Data Mining Project

Jacopo Donà
jacopo.dona@studenti.unitn.it
LM: Artificial Intelligence Systems
Mat: 229369

Jacopo Clocchiatti
jacopo.clocchiatti@studenti.unitn.it
LM: Computer Science
Mat: 229701

Keywords

datasets, recommendation systems, data mining

ACM Reference Format:
Jacopo Donà and Jacopo Clocchiatti. 2023. Data Mining Project. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Abstract

This document is the report of the project work conducted for the Data Mining course at the University of Trento.

This report presents a recommendation system that utilizes user behavior data and query results analysis to make personalized recommendations. The final system employs a combination of collaborative filtering and content-based filtering techniques to generate a list of queries that are predicted to be of interest to the user.

For evaluating the performance of the system we use metrics based on accuracy on both true and synthetic datasets, generated using pseudo-random statistical behavior. Our results show that the proposed system outperforms the baseline, demonstrating its effectiveness in providing more accurate and relevant recommendations. We also discuss applications, potential issues and future directions for the system.

2 Introduction

With the constant content growth of online stores, social networks and streaming platforms, recommendation systems have become more and more necessary in guiding the customer through data. As content availability increases, a new problem for online users has surfaced: information overload. Poorly designed web services can "drown" the user with information, making the navigation overly complicated and leading to the customer leaving the website.

Recommendation systems aim to improve the user experience by aiding users in finding what they are looking for and discovering new items of interest.

These tools aim to provide personalized suggestions for what to search for or purchase, helping users to find what they are looking for more quickly and easily, and can also help them discover new items they may be interested in.

Despite being heavily focused on improving the user experience, recommender systems are beneficial to both service providers and users. By helping users find what they are looking for more easily, these systems can increase the likelihood that users will complete a purchase or take other desired actions. Recommendation systems can also help businesses better understand the needs and interests of their users, which can be used to improve the overall user experience and drive more traffic to the platform. Moreover, a successful user experience leads to the customer using the same service in the future once a similar need resurfaces.

Recommendation systems face many challenges when dealing with user preferences and needs. Query recommendation systems need to be able to adapt to each individual customer's preferences. This first objective is non-trivial, as customers tend to have different tastes and different priorities when looking for a particular item, which are unknown to the system when the user logs in for the first time.

In addition to that, another complex challenge is maintaining the quality and relevance of the recommendations over time, as users may become less satisfied with recommendations that are not accurate or useful. Moreover, queries that led to successful results for a customer may become less effective over time, especially if they are frequently reproposed.

In this report, we propose 3 different solutions we tried for solving the problem, explain their main concepts, as well as the pros and cons, and evaluate them on a synthetic dataset we generated for the problem.

3 Formal Definition

The goal of this project is to design and implement a query recommendation system able to operate in a generic contest.

The items available in the system are stored in a table T and are described by a set of m attributes $\{a_1, ..., a_m\}$. We assume the items do not contain empty attributes value and do not duplicate, i.e. the dataset does not contain multiple items who share the same identifier.

The algorithm makes use of a set $S = [u1, ..., un]$ to keep track of the n registered users in the system and of a set $Q = [q_1, ..., q_x]$ containing queries that have been posed to the system in the past.

Each query is a stored in the format $q = \{id, cond_1, cond_2, ..., cond_i\}$ with $i \in [0, ..., m]$. Queries can have different length and conditions $cond_i$ are in the form $(attribute = value)$. Conditions are logically connected by an AND operator.

For each query the user user has executed, a query rating is being stored, representing the user satisfaction based on the results. This information is stored in a utility matrix $Q^{m \times n}$, where the cell (i, j) contains the rating user j assigned to query i. The cell value may be empty if the user has not rated the query yet.

Given these inputs, the goal of the project is to:

- Generate the missing entries of the utility matrix U, using the available user ratings to predict the unrated queries.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

- Predict for a given user the top-k queries that may be of interest to him.
- Define a utility function $f : (q_i, U) \rightarrow s$ which predicts the utility score s of a query q_i for all the users in the utility matrix U.

The score value s could be of use to propose globally appreciated queries in the case of a cold start, where the user has just signed in on the service and has not yet rated any query. On the other hand, once the user has started rating, the system can begin recommending queries by filling the utility matrix based on his preferences.

4 Related work and technologies

In this section we give a brief introduction of algorithms we integrated into the project that we integrated from other fields of Data Science.

4.1 Cosine and Pearson Similarity

The cosine similarity is a measure of similarity between two non-zero vectors of an inner product space that measures the cosine of the angle between them.

$$\cos(\theta) = \frac{A \cdot B}{\|A\|_2 \|B\|_2}$$

This metric is commonly used in information retrieval and text mining to measure the similarity between items represented as vectors.

