1. what problem you are solving; why it is an important or challenging problem;

The intent of this project was to compare neighborhood-based(NB) recommendation systems with matrix factorization(MF) based recommendation systems. The purpose of that is to see where the performance differences lie between the two. The evidence shows that NB systems are typically very good performers when users have a significant amount of data already in the system. The main reason for this is because the user’s abundance of data provides more resources for the system to determine the highest ranked items based on similarity. Unfortunately, NB systems perform horrifically when applied to cold start scenarios. Cold start is the terminology to describe a user or item with no previous data. NB systems do not perform well with cold start because their reasoning for making recommendations is that the users have a history of selecting specific types of items.

Another very common recommendation system is the MF system. The key concept when using MF is that both users have latent factors that describe their preferences. If you can create a vector that represents those factors, you can predict how a user will feel about an item multiplying those two vectors. Where this is extremely useful is in cold start scenarios. While there is no easy way to create recommendations for NB systems, it is relatively easy to do so in MF based. You can assume that a new user will be a relatively average user. With this assumption, you can assign them a latent factor that is the average user. Then, as they rate items you can adjust the latent factor to describe them more closely. This initial latent factor can be used to determine what an ideal set of initial recommendations is. If they are an average user, it is possible to have some success. Unfortunately, one major downfall to MF is that it can be very expensive to do the initial training. After initial training, updating and maintaining the matrix is relatively cheap. Also, there is relatively low cost on memory in comparison to the NB or even the sparse representation of data itself. The MF also tends to fall behind when users have an abundance of data. The NB system will start to shine in this scenario.

Why is this comparison important? Well not only is it valuable to understand the differences so that you can pick the better system for your application, it is also possible to create a hybrid for your system. A hybrid is a recommendation system that will use multiple different recommendation systems to create recommendations and depending on the scenario for user, it will favor one system over another. In a sense, it takes the best of all the worlds. In cold start, it will probably rely on MF. Whereas in a scenario where the users have a lot of data it will rely on the NB system. Our goal with this project was to, after comparing the two primary systems, build a hybrid that utilized both and provided the best of both worlds.

1. what methods/algorithms you are using; the details of the methods/algorithms;

In the development of the MF model, we started with a version of the Programming Assignment 2. We added a noticeable amount of extra functionality. The simplest and most straight forward set of functionality was to implement K-folds cross validation using multiple versions of the same data set. From there we added a large amount of other functionality to allow for clear comparison with the NB system. Including, an algorithm for creating a list of recommendations based on similar users, a quicksort algorithm for sorting recommendations, a method for training individual users with new entries, a simple cosine similarity method, the ability to determine the average user, and an algorithm for systematically testing the system with a set of cold start users. The vast majority of the methods are very straight forward but the few that aren’t we will go over below.

The first method we will talk about is the algorithm for creating a list of recommendations based on similar users. The method we used is a method derived from the method provided by Lei Guo, in MF Techniques for Recommender Systems. In their method, they take the K most similar users to the primary user. They then do a frequency report on all the items those users have viewed. They then use the most frequent items amongst those users to create the N recommendations. As a simple neighborhood type algorithm, it takes advantage of the user’s latent factors to determine the recommendations. The belief being that if all the latent factors for two users line up, they will more than likely enjoy the same things. It is a little risky to use for cold start but if most of the new users are average users to some degree it is likely that we could provide some successful recommendations early on. We then made sure to train the new user after every new item the user picked. This way after a few iterations the user’s latent factors start to paint a clearer picture of what their preferences are. A more robust method would be desirable but due to time constraints, we were not able to develop one.

The next algorithm we would like to explain is the method for training individual users. While this method is a simple method it is has a few differences from the primary gradient descent method. Obviously, it only affects one user and item at a time. This makes it’s computation time extremely fast. It operates under the same principle as the gradient descent method. The other major difference is with the learning rate for the new user. It is multiplied by roughly 100 under the concept that with a brand-new user it may be worthwhile to speed up their learning process. That provided successful results in having more unique recommendations early in the learning process. Unfortunately, as a user is getting closer to converging on their ideal latent vector it could cause significant error in finding that ideal vector. There is a concept for a variable learning rate that would decrease as the user gets closer to the ideal vector. Unfortunately, due to time constraints we were not able to implement this feature.

