Amazon reviews recommender systems

Anna Gotti[1]

Department of Statistical Sciences, University of Padua anna.gotti16@gmail.com

Sommario This report aims to build recommender systems using an *item-based collaborative filtering* approach for data from Amazon.com. In particular, different distance measures applied to *items* will be implemented and combined with the *k-means* clustering method.

Keywords: Utility matrix · Collaborative filtering · Predictions · RMSE · Similarity measures · k-means

1 Introduction

In the age of digital information and personalized experiences, recommender systems have become essential tools in domains such as e-commerce, streaming services, and social platforms. Their primary objective is to suggest relevant content or items to users by leveraging past interactions, preferences, or behavior. Among the various recommendation techniques, collaborative filtering stands out for its effectiveness and scalability, particularly in systems where content-based features are scarce or unavailable. This project focuses on implementing and evaluating a collaborative filtering model based on item-based predictions for data from Amazon.com, which are characterized by a high number of *users* compared to the number of *items*. The approach estimates missing ratings by analyzing similarities between items rated by users and using this information to infer preferences. Several similarity measures have been explored and compared, including Cosine Centered Similarity (CCS), Jaccard Similarity (JS), Jaccard for Bags (JSB), and Euclidean Distance (ED). Each of these measures offers different characteristics in capturing relationships between items.

In addition to measuring the performance of these similarity metrics, the study also investigates the impact of applying clustering algorithms to group similar items before prediction. This aims to reduce the computational burden and potentially improve the quality of recommendations by focusing on local item neighborhoods.

The performance of the proposed methods is assessed through two key aspects: **accuracy**, evaluated via the Root Mean Squared Error (RMSE), and **efficiency**, measured in terms of execution time. By comparing clustered and non-clustered approaches, as well as different distance metrics, this work provides insight into the trade-offs between computational cost and prediction accuracy.

Overall, the objective of the project is twofold: to implement a flexible framework for collaborative filtering using various similarity metrics, and to analyze the effects of clustering on both the quality and speed of predictions in recommender systems.

The recommendation problem can be defined as predicting a *user*'s response to a new *item*, based on the information that *users* and *items* carry, suggesting new and original items to the *users* for which the predicted *rating* is high. The chosen approach is ***item-based collaborative filtering (CF)***, where the key idea is that a *user* $\boldsymbol{u}$ rates two items $\boldsymbol{i}$ and $\boldsymbol{j}$ similarly if other users have given similar ratings to both items.

A major advantage of the *CF* approach is that items for which no information is available, or it is difficult to obtain, can still be recommended to users through the feedback from other users. Therefore, the *user-item* rating stored in memory can be directly used to predict ratings for new items. The *item-based* variant is based on the idea that a small number of highly similar users is preferable to a large number of users with unreliable similarity measures. For this reason, when the number of users is much larger than the number of items, as is the case with Amazon.com data, an *item-based* approach produces more accurate recommendations. In terms of efficiency, *item-based* recommendation is also better when the number of users exceeds the number of items, as it requires significantly less memory and time to compute similarity weights.

Various similarity measures can be used. The choice of the measure is crucial because, under the same information conditions, different measures can lead to different results. In general, the choice should be based on the working context and data characteristics. For each distance type chosen, a rating prediction will be computed and evaluated in terms of accuracy.

Additionally, the *k-means* clustering method will be implemented, which groups *items* based on distances to representative cluster centroids. These groups may reveal patterns not otherwise detectable. The choice to implement clustering algorithms within recommender systems arises from the need for greater computational efficiency, as they reduce the number of comparisons by limiting calculations to entities within the same *cluster*.

The goal of the report is to evaluate how combinations involving different distance measures and the presence or absence of *clustering* affect both efficiency and predictive accuracy. Efficiency will be evaluated in terms of computation time, and accuracy in terms of *RMSE*.

2 Starting Point

For the procedures described in the following section, we refer to Chapter 4 of book [**1**] and Chapter 2 of book [**2**].

The key elements in implementing a recommender system include:

- normalization of ratings and handling of missing data;
- computation of similarity weights;
- selection of the most similar items.