The Pearson similarity has an almost identical procedure but with an important pre-processing step. For each vector, the mean value μ_i is computed i by simply averaging the votes the user has given to the available entries. Then, each entry rating of the vector is 'rescaled' by subtracting μ. In the utility matrix case, for every i user the rating of the j query is computed as such:

$$\gamma_{ij} = \alpha_{ij} - \mu_i$$

where α_{ij} is the original rating user i gave to query j.

This operation has a similar effect as normalization. Each user has a different way of rating items. Some people might vote in between a strict range of medium values (example: rate $\in [40, 70]$), while others may have a wider range for the rating (example: rate $\in [10, 100]$). By re-scaling the ratings to the user average, we remove the bias generated by the user's personal way of voting and focus on whether or not the item has left a positive impression on the user.

Furthermore, rescaling the empty values doesn't influence negatively the user's tastes, enabling us to have a clearer computation of the user similarity.

In our project, these similarity measures are used when computing the user-user similarity matrix (baseline and final solution) and for the query-query similarity matrix (final solution).

4.2 Weighted K-Nearest neighbour

The K-Nearest Neighbour is a distance-based learning method where we place a set of learning points into a N dimensional space, storing for each point information about its class/value.

During inference, a test point is placed in to the dimensional space and the distance of each point of the learning set is measured with respect to the newly inserted point by computing the Euclidean distance along its dimensions. The K-NN uses a parameter k (typically a odd number to avoid even results) and measures the frequency of each class appearing in the neighbourhood of the test point, and assigns it the most dominant one.

In the Weighted K-Nearest neighbour, the same procedure is followed but instead of simply counting the class appearances, we make a weighted average of each entry by measuring the distance w.r.t the test point. In this way, points being closer to the test point have an higher influence on the final decision, and outliers have less effect on the final assignment.

$$\hat{y} = \sum_{i=0}^{k} w_i * c$$

where:

- $\hat{y_i}$ is the predicted class label/value of test point i
- c_j is the class label/value of the jth point
- w_j is the weight assigned to the jth nearest neighbour
- k is the number of nearest neighbours to consider

4.3 SVD

Singular Value Decomposition (SVD) [2] is an algorithm used to reduce the dimensionality of a matrix. It is a powerful tool that can be used to reduce the complexity of a problem, allowing faster computations.

The SVD of an $m \times n$ real matrix M is a factorization of the form $M = USV^T$, it decomposes the matrix M into a set of three matrices: U, S, and V. U and V are orthogonal matrices (V^T is the transpose of V) of dimension $m \times m$ and $n \times n$ respectively while S is a $m \times n$ diagonal matrix with non-negative real numbers, called singular values, along its diagonal. The singular values represent the strength of the relationship between the columns and rows of the original matrix.

SVD can be used to reduce the dimensionality of a matrix by eliminating the columns and rows with the smallest singular values. This process is known as truncation. Truncation allows us to reduce the complexity of the matrix without losing too much information. Additionally, SVD can be used to identify the underlying structure of a matrix. By analyzing the singular values of the matrix, we can identify the features that are most important to the data.

5 Solution

In this section we propose 3 different solutions we tried to solve the utility matrix filling problem.

These 3 solutions share a collaborative approach to solving the problem, where we use the ratings of users to compute a similarity metric to fill in the missing values based on votes of similar users/queries.

5.1 Baseline

As a Baseline, we decided to treat the problem as a standard item recommendation system. That is, the system ignores the existence

of the table of items and treats query ratings as if they were item ratings.

To implement this solution, the utility matrix containing the user-query ratings is first loaded onto the memory, with empty entries having assigned the Python NaN value (Not a Number).

Once every rating is loaded, the empty entries of non-rated items are converted from NaN to 0.

From there, we use each user's ratings to compute a similarity matrix between users. To do so, each user row of the utility matrix containing the user ratings for every item in the dataset is extracted and treated as a single vector.

The cosine similarity is then applied to each pair of vectors, and produces as output the degree of correlation between two users in the range $[-1, 1]$ where:

- 1 indicates a strong correlation between users ratings (users have similar tastes).
- 0 indicates a the absence of correlation between user ratings (users have different tastes).
- -1 indicates a strong negative correlation between users (users have opposite tastes), this type of correlation is not treated in the algorithm.

Once the user similarity matrix has been calculated, we use the Weighted k-Nearest Neighbour algorithm described before to compute the query prediction. The k most similar user share their vote and based on the similarity a weighted average is computed to propose a prediction.

In Figure 1 we show the pseudocode of the algorithm used for obtaining the vote proposal for every query. In the event no user in the k similar group has voted the query (possible with low k), a random vote is generated for the query. This process is repeated for the whole utility matrix until complete filling.

5.2 SVD trial

Our first initial solution for solving the problem consisted in using a collaborative filtering method based on matrix factorization. We took inspiration from the Netflix challenge and seek to implement an SVD [2] where the utility matrix was used to extract features that correlated the sensibility of each user to each feature concept.

