Data Mining: query recomendation

Pietro Demurtas*
pietro.demurtas@studenti.unitn.it
University of Trento
Trento, TN, ITA

Matyas Vincze*
matyas.vincze@studenti.unitn.it
University of Trento
Trento, TN, ITA

KEYWORDS

data mining, query recommendation, datasets, big data, data exploration

ACM Reference Format:
Pietro Demurtas and Matyas Vincze. 2018. Data Mining: query recomendation. In *Trento '22: Dataming a course about leveraging data, December 08–20, 2022, Trento, IT*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

1.1 Highlights

Query recommendation refers to providing suggestions to users in the form of queries over a given dataset. The recommendation is based on a set of query logs unique to each user. The evaluations of the users are not referred to the queries themselves, but instead to the queries' results. It differs from other recommendation systems because the evaluation of the user is not about the recommended item itself (i.e. a movie in a classical recommendation system) but rather on its results. In addition, the recommended query should be novel to the user and may be entirely generated ex novo by the algorithm. Our approach focuses on data reduction and summarization techniques to compute the latent factors, or "archetypes" of queries, how much each one is associated with each specific query, and how much a single user is interested in the specific archetype. Namely, our approach consists in filling the utility matrix using the SVT methods proposed by [1]. This well-experimented procedure allows us to iteratively fill the utility matrix in a way that scales efficiently to high-dimensional problems and was devised with the "Netflix problem" in mind. Up next, in order to generate the queries, our system selects a fixed number of latent factors with the strongest association with the user. From each of the chosen latent factors, our algorithm selects the queries with a high utility for the user. The queries get then scored on the dataset and all the results are merged according to the latent factor mostly associated with the query generating such results. Then, according to a "majority vote" system, explained in detail later, new queries are generated. The queries are generated in such a way that each query "describes" a set of results associated with the same latent factors. The expected quality of the generated queries is heavily dependent on the quality of the reconstructed utility matrix provided by the SVT. The quality of the SVT depends on the sparsity and dimensionality of the utility matrix. With an SVT RMSE of 0.28 we were able to score an average 0.88 rating across all the users for suggestions of 5 queries each.

1.2 Motivation and difficulties

Query recommendation systems have many real-world applications. They can be used to significantly speed up the search into a novel database where the user does not exactly know what they are looking for. A meaningful example is the role query recommendation could play during data exploration tasks. They could help the data scientist by suggesting queries associated with a portion of the data that is likely to be more interesting for the user. One of the main difficulties of the query recommendation problem is to find a mapping and link the domain of the recommended item, namely the query, and the domain in which user evaluation takes place, i.e. the slice of the dataset returned by the query. The link between these two domains and how to establish an information flow are nontrivial. Specifically, the most complex part is how to "break down" each query evaluation into its components and understand which part of the query yields data interesting to each user. A second non-trivial problem that we stumbled upon, once found the general interests of each user, is how to generate a query in such a way that describes or "satisfies" as much as possible each of those interests.

2 PROBLEM STATEMENT

In order to formally describe the query recommendation problem we first have to introduce the following entities:

- A **dense matrix** D representing a **relational table** populated by tuples. Each column of D corresponds to a different attribute and each row defines a distinct entry in the table such that $d_{i,j}$ represents the value of the i^{th} entry for the j^{th} attribute.
- A **set of unique users** U. Each user is associated with a log of its past queries such that u_i is the i^{th} query posed by the user u.
- A **query set** Q containing all the queries posed in the past. Each query is composed of a unique query id and the conditions of the query itself. The conditions of a query are expressed as a conjunction of conditions over the attributes of D such that

  $$Q_{cond} = ((attr_1 = value_1) \wedge ... \wedge ((attr_p = value_p))$$

  with $attr_i \in D_{columns}$ and $p \leq |D_{columns}|$. Each query can be seen as a summary representation of all the rows of D whose attributes $\models Q_{cond}$.
- A **sparse utility matrix** $M_{|U|x|Q|}$ containing the evaluation of the queries in the query log by each user expressed on a

*Both authors contributed equally to this research.