Once the utility matrix is built, i.e., the matrix containing the ratings indexed by users (rows) and items (columns), the high number of missing values becomes

evident. This sparsity will be handled by normalizing ratings through mean-centering, where the idea is to determine whether a rating is above or below the item's average. Specifically, using an *item-based* approach, the centering will be around the mean for each *item*, setting missing ratings (i.e., those not rated by a user) to 0 — equivalent to the item's average. For the remaining ratings, given an uncentered rating r_{ui} for user u and item i, a centered version $h(r_{ui})$ is obtained by subtracting the item's mean rating $\bar{r}_i$ from r_{ui}, where $\bar{r}_i$ is the average rating given to item i by a set of users U_i:

$$h(r_{ui}) = r_{ui} - \bar{r}_i.$$

This normalization technique is used for the *item-based* recommender system, where the **predicted rating** r_{ui} for item i and user u is written as:

$$\hat{r}_{ui} = \bar{r}_i + \frac{\sum_{j \in \mathcal{N}_u(i)} w_{ij}(r_{uj} - \bar{r_j})}{\sum_{j \in \mathcal{N}_u(i)} |w_{ij}|}, \tag{1}$$

where $\bar{r}_i$ is the mean for item i, w_{ij} is the weight of item j in the prediction of item i, and $\mathcal{N}_u(i)$ is the set of items rated by u that share at least one user in common with i. The presence of $\bar{r}_i$ is due to the previous mean-centering operation.

The weight w_{ij} will be calculated using various similarity measures, which will be compared based on the accuracy of the predicted ratings. In particular, this weight emphasizes that more similar items are given more importance in the final prediction.

When the clustering method is introduced, a sub-partition of the utility matrix will be considered. Then, in formula (1) for the predicted rating, $\mathcal{N}_u(i)$ will be replaced with $\mathcal{N}_u(i)'$, i.e., the set of items rated by user u that belong to the same cluster as item i and have at least one user in common with i. The predicted rating in this case becomes:

$$\hat{r}_{ui} = \bar{r}_i + \frac{\sum_{j \in \mathcal{N}_u(i)'} w_{ij}(r_{uj} - \bar{r_j})}{\sum_{j \in \mathcal{N}_u(i)'} |w_{ij}|}. \tag{2}$$

2.1 Measuring the Accuracy of Rating Predictions

To evaluate the performance of the various versions of the recommendation system that will be implemented, the dataset was split into two parts. The *training set* R_{train}, which contains 80% of the observations and is used to train the model to make predictions, and the *test set* R_{test}, containing 20% of the observations, which will be used for comparison with predictions based on the *training*.

To assess the quality of the results obtained in this project, an accuracy metric is used to evaluate the rating predictions: the **Root Mean Squared Error (RMSE)**, defined as

$$RMSE = \sqrt{\frac{1}{|R_{test}|} \sum_{(u,i) \in R_{test}} (\hat{r}_{ui} - r_{ui})^2}, \tag{3}$$

where $\hat{r}_{ui}$ represents the predicted rating r_{ui} and $|R_{test}|$ is the cardinality of R_{test}.
This metric is calculated only for ratings that users in R_{test} have already given to certain items, since it is not possible to measure the accuracy of predictions for ratings that are not yet available.

3 Problem Addressed

For this project, the developed recommendation system can be divided into two main components. The first concerns the *creation and cleaning of the utility matrix*, and the second concerns the *calculation of predictions*. This section describes the functions that make up the first component which, being related to data processing, will also include an exploratory analysis to highlight the nature of the data.

The most commonly used modules for defining the functions, based on the `Python` language, are `pandas` [8] and `numpy` [9]. The former was used exclusively for working on the data structure, while the latter was chosen for performing matrix and vector operations.

The `pandas` library was chosen because the following algorithms heavily rely on the indexing of the objects involved in various steps. In particular, the `pandas.DataFrame` structure allows for intuitive and simple operations on the available data. `pandas` was not used outside of indexing operations. For all other matrix operations, functions from the `numpy` library were used, after appropriately converting the `pandas.DataFrame` into a `numpy.array` object. The `pandas.DataFrame` structure enabled the development of an important feature of the code: it can be used with any dataset containing the few necessary variables for the analysis, requiring only minimal modifications to the code.