This approach follows an Expectation-Maximization approach, where the SVD is computed for multiple steps until minimal changes or convergence.

Once the user activation and query activation to each feature was computed, we could have used a simple dot product for obtaining predictions or used some machine learning regression approach to estimate the utility matrix entries for each user.

Unfortunately, we found various challenges in implementing this solution, which ultimately led us in abandoning the idea due to time restrictions:

- Even on our smallest dataset, the expectation-maximization approach was very slow.
- We found SVD to be very sensitive to data sparsity, lowering the number of training steps allowed us to achieve results that were however corrupted, mixing float values with NaN, therefore incorrectly filling the utility matrix.

Given more time, we would like to re-try this approach following a different pipeline, however, in order to deliver a fully functional method, we decided to change our final solution algorithm.

5.3 Final Proposed Solution

In the baseline, we limited the problem to a standard item recommendation system, considering only the information contained in the utility matrix and completely ignoring the fact query and item table data.

However, we still liked the idea of being able to categorize users and use only a portion of the user pool as suggestion for a given user when suggesting new content.

As a final solution, we propose to mix the collaborative and content-based approach by combining all the data available in input to better model dependencies between rows/columns of the utility matrix.

In this solution, we kept the user similarity matrix procedure as it was in the baseline, but decided to replace the cosine similarity with the Pearson similarity. One major downside of the cosine similarity when dealing with missing data (in the case of utility matrix filling, dealing with not rated queries) is that empty entries value (in our case, the 0 value of a non-rated query) are treated as a negative rating. This can cause unwanted effects, as two users with pretty different tastes may have a high similarity score due to the fact they share many missing queries. As mentioned before, the normalization effect introduced by the Pearson similarity helped us in better modeling dependencies between user rows, resulting in a small recommendation improvement with close to no additional computing cost.

In addition to the users, we wanted to also generate a similarity matrix between queries.

The easier and faster approach would be to apply the same procedure of the user similarity matrix, by extracting column values from the utility matrix and computing dependencies between votes.

However, this approach has two main problems:

- We believe that, by computing two similarity measures from the utility matrix in the same procedure, their results would have been in some way correlated, limiting the efficiency of the additional information.
- As for the baseline, this solution would completely ignore the query conditions and their results when executed on the data table, treating each query essentially as an item

Instead, we opted for using the query results for computing the similarity matrix.

To do so, each query is executed and the Jaccard Similarity (also known as Intersection over Union) it's computed on its results. Since the matrix is symmetric, to save computational time we compute only half of the matrix and replicate their results on the "flipped" side of the table.

The first implementation of this idea had two main problems.

Our code is run on Python, and to load and query the table we first thought of using the pandas dataframe [4]. What we did not know is that this implementation, even if converted on a numpy array, was extremely slow. We believe this to be due to the fact that the pandas library is Python native.

```
var matrix # Contains full utility matrix
var similarity_matrix # Contains full similarity matrix of users
var k # Number of neighbours to consider when making prediction

for i in range(0, len(original_matrix)):
    row = matrix[i] # Contains row of utility matrix with given ratings and empty entries of a user
    for j in range(0,len(row)): # For every query in the utility matrix
        rating = row[j]

        if rating is NaN: # Make predictions on missing entries
            indices, weights = getKUserneighbours(similarity_matrix[i], k)
            # indices contains the indexes of most similar neighbours
            # weights contains the measure of similarity
            neigbor_votes = matrix[indices, j] # Get list of votes of selected query j for all neighbours

        for h in range(0,k)): # For every neighbour, get his contribution to the prediction vote
            vote = neigbor_votes[h]
            prediction += vote * weights[h] # Accumulate sum of votes
            weights_sum += weights[h] # Accumulate sum of weights

        prediction = prediction / weights_sum # Make weighted average and get query prediction
```

Figure 1: Pseudocode of baseline prediction algorithm

Fortunately, while experimenting with other implementations we found out about Polars [5], a similar dataframe library written in Rust with Python API support. We found out that this architecture was considerably faster compared with Pandas, even when dealing with large query results, allowing us to rapidly execute queries onto the loaded table.

The second issue was encountered when computing the prediction using the Weighted k-NN. Our idea is to combine the user similarity and the query similarity in a linear way, by extracting the top k users and top k queries, giving each similarity the weight proportional to the similarity score.

In our first implementation, we thought of using as weights directly the values of the similarity matrices, as both the Jaccard Similarity and the Pearson similarity (only the positive correlations) belong in the range $[0, 1]$. However, when dealing with large query results, the Jaccard Similarity tends to have really small values (< 0.1) resulting in queries having way less of an impact when computing the weighted average.

To fix this behavior, we introduce a normalization procedure when extracting the neighbours.

