Data Mining HW3

# Asish Addanki

### ASUID:1230916442

The drive link for task1: <https://drive.google.com/drive/folders/1zrFBTAu49bNPn3nsj9yciCCkAtuoMK08?usp=sharing>

**Task 1:**

1. The obtained values of SSE after running it against kmeans clustering with Euclidean, cosine and jaccard similarities are as follows:

SSE (Euclidean): 5565072.345041299

SSE (Cosine): 5598833.603724479

SSE (Jaccard): 6679332.000000004

Lower SSE values suggest better clustering quality, indicating that data points within each cluster are closer to the centroid. In this specific instance, EuclideanKmeans exhibits the lowest SSE, implying more welldefined clusters for the dataset.

Therefore, based on the SSE values, the conclusion is that Euclidean Kmeans offers superior clustering quality for the given data.

1. The obtained accuracies for the corresponding below SSE’s are:

Euclidean similarity SSE: 5569451.3558367565

Cosine similarity SSE: 5604872.572570423

Jaccard similarity SSE: 6448978.453950518

Accuracy (Euclidean): 0.5056505650565056

Accuracy (Cosine): 0.5533553355335533

Accuracy (Jaccard): 0.2222222222222222

The code performs Kmeans clustering on a dataset using three different similarity measures: Euclidean, Cosine, and Jaccard. It initializes the KMeansClustering object, applies Kmeans with each similarity measure, assigns labels to clusters, and computes the accuracy of clustering. The Index Error likely occurs due to incorrect indexing of `true\_labels`, possibly indicating a missing or incorrect column reference.

From here, we can see that the accuracy of the cosine is highest which is 0.53 and since it is the highest, cosine similarity has the better accuracy among others.

1. The KMeans class is defined with methods to compute distances based on different similarity metrics (Euclidean, Cosine, Jaccard) and perform Kmeans clustering. The fit method iterates until convergence or the maximum number of iterations is reached, updating centroids and calculating SSE. The iterations for convergence are:

Iterations for convergence:

Euclidean: 62

Cosine: 32

Jaccard: 2

Based on the provided iterations for convergence, the JaccardKmeans method requires the fewest iterations and time to converge, followed by CosineKmeans, and then EuclideanKmeans. Hence here, Euclideankmeans takes more iterations here.

1. After reading the data from CSV files and standardizing the features, the code initializes `KMeans` objects for each similarity metric. It then applies the Kmeans algorithm using each similarity metric, iterating until convergence or reaching the maximum number of iterations. The SSE and the number of iterations required for convergence are computed and printed for each method.

The code satisfies the three termination conditions outlined in the question:

1. No Change in Centroid Position: The `\_check\_convergence` method ensures that the algorithm stops iterating if there is no significant change in the positions of the centroids between consecutive iterations. This condition prevents unnecessary iterations when the centroids have stabilized.

2. Increase in SSE in the Next Iteration: The `\_check\_convergence` method also stops the iteration if the SSE value increases in the next iteration compared to the previous one. This condition indicates that further iterations may lead to worse clustering results, prompting an early termination to prevent divergence.

3. Maximum Preset Value of Iteration: The algorithm terminates if the maximum preset number of iterations is reached. This condition prevents the algorithm from running indefinitely, ensuring that the clustering process stops after a predefined number of iterations, even if convergence has not been achieved.

Comparing the SSEs of EuclideanKmeans, CosineKmeans, and JaccardKmeans under different termination conditions sheds light on their convergence behavior and clustering quality. By examining how each method responds to termination conditions such as no change in centroid position, an increase in SSE in the next iteration, or reaching the maximum preset number of iterations, we can assess their effectiveness in clustering the given dataset. Below are the results:

SSE (Euclidean): 5591786.700961408

Iterations (Euclidean): 36

SSE (Cosine): 5585647.416890697

Iterations (Cosine): 29

SSE (Jaccard): 6437092.910866872

Iterations (Jaccard): 2

Hence, Cosine<Euclidean<jaccard

**Summary of the results:**

* SSE Comparison:

EuclideanKmeans yields the lowest SSE, indicating better defined clusters for the dataset.