Trento '22, December 08–20, 2022, Trento, IT

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

scale to 100 such that $M_{i,j}$ is the rating of the i^{th} user for the j^{th} query and $0 \leq M_{i,j} \leq 100$. Each user does not submit all the queries and thus the matrix is sparse: if user i has never posed query j $M_{i,j}$ will be empty.

After defining such entities the query recommendation problem can be divided into two parts.
The first part involves filling the utility matrix inferring the missing values; in other words, it means predicting the rating of a user regarding the results of queries that he has not posed.
More formally part 1 can be summarised as computing M' such that

$$M'_{i,j} = \begin{cases} M_{i,j}, & \text{if } M_{i,j} \neq \varnothing \\ f_\theta(M, i, j), & \text{if } M_{i,j} = \varnothing \end{cases} \tag{1}$$

Where $f_\theta(M, i, j)$ is the estimate that we wish to calculate with respect to M, i, j being respectively the sparse utility matrix, the user, and the query.
The second part of the problem consists in generating a new set of novel queries to recommend to each user starting from the dense utility matrix. These queries should maximize the expected utility (evaluation) of the user they are recommended to. The task of the second part is to define a general way to compute utility for each user and to generate queries maximizing such utility.

Part 2 can be formally defined as computing for each user u a set r of recommended queries such that

$$r = \underset{g_\phi(u,M')}{\arg\max} f_\theta(M', u, g_\phi(u, M')) \tag{2}$$

with

$$g_\phi(u, M') \notin u$$

(the generated query is not in the query log of the user).
f_θ is the same estimate as in part one and g_ϕ is a query generation function. Our overall goal thus sums up in developing and estimating f_θ and g_ϕ.

3 RELATED WORKS

3.1 Singular Value Decomposition (SVD)

Let A be a rectangular matrix of dimensions $m \times n$, then the SVD of the matrix A is given by $A = U\Sigma V^T$, where U is an orthogonal matrix of shape $m \times m$ containing the left singular vectors, V, is an orthogonal matrix of shape $n \times n$ containing the right singular vectors and Σ is a diagonal matrix of shape $m \times n$ with non-negative entries, containing the singular values of A in decreasing order. This formulation of the SVD can be expressed as

$$A = \sum_{i=1}^{r} s_i \cdot u_i v_i^T$$

where $r = \min(m, n)$ represents the rank of the matrix A, s_i is the ith singular value and $u_i v_i^T$ is the outer product of the ith left and right singular vectors. $u_i v_i^T$ by definition is a rank-1 matrix, therefore the SVD gives us the original matrix A as a linear combination of rank-1 matrix-es.

This opens the opportunity to approximate A using only the sum of the first k outer products where $k < \min(m, n)$ - effectively means that we are zeroing out some of the singular values by assuming that the contribution to the sum is negligible.

In our case, using SVD on the utility matrix can be interpreted as searching for latent factors between the users and queries in a lower-dimensional space. U represents the relationship between users and latent factors, S is the strength of the latent factors and V indicates the similarity between queries and latent factors. The lower-dimensional our approximation is, the more global information we are using to relate the users to the queries.

3.2 Singular Value Thresholding (SVT)

We can estimate the missing values of a sparse matrix using SVD, assuming that our original matrix had a low rank or a decently good low-rank estimation. [2], [1] has proven that most matrix-es M of rank r can be reconstructed by solving the following optimization problem:

$$\begin{aligned} &\min \|X\|^* \\ &\text{s.t.} \quad X_{i,j} = M_{i,j} \quad \forall (i,j) \in \Omega \end{aligned} \tag{3}$$

provided a sufficient, but surprisingly low number of samples from M making up the sparse matrix $M_{i,j}$. The operation $\|\cdot\|^*$ is the nuclear norm, which is the sum of the singular values.

SVT is an iterative method, applying a soft-thresholding rule to the singular values until the rank of the matrix is reduced to a desired level. It is not computationally heavy and can work well with huge matrixes.