Once the weights of the top k neighbours have been extracted, they are normalized.

In this way, even small values of intersection over union queries have equal influence to user neighbours of the selected query-user prediction to compute.

The rest of the procedure is the same of the baseline.

6 Part B concept

As mentioned on the forum, for part B of the project, the one that requires designing a global utility score for a given query in the system, we just propose our own theoretical idea for how it could be implemented, in a similar way as we implemented our overall predicting system.

Once the utility matrix has been filled with the predictions for missing entries, each query j has for every user i a rating, that may be real if the user has originally voted it, or estimated if it was not rated yet.

To generate a utility score for every query, we propose to use once again a Weighted Average method where the initial and filled utility matrices are considered.

Firstly, we begin by computing the query overall perception among the users. To do so, we use both matrices to compute the vote as such:

$$p_j = \sum_{i=0}^{N} s_{ij} * \alpha_{s_{ij}}$$

where:

- p_j is the global perception of query j.
- N is the number of users.
- s_{ij} is the score user i has for the query j.
- $\alpha_{s_{ij}}$ is either 1 if the entry is present in both the original and filled matrix, or 0.5 is the entry is empty in the original utility matrix but present in the filled one.

In this way, we use the system's full knowledge of the scenario to make a prediction on the query's actual value. To account for possible system mistakes, we give higher relevance to ratings effectively given by users, while still keeping into consideration the generated predictions.

Once p_j is computed for every query j is computed, the values are normalized and returned in a $[0, 1]$ range, where lower values correspond to less 'useful' queries from the user perspective, and

```
var matrix # Contains full utility matrix
var user_similarity_matrix # Contains full similarity matrix of users
var query_similarity_matrix # Contains full similarity matrix of queries
var k # Number of neighbours to consider when making prediction

for i in range(0, len(original_matrix)):
    row = matrix[i] # Contains row of utility matrix with given ratings and empty entries of a user
    for j in range(0,len(row)): # For every query in the utility matrix
        rating = row[j]

        if rating is NaN: # Make predictions on missing entries
            user_indices, user_weights = getKUserneighbours(user_similarity_matrix[i], k)
            # user_indices contains the indexes of most similar neighbours
            # user_weights contains the measure of similarity
           user_neigbor_votes = matrix[user_indices, j] # Get list of votes of selected query j for all neighbours

            query_indices, query_weights = getKQueryneighbours(query_similarity_matrix[j], k)
            # Get list of votes of selected query j for all neighbours
            query_neigbor_votes = matrix[query_indices, j]


        for h in range(0,k)): # For every user and query neighbour, get his contribution to the prediction vote
            user_vote = user_neigbor_votes[h]
            query_vote = query_neigbor_votes[h]
            prediction += user_vote * user_weights[h] + query_vote * query_weights[h] # Accumulate sum of votes
            weights_sum += user_weights[h]+ query_weights[h] # Accumulate sum of weights

        prediction = prediction / weights_sum # Make weighted average and get query prediction
```

Figure 2: Pseudocode of our final solution prediction algorithm

high values correspond to globally appreciated queries. Therefore, we use the user behavior to describe the utility of the queries, as less appreciated queries will be categorized as less useful.

We believe this solution to also be a good approach for dealing with the cold start issue, where a user just registered onto the system and has not rated anything yet.

7 Experimental evaluation

In this section, we describe the evaluation process of our project. We first introduce the dataset we used to test our solution and describe the metrics and the validation process we used to analyze the overall project performance.

7.1 Dataset

For the evaluation of the algorithm, we first decided to use both a synthetic generated dataset and a realistic one, available online, in order to have a better understanding of the algorithm's performance.

Our initial objective was to make our algorithm as flexible as possible, so we started by generating our own data to create 3 different scenarios:

- `people` dataset represents a LinkedIn-style social network, where we have a table with a number of people stored by full name, occupation, nationality, and age. In this scenario, the users represent companies looking for new employees by querying the database in search of appropriate candidates, so we assume that the number of users is much larger than the number of actively searching companies. We generated 5000 active users and 50000 table items.
- `music` dataset represents a Spotify-style media platform, where we store in the tables attributes relative to songs name, genres, authors, and the decade of when it was produced. In this scenario, we imagined we have an even distribution between items in the tables and users. We generated 10000 active users and 10000 table items.
- `movies`: dataset represents a Netflix-style media platform, where we store in the attributes of the table information relative to movie name, genre, duration, and nationality. In this scenario, we imagined we have a large number of active users, with a limited availability of movies in the database. We generated 20000 active users and 5000 table items.

For all 3 scenarios, we generated 500 queries that are guaranteed to return at least one result (queries do not return empty results) that are generated randomly by iterating through the available attributes/items.

Furthermore, we followed a closed-set assumption for the queries. The queries available to the system are finite and are all given as

input, the system is not able to recommend or analyze queries that have not been included in the initial input.