Aspects such as execution time or order of complexity will be addressed in paragraph 6.

3.1 Exploratory Data Analysis

The dataset used for this project is `DigitalMusic`, downloaded from [7]. This dataset contains ratings left by users on products in the "digital music" category. Each row in the dataset represents a record, with information available on the *user* (`reviewerID`), the *item* (`asin`), the *rating* (`overall`), and the time the rating was given (`reviewTime`). The dataset contains 5541 *users* and 3568 *items*. The total number of records is 64706. Only items with at least 11 ratings were considered. Therefore, 12708 records were excluded, corresponding to 5.1% of the initial dataset, referring to items that do not meet this condition. After this initial cleaning, 5401 *users* and 1715 *items* remain.

Next, the training and test sets were created. The training set contains 40569 records, while the test set contains 11429. Their respective utility matrices have a sparsity of 99.56% for the training set and 99.81% for the test set.

An additional cleaning step was performed to eliminate from the utility matrices the items that, ideally, received no ratings and the users that, likewise, left no ratings. In the training set, the percentage of lost items and users is 1.87% and 0.13%, respectively. Thus, the number of items goes from 1715 to 1683, and users from 5401 to 5394. In the test set, the losses are 13.76% for items and 5.26% for users, resulting in 1479 items and 3387 users. Finally, to make predictions from the training set onto the test set, only the users and items common to both sets were retained. The final dataset includes 1466 items and 3284 users.

3.2 Dataset Splitting

This section illustrates the solution devised to split the dataset into a training set and a test set.

Algoritmo 1 Splitting data into training and test sets

Input: Dataframe containing the data
Output: *test set*, *training set*, number of ratings excluded at the end of the split
1: Set the fraction `testratio` and initialize the counter for excluded ratings `canc`
2: **for** each *item* i **do**
3: Determine the number `n` of ratings given to item i
4: **if** `n` ≤ 10 **then**
5: Update `canc` by `n`
6: **end if**
7: Calculate the number of ratings for item i to be included in the test set by multiplying `testratio` by `n`
8: Sort the ratings of item i by time
9: Form the training set and test set dataframes
10: **end for**

In this case, the timestamp information is also used. Therefore, the training set will contain "older" ratings, while the test set will consist of more "recent" ratings. The idea is that past ratings can help in making recommendations for new items for a user.

3.3 Creation of the Utility Matrix

This section outlines the implementation of the functions used to create and clean the utility matrices. The utility matrix defines user-item pairs and their associated ratings. The following function creates the utility matrix from the input dataframe:

Algoritmo 2 Creation of the Utility Matrix with `NaNs`

Input: Dataframe of data
Output: Utility matrix, user indices, item indices
1: Extract the indices of the *users*
2: **for** each row i of the dataframe **do**
3: Extract the i-th *user*, i-th *item*, and i-th *rating* from the previously created lists.
4: Save the *rating* in position *(i,j)* of the utility matrix.
5: **end for**
6: Extract the indices of the *items*

The matrix built using this function will have dimensions $n \times m$, where n is the number of *users* and m is the number of *items*. The main characteristic of these objects is their high sparsity, that is, the presence of many missing values indicated by `NaN`, due to certain *items* not being rated by some *users*.

The following algorithm offers a way to address this issue:

Algoritmo 3 Centering of the Utility Matrix

Input: Utility matrix, user indices, item indices
Output: Centered utility matrix without `NaNs`
1: Mask the `NaN` values.
2: Replace the masked values with the column-wise mean.
3: Subtract the column-wise mean from each *rating*.

This function returns an $n \times m$ matrix in which the missing *ratings* are set to 0. Another implemented solution simply assigns a default value of 0 to missing data, leaving the actual ratings unchanged. The difference lies in the interpretation of the zero value. In the first case, 0 represents the average *rating* per *item*, since the mean has been subtracted from each existing *rating*. In the second case, it is merely a default value, which can introduce more distortion.

Indeed, the first solution is methodologically more correct, since the average value is generally closer to the real *ratings* than 0, which in the worst case (e.g., when the maximum *rating* is 5) could differ by up to 5 points.