* Accuracy Comparison:

cosine K means demonstrates the highest accuracy, suggesting better alignment with true labels

* Iterations for Convergence:

Euclidean K means requires the highest number of iterations for convergence compared to Cosine and Jaccard.

* Impact of Iterations on Convergence:

Euclidean K means exhibits slower convergence, while Cosine and Jaccard converge more quickly.

* Consideration of Terminating Conditions:

Despite different iteration counts, terminating conditions have consistent effects on convergence patterns.

* SSE Results with Terminating Conditions:

Euclidean K means maintains lower SSE values even with additional terminating conditions.

General Analysis:

Euclidean distance appears more effective for this dataset based on both SSE and accuracy.

**Takeaways:**

**1) Clustering Quality:**

* The SSE values indicate that EuclideanKmeans exhibits the lowest SSE, suggesting more welldefined clusters for the dataset.
* While CosineKmeans has a higher SSE compared to EuclideanKmeans, it achieves the highest accuracy among the three similarity metrics, indicating better clustering performance in terms of accuracy.
* JaccardKmeans, despite having a higher SSE than EuclideanKmeans and CosineKmeans, demonstrates the lowest accuracy, suggesting lower clustering quality.

**2) Convergence Behavior:**

* JaccardKmeans converges in the fewest iterations, followed by CosineKmeans and then EuclideanKmeans.
* The termination conditions, including no change in centroid position, an increase in SSE in the next iteration, and reaching the maximum preset number of iterations, are effectively met by the Kmeans algorithm for all three similarity metrics.

**3) Comparison of SSEs under Different Termination Conditions:**

* Under various termination conditions such as no change in centroid position, an increase in SSE in the next iteration, and reaching the maximum preset number of iterations, the SSEs and iterations of EuclideanKmeans, CosineKmeans, and JaccardKmeans are compared.
* The results show that EuclideanKmeans has the lowest SSE and requires more iterations compared to JaccardKmeans and CosineKmeans.

4) **Robustness to Outliers:**

* Euclidean distance is sensitive to outliers, as it computes the straightline distance between points. Outliers can significantly impact the position of centroids and the resulting clusters.
* Cosine similarity is more robust to outliers since it measures the angle between vectors rather than the distance. Outliers have less influence on the clustering process as long as their direction is not aligned with the cluster structure.
* Jaccard similarity may be affected by outliers in binary data, particularly if outliers introduce noise in the dataset. However, its impact can be mitigated by preprocessing or outlier detection techniques.

In summary, while EuclideanKmeans offers superior clustering quality in terms of SSE, CosineKmeans achieves better accuracy. JaccardKmeans converges in the fewest iterations but exhibits lower clustering quality. Overall, the choice of similarity metric should be based on the specific requirements and characteristics of the dataset, considering factors such as clustering quality, convergence efficiency, and computational complexity.

**Task 2:**

Kaggle link for the task2: <https://www.kaggle.com/code/asishaddanki/dm-hw3-task-2>

1. .
2. .
3. The provided code in the link performs collaborative filtering on a movie rating dataset using three algorithms: Probabilistic Matrix Factorization (PMF), User-based Collaborative Filtering (User CF), and Item-based Collaborative Filtering (Item CF). It utilizes the Surprise library to load the dataset and define the algorithms with their configurations. The `evaluate\_algorithm\_performance` function cross-validates each algorithm and computes Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE). The results are then printed for each algorithm. Finally, the MAE and RMSE values for each algorithm are assigned to respective variables and printed. The code facilitates the comparison of algorithm performance in terms of prediction accuracy (MAE, RMSE) across the three collaborative filtering methods. Below are the results:

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

1. This code defines a dictionary named `algorithm\_results` to store the Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE) results for each algorithm: User-based Collaborative Filtering, Item-based Collaborative Filtering, and Probabilistic Matrix Factorization. It then iterates over the dictionary and prints the mean MAE and RMSE for each algorithm. Finally, it compares the models based on their MAE and RMSE values, determining which algorithm has the minimum MAE and which one has the minimum RMSE. This comparison provides insights into the relative performance of the algorithms in terms of prediction accuracy. The results are:

A black text with numbers

Description automatically generated with medium confidence

1. This code performs collaborative filtering using the K-Nearest Neighbors (KNN) algorithm with different similarity metrics: cosine, MSD (Mean Squared Difference), and Pearson correlation coefficient. It evaluates the performance of both user-based and item-based collaborative filtering approaches using these similarity metrics.

The `cross\_validate` function from Surprise library is used to perform 5-fold cross-validation, computing the Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE) for each combination of algorithm and similarity metric. The results are then plotted using bar charts for User CF MAE, User CF RMSE, and Item CF MAE against the similarity metrics. The visualization allows for the comparison of algorithm performance across different similarity metrics, providing insights into which metric yields the best results for collaborative filtering.

A graph with numbers and a few red and blue bars

Description automatically generated with medium confidence

1. Loading Data: Data from a movie ratings CSV file is loaded using the Surprise library.
2. Defining Parameters: It defines a list of numbers of neighbors (num\_neighbors\_list) to iterate over.
3. Evaluation Function: A function evaluate\_algorithm\_performance is defined to evaluate the performance of each algorithm (user-based and item-based CF) using cross-validation. It computes the Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE) for each configuration.
4. Algorithm Evaluation Loop: The code iterates over each number of neighbors in num\_neighbors\_list. For each value, it trains both user-based and item-based KNNBasic models with the specified number of neighbors and evaluates their performance using the defined function.
5. Results Collection: MAE and RMSE values are stored for plotting.
6. Plotting: Subplots are created to visualize MAE and RMSE vs. neighbor counts.

The code assesses how the number of neighbors impacts the performance of user-based and item-based collaborative filtering algorithms using KNNBasic from the Surprise library. Below are the results:

A graph with numbers and lines

Description automatically generated

A graph of a number of neighbors

Description automatically generated

1. Explanation of the code:
2. Parameter Grid Definition: The code defines a parameter grid `param\_grid` containing a range of values for the number of neighbors (`'number\_of\_neighbors'`).
3. Result Containers Initialization: Empty dictionaries `user\_cf\_results` and `item\_cf\_results` are initialized to store the Root Mean Squared Error (RMSE) results for user-based and item-based collaborative filtering, respectively.
4. Neighbor Count Iteration: The code iterates over the values of the number of neighbors specified in the parameter grid.
5. Algorithm Evaluation: Within each iteration, user-based collaborative filtering and item-based collaborative filtering models are created using the `KNNBasic` algorithm from the Surprise library. The models are trained and evaluated using the `evaluate\_algorithm` function, which computes the RMSE for each model.
6. Best Parameter Identification: After evaluating each model, the code identifies the best number of neighbors for user-based collaborative filtering (`best\_k\_user\_cf`) and item-based collaborative filtering (`best\_k\_item\_cf`) based on the neighbor count with the lowest RMSE.
7. Results Printing: The code prints the best number of neighbors for both user-based and item-based collaborative filtering, and then checks if they are the same or different. This comparison provides insights into whether the optimal number of neighbors is consistent across both collaborative filtering approaches.

A screenshot of a computer

Description automatically generated

**Links:**

task1:<https://drive.google.com/drive/folders/1zrFBTAu49bNPn3nsj9yciCCkAtuoMK08?usp=sharing>

Kaggle link for the task2: <https://www.kaggle.com/code/asishaddanki/dm-hw3-task-2>