes. In reality, the library would not suffice for real world applications. Below are a few reasons why:

- The code was written especially for this collaborative filtering problem, and so the code is not fully implemented or cleaned up. For example, the sparse matrix implementation is missing many functions as they are not required to do collaborative filtering.
- The library requires that the dataset be able to rest in memory. Therefore, there cannot be too much data, else we would run into overflow errors. This can be remedied by modifying the matrix data structures to hold values on disk as opposed to in RAM.

- This library only supports computations on a single computer. Having multiple machines available as nodes would allow us to scale horizontally in computational power and memory
- Some matrix operations have not been optimized, such as finding the inverse or performing decompositions. There are a number of already existing linear algebra libraries, such as LAPACK, that can handle all of our intense matrix computing needs. In future iterations of this project, we could outsource some of the comptutations to LAPACK.
- Some computations are bound to break, so many large scale computing engines provide some sort of fault tolerance. It is unclear how this library would handle faults in computation, but fault tolerance is a feature that could be introduced in the future

If there is any interest in making these modifications, all code is available for public view and reuse at `https://github.com/ssingal05/collaborativefiltering`.

Of course, there is also the option of using already existing computing engines and libraries to build recommender systems. Apache Spark and Apache Hadoop (along with machine learning libraries like MLLib and Apache Mahout) have libraries that can build more scalable recommender systems. If a Python solution is needed, scikit-learn is also a great option.

References

[1] Recommendation Systems. `http://infolab.stanford.edu/~ullman/mmds/ch9.pdf`

[2] Bokde, Girase, Mukhopadhyay. Role of Matrix Factorization Model in Collaborative Filtering Algorithm: A Survey. `https://arxiv.org/pdf/1503.07475.pdf`

[3] Wills, Ryza, Owen. *Advanced Analytics with Spark*, Chapter 3

[4] Regularization. `http://www.holehouse.org/mlclass/07_Regularization.html`

[5] Aberger, Christopher R. Recommender: An Analysis of Collaborative Filtering Techniques. `http://cs229.stanford.edu/proj2014/Christopher%20Aberger,%20Recommender.pdf`

[6] Cholesky Decomposition. `https://rosettacode.org/wiki/Cholesky_decomposition`

[7] Solving Systems of Linear Equations. `https://www.cs.mtu.edu/~shene/COURSES/cs3621/NOTES/INT-APP/CURVE-linear-system.html`