Algorithm 1: Simplified Singular Value Thresholding

Input : sparse matrix X
maximum iterations $k_{\max}$
increment l
Output : dense matrix X^{opt}

$Y_0 = X$;
$r = 0$;
for $k \in [1, k_{\max}]$ **do**
  $s = r + 1$;
  while $\sigma_{s-l}^{k-1} \geq \tau$ **do**
    compute $SVT(Y_{k-1}) = [U_{k-1}, \Sigma_{k-1}, V_{k-1}]_s$
    $Y_{k-1} = U\Sigma V_T$
    $s = s + l$
  end
  $r = max\{j : \sigma_j^{k-1} > \tau\}$;
  $X_k = \sum_{j=1}^{r} (\sigma_j^{k-1} - \tau)\mathbf{u}_j^{k-1}\mathbf{v}_j^{k-1}$;
  if $\|X_k - X\|_F / \|X\|_F \leq \epsilon$ **then**
    break
  else
    $Y_k = Y_{k-1} + \delta(X - X_m)$, where X exists, else 0
  end
end

4 SOLUTION

We have decided to dived our solution in two parts describing our approach for filling the matrix and our approach to generate new queries. For simplicity the two solutions and algorithm are presented

as two distinct entities but they are actually part of the same implementation. The two parts can be ran in distinct moments in time or also sequentially using the SVT approximation also for query generation without the need to recompute the three matrices. The python implementation can be found in our Github repository .

4.1 Filling the utility matrix

In order to fill the utility matrix we implemented an SVT algorithm based on [1]. First, the missing values were filled with zeros, and we used the *scipy* library to store the matrix in a sparse format, for memory efficiency. Our implementation follows 1, with parameters:

- $\tau = 5 * \sqrt{\text{number of users} * \text{number of queries}}$ ([1])
- $k_{\max} = 1500$
- $\epsilon = 1e{-}3$
- $l = 5$

Based on our experience, the only one of these worth changing based on the use-case is $k_{\max}$, with reasonable values between 300 and 2000. The more iterations we do, the less our originally-known values will change.
At each iteration:

- SVD is calculated for Y_{k-1} with increasing number of singular vectors, until we have a singular value less than τ
- reconstruct the utility matrix
- create a Y_k using Y_{k-1} and the difference between the new and original utility matrixes
- stop if the difference between the new and original utility matrixes is negligible

4.2 Generating new queries

Once filled the utility matrix we resort back again to the Singular Value Decomposition of the matrix. Our solution for generating new queries can be divided into several different phases:

4.2.1 Highest associated factors identification. Our initial idea was first to identify the n latent factors more strongly associated with each user. n is a tunable hyperparameter. In order to do so we search for the columns of U, the right singular matrix, which contain the highest value across every row. In our case, the columns represent the latent factors or "properties" of the data and the rows represent the users, the values in the matrix represent how much each user is associated with each latent factor of the data. In other words, we somewhat search for $\arg\max_j U_i$ where U_i represents a user-row. In practice, we do not look just for the index of maximum value but for the indexes of the n highest values. These indexes correspond to latent factors mostly associated with a specific user i. We search for the most associated latent factor for all the users (rows of U) and store the results. (listing 2 lines 1-7)

4.2.2 Retrieval of the highest scoring associated queries. Once retrieved the latent factors each user was "interested" in, we decided to collect a set of queries strongly associated with each

[1] In the actual implementation we do not use argsort but argpartition since for our application they solve the same purpose and argsort runs in $O(n)$ instead of $O(nlog(n))$ as argsort does.
[2] We use the indexing notation of python to describe slices of array, i.e. assuming that it is sorted in ascending order $A[-k:]$ is the slice containing the k highest values, the last k elements of A
[3] N.B. the queries are sorted with respect to the ratings of user i

Algorithm 2: Query SVD Recommendation System

Input : dense utility matrix X,
query list Q,
data set D,
length of recommendation L
number of factors per user k,
number of queries per concept n,
number of queries per user-concept pair m with m<n,
Output : list R of recommendation for every user