In this project, we aim to prioritize the centered utility matrix for distance calculations (i.e., the matrix derived from the *training set*). On the other hand, the utility matrix with default zeros will be used for RMSE calculation (on the *test set*) and for group formation using the *k-means* method.

Algoritmo 4 Cleaning the Utility Matrix of Rows and Columns with All-Zero Elements

Input: Utility matrix
Output: Cleaned utility matrix, item indices
1: Remove columns from the dataframe that consist entirely of zeros.
2: Based on step 1, remove rows that consist entirely of zeros.
3: Extract the indices of the *items*.

Algorithm 4, which summarizes the cleaning function, implements the removal of columns in the centered utility matrix where all values are 0—that is, *items* that have not received any *ratings*, or more likely (given the preprocessing steps), *items* for which all assigned *ratings* matched the item-wise mean.

Rows composed entirely of 0s are also removed. These ideally represent users who gave no *ratings*, or always gave *ratings* equal to the item-wise mean.

This operation was necessary for computing certain distances, which would not be feasible with item vectors made of all 0s. Additionally, it was considered that removing these rows and columns would not significantly reduce informativeness, since during prediction, all 0s in the utility matrix are treated as missing values. Therefore, the origin of the 0—whether due to a "mean" rating or a true `NaN`—will not be considered.

4 Proposed Methods

We now present the similarity measures studied in this report and define the clustering algorithm used to group the items. The procedures described in the following paragraph are based on article [**3**] and chapters 3 and 9 of book [**5**] for the similarity measures, and chapter 7 of book [**4**] and book [**5**] for the clustering algorithm.

4.1 Similarity Measures

Several similarity measures can be used in CF recommendation systems, and choosing the most suitable one is crucial since, given the same input, different measures can yield different results.

It is therefore of interest to investigate the properties of the main similarity measures and see how the quality of the procedure varies with different chosen functions.

In particular, the similarity measures will be computed based on the utility matrix centered by item mean in the case of the ***Centered Cosine Similarity*** and the ***Euclidean Similarity***. For the ***Jaccard***, whether the matrix is centered is irrelevant because, as we will see, the rating given to the item does not affect the measure's calculation. Lastly, the ***Jaccard for Bags Similarity*** will use the non-centered utility matrix, as the original rating value is needed to compute a correct version of the Jaccard Similarity.

These similarity measures will also be used to determine the weight w_{ij} of the j-th item in predicting the i-th item. The exception is the *Euclidean Similarity*, for which we chose to set the weight w_{ij} as the reciprocal of the obtained distance value.

Jaccard Similarity (JS) The *JS* measures the similarity between two items **i** and **j** as the ratio between the cardinality of the intersection and the union of x and y, where x and y are the vectors of users who rated items **i** and **j**, respectively. It is defined as:

$$SIM(x, y) = \frac{|x \cap y|}{|x \cup y|}.$$

This similarity measure ranges from 0 to 1, where 0 indicates maximum dissimilarity (no user has rated both items, making comparison impossible), and 1 indicates maximum similarity (identical users rated both items).

The main limitation of *JS* is that it does not consider the rating value, only the number of users who rated both items. This affects prediction and precision evaluation.

Jaccard Similarity for Bags (JSB) To overcome the limitations of *JS*, the *JSB* variant can be used. Unlike *JS*, *JSB* considers the rating value assigned by a user to an item. In this case, x and y are vectors where each user appears as many times as the rating they gave to items **i** and **j**, respectively.

JSB is defined as the ratio between the sum of the minimum number of times users appear in both vectors x and y and the total number of appearances in both vectors.

This similarity measure ranges from 0 to $\frac{1}{2}$, where 0 indicates maximum dissimilarity, and $\frac{1}{2}$ indicates perfect similarity.

Centered Cosine Similarity (CCS) As the name suggests, *CCS* is equivalent to Cosine Similarity applied to mean-centered data, i.e., ratings with the item mean subtracted.

CCS between items **i** and **j** is defined as:

$$SIM_{cos}(i, j) = \frac{\sum_{u \in U_{i,j}} (r_{ui} - \bar{r}_i)(r_{uj} - \bar{r}_j)}{\sqrt{\sum_{u \in U_{i,j}} (r_{ui} - \bar{r}_i)^2 (r_{uj} - \bar{r}_j)^2}},$$

where $U_{i,j}$ is the set of users who rated both **i** and **j**, r_{ui} is the rating by user u for item i, and $\bar{r}_i$ is the average rating of item i.

