Expedia Hotel Recommendations
Phase II
Yunwen Cai (yc3388), Zihan Ye (zy2293), Anna Zhou (yz3220)

1. Project Objective

In project phase II, we continued to use the Expedia Hotel Recommendations dataset to give recommendations of potential hotels users like. The dataset was given as a Kaggle data challenge and Expedia is interested to know which hotels users would prefer[1]. Moving forward from phase I, we considered three types of approximate nearest neighbors algorithms, k-d trees, k-means and locality sensitive hashing(LSH) and two latent factor models, SVD and SVD++. In order to get ideas of scalability of different algorithms, we implemented our models on randomly selected 10,000 and 100,000 unique users search records. There are 100 hotel clusters in the original datasets, which are treated as items. From the dataset, we got information of whether users clicked and booked hotels labeled by clusters. Our goal is to find an algorithm that can accurately predict whether users will book each hotel cluster given their previous search and book records.

2. Data Preparation

2.1 Data Sampling

For scalability purpose, we randomly selected 10,000 and 100,000 unique users from the original datasets and fetched all their records. 10k dataset is around 30MB and 100k dataset is around 300 MB. We only looked into three columns, 'user_id', 'is_booking', 'hotel_cluster'. 'Is_booking' is a binary column, with 1 represents the user made a search and a booking, and 0 represents the user made a search and DID NOT book. If the user never search the hotel cluster, there is no record in the dataset. Hotel_cluster is hotel cluster labels which ranges from 0 to 99. We used this information to build the user-item matrix. In addition, we want the user-item matrix to differentiate frequencies the user booked a hotel, and whether the user ever searched the hotel cluster. Thus, we performed the following operation: if the user booked the hotel, the corresponding user-hotel entry value is increased by one. If the user did not search or book the hotel cluster , the entry remains the same. At the end, all entries of search records in original datasets are increased by one, reflecting users' search interests. Missing records result in 0 in corresponding entries.

2.2 Train-test datasets split

We uniformly and randomly assigned probabilities between [0,1] to each entry of user-item matrix. Then entries with probabilities larger than certain probability were

selected for train datasets and the left entries were used as test datasets. We chose the split probability to be 0.2, so the train/test ratio was 0.8/0.2. Test datasets were remained untouched before cross-validation.

3. Model Implementation

3.1 Approximate Nearest Neighbors

Due to the nature of Expedia Hotel dataset, items, which are hotel clusters in this case, were independent of each other. Additionally, further information of each hotel cluster was not given. Thus, we concluded that there is not much interest to gain if we compared item similarity in item-based collaborative filtering algorithm. On the other hand, we expected that users may share some similarities in their tastes in choosing hotels.

Differently from phase I, we tried to find nearest neighbors for each user using approximate methods before collaborative filtering was applied. The purpose of finding nearest neighbors in a large dataset is to reduce the candidate set of nearest neighbors from the full set to a much smaller set. We used a library implemented in python called FLANN[4][5], which is known for performing fast approximate nearest neighbor searches in high dimensional spaces. K-d trees and k-means trees were implemented.

After finding nearest neighbors for each user, we used the following equation for predicting booking scores:

$$P_{u,i} = \frac{\sum_{k\ most\ similar\ users}(S_{u,k} * R_{u,i})}{\sum_{k\ most\ similar\ users} |S_{u,k}|}$$

$P_{u,i}$: *Prediction on hotel cluster i for user u*
$S_{u,k}$: *similarity between user u and other k most similar users*
$R_{u,i}$: *rates, 1or 0, k most similar users give on hotel cluster i*

The prediction on hotel cluster i for user u was calculated by the sum of rating top k users gave on item i. Each rating was weighted by similarities between user i and j. $S_{u,k}$ is the cosine similarity score.

3.1.1 K-d Trees

K-d trees can be used to perform nearest-neighbor searches by parallelly searching multiple and randomized trees.[7] Two parameters need to be tuned are the number of neighbors for each user and the number of randomized k-d trees to create.