Note: This final dataset is smaller than the initially delivered one, this was due to the fact we initially had 50000 users in the movie's scenario, which generated a bigger utility matrix that caused our hardware to freeze when calculating the user similarity matrix through cosine/Pearson similarity. Therefore, we later re-scaled each scenario to have smaller utility matrices able to fit onto the memory of our machines.

One of the main challenges in generating the data was being able to generate a realistic dataset. While this task was trivial for generating semi-realistic people/song/movie entities, the same could not be said for the utility matrix.

To differentiate users, we start by randomly rating a small subset of queries that represent the user "unique tastes", these queries are selected through a uniform distribution.

From there, our goal was to make a semi-realistic set of queries that are influenced by the previously rated queries of the user. To do so, we first tried to compute a query similarity metric by using a Intersection over Union on query results on the dataset. However, computing this operation for 500 queries with potentially hundreds of results caused the generation process to be very slow on our hardware (this was before we discovered the Polars architecture).

To solve this problem, we decided to base the query similarity on the attributes-values combination, based on the fact that for example if a user likes a music query `genre='rock' AND year='80s'`, he probably will give a similar rating to a query of 90s rock music.

The similarity of between a pair of queries i, j during the generation is based on a score, computed as such:

$$s_{ij} = \sum_{k=0}^{m} \sum_{h=0}^{n} \eta_{kh}$$

$$\eta_{kh} = \begin{cases} 0 & attribute_k \neq attribute_h \\ 1 & attribute_k = attribute_h \\ 5 & attribute_k = attribute_h, value_k = value_h \end{cases}$$

where m, n are the numbers of conditions of each query, respectively.

Once the s score is computed for each pair of queries, the rating new query is generated by sampling from a Gaussian which has mean equal to the vote of the most similar query and the standard deviation $\sigma = 1/(2 * s_{ij})$.

In this way, the more similar a query is to a voted one, the more its rating will be similar. If a query it's very different from the currently rated ones, s_{ij} will be low which will cause the standard deviation value to remain large enough to generate a value not-correlated to the original rating.

For the utility matrix, we created various scenarios based on utility matrix sparsity to test our system on. One where each user has rated <10% of the available queries, one where each user had rated ~ 30% of the entries, one where the ratio of rated/unrated items by every user was ~ 50%, and a final one where each user rated ~ 75% of the entries, to see how much of an impact the data sparsity had on our algorithm.

After first testing our algorithms on our synthetic data we started looking for an analogous dataset online to use. However, due to the detailed formulation of this project, we were not able to find a dataset able to fully provide samples for this problem. In particular, we found a vast availability of item recommendation datasets but all of them focused on item recommendation rather than queries.

Since our method of generating the dataset involves using pseudo-random votes for users, we still wanted to be able to compare our algorithm with a realistic one, in order to understand if the poor initial results we encountered were caused by the poor efficiency of our solution or whether the generated dataset was effectively difficult to interpret by an automated system.

The movies dataset(IMdB) we built from [3] is an item recommendation type of dataset. We built the utility matrix from the available ratings file, obtaining a matrix somewhat different from the one we generated ourselves. Ours was more "vertical" (more users than queries), this one is more "horizontal" (more items than users). After some runs of the baseline, which is not very domain specific on queries, we obtained worse results than our synthetic datasets. These performances are caused by the lower number of users, users that the baseline uses to obtain the predicted value, and its sparsity.

7.2 Metrics

To evaluate our algorithms on the dataset, we performed validation by masking a portion of the utility matrix.

We used an 80-20 split with the testing users providing information on just 70% of the queries to make a prediction. By doing this way, we are able to preserve ground truth values to test the algorithm predictions. The only downside of this approach is that for some users performing this type of mask can 'hide' a significant portion of their queries used for learning its preference.

However, once the script is deployed and active we must expect users with very few ratings, so we decided to maintain this approach unaltered.

The error measures are computed **only on the masked entries**, the values contained in the original utility matrix (the true user ratings) are not considered when measuring the errors.

7.2.1 *RSME*

The Root Mean Square Error is the square root of the average of the squared differences between the true values and the predicted values. It squares the errors before averaging them, which gives more weight to larger errors. This makes it more sensitive to outliers and can be affected by large errors.

$$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2}$$

where:

- N is the number of samples
- y_i is the ground truth value of the i^{th} sample
- $\hat{y}_i$ is the predicted value of the i^{th} sample

7.2.2 *MAE*

The Mean Absolute Error is an error metric that measures the average of the absolute differences between the true values and the predicted values. It simply adds up the absolute errors and provides a measure of the magnitude of the errors, regardless of

their direction (positive or negative), and tends to "penalize less" the outliers by avoiding the square operation.