This similarity ranges from -1 to 1. A value of 1 means identical ratings, -1 indicates opposite ratings, and 0 means no common user rated both items.

This measure has limitations when missing ratings are present, as its definition becomes inapplicable. Missing values must be imputed before computing the similarity.

Euclidean Distance (ED) *ED* is the simplest measure among those presented. Given two items **i** and **j**, it is defined as:

$$d(i,j) = \sqrt{\sum_{u \in U_{i,j}} (r_{ui} - r_{uj})^2},$$

where r_{ui} and r_{uj} are the ratings by the same user on two different items.

Unlike previous measures, *ED* measures distance rather than similarity. Smaller distances indicate higher similarity, while larger values imply dissimilarity. By definition, $d(i,j) = 0$ when the two items are identical.

4.2 Cluster Analysis

As previously emphasized, a crucial step in making predictions is computing item-item distances. In an item-based CF approach, this can become computationally expensive when the number of items is large.

To address this, traditional methods are augmented with clustering algorithms to group similar items. Defining such groupings reduces the number of similarity measures to compute and thus the processing time. Work will focus on utility matrix segments defined by how items are distributed across clusters. Unlike execution time, we cannot predict the effect of item partitioning on prediction accuracy. The outcome depends on the effectiveness of the clustering process. If the identified groups capture actual latent similarities between items, predictions may improve by focusing on "nearby" items in terms of features. Conversely, if clustering fails to capture relevant latent factors, accuracy may worsen by excluding influential items from the prediction.

k-means This algorithm defines a partitional clustering technique where, given a predefined number of clusters, data is organized into groups such that points within a cluster are more similar to each other than to those in other clusters, with no overlaps. This approach offers several advantages, including implementation simplicity, fast computation, and good result quality. The core idea is as follows:

We have n data points, typically vectors, and aim to group them into k clusters. The implementation follows three main steps:

- **Step 1:** Randomly select initial centroids for the k clusters.
- **Step 2:** Compute the *ED* from each point to the k centroids and assign each point to the nearest cluster.
- **Step 3:** Recompute the centroids as the mean of the points within each cluster.

Steps 2 and 3 are repeated until the centroids do not change or the change is infinitesimal.

As already mentioned, the number of clusters must be predefined. In this case, we set $k = 2$ since, based on data analysis, higher values of k result in

clusters that are too small, sometimes containing only one item, making the chosen methods inapplicable.

During clustering, the non-centered utility matrix was used because the centered version led to the aforementioned issue of small-sized clusters. As noted in section 3, we chose to work with the non-centered utility matrix to avoid such situations.

5 Experiments

As previously stated, this section describes the algorithms implemented in the functions for *prediction computation*.

Once all necessary predictions have been obtained for comparison with the test set, the *RMSE* is used to assess the accuracy of the implemented distance measures, both with and without clustering.

6 Experiments

As previously stated, this section describes the algorithms implemented in the functions for *prediction computation*.

Once all necessary predictions have been obtained for comparison with the test set, the *RMSE* is used to assess the accuracy of the implemented distance measures, both with and without clustering.

6.1 Prediction Computation

The algorithms developed to compute the prediction of a missing *rating* are illustrated below.

Algoritmo 5 Prediction Computation for a *rating*

Input: Item i, user u, Distance Matrix, Utility Matrix, similarity measure
Output: Prediction p for user u on item i

1: Compute similarity between item i and all items rated by user u, and store them in a vector.
2: **if** the similarity vector is empty **then**
3: p = mean of item i.
4: **else**
5: Compute the prediction according to the definition.
6: **end if**

The definition referred to in **step 3** of **Algorithm 5** corresponds to the one given in paragraph 2, summarized by formula (1) for rating prediction without *clustering*, and formula (2) in the case of partitioning by *clusters*.

The prediction is adapted to all similarity measures defined in paragraph 4, particularly in section 4.1, simply by specifying the chosen similarity measure as an input parameter when computing distances.