3.1.2 K-Means Trees

The hierarchical k-means tree is constructed by splitting the data points at each level into K distinct regions using a k-means clustering, and then applying the same method recursively to the points in each region[6]. Parameters we need to tune are:

Num_neighbors - the number of neighbors for each user
Branching - the branching factor to use for the hierarchical k-means tree creation
Iterations - the maximum number of iterations to use in the k-means clustering stage when building the k-means tree.
Centers init - the algorithm to use for selecting the initial centers when performing a k-means clustering step.[5]

After a few trial runs, we found that the RMSE did not show significant change with different number of iterations and different algorithms for initial centers. However, using a small number of iterations instead of performing until converge significantly reduced run time. So we decided on fixing the number of iterations to 5 and randomly choosing initial centers. Then our analysis mainly focused on tuning the number of neighbors and the branching factor.

3.1.3 MinhashLSH

Minhash LSH is a technique that combines Locality Sensitive Hashing (LSH) with Minhash to find the nearest neighbors of a query set in a large collection of sets. The metric that the model uses to select candidate nearest neighbors is Jaccard Similarity, which equals to the area of the intersection divided by the area of the union. The general procedure is using a series of hashing function to create a signature matrix that contains minhash value for each user's record and outputting the candidate users whose minhash result satisfies the similarity threshold, or at least matches with the query in one band.

We implemented the MinHash LSH function from the public datasketch package written by Eric Zhu for this task.[3] First, we created a MinHash instance, given a specified number of permutation (hash function), for each user with the corresponding row in the user-item matrix. Each instance was then inserted into the MinHash LSH model with threshold parameter and number of permutation, which should be consistent with the MinHash instance. After that, we looped through each user as the query set to the LSH model and stored the resulted neighbors into a matrix. The final step was to run the user-based collaborative filtering again, but for each user's empty entry, calculate the predicted value by using the neighbors found before. The final

recommendations were made according to the resulted predicted matrix. We used the RMSE metric for accuracy and also examined on the coverage of recommendations.

In phase I of the project, we selected K users with top cosine similarity values for prediction. While trying out some random parameter values to test out the MinHash LSH model, we found that the size of the output nearest neighbors varies and may not meet the K constraint for collaborative filtering. So for the last step, we handled these in three cases:

1. The size of the nearest neighbors is equal to zero, which means the Jaccard similarity between the query's minhash value and that of other users does not meet the given threshold. In this case, we just performed the collaborative filtering procedure we used in phase I, making the prediction based on the K users with top cosine similarity with the current user.
2. The size of the nearest neighbors is less than or equal to K. Then we just used the information from these set of neighbors for prediction.
3. The size of the nearest neighbors is larger than K. We applied an additional step to choose K users in this neighbor set with top cosine similarity. The final prediction is calculated using the selected users after this combined process.

Besides, we mainly tuned three parameters for this entire model: the threshold value, the band/row size, and the number of permutations used. Threshold and band/row size are actually substitutes. If a threshold is given, the package will include an optimization step to minimize false positive and false negative rates. We also manually put in different values for band/row size and compared the model performance. After choosing the optimal threshold, we then examined whether the number of hash functions will have significant impact on the resulted RMSE. In addition, we also looked at the movement of running time when trying different parameter values.

3.2 Latent Factor Models

3.2.1 SVD

Singular Value Decomposition is a well known matrix factorization technique that reduces dimension by factorizing an $m \times n$ matrix X into three matrices as $X = USV^T$. Specifically, the U matrix represented the latent vectors corresponding to users and the V matrix represented the latent vectors corresponding to items. In the project phase I, we use the accuracy to tune the parameter k, which is the first k ($0 < k < r$) singular values in S we would like to keep. Then, by multiplying these two low-rank matrices together, we got a prediction matrix $\widehat{X}$ with missing entries in the original user-item matrix filled up.