$U, \Sigma, V = SVD(X)$;
/* Fetch the highest associated factor for each user */
$top\_factors\_per\_user = []$;
for $i \in [0, |U_{rows}|]$ **do**
  $top\_factors\_per\_user[i] = ARGSORT(U_{i,-})^1[-k:]^2$;
end
/* Fetching the most associated queries for each latent factor */
$top\_queries\_per\_factor = []$;
for $i \in [0, |V_{rows}|]$ **do**
  $top\_queries\_per\_factor[i] = ARGSORT(V_{i,-})^1[-n:]^2$;
end
/* Create the query list for every user-factor pair and the list with the results */
queries_per_user-factor_pair = [] ;
$results\_per\_u - f\_pair = []$;
for $i \in [0, n\_users]$ **do**
  for $j \in top\_factors\_per\_user[i]$ **do**
    $queries\_per\_user - factor\_pair[i][j] = SORT(top\_queries\_per\_factor[j])^3[-m:]$;
    for $q \in queries\_per\_user - factor\_pair[i][j]$ **do**
      $results\_per\_u - f\_pair[i][j].append(D.query(q))$
    end
  end
end
/* Generate the new queries, one at the time from each concept */
$Recommendations = []$;
for $u \in [0, |U_{rows}|]$ **do**
  $generators = []$;
  for $f \in top\_factors\_per\_user[u]$ **do**
    $generators.append($**MajorityVote**$(results\_per\_u - f\_pair[u][f])$;
  end
  $i = 0$;
  while $len(Recommendations[u]) \neq L$ **do**
    $new\_r = next(generators[i\%k])$;
    if $new\_r \notin Recommendations[u]$ **then**
      $Recommendations[u].append(new\_r)$;
    end
    $i++$
  end
end
return Recommendations

Algorithm 3: MajorityVote

Input : table of data points Dr associated with given factor
Output : yield one recommended query R at each call

```
pert = [1] ;        // array full of 1 of len = |dr_rows|
sorted_Dr = Utility_sorted_counts(Dr) ;
while True do
    sorted_Dr.weighted_utility = sorted_Dr.weighted_utility * pert ;
    conditions = [] ;
    for attribute ∈ Dr_columns do
        candidate_value = Dr[attribute][0] ;
        candidate_votes = sum(sorted_Dr[attribute == candidate_value].weighted_utility)) ;
        quorum = sum(sorted_Dr.weighted_utility)/2 ;
        if candidate_votes > quorum then
            conditions.append(attiribute == candidate_value)
        end
    end
    Yield Query(conditions) ;
    pert = random[0.2,1.8] ; // updates pert as an array of bounded random numbers of len = |dr_rows|
end
```

Algorithm 4: Utility sorted counts

Input : table of data points Dr associated with given factor. every row represents a datapoint and every column is an attribute
Output : table of data points Dr associated with given factor sorted wrt to the weighted utility
return input data points with a new column "weighted utility" containing the utility of the query generating each row. If two or more rows have the same values for all the attributes they get collapsed in a single row with the corresponding weighted utility of (number of identical rows * utility of the query generating them). The returned table is sorted with respect to the weighted utility column.

factor. Our first idea was to pre-compute all the queries associated with each latent factor by searching in a similar way as before by looking for the columns of V, the left singular matrix, which contain the highest value across every row. In this case, every row of V represents a latent factor and the columns represent the queries thus the values in V represent how much each latent factor is associated with each query. In other words we sort of search for $\arg\max_j V_i$ where V_i represents a concept-row. As in the phase before we do not just look for the index of the max value but rather for the indexes of the n highest values. This possible implementation could be very handy since all the most associated queries for each concept could be computed beforehand and when prompted for a recommendation just use the n factors mostly associated with the specific user to produce the recommendation. However, after some further development and testing, we recognized the fact that some collected queries for each latent factor had very different utility even across all the users even if they shared the association with the specific latent factor. This introduces some instability during the query generation phase. For this reason the final version of our algorithm employees a slightly different strategy and uses this model as a baseline to test improvements. (listing 2 lines 7-12)
Instead of just looking for the most associated queries with each concept we decided to look for the most associated queries which have been assigned a high utility score by a specific user. In this way, the set of queries is unique for each pair user-factor (across only the top n factors associated with the user). In other words, after obtaining the indexes corresponding to the top queries associated with a latent factor we sort the queries wrt the rating of the user we are generating the recommendation for, and then we take only the top m scoring queries where m (number of top scoring queries per user per concept) < n (number of most associated queries per concept). (listing 2 lines 13-27)