This algorithm can be used both when computing predictions on the full *dataset* and when working on a subgroup defined by the *clustering* algorithm. However, it is crucial to distinguish between the two scenarios: in the first case, `Utility Matrix` will be the centered utility matrix, and the prediction must be rescaled by adding back the mean to map it onto the original data space. In the second case, `Utility Matrix` must be the non-centered utility matrix, making rescaling unnecessary. It is important to reiterate that the choice of using this second matrix in the clustering case was driven by data considerations rather than methodology.

Algoritmo 6 Construction of the Prediction Matrix

Input: Item i, user u, Matrix1, Matrix2, m rows of *train*, New_{test}, similarity measure
Output: Prediction matrix

```
Compute the distance matrices.
for each user u and each item i do
    if rating pair u-i equals 0 then
        Do not compute the prediction.
    else
        Compute the rating prediction using Algorithm 5
    end if
end for
```

This algorithm enables construction of the "pseudo" utility matrix for the *test set*, which is then compared to the "true" utility matrix during validation of the method's effectiveness. It is essential that this matrix is built only for items and users common to both the *test set* and the *training set*; otherwise, the comparison would not be feasible.

The parameters `Matrix1` and `Matrix2` are necessary when the chosen similarity measure is *JSB*, since the utility matrix used to compute item similarities differs from the one used to compute predictions. `Matrix1` will be the non-centered utility matrix of the *training set*, while `Matrix2` will be the centered utility matrix. For similarity measures other than *JSB*, both `Matrix1` and `Matrix2` will be the centered utility matrix of the *training set*.

6.2 Accuracy and Efficiency of the Algorithms

This section presents the results of the **accuracy measures**, calculated using *RMSE*, with the goal of observing how different distance metrics and *clustering* methods affect the predictions. The RMSE values without any grouping algorithm are: 0.04876 (*CCS*), 0.04772 (*JS*), 0.04795 (*JSB*), and 0.04749 (*ED*), with *ED* achieving the lowest RMSE among all measures. When clustering is

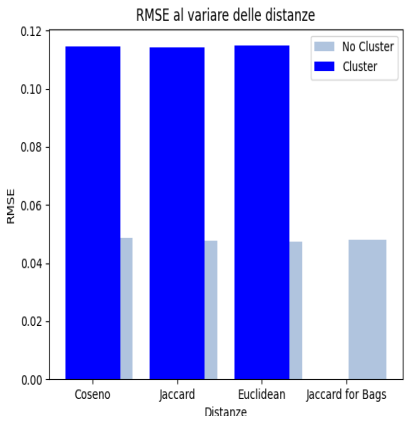
applied, the following RMSE results are obtained: 0.11444 (*CCS*), 0.11428 (*JS*), and 0.11478 (*ED*). The *Jaccard for Bags* similarity was not implemented with *clustering* because, as seen below, it is already highly computationally expensive without clustering.

In general, the obtained values highlight better accuracy for the method implemented without clustering and a minimal difference between the chosen distance metrics. It is also noticeable that the RMSE values obtained when clustering items are about twice those obtained without clustering, emphasizing the greater accuracy of the method that does not involve clustering. **Chart 1** shows these results.

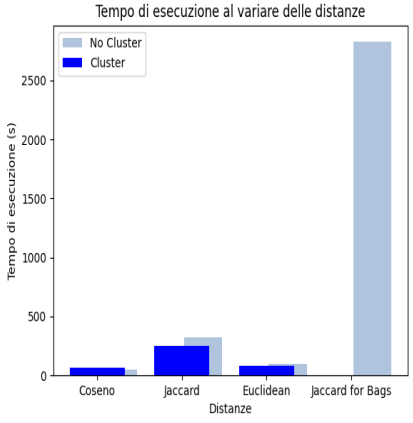
Another important aspect of the proposed methodology is the **efficiency** of the process, evaluated in terms of execution time of the implemented algorithms for the similarity measures, both with and without item grouping. This analysis was conducted using the `time()` function from the `time` module, which returns the number of seconds elapsed since the epoch (on `Unix` systems, 00:00 on January 1, 1970) as a floating-point value. Execution time was calculated as the difference between the times recorded at the start and end of the algorithm for each implemented method combination.