However, the basic SVD mentioned in the above paragraph always raises difficulties due to the sparsity of our user-item matrix. (reference to paper 1). Moreover, this can easily result in overfitting. Recent works suggested that modeling directly on the available recorded ratings, while avoiding overfitting through a regularization model[8]:

$$\sum_{r_{ui} \in R_{train}} (r_{ui} - \widehat{r_{ui}})^2 + \lambda(b_i^2 + b_u^2 + \|q_i\|^2 + \|p_u\|^2)$$

where:

$$\text{constant } \lambda \text{ controls the extent of regularization;}$$
$$\text{estimated rating: } \widehat{r_{ui}} = \mu + b_i + b_u + q_i{}^T p_u$$
$$\text{bias } b_u\text{: average rating performed by user } u$$
$$\text{bias } b_i\text{: average rating for item } i \text{ performed rated by users}$$
$$q_i \text{: } \epsilon\ \mathcal{R}^f, \text{item factors}$$
$$p_u \text{: } \epsilon\ \mathcal{R}^f, \text{user factors}$$

Then we performed stochastic gradient descent to minimize the squared error on available ratings of the training set. In each iteration, parameters were updated through:
.

$$b_u \leftarrow b_u + \gamma(e_{ui} - \lambda b_u)$$
$$b_i \leftarrow b_i + \gamma(e_{ui} - \lambda b_i)$$
$$p_u \leftarrow p_u + \gamma(e_{ui} \cdot q_i - \lambda p_u)$$
$$q_i \leftarrow q_i + \gamma(e_{ui} \cdot p_u - \lambda q_i)$$

$$e_{ui} = r_{ui} - \widehat{r_{ui}}$$
$$\gamma\text{: learning rate to modify parameter in the opposite of the gradient}$$

3.2.2 SVD++

To cope with cold start problem, wherein many users offer too few ratings, SVD++ algorithm incorporates implicit feedback to relieve this problem. In our case, implicit feedback is a boolean value indicating whether the user searched or booked the hotel, regardless how many times the user searched or booked the hotel.
Now the rating estimation function is modified to:

$$\widehat{r_{ui}} = \mu + b_i + b_u + q_i^T(p_u + |I_u|^{-\frac{1}{2}} \sum_{j \in I_u} y_j)$$

$$y_j: implicit\ feedback$$
$$I_u: set\ of\ items\ rated\ by\ user\ u$$

We still performed stochastic gradient descent to minimize the regularization model presented in the SVD module.

We implemented both SVD++ and SVD algorithms from the open source Surprise python package. Iterations, learning rate and regularized parameter for both SVD++ and SVD models were tuned in the grid search.

3.3 Baseline

$$\widehat{r_{ui}} = b_{ui} = \mu + b_i + b_u$$
$$\mu: overall\ average\ rating$$
$$bias\ b_u: average\ rating\ performed\ by\ user\ u$$
$$bias\ b_i: average\ rating\ for\ item\ i\ performed\ rated\ by\ users$$

The baseline estimation exploits user u's bias to item i. It takes accounts of overall average ratings in user-item matrix, observed deviation of user u and item i.
We compute the baseline RMSE from the function provided in Surprise python package.

4. Evaluation

4.1 Accuracy:

We chose to use root-mean-square error (RMSE) to evaluate our approaches. For each run, we applied five-fold cross validation. We randomly assigned probabilities to user-item matrix to obtain train and test datasets. The overall RMSE was the average of the RMSE from five-fold cross validation.

4.1.1 Baseline:

The baseline RMSE is 0.5388 and 0.55886 for 10k and 100k datasets respectively.

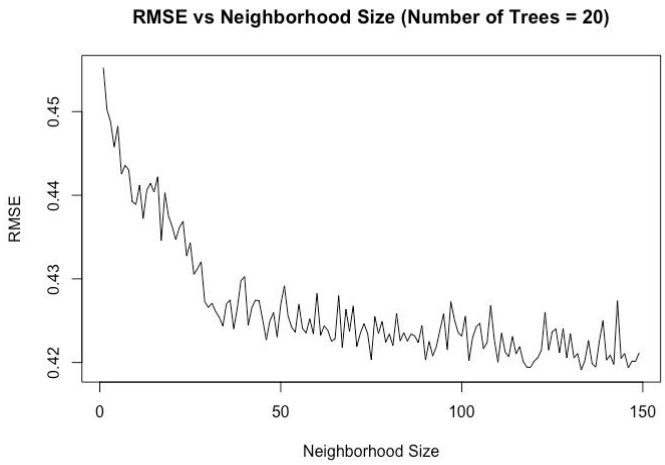
4.1.2 Approximate Nearest Neighbors:

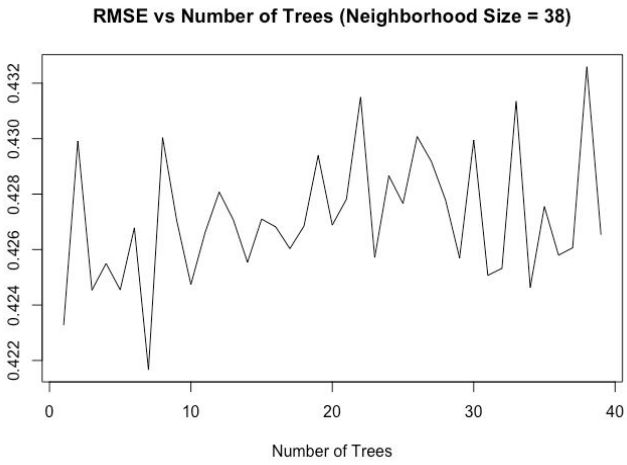
4.1.2.1 K-d Trees:

We first fixed the number of trees to be 20 and focused on how RMSE changed with the number of neighbors for each user. We can see from the plot that the RMSE has a decreasing trend as the neighborhood size increases when the neighborhood size is less

than 35. RMSE oscillates but the trend gets flat with different neighborhood size greater than 35. Since we want our approach to work fast, we inclined to work with a smaller neighborhood size.

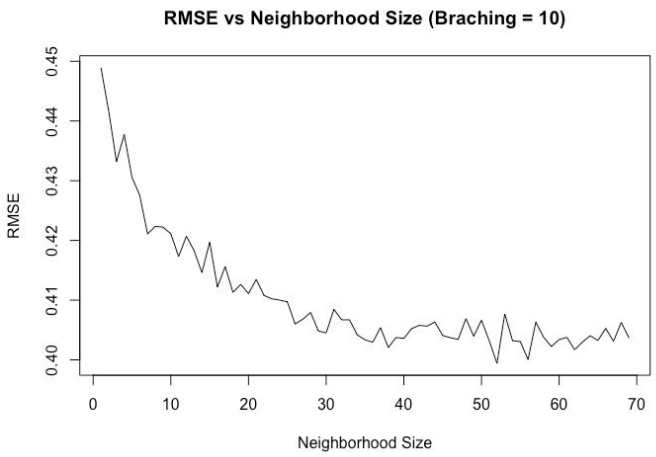
Next we fixed the neighborhood size to be 38 and focused on how RMSE changed with the number of trees we built. RMSE oscillates but shows no clear trend as we increased the number of trees. So we conclude that the number of trees does not have a significant impact on our prediction.

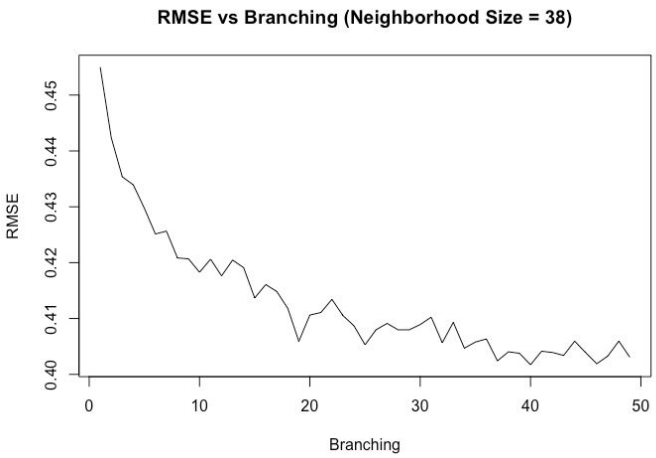




4.1.2.2 K-Means Trees

First we fixed the branching factor to be 10. The RMSE shows a decreasing trend as the neighborhood size increases to 35, and then stays roughly flat for neighborhood size greater than 35. Next we fixed the neighborhood size to be 38 (same as the neighborhood size for k-d trees). The RMSE exhibits a similar trend as the is was for different neighborhood size. The RMSE shows a decreasing trend as the branching factor increases to 35, and then stays roughly flat for branching factor greater than 35. For simplicity, we prefer a smaller neighborhood size and branching factor when there's no clear difference in RMSE.