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|$$

where:

- N is the number of samples
- y_i is the ground truth value of the i^{th} sample
- $\hat{y}_i$ is the predicted value of the i^{th} sample
- $|\cdot|$ denotes the absolute value

7.2.3 *Accuracy*

As an additional metric, we wanted to "simulate" the behavior of our algorithm on recommending items. In our opinion, if the algorithm makes for a query with a ground truth value = 90 a prediction with a value = 80, it should not be treated as a "bad mistake", as the algorithm still thinks the selected query may be a good suggestion for the user.

To test this behavior, we developed a custom metric we called Accuracy which consisted in classifying every proposed query into "good suggestion" and "bad suggestion". To make this separation, we inserted a threshold at the rating of 50. If both the proposal and ground truth are above/below the threshold, it counts as a correct suggestion, if one of the two values is on the other side of the threshold, instead it is classified as a mistake.

Once all the predictions are classified, we perform a standard accuracy metric:

$$Accuracy = \frac{TP}{TP + FP}$$

where:

- TP is the number of true positive recommendations, i.e. the number of correct predictions.
- FP is the number of false positive recommendations, i.e. the number of incorrect predictions.

7.2.4 *User iterations per second*

As a metric to measure the system speed, we recorded how many user rows of the utility matrix the system was able to fill at each second. Our main goal was to be able to compute proposals for multiple users at every second, as a realistic system has to handle multiple users at a time and cannot afford to make them wait multiple seconds at login.

7.3 Results

In Table 1 we report the evaluation of the various proposed metrics on all the generated scenarios for both the baseline and the final solution.

As we can see, the final solution provides a sizable improvement on all metrics, being more robust to both smaller and larger mistakes, and therefore being able to improve its accuracy score. The only downside of our method compared to the baseline is the fact that the user predictions per second decrease, but we believe the overhead to not be overly excessive.

Unfortunately, despite the gain over the baseline, our algorithm still has a large margin of improvement on the dataset.

We believe this to be caused by the fact that our prediction algorithm and our dataset generation use different methods to measure the similarity between queries.

In fact, recalling 7.1, during the generation of the utility matrix, two queries are similar if they share multiple attribute-value pairings in their condition, meanwhile in our algorithm we measure the similarity by computing the Jaccard Similarity in the query result.

For example, a query $q1$ first-name='John', last-name='Brown', job='plumber' and a query $q2$ first-name='John', last-name='Brown', job='teacher' are very similar when generating the dataset, but since they are mutually exclusive (as a person can only have one job) they return an empty Intersection over Union.

This "similarity replacement" when designing the algorithm was intentional, as our main goal was to design a recommendation system that could generalize well. However, this will cause major differences when creating the query similarity matrix, and can be treated as a sign of the importance of domain knowledge when working on data mining.

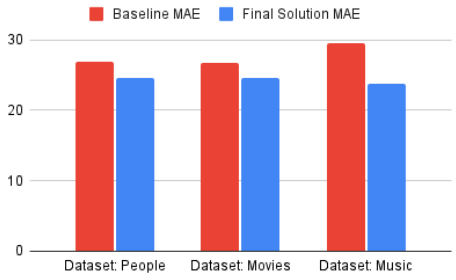

Figure 3: MAE for each of the datasets

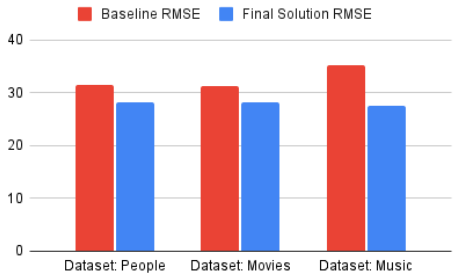

Figure 4: RMSE for each of the datasets

Table 1: Results of baseline and final proposed solution, for each domain and for each solution the MAE, RMSE, Accuracy and number of user predictions per second is reported

	People				Music				Movies			
	MAE	RMSE	Accuracy	User/s	MAE	RMSE	Accuracy	User/s	MAE	RMSE	Accuracy	User/s
Baseline	27.0	31.54	49.7%	11.02	29.57	35.17	50.0%	5.13	27.0	31.28	49.0%	2.5
Final Solution	24.55	28.13	54.3%	9.71	23.78	27.44	58.3%	4.78	24.59	28.27	52.6%	2.3