4.2.3 Generating new queries from latent factors. The next phase in our algorithm involves querying the data-set to collect the pieces of data associated with each concept. We produce a set of data points containing all the results of the queries for each user-factor pair. All the data points are associated with the utility score of the query generating them. Next, each set is sorted according to the expected utility of the generating query (the query that "pointed" to those data points), if there are multiple data-points with the same set of attributes (which can happen and is application-specific) we sort the data according to the counts weighted by the utility of each query. Next, we do employ a system of "majority voting" to generate new queries from each set. Our underlying idea is that all the data-point with high utility and strong association wrt to a user-factor set represent what a specific user does like about a specific factor. In addition, we make the assumption the data-points in this set should lie in a confined region of the feature space where the utility function is very high by construction. The fact that they lie close to each other in the feature space is very important as it allows us to define a "common description" in terms of features (i.e. the values for the attributes). With this in mind, we devised a way to generate a query that "describes as much as possible" the underlying data. Starting from the sorted set first k elements we iterate over the attributes of the data set to define which value will be set for the conditions of that attribute in the generated query. For a given attribute, the condition of the query is set to a given value if the "majority" of queries exhibit that value, else the query will have no condition for that attribute. It is important to remark that during the "majority voting" the value of the vote of each query is the weighted count described before. The quorum of the voting can be changed to different values from the default absolute majoritarian setting (the votation is successful if 50%+1 agrees). In this way, we are able to select the only attribute that describes and thus points to, at least half of the data of interest. It is worth noting that we are able to produce more than one query from each set of user-factor data by injecting noise in the weighted counts of the queries resulting in different outcomes of the majority vote. We expect the resulting queries to have a slightly worst quality compared to the one generated without perturbation, however since

we employ multiple filters on the association and utility during the selection of user-factor data we expect nonetheless queries with high enough utility. (listing 2 lines 28-45)

Tested and rejected ideas - the user tree. While developing the majority vote system we also experimented in other ways of producing a description, and thus a query, of the user-factor sets. One promising idea was to grow a decision tree for every user in such a way that the leaves would have data with low entropy wrt factor-association of the data points. (i. e. a leaf would hold only the data points associated with latent factor 1). Once the tree was grown, in order to provide a description of the dataset we could just walk the path from the root to each leaf and generate the condition of the generated query on the basis of the splits of the nodes in the path. However, due to the low expressivity of decision trees and the complex nature of the problem, the trees were not able to segment the data set with high enough accuracy and this would have had a huge impact on the quality of the generated query and thus we ditched the idea.

The pseudo-code for the whole new query generation algorithm and auxiliary functions can be found in listing 2, 3 and 3. It is important to note that the aim of the pseudo-code is to clarify the workflow of our idea and differs slightly from the python implementation. In the actual python implementation, some generic entities (i.e. the top queries associated with each factor) get pre-computed while the user-specific ones (i.e. the queries in the user-factor set) get computed when prompted for the recommendation over a specific user.

5 EXPERIMENTAL EVALUATION

In order to develop and test our algorithms we have decided to employ both synthetic and real-world data. We tested the matrix filling SVT on both synthetic and real-world data, while due to the scarcity of real-world data, we tested the new query generation only on synthetic data.