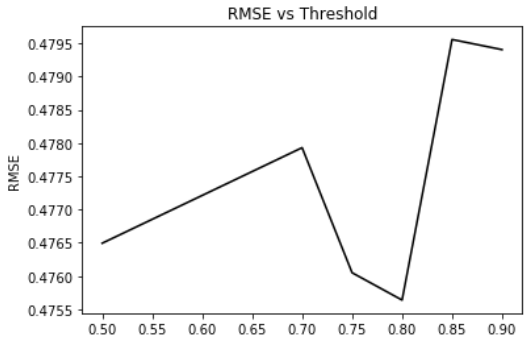




Comparing k-d trees and k-means trees, we can see that the RMSE for both methods decreases gradually as the neighborhood size increases up to 35, and then stays roughly flat. Overall, k-means trees have a lower RMSE than k-d trees. The RMSE of k-mean trees can be as low as 0.4, while the lowest RMSE of k-d trees is 0.42.

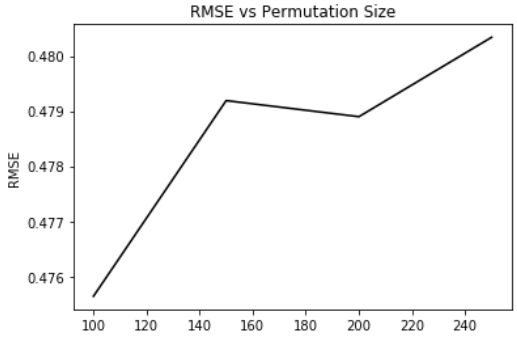
4.1.2.3 MinhashLSH

We first tuned the threshold parameter by fixing 100 hash functions with 5-fold cross-validation and tried 8 values in the range from 0.5-0.9. RMSE exhibits a decreasing trend as threshold increases. This makes sense since as we are more strict about the similarity level, more accurate neighbors will be selected and the accuracy should increase. As shown by the lots below, the optimal threshold is at 0.8.

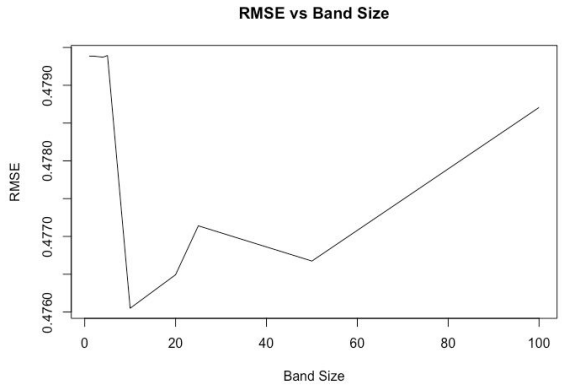


Then, we fixed the threshold at 0.8 and varied the number of permutations used. Intuitively, increase the permutation size will increase the model performance by decreasing the probability of collision and providing more accurate Jaccard similarity comparison result. However, too many permutations and in turn comparisons might

overfit the data, which reduces the accuracy. From the plots, the best performed permutation size is at about 100.



Lastly, we fixed the number of hash functions at the optimal value found above and tried several tuples of brand-row sizes in order to gain more insights than the threshold with optimized step provided by the package. The resulted RMSE trend shows that 10 gives the best performance. The band size being too narrow puts little constraints on candidate users since they only need to match with the query user in one band. If the band size is too large, it might cause overfitting on the training set and RMSE increases.



With optimal threshold and permutation size, our 5-fold cross validation RMSE result is 0.4756. With optimal band size and permutation size, our 5-fold cross validation RMSE result is 0.4760.

