191 Final Report

By Lauren Dunlap, Kevin Stangl, Aaron Zhou Qian

-Motivation, definition, and significance of the algorithm

The motivation of the softimpute alternating least squares algorithm is to complete a sparse matrix. One of the applications of such matrix completion algorithms is to provide predictions on user-item ratings, such as in the context of moving recommendations. In such contexts, the movie ratings form a sparse matrix with the rows representing the users and the columns representing the movies. We assume the matrix that contains the ground truth of all the ratings to have low rank, which is the key for such matrix completion algorithms.

There are usually many users and movies so the resulting matrix would have very big dimensions. So one of the advantage of using the alternating least squares is that we can store the matrix across the iterations of the algorithms as sparse plus low rank. This makes the storage and multiplication of the matrices computationally faster.

In the original paper, the authors implemented the algorithm in the statistical programming language R. Our group tried to implement the algorithm in python. Later on we upgraded the speed of our implementation by using the Cython interpreter.

-Algorithm description

We wrote a function “generate_training_dataset(filename)” that randomly select 20% of the 100,000 data points to serve as testing set and the remaining 80% for training. Note that this piece of code is included in the python script although it should run in the terminal separately before the whole algorithm to generate the training/testing data sets.

Afterwards, we successfully parsed the data as the data was clearly delimited by white spaces.

With the scipy/numpy packages we successfully created a random matrix as the start and implemented Algorithm 3.1 in a non-sparse fashion, in parallel with the non-sparse implementation by the authors in the R code.

In the main() function we wrote a code to automatically test a range of regularization parameters lambda and computed the training/testing RMSE with the help of another function that we wrote.

In the end, we used matplotlib to plot the rates for the algorithm to converge with respect to different choices of lambda. The y-axis is the relative changes of the estimators measured in terms of the Frobenius norm, while the x-axis is time in seconds.

We also plotting the training/testing RMSE with fixed lambda but against increasing ranks.

After finishing the non-sparse version of the implementation, we upgraded the speed of the implementation by using the Cython interpreter. When iterating over the loops we used C loop so the iteration was significantly faster.

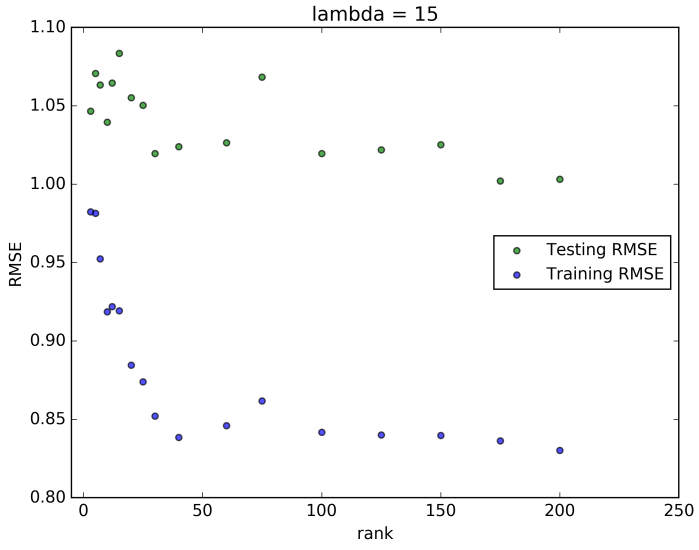
-The complete reproduced experiment results