In order to produce ratings for our synthetic dataset we have decided to define a seed for each user which consists of results of a given number of queries; the rating of each query by a specific user is then computed as $\frac{size-of(query-results \cap user-seed)}{size-of(query-results)}$. This method of generating ratings allows us access to ground truth ratings for all the queries not rated by the user in the utility matrix. This same mechanism could be used to estimate the utility of a generated recommended new query. All the python scripts used for the generation of the dataset and the utility matrix as well as the data itself are provided in our Github repository. For a more detailed description of synthetic data please refer to Appendix A

5.1 Testing on real data

We evaluated the SVT on multiple real recommendation system datasets from Kaggle. A portion of the known values was masked and treated as Nan, so we could measure the reconstruction quality of the originally-known values and how well it can fill not-known values at the same time.

The first dataset is [4], with 610 unique users, and over 194k movies. 1.7% (100836 values) of ratings were available, out of which we masked 1% (1008 values).

The second used dataset is [3], where we sampled 30k available ratings, to fit in the memory. 2.35% (30000 out of 1277694) ratings were available, and 1% (300) got masked for evaluation purposes. We run the SVT for 1000 iterations each, with the parameters described in 4.1.

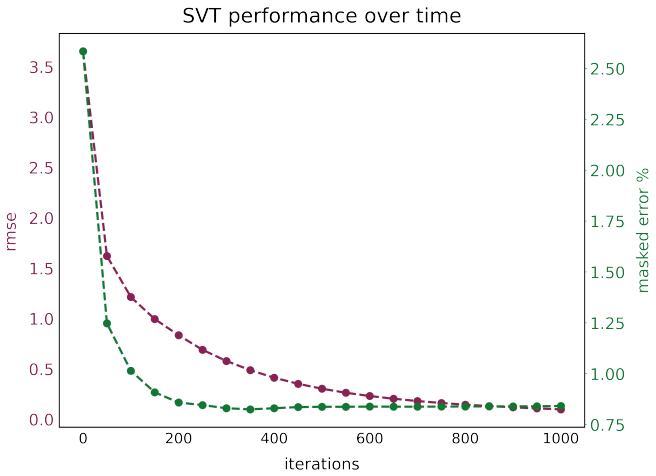

Figure 1: Plot of reconstruction RMSE and prediction quality for the Movies dataset

Dataset	Size	Nan %	reconstruction RMSE	masked RMSE
MovieLens-20m	(239, 5346)	97.65	0.02703	1.095
MovieRecom	(610, 9724)	98.3	0.09893	0.8415

Table 1: SVT evaluation on real data.

The reconstruction RMSE was very low in both cases, and the masked RMSE is comfortably low which shows that the SVT could be used as a backbone for making recommendations in real-life applications. Furthermore, the reconstruction RMSE in the case of the MovieLens-20m dataset is higher, as we took a random subset of the available ratings, resulting in a less complete dataset with fewer available ratings for each movie. This is the reason why although the MovieRecom dataset is bigger in scale, the SVT does better in prediction.

We visualize the progression of the SVT every 50 iterations at 1, where we can observe that the masked RMSE decreases quickly, and stagnates over time while the reconstruction RMSE decreases slowly. This suggests that early stopping criteria and re-usage of the originally-known values could give a competitive and even more cost-effective solution to the utility-matrix filling problem.

5.2 Testing on synthetic data

We tested the whole recommendation system on several instances of synthetic data generated with a custom python script. Table 2 reports the results of the system quantified in terms of Rooted Mean Square Error for the filling of the utility matrix and average user rating over 5 recommend queries per user for the generation methods. The algorithms taken into consideration are:

- Baseline algorithm
- QSRS(c) in which the function `utility sorted counts` sorts according to frequency only