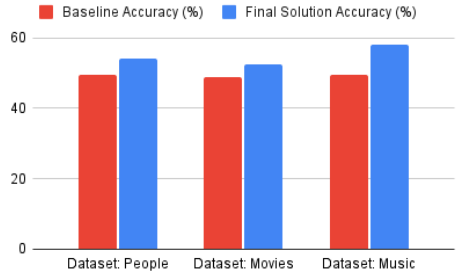

Figure 5: Accuracy for each of the datasets

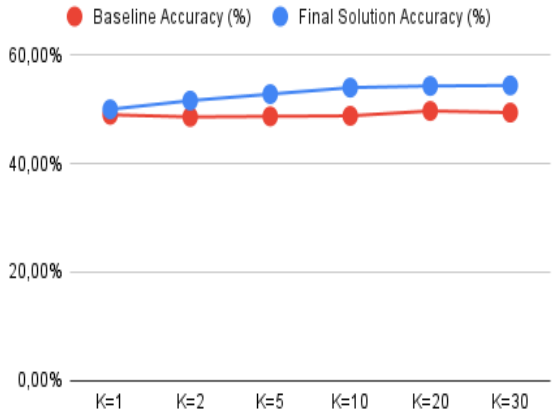

Figure 7: Accuracy for different values of k

7.3.1 K parameter

In Figures 7 6 we plot the behavior of the evaluation metrics w.r.t. the parameter k used for predicting the vote with the Weighted k-Nearest Neighbor algorithm.

It is interesting to note how, contrary to typical data science applications where usually small values of k work the best (example: $[3, 5, 7]$) in our algorithm it appears as higher k values produce more accurate prediction, with the best performance achieved with $k = 20$ and with a saturating behavior afterward.

We believe this is in part caused by the fact that we use a sparse matrix and, as mentioned, if the k closest user neighbors did not rate the query in consideration or if the nearest queries were not rated by the user, a random vote is generated. Therefore, larger k values guarantee to have a prediction being performed by collaborative filtering.

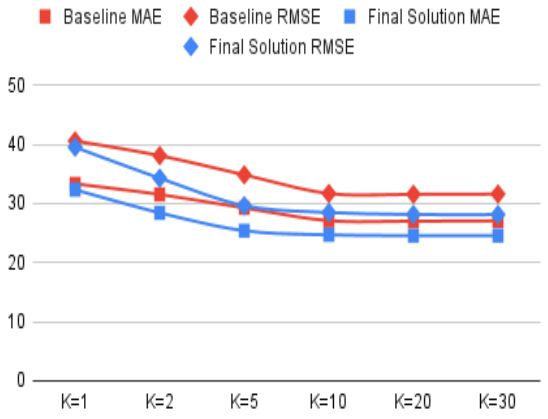

Figure 6: Errors for different values of k

7.3.2 Utility matrix density

In Figures 8 9 we plot the behavior of the evaluation metrics w.r.t. the density of the utility matrix given as input.

Interestingly, in this experiment we witness a contradicting behavior as the evaluation metrics, in particular MAE and RMSE, progressively get worse as the utility matrix gets denser.

Having generated our own dataset, we believe this behavior could be caused by the Curse of Dimensionality effect.

As users give more and more reviews, their rows (and therefore vectors) inside the utility matrix become bigger. This may cause the Pearson similarity to return small values because of the poor correlation between users, as their ratings may bring multiple tastes that with high dimensionality become less and less correlated between users. This experiment was conducted only in the people scenario, so we think that having not many users at our disposal (in people it's just 5k users) the user similarity matrix may not capture good user correlations.

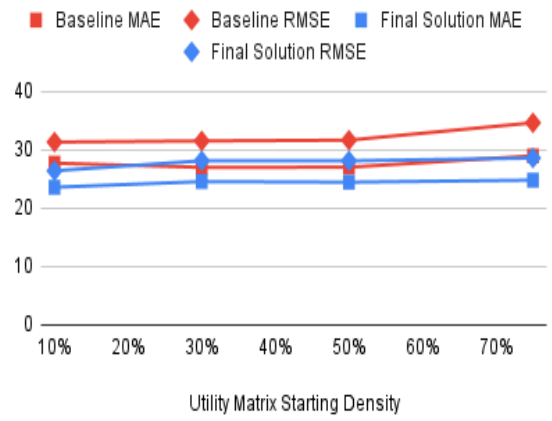

Figure 8: Errors for different density of the starting utility matrix

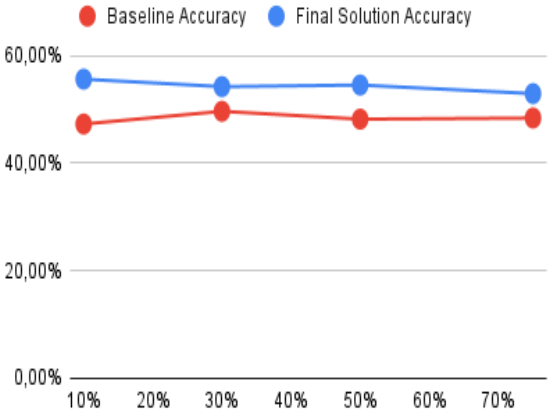

Figure 9: Accuracy for different density of the starting utility matrix

7.3.3 *Time of execution*

In Figure 10 we plot the time of execution of each scenario.