Execution times (in seconds) obtained without clustering were: 51.924s (*CCS*), 322.158s (*JS*), 2826.765s (*JSB*), and 94.066s (*ED*). With clustering, the results were: 67.137s (*CCS*), 252.485s (*JS*), and 85.476s (*ED*). Note the significant improvement in time for (*JS*).

In light of the considerations made in Section 4.2, the use of the *clustering* algorithm, as expected, leads to a significant reduction in prediction execution time for the various distance metrics. **Chart 2** shows these results.



(a) Chart 1.



(b) Chart 2.

7 Conclusions

This work focused on the implementation and evaluation of collaborative filtering algorithms for rating prediction, comparing multiple similarity measures and the use of clustering techniques.

The project successfully implemented algorithms for computing missing rating predictions using several distance metrics: Cosine Centered Similarity (CCS), Jaccard Similarity (JS), Jaccard for Bags (JSB), and Euclidean Distance (ED). Each of these measures was tested both with and without the application of clustering to group items.

The results demonstrate the following key findings:

- **Accuracy:** Prediction accuracy, measured via RMSE, was consistently higher (i.e., RMSE was lower) when clustering was not used. The best result was obtained using Euclidean Distance (ED) without clustering, achieving an RMSE of 0.04749. Conversely, all distance metrics showed significantly higher RMSE values when clustering was applied, approximately doubling the error rate.
- **Efficiency:** The clustering approach notably improved computation times, especially for computationally intensive similarity measures such as JS and JSB. For instance, the execution time for JS dropped from 322.158s to 252.485s when clustering was applied.
- **Trade-off Analysis:** There exists a clear trade-off between prediction accuracy and computational efficiency. While clustering accelerates processing, it tends to degrade accuracy. This suggests that clustering may be beneficial in large-scale systems where speed is critical, but not ideal when high accuracy is required.
- **Suitability of Similarity Measures:** Among the similarity measures, Euclidean Distance (ED) performed best in terms of both accuracy and efficiency (especially with clustering). In contrast, Jaccard for Bags (JSB) was the most computationally expensive and did not offer sufficient accuracy improvements to justify its cost.

Future Work: Future improvements could include experimenting with alternative clustering strategies (e.g., density-based or hierarchical clustering), applying dimensionality reduction techniques such as PCA or SVD, and incorporating user-based collaborative filtering to compare against the item-based approach. Additionally, hybrid models combining content-based information with collaborative filtering could be explored to enhance both accuracy and personalization. In conclusion, the project highlights the practical trade-offs involved in recommender system design and underscores the importance of choosing the right balance between algorithmic complexity and predictive performance depending on the application context.

Riferimenti bibliografici

1. Ricci, Francesco, Rokach, Lior, Shapira, Bracha, and Kantor, Paul B.. Recommender Systems Handbook (1st. ed.). Springer-Verlag, Berlin, Heidelberg, (2010).

2. Aggarwal, Charu C. Recommender systems. - The Textbook (1st. ed.). Springer International Publishing, (2016).
3. Sondur, Mr Sridhar Dilip, Mr Amit P. Chigadani, and Shantharam Nayak. Similarity measures for recommender systems: a comparative study. Journal for Research 2.3 (2016).
4. Tan, Pang-Ning, Michael Steinbach, and Vipin Kumar. Introduction to data mining. Pearson Education India, (2016).
5. Leskovec, Jure and Rajaraman, Anand and Ullman, Jeffrey David. Mining of Massive Datasets. Cambridge University Press, USA, (2016).
6. Aho, Alfred V. and Ullman, Jeffrey D. Foundations of Computer Science. W. H. Freeman Co., USA, (1994).
7. Amazon Review Data (2018): `https://nijianmo.github.io/amazon/index.html`
8. Wes McKinney. Data Structures for Statistical Computing in Python, Proceedings of the 9th Python in Science Conference, 51-56 (2010). `http://conference.scipy.org/proceedings/scipy2010/pdfs/mckinney.pdf`
9. Jones, E., Oliphant, T., Peterson, P., et al.: NumPy Reference Guide release 1.14.5. `https://docs.scipy.org/doc/numpy-1.14.5/numpy-ref-1.14.5.pdf`