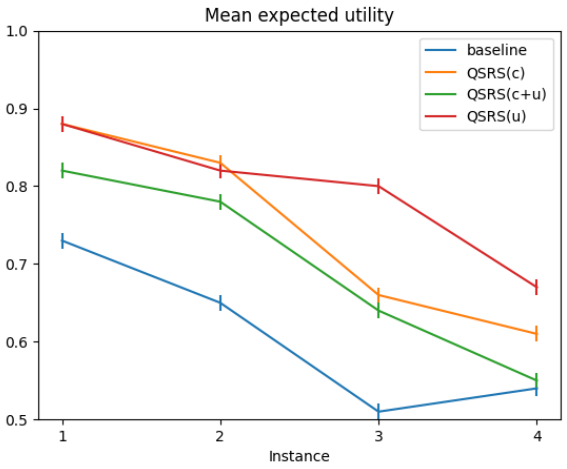

Figure 2: Plot of mean utility for the new recommended queries on several instances of the data

- QSRS(c+u) in which the function `utility sorted counts` sorts according to frequency and utility, identical to the one described in pseudocode
- QSRS(u) in which the function `utility sorted counts` sorts according to utility only

Table 4 reports the parameters of each instance of the synthetic data.

Instance	reconstruction RMSE ↓	baseline ↑	QSRS(c) ↑	QSRS(c+u) ↑	QSRS(u) ↑
1	0.28	0.73	0.88	0.82	0.88
2	0.29	0.65	0.83	0.78	0.82
3	0.31	0.51	0.66	0.64	0.80
4	0.35	0.54	0.61	0.55	0.67

Table 2: Reconstruction RMSE and mean utility for the new recommended queries on several instances of the data

As we can see from Image 2 the tree improved algorithms managed to achieve very high results for simpler instances of the data-set. However, as the instances become more complex and the utility matrix becomes more sparse we can clearly see that all the variants are subject to a decrease in performance. We can observe from the plot that the variant based only on utility sorting manages to achieve better results on more complex data-sets and to achieve similar results to the other variants. In general, we can say that all tree tested variants managed to achieve satisfactory results and all of them showed significant improvement over the baseline.
We also wanted to investigate the behavior of our algorithm and its variants with a data-set consisting only of unique entries. Results are reported in table 3 the results of this second test partially confirm what we have found so far. One major difference is the fact that QSRS(c) and QSRS(c+u) managed to outperform QSRS(u) on simpler instances. This may sound counterintuitive at first since we do not have counts to keep track of since all data-points are unique, however, it is justified by the fact that without sorting by count (or rather doing it with all equal counts) does not change the structure of

Instance	reconstruction RMSE ↓	baseline ↑	QSRS(c) ↑	QSRS(c+u) ↑	QSRS(u) ↑
1	0.28	0.78	0.93	0.94	0.88
2	0.29	0.67	0.85	0.89	0.82
3	0.31	0.51	0.68	0.69	0.78
4	0.35	0.57	0.63	0.64	0.67

Table 3: Reconstruction RMSE and mean utility for the new recommended queries on several instances of the data with unique entries

the set and thus more similar data points are chosen for majority voting. This results in less variation in the recommendation which may lead to improved performance on simpler instances but is penalized on instances with sparser data and fewer ratings per user.

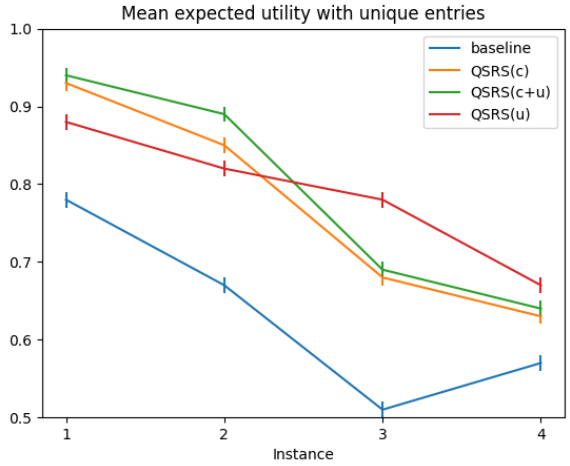

Figure 3: Plot of mean utility for the new recommended queries on several instances of the data with unique entries

In general, we can say that for both tests the QSRS(u) variant performs better on more complex instances and is more robust to variation in the sparseness of the data. We can see that in both cases the change in performance of QSRS(u) reflects the change in performance of the SVT algorithm as we expected. As the complexity of the data-set increases there are lees high utility results also due to estimates of the SVT, this may be one of the reasons for the better performance and stability of QSRS(u) which is able to exploit at best the fewer high utility estimates. We wanted to investigate the behavior of our algorithms by testing more complex data-set with higher dimensionality however even if we were able to obtain recommendations in a reasonable time for all the variants, bench-marking the results and quantifying the performance became too computationally expensive to run on our local machines.