The difference in execution times is mainly correlated to the number of users of the utility matrix. Larger active users require more time in execution as the Weighted k-NN has more points to analyze when computing the difference (this can be noted by looking at the $User/s$ metric in Table 1, and by the fact that the Movies scenario is 5x bigger w.r.t People.

In addition to that, the user and query similarity matrix also had their time recorded:

- *people* user similarity matrix took less than a second to be computed.
- *music* user similarity matrix took 7 seconds.
- *movies* user similarity matrix took 37 seconds.

These values are entirely dependent on the number of users of the scenario, and are not affected by the degree of density of the utility matrix.

The query similarity matrix had some slight variation of time calculation on each scenario due to different sizes of query results (for example: the People scenario, even though it had more items in the table, tended to return smaller results due to the variety of data fulfilling all query conditions), but they all averaged to be in the 3-4 minutes range. This may seem like too much time, but we believe that by having a closed-set query scenario, each query is declared before any operation regarding the users begins. This assumption allows us to have the system independently compute the query Jaccard similarity before the service launches, and store in memory just the query similarity matrix to use at inference time, which is composed of just numbers.

In the case additional queries have to be added, the system will have to re-compute the similarity matrix and publish it once the computation is complete.

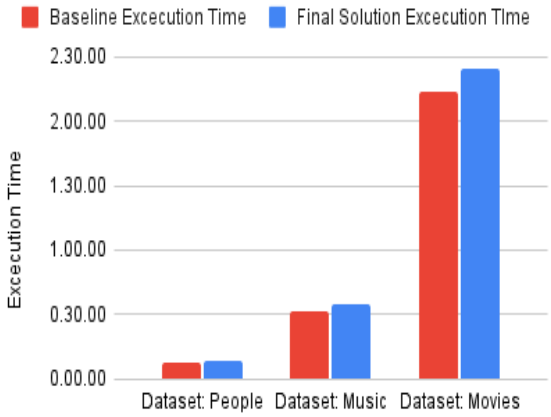

Figure 10: Execution time for each dataset

8 Known issues and future improvements

The main issue of our proposed method is its scalability. Our method, leveraging two similarity matrix, needs to compare each couple of user and each couple of queries and this means that it is asymptotic time is at least $O(n^2)$. Needless to say for a massive number of users and/or queries our solution is not feasible. One improvement to our solution that also addresses this issue is to use a distributed or parallel computing library like Dask[6] or Apache Spark[7].

Another issue is when giving a proposal for a user, it does not take into consideration the fact that a user may have different benchmarks for voting which results in some proposals being far from their ground truth (the user range of votes we discussed in section 4.1), we should implement some sort of scalar multiplication to take into account this complication.

Even if faster, our method of computing the query similarity matrix using polar still takes time to execute queries. Our main flaw is that we compute every time the solution of the query, but it could be smarter to use some form of mapping/hashing for storing results

in a lighter format to speed up computations while performing the Jaccard similarity.

In order to further improve our solution we could extract some sort of tag and use them for building similarity matrix instead of executing the queries. This could lead to faster comparisons due to the lack of need to execute the queries multiple times. Another improvement on the comparisons could be made on the user's one. If we have information about the users, like their geographical location or their date of birth, we could cluster them on those information thus reducing the number of comparisons needed.

Lastly, some deep learning techniques could be incorporated to address the problem. Recent works study the applicability of techniques such as Autoencoders, Natural Language Processing and Graph-based Neural Networks to explore solutions for recommendation systems.

9 Conclusion

Throughout the course of this project work, we faced various challenges, from fully understanding the problem to generating an adequate dataset and evaluation process for the testing phase. We found out that it is difficult to have a 'universally good' algorithm due to differences between domains, as well as how users may interact with data, which can greatly affect how a system can perform a recommendation task.

This project helped us understand how complex it can be in dealing with a large amount of data (for our hardware), and how difficult it can be to develop a solution without making assumptions about the data.

Also, it is not easy to find a correct trade-off between a fast-to-execute algorithm and a precise one, as precise recommendations may require a deep data analysis that cannot be executed if the main goal is to maintain a fast system.

We believe a single query recommendation algorithm can't be successfully applied in every contest, as items/queries from different domains share different correlations between them that can variate across different scenarios.

References

[1] S.A. Dudani. "The Distance-Weighted k-Nearest-neighbour Rule"
[2] V. Klema, A. Laub. "The singular value decomposition: Its computation and some applications"
[3] User Rounakbanik, Kaggle. "The Movies Dataset" (www.kaggle.com/datasets/rounakbanik/the-movies-dataset)
[4] Pandas "Pandas library" (pandas.pydata.org/)
[5] Polars "Polars: Lightning-fast DataFrame library for Rust and Python" (www.pola.rs/)
[6] Dask "Dask library" (www.dask.org)
[7] Apache Spark "Apache Spark library" (spark.apache.org)