Repository link: https://github.com/conradsmi/ML-hw3

Task 1

The k-means code itself is located in kmeans.py, and the code relating to distance measurements is in distances.py. NumPy was used to significantly increase the speed of both distance calculations and k-means related algorithms, sometimes showing a multiple times improvement (it was otherwise considerably slow, ranging between 10-30 minutes depending on the distance metric chosen). I did not use any other outside libraries.

If the “results” folder does not exist in the task1 directory, centroids for each cluster and each distance measurement function will be calculated and saved into a csv file, with the last line being the total SSE and total number of iterations taken. Otherwise, centroids will be loaded from those csv files, and the predictive accuracy of the total group of clusters will be calculated. Results for every question (and part, where relevant) are saved in folders “results_q<question number>p<part number, if any>”.

Q1. According to the numbers shown in “task1/results_q1q2”, Euclidean distance had the smallest SSE, with roughly 391697. Next was cosine similarity had 393435, followed by generalized jaccard similarity at 393693.

Q2. By renaming “task1/results_q1q2” to “task1/results” and running the kmeans.py script, we can see that the accuracy of the model using euclidean distance is 63.34%, followed by the one using generalized jaccard similarity at 62.06%, followed by the one using cosine similarity at 59.89%. Euclidean is clearly better when looking at prediction accuracy alone.

Q3. According to “task1/results_q3”, generalized jaccard similarity converged quickly at 17 iterations and has a SSE of 394957. Cosine was next at 21 iterations and has a SSE of 391979, followed by Euclidean at a considerably large 85 iterations and a SSE of 389447.

Q4. When the stop criteria is “no change in centroid position” (“task1/results_q4p1”), generalized jaccard similarity takes 64 iterations and has a SSE of 393477, cosine similarity takes 118 iterations and has a SSE of 393742, and euclidean takes 134 iterations and has an SSE of 389448.

When the stop criteria is “SSE increases between iterations” (“task1/results_q4p2”), generalized jaccard similarity takes 23 iterations and has a SSE of 393577, cosine takes 24 and has a SSE of 393742, and euclidean takes 41 and has a SSE of 389448.

When the stop criteria is “iteration counts reached” (“task1/results_q4p3”), with an iteration count of 100, generalized jaccard similarity’s SSE is 390870, euclidean’s SSE is 391112, and cosine’s SSE is 395419.

Q5. From the results, it is apparent that using Euclidean distance results in the lowest SSEs across the board, generally ignoring stop criteria. However, it also tended to be the slowest algorithm to converge by far; for Q3, the model that used euclidean distance took 5x longer to converge than the one that used generalized jaccard similarity, for a marginally better SSE. Generally speaking, all three models had roughly the same predictive accuracy (ranging between 58-63% usually). Both the differences in predictive power and SSE between the models may be considered negligible in a lot of cases, but the training time difference was critically different between them.

I would certainly use generalized jaccard similarity in most time-sensitive or computationally-expensive training routines, but would consider using euclidean distance for applications where the absolute greatest predictive power is needed.

Task 2, Q3

For this task, I made heavy use of the Surprise library for most algorithms and models. Different experiments can be created by supplying the custom training routine with a list of instantiated Surprise library models, a list of accuracy metrics to compute (RMSE and MAE for this task typically), the number of times to train each model / number of folds for k-fold cross validation, and the dataset itself (loaded using the Dataset class from the Surprise library). The plot function works similarly and will produce a bar chart given each model's score for each accuracy metric, the names of each metric, the names of each model, and the width of the bars to be displayed.

All parts can be ran by uncommenting or commenting each experiment and looking at the raw accuracies table for each. Each column of the table is an average score over the number of runs provided for each accuracy metric (in order of the list provided), and each row is a different model.

Part D. By looking at the accuracy table generated in part C ("results_c.txt"), we see that the item-based collaborative filtering model performs comfortably better than the other two, having an average RSME of 0.9346, with user-based collaborative filtering following it at 0.9684 and PMF following that at 1.0075.

Part E. By looking at the plot "results_e.png", we can see that each similarity metric affects both models very similarly. The average RSMEs are the same for both when cosine is used, while their MAEs are marginally different (0.768 for user-based and 0.773 for item-based). The item-based model performed somewhat better than the user-based model when MSD was used, with 0.935 RMSE/0.721 MAE and 0.968 RSME/0.744 MAE respectively. Both models performed virtually the same using pearson as they did using cosine. It seems the item-based model using MSD is the best option.

Part F. By examining "results_f_user.png" and "results_f_item".png, it can be said that k has a somewhat substantial impact on performance. The user-based model had the best results at k=15 and the item-based model had the best results at k=70.

Part G. By examining the plots from part F ("results_f_user.png" and "results_f_item.png") as well as the raw tables in "results_g.txt", it is apparent that both models use radically different values for k, with item-based models using k=70 (with RMSE of 0.9316) and user-based models using k=15 (with RMSE of 0.9623).