6 CONCLUSION

In conclusion, we have seen how the SVD framework could be applied to the query recommendation task. Testing on real-world data showed us that the SVT algorithm is able to fill the utility matrix with high fidelity even in the case of very sparse data and is able to scale well with dimensions. Testing on synthetic data allowed us to

Instance	Queries	Users	Queries per user
1	2000	50	1000
2	2000	100	500
3	2000	300	200
4	2000	300	100

Table 4: Parameters of the instances of the synthetic data.

test the generation capabilities of our approach, we have seen that the performance of the more robust variant of our algorithm is tightly linked with the performance of the underlying SVT backbone in reconstructing the utility matrix. For this reason, we speculate that in real-world applications the utility of the recommended queries could be better than the one obtained on synthetic data as the reconstruction RMSE is lower. However in order to confirm this we would need extensive testing on a complex real-world query dataset. In addition, to further quantify the performance, could be useful to test the whole algorithm on synthetic data generated in a different way and with more complexity. Unfortunately due to time and computational constraints, we were not able to generate very complex and big synthetic data and test our full pipeline on it. Another possible future direction could be, once identified the interests of each user by the means of SVD, to experiment with different methods to generate the queries, different from our proposed majority vote system.

REFERENCES

[1] Jian-Feng Cai, E Candes, and Zuowei Shen. 2008. A singular value thresholding algorithm for matrix completion. arXiv, math. *OC* 810 (2008), v1.
[2] Emmanuel J Candes and Benjamin Recht. 2008. Exact low-rank matrix completion via convex optimization. In *2008 46th Annual Allerton Conference on Communication, Control, and Computing*. IEEE, 806–812.
[3] GroupLens. 2019. MovieLens 20M Dataset, Kaggle. https://www.kaggle.com/datasets/grouplens/movielens-20m-dataset.
[4] Shinigami https://www.kaggle.com/gargmanas. 2021. Movie Recommender System Dataset, Kaggle. https://www.kaggle.com/datasets/gargmanas/movierecommenderdataset.

A SYNTHETIC DATASET

In this section we will provide a detailed description of the synthetic data files and their generation. The same information can be found in the readme file of the data folder of our Github repository. Please note that even if our data generation script is able to generate also continuous attributes we tested our algorithm only on discrete ones.

the main folder contains all the data generated by the `generator.py` part of `datagen` module. The files in the data folder are:

- **dataset.csv** contains the relational table upon which the rest of the data is generated from. It is composed by several entries each one possessing a given number of attributes. Attributes can either be discrete or continuous. It is possible to generate data consisting of mixed attributes or just discrete or continuous attributes. Discrete attributes values for each row are sampled with a random probability from a predefined list of possible values. The length of the list of possible values is a tunable parameter. Continuous attributes are drawn from a normal distribution with fixed standard deviation and mean.
- **query_log.csv** contains a list of non-empty queries generated on the base dataset. Conditions over discrete attributes are expressed as equalities while conditions over continuous attributes are expressed as inequalities for simplicity.
- **user_list.csv** contains the list of unique id users.
- **utility_matrix.csv** contains the sparce utility matrix with the rating of each user. It is possible to tune how many queries rates each user. Ratings are calculated by associating a seed set of queries for each user representing the ideal query with maximum utility. Comparing the indexes of the data returned by the seed queries and the indexes of the data returned by the rated query is it possible to calculate the rating as $\frac{size-of(query-results \cap user-seed)}{size-of(query-results)}$.