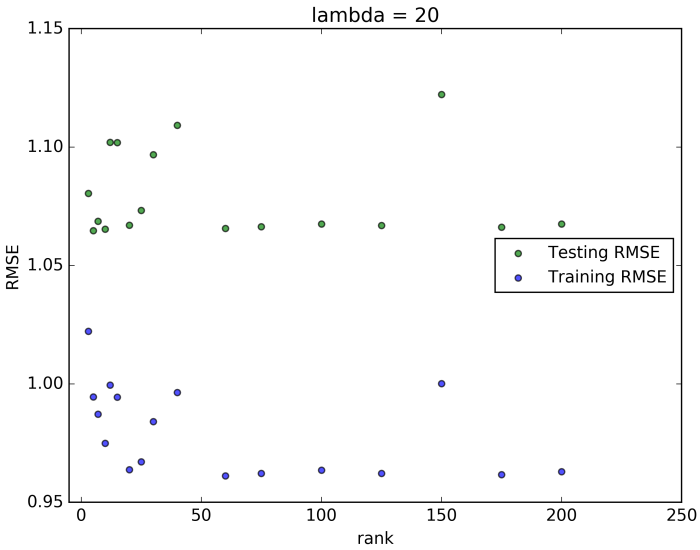
In the beginning, we implemented the algorithm without creating any sparse matrix multiplication function. In other words, the only place we took advantage of the sparse matrices is when we were initializing the matrix X, which contains the observed entries from the dataset. Afterwards, when we were modifying the entries in the matrix X as in step 2 of algorithm 3.1 equation (22), we converted the matrix to a dense matrix and multiplied out A and B transpose, and performed direct matrix assignment with the index slicing feature in python, i.e. we did xfill[row,col]=xhat[row,col] where row and col are lists containing the row indices and the column indices of the unobserved entries.

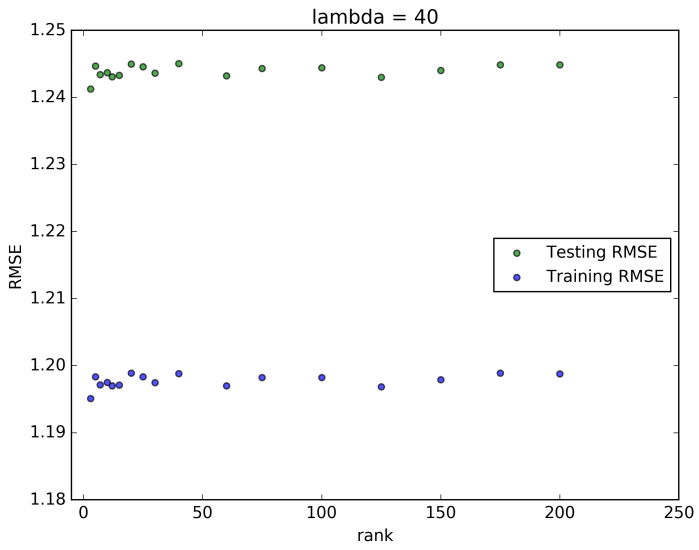
The reason why we didn't implement the sparse matrix multiplication was because such implementation would require a loop that iterates over the positions of the observed entries. Even though the matrix is sparse, perform such iterations with a for loop in python was very slow.

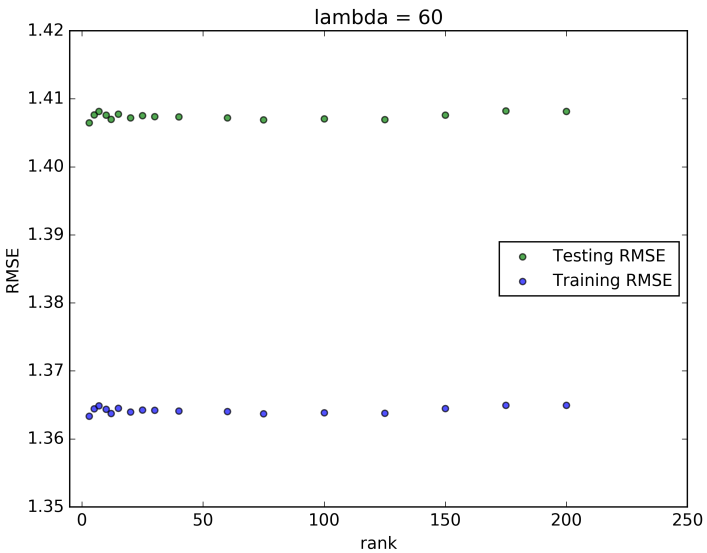
Therefore with our initial implementation of the algorithm, our algorithm did not perform at the speed the authors reported in the paper. However, in terms of the accuracy of the prediction results, measure in term of the root mean squared error (RMSE), our implementation matched the results reported in the paper with good accuracy.