4.1.3 Latent Factor Models:

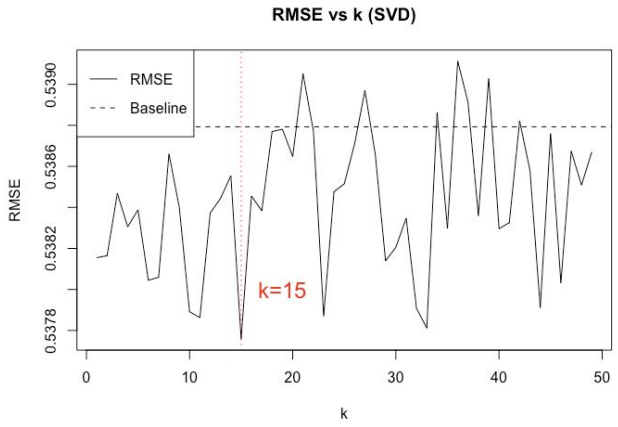
We applied SVD and SVD++ modules from the surprise-sklearn package for tuning parameters, computing RMSE accuracy and output recommendations. [9]

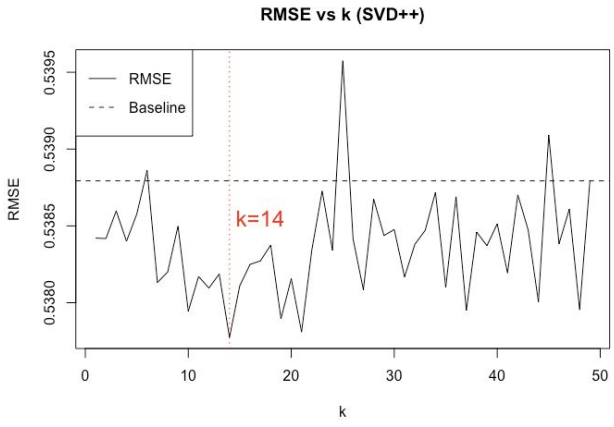
We used grid search to tune the SVD and SVD++ model and the best parameter combinations achieved the lowest RMSE. The running iteration is set to 10 epochs,

learning rate is 0.005 and regularization is 0.05. We applied the same parameters set to SVD, SVD++ and baseline algorithms.

For each train/test split, we ran SVD++, SVD on 10,000 users datasets with k values in a range from 1 to 50, and computed RMSE using 5-fold cross validation. The RMSE vs k plot is as follows:

Both RMSE plots do not show an obvious trend versus k value. With k value increases from 1 to 50, RMSE oscillated in a rough range from 0.5370 to 0.5395. Because baseline algorithm RMSE lies between this range and there is no clear increase or decrease trend in RMSE versus increases in k, for both SVD++ and SVD algorithms. As running time increases with increase in k, we chose k value as 15 for SVD and k value as 14 for SVD++ algorithm. When k is 15, RMSE of SVD is 0.5377. When k is 14, RMSE of SVD++ is 0.5377. We applied these two k values in SVD and SVD++ algorithms respectively on 100k users dataset for scalability purpose.



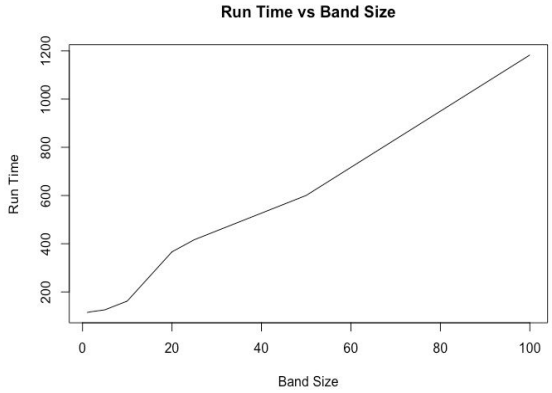


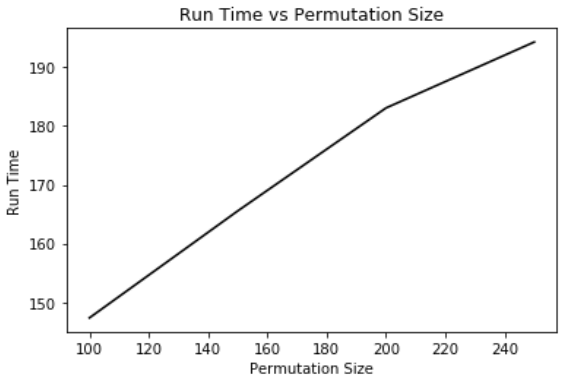
4.2 Scalability:

4.2.1 Approximate Nearest Neighbors

To test the running time of ANN, we chose to user a neighborhood size of 38, 20 trees for k-d trees and branching factor of 20 for k-means trees. The average running time to make a recommendation using k-d trees for our dataset of 10000 users is 49 seconds, which is the same as using k-means trees. We tried to apply our approaches to a larger dataset with 100000 users and analysis the running time, but too much computation was required, which made it impossible for our personal computers to complete. But the impossibility implied that when the dataset gets larger, the complexity of the algorithm increases dramatically. With threshold at 0.8 and permutation size at 100, the average

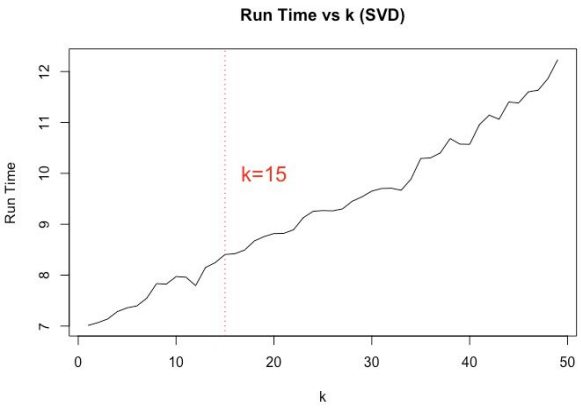
running time is 150.53. If using customized band size at optimal with permutation size 100, the average running time is 170.06. As band size increases, running time has an increasing trend. This combined technique of minhash LSH and collaborative filtering requires large computational power, so we are only able to get results from dataset of 10K users.

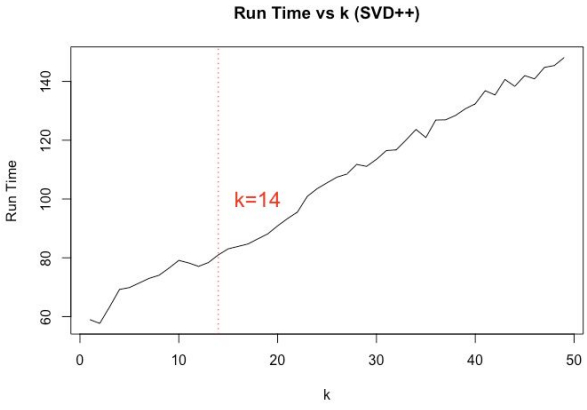




4.2.2 Latent Factor Models:

We plot the running time of both SVD and SVD++ algorithms on 10k users dataset with k increases from 1 to 49. When k is 15, the running time of SVD is 8.4s. When k is 14, running time of SVD++ is 81s. With every k, running time of SVD++ is significantly larger than that of SVD. Running time increases almost linearly with increases in k.





We tested the scalability for SVD and SVD++ models by comparing both algorithms' running time on 10,000 users and 100,000 users datasets. Running time of chosen k values on datasets are presented in the following table.

Running Time (s)	SVD (k=14)	SVD++ (k=15)
10,000	8.41	81.00
100,000	86.88	805.83

With dataset size increases in 10 folds, both SVD and SVD++ algorithm running time increase 10 folds.

4.3 Coverage:

In order to examine the coverage of our prediction and the original dataset, we created a prediction matrix by splitting the dataset into 5 folds and perform prediction for each fold using the rest 4 folds. Then we put the predicted value for each fold together to get an overall prediction for the original dataset. Then we looped over each user and marked the top 5 recommendations in our prediction matrix and original matrix.

By taking the unique values of all recommendations, we were able to find out the number of hotel clusters covered. This evaluation procedure gave us a 100% coverage rate using the original matrix. All algorithms implemented in this project can output 100 unique labels, meaning that each hotel type is recommended at least once to some users. For the purpose of this project, we hoped to build a recommender system that is able to give personalized hotel type recommendations to each user instead of just the popular items, so the resulted coverage using user-based collaborative filtering matches with our expectations.