The last method in the MF system worth going over would be the algorithm for testing the system with a set of cold start users. This is the method that was proposed for both the NB and the MF systems. The problem was how to test the systems in a cold start environment and monitor their success rate as more items came in. The method we proposed was:

For Every User U in Cold Start Set:

While (user has untrained items):

* + Create a set of recommendations and predicted ratings for user
  + Calculate the number of recommendations that are in untrained items as hits.
  + Determine position of hits in recommendations and error for predicted values and actual ratings
  + From there calculate and save potential HR, potential ARHR, MSE, and RMSE for that set of recommendations
  + Then pick a random value within untrained items and train the user as if it was their next item remove item from untrained items

Ideally, this method can give us a few measures that would help determine how successful our systems are in cold start scenarios along with at what point do they stop improving their performance. This would give us a baseline for weighting the systems within the hybrid. There is one major concern with this method related directly to the data set we are using. Picking a random value from our untrained items to train with limits the measurements we can accomplish. Unfortunately, not knowing the order in which users rated the items, we can’t justify picking specific items. Thus, we must choose randomly. Unfortunately, choosing randomly means we cannot get true measurements of HR or ARHR. Instead we measure a potential hit rate under to determine if any of the recommendations exist within the untrained items. This tells us that the user at some point will at least pick a few of what we are recommending. MSE and RMSE tell us, out of the items that the user will pick, how accurate our current rating predictions are. These metrics could have been greatly improved upon by having a dataset that included the order in which a user rated items.

We also use Item-Item based Collaborative Filtering to generate recommendations. This is based on the idea that people who like a set of items are also more likely to like other items similar to the set of items. The ratings are generated from items that are similar to a users items but are also unrated to a user. The training and testing is done in the same way as Assignment 1.

The biggest challenge with NB is how to handle cold start cases. Some of the research that we came across involved Interactive CF and Two-Stage Filtering as mentioned here: <http://discovery.ucl.ac.uk/1474118/1/Thesis_final_revision.pdf>. Due to shortage of time, we ended up deciding on picking a form of average for an item. The items with higher scores, in a cold start, scenario would be the ones picked as recommendations. To calculate the score, we used Lower Bound of Wilson Score Interval. This algorithm has been used by other notable companies in the past such as Reddit and Yelp(Sources: <https://redditblog.com/2009/10/15/reddits-new-comment-sorting-system/>, <https://www.yelpblog.com/2011/02/the-most-romantic-city-on-yelp-is> ). The benefits of this type of averaging involve normalized results with filtering away extraneous cases and other pitfalls that come from classical averages. We did our calculations for the Wilson Score Interval with a 95% Confidence Interval. One benefit of this is that we can calculate meaningful averages very efficiently.

The actual integration of Wilson score interval lower bound(WSI) was relatively simple. The items were ranked by the following score:(cosineSimilarity + .5) \* WSI. Using this equation, there is no effect on the rankings when the system is fully trained since the order of the scores are exactly as before integrating WSI. This score becomes more powerful when cosineSimilarity is 0, at which point the final rating for an item is just WSI. With how low the values of WSI are (in the hudredths of percents), all of the pure WSI ratings get pushed back and replaced by ratings that are higher than 25%, of plain cosine similarity.

Using this method, we seamlessly push an average item to the user when there are not enough of good items to recommend specifically to the user.

1. what data you are using to test your methods; the details of the data;

In this project, we are using the same data that was used in programming assignments 1 and 2. This data allowed for easier development of programs and some sort of expectations in the results we would be getting. The one thing we did to change the data was to split the data for K-folds cross validation. This was done using a simple python script that split everything into 5 versions. The items that would go into each K-folds sets were randomly chosen for each file. They do should not have any overlap for each testing set.

4. the results presented in multiple tables and/or figures

5. what problems you still have

With no way to truly tell how successful our system is based on what order a user rates items, it is next to impossible to truly prove the performance of each system in a cold start scenario. If we had noticed this flaw earlier, we could have easily found a data set with time stamps and rebuilt our parsing to accept the new dataset. Also, due to a few pervasive bugs we were not able to fully implement the hybrid system(edit as seen fit). There are also interesting behaviors in the MF system where if K = 50, MSE and RMSE score off the charts. We were not able to find the origin of this bug.

6. each team member is required to explicitly specify his contribution. Grading will be for each team member, not for each team as a whole.

Dimitry’s Contribution – Convert Evan’s Assignment 1 to C++ (everything under the NB/CPP directory. Solve the problem of cold start for Collaborative Filtering (Wilson’s Score Interval), and creation of sample cold starts for some benchmarking for untrained systems (CreateColdStarts.cpp file).