Plots of the RMSE against some ranks are displayed below:

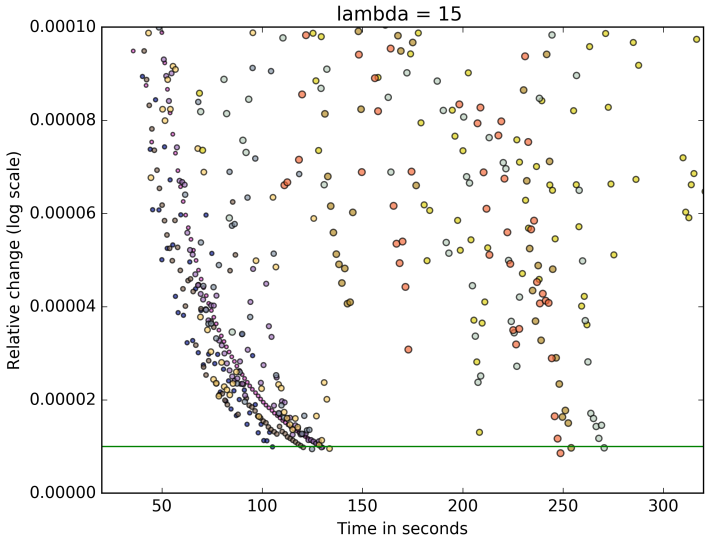


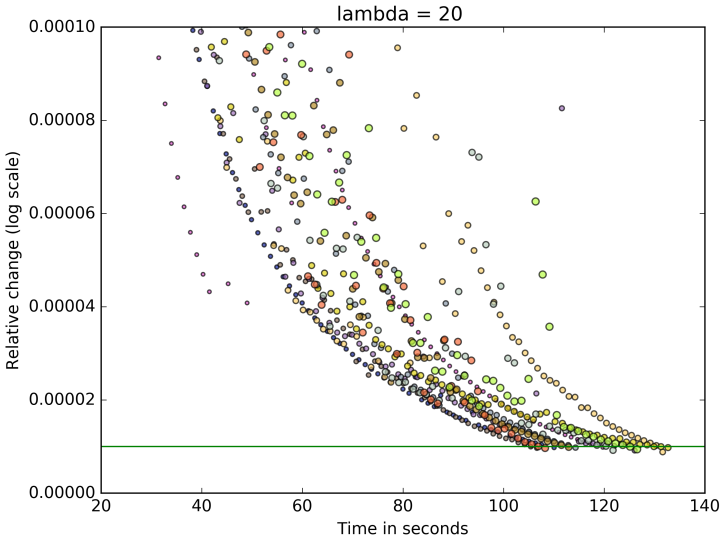


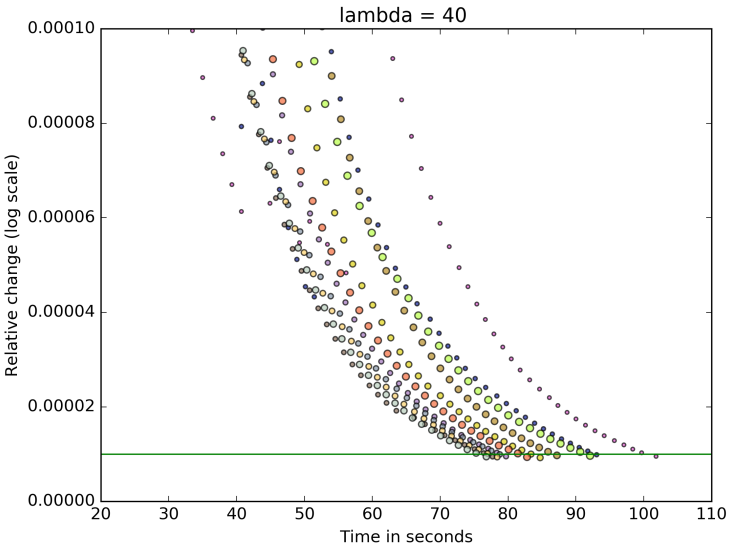


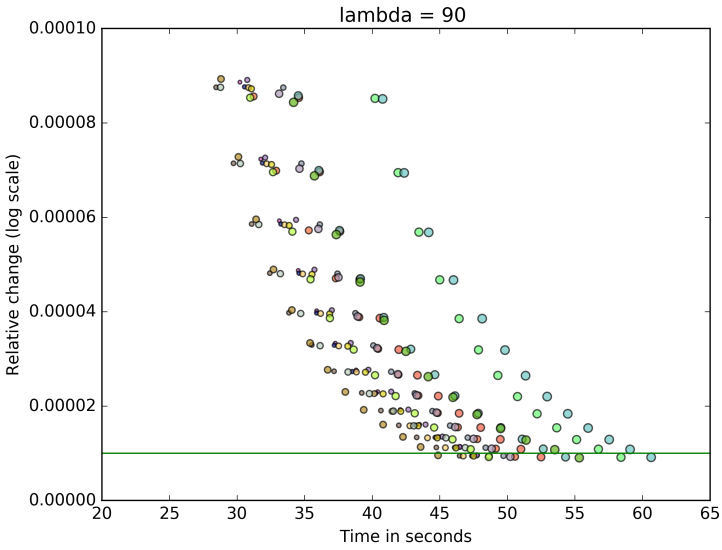


Here are some of the plots for the rates of convergence without Cython:

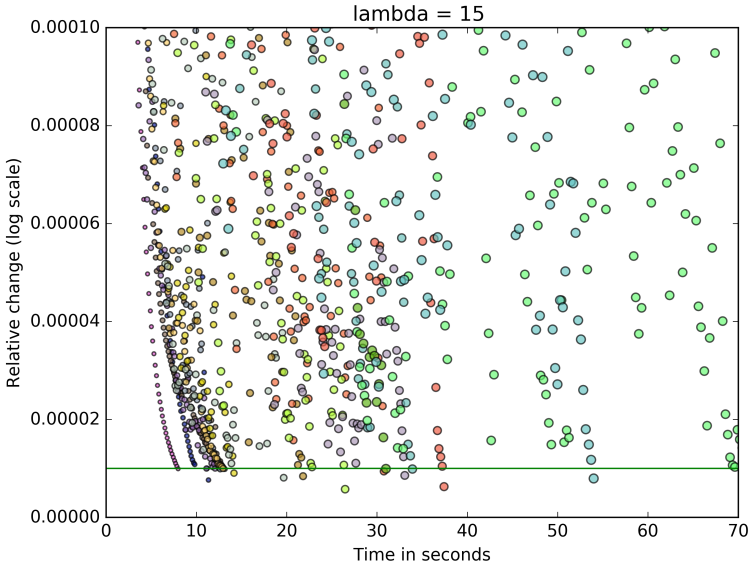


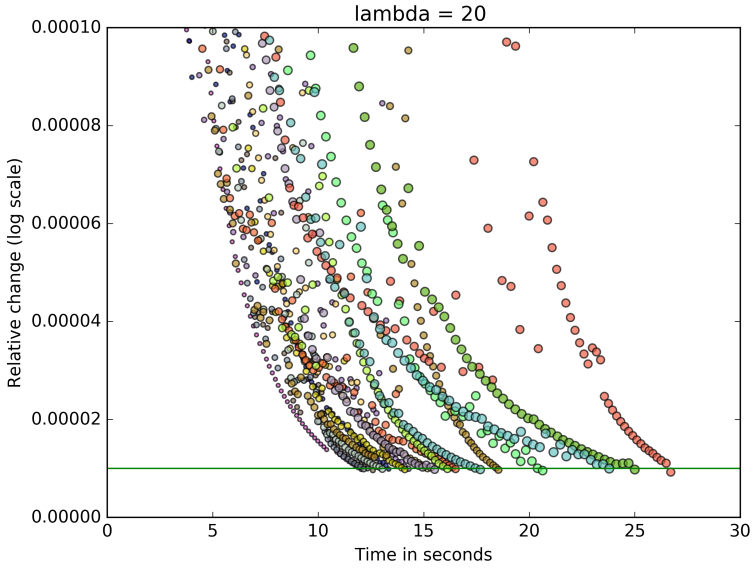


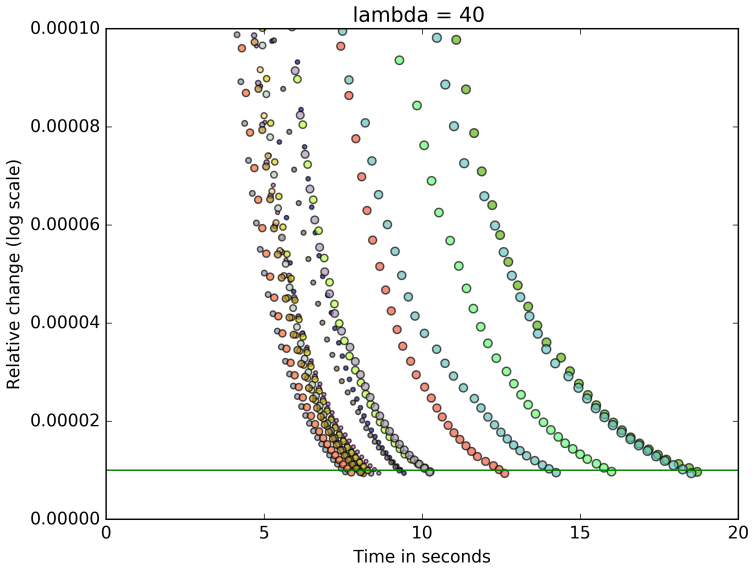


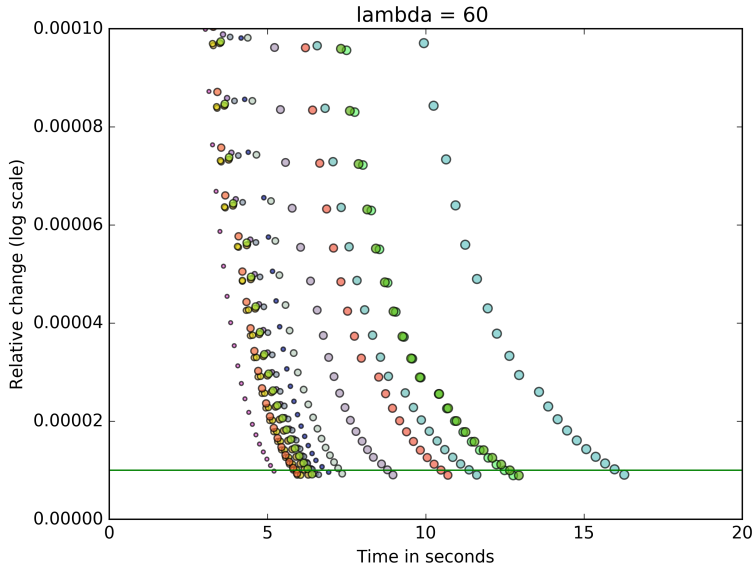


However, after using the Cython interpreter, our speed improved dramatically. Here are the plots for the upgraded rates of convergence, notice the change in the x-axis scale:





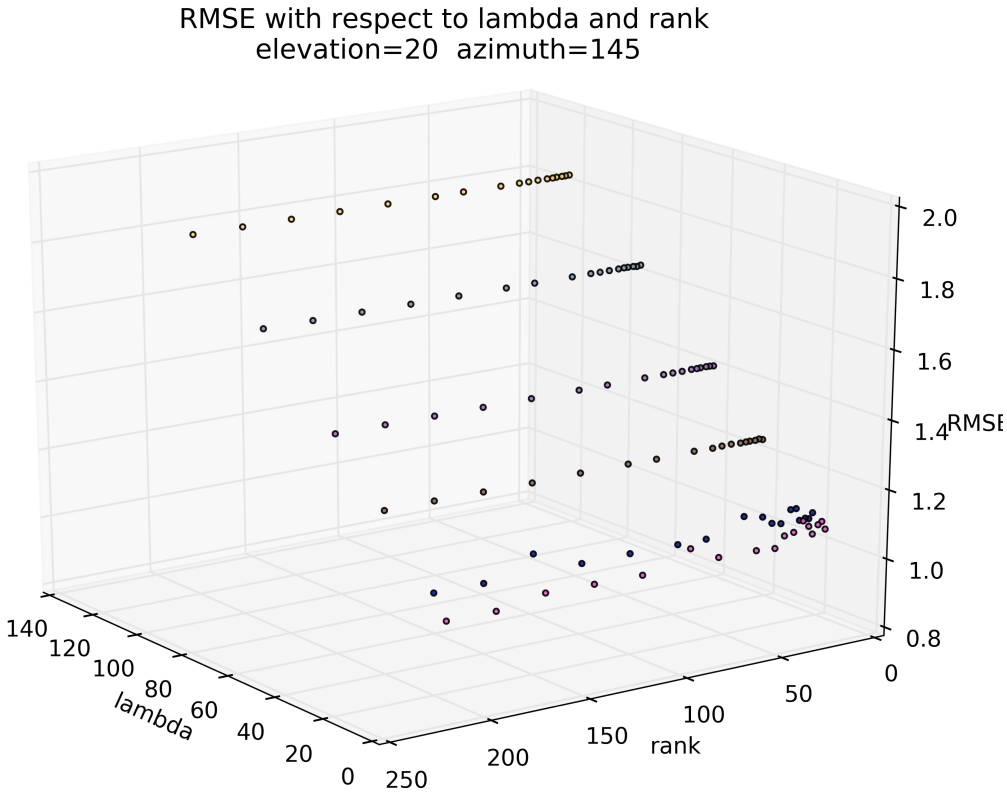


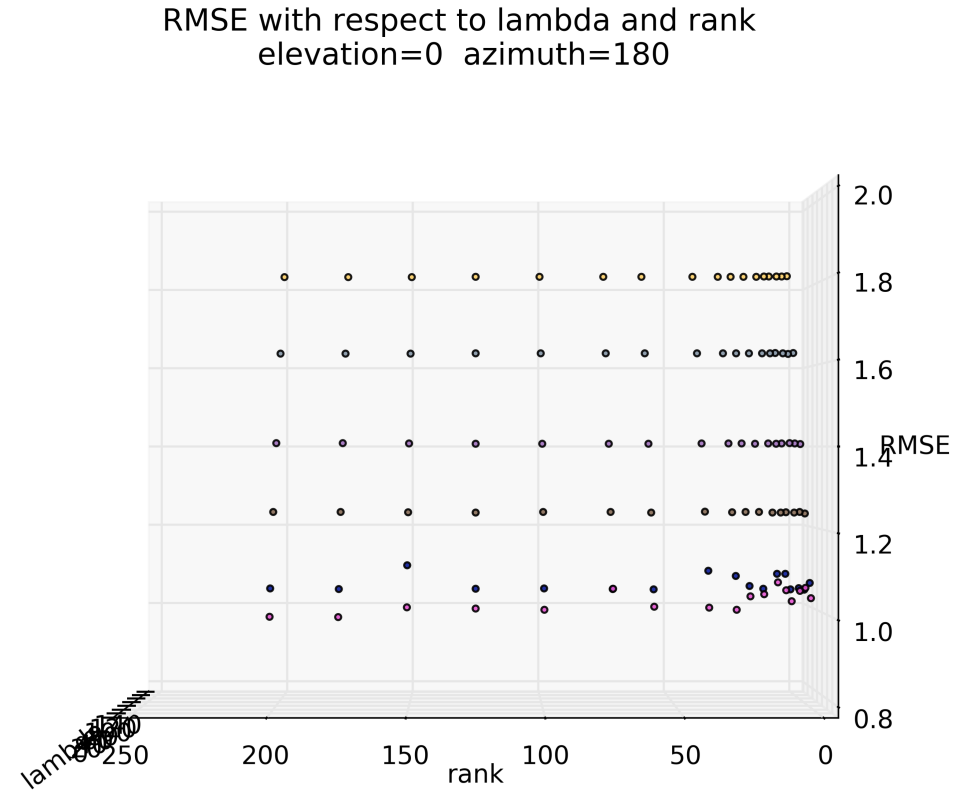


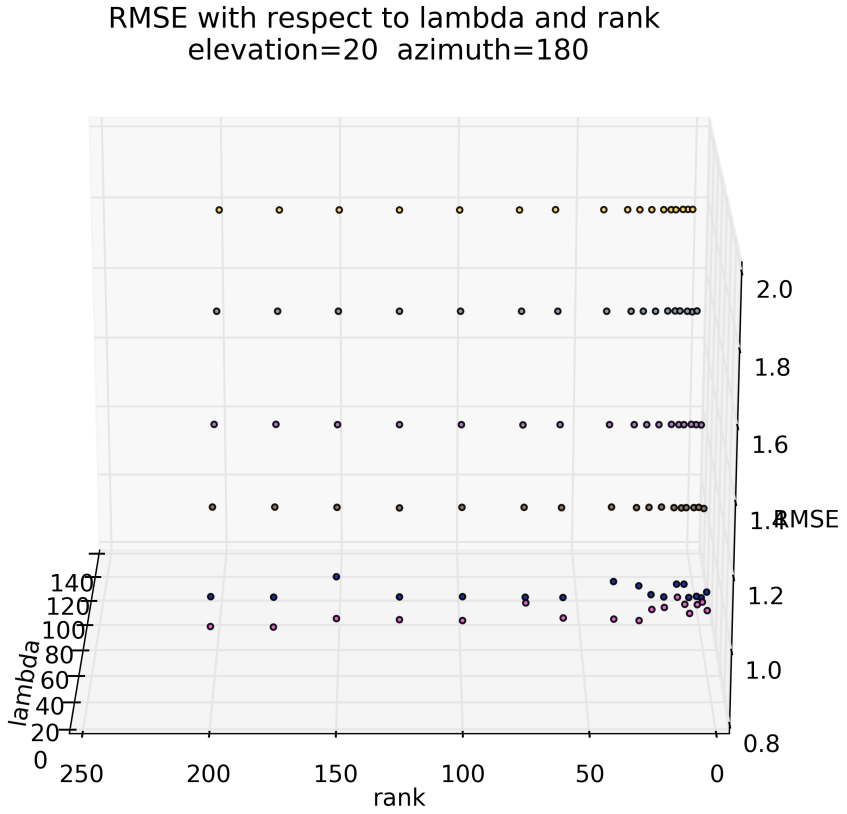
-Results from exploring the datasets if any
With our initial implementation, we realized that with fixed regularization parameter lambda, the RMSE decreased as the rank increased from 3 to 200 in a monotone fashion.

With fixed rank, the RMSE decreased in a monotone fashion as the regularizer lambda increased. These are summarized in the 3D plots below:

In terms of run speed, the lower the value of lambda, the longer and more iterations it takes to complete the algorithm, but the prediction results were more accurate.







-Lessons learned

We realized that the bottleneck of the algorithm is actually modifying the entries of the sparse matrices. This is because the for loop in python has a lot of overhead.

Under the advice of our instructor Da Kuang, we tried implement the algorithm with the Cython interpreter, which is a superset of the python programming language that supports static type declarations and thus reduces runtime overhead.

We statically typed the numpy ndarray when we tried to modify the entries in the high dimensional sparse matrices, and the speed of the algorithm improved dramatically.