Besides, we also compared the top 5 recommendations given by the prediction matrix and the original matrix for each user. We considered our predicted recommendation to be similar to the recommendation given by original matrix if they have one overlapped items. When we applied k-d trees with a neighborhood size of 38 and 20 trees before collaborative filtering, 3396 out of 10000 users received similar recommendations. When we applied k-means trees with a neighborhood size of 38 and branching factor of 20 before collaborative filtering, 4421 out of 10000 users received similar recommendations.

MinHash LSH has a very impressive performance on recommendation accuracy and coverage. Compared to the original dataset, the predicted user-item matrix gives the same general popular items. Union all the 5 recommendations it generates for the 10K users, the entire recommendation set covers all the hotel clusters, which matches our objective. This combined model is also able to give similar recommendations for 9864

users out of 10000, and even when we compare the exact 5 recommendations, 9567 out of 10000 users received the same as the original dataset provided.

Both SVD++ and SVD present good recommendation coverage. When we applied SVD++ with factor of 14, 7688 out of 10,000 and 76964 out of 100,000 users received similar recommendations. When we applied SVD with factor of 15, 7711 out of 10,000 and 76963 out of 100,000 users received similar recommendations. Thus, both SVD and SVD++ algorithms presented relative high and consistent coverage rates on datasets with different scales.

5. Conclusion

Moving forward from project phase I, we continued to explore different recommendation models to overcome disadvantages of user-based and model-based collaborative filtering. In phase II, we applied three approximate nearest neighbors algorithms, k-d trees, k-means, locality sensitive hashing(LSH) and two latent factor models, SVD and SVD++ on 10k and 100k users datasets. We aim to improve accuracy and test different algorithms scalability beyond the phase I. We evaluated algorithms by computing RMSE values and predicted results coverage.
Although, k-d trees and k-means algorithms presented relatively high accuracy compared to other performed algorithms, both coverage results are relatively low compared to those of SVD and SVD++ algorithms. LSH presents the best recommendation performance with relatively good prediction accuracy, but it does not significantly decrease the running time when combined with collaborative filtering. Due to the limitations on the laptop's hardware, we were able to apply SVD and SVD++ on both 10k and 100k datasets, and we were not be able to apply k-d trees, k-means and LSH algorithms on 100k datasets. The running time of SVD and SVD++ algorithm scaled 10 times with the datasets size, while the running time of the test algorithms takes forever. This observation, on the other hand, reflects SVD and SVD++ algorithms' good scalability.

6. Next Step

User-based collaborative filtering can be very time-consuming dealing with large amount of users. It calculates pairwise similarities with all rest data, which cost $O(n^2)$ complexity and space. The increase in complexity can be almost exponential for datasets with millions of users and items. Even combined with an LSH step to reduce neighborhood size, the collaborative filtering procedure still take a long time. Hence a

more efficient alternative for prediction can be a future goal. Moreover, with k-d trees and k-means trees, the number of trees increases as the the number of users increases, which leads to a dramatic increase in running time. In this project we were not able to run collaborative filtering with a dataset of 100000 users because of the limitation of computing power of our personal computers. In the future we could exploit resources such as clusters or computer labs of the university to analyze our algorithms.

7. Reference

1. https://www.kaggle.com/c/expedia-hotel-recommendations/data
2. https://cambridgespark.com/content/tutorials/implementing-your-own-recommender-systems-in-Python/index.html
3. https://ekzhu.github.io/datasketch/index.html
4. https://www.cs.ubc.ca/research/flann/uploads/FLANN/flann_manual-1.8.4.pdf
5. https://github.com/primetang/pyflann
6. https://www.cs.ubc.ca/research/flann/uploads/FLANN/flann_visapp09.pdf
7. https://www.cs.ubc.ca/research/flann/uploads/FLANN/flann_pami2014.pdf
8. https://datajobs.com/data-science-repo/Recommender-Systems-[Netflix].pdf
9. http://surprise.readthedocs.io/en/stable/getting_started.html