Evaluating Dataset portions based on query logs

Samuele Angheben 240268
Marco Frassoni 194513

ACM Reference Format:
Samuele Angheben 240268 Marco Frassoni 194513 . 2023. Evaluating Dataset portions based on query logs. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTIONS

1.1 Query recommendation system

The use of relational databases is known to be widespread in all kinds of applications where it is needed to store data and be able to quickly access it through a specific search called query.

An example are the web applications that need to store a lot of products/items and the various information about the users, in this case when a user filters by features the items a query is used and the Result-Set, composed by some items, is shown to the user.

More specifically a query to a database is a request for specific data and the result of this operation is a subset of the data, called Result-Set, that satisfies the restriction of the query.

To come up with a specific query that returns the Result-Set that we would like can be challenging, moreover if the database is very large. Furthermore it can be interesting to have an automated way to suggest a query to the user, on the basis of its past interactions with the system.

Since the data is very large and usually the query done by each users are a lot, an idea to improve the Result-Set for a specific user and to understand if the Result-Set would be liked, is to use a query recommendation system. Query recommendation systems can be particularly useful for users who are not sure what to search for, or who are looking for specific information but are not sure how to phrase their query.

1.2 Goal and solution

The goal of this work is to develop a sophisticated query recommendation system with the hypothesis of using a single relational table to contain the data and that each query can only be the conjunction of a set of equals attribute-value condition.

To achieve this goal, one possible strategy is to implement collaborative filtering by harnessing the ratings expressed by the users in order to determine the utility of a query. Collaborative filtering takes into account the ratings expressed by similar users, where similarity is meant with the user for whom the estimation is done and it is calculated using a proximity distance measure - such as Jaccard similarity or Cosine similarity - when declined in its user-user view. Collaborative filtering could also be viewed from an item-item perspective: in this case, the utility of a query would be calculated on the basis of the ratings given to queries that are similar to it. To create groups of users or items, such that members of each group are close/similar to each other while are dissimilar to members of other groups, is is possible to use clustering techniques.

Another possibility is to use content-based filtering, which takes into account the characteristics of a query and deduce the utility based on what the specific user is interested in, known as user-profile. In this problem there are two ways to consider the characteristics of the query: retrieve the information by the logic form of the query or take in consideration the answer set returned by query. This last approach treats a query as a set of tuples, so different analysis can be performed, for example Jaccard similarity to find similar query and frequent itemset algorithm to find the most retrieved tuples of a user, and so deduce the user profile that can be a single tree with at the root the frequent itemset or a forest of differents trees.

Our idea to implement the query recommendation system, at the moment, is to adopt an hybrid approach and combine the result of the two methods, collaborative filtering and content based. Then with the experiments and tests manually tune the coefficient of the combination to maximize the performance.

1.3 Relevance and similar problems

This type of solution is relevant because can be applied and adapted to many different applications, even considering the restriction about the data and the type of the query, moreover can be used as a base for solution in a more general environment.

This study differs from the known general recommendation system because what do users rates are sets of tuples returned with the answer set and not a single item, whilst the aim of the recommendation system is to help users to build the query, finding the right conjunction of simple yet complex conditions - given that there are a lot of possible values for each single attribute - that most fulfill their needs returning data of interest. This differentiates this project from a common recommender system, such as for example streaming services or online stores - where the goal is to foster users' explorations of new items to be then consumed or buyed.

2 PROBLEM STATEMENT

In this section a formal definition of the problems is described.

2.1 Definitions

Definition 1 (Relational Table). *A **relational table** is a structure used by DBMS to store the data in an efficient way, for this work*

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

*is sufficient to consider it as a matrix $T_{m,n}$ with m rows, called **tuples**, and n columns.*
*The **attribute vector** is the vector $A_n = [AT1\ AT2\ ...\ ATN]$*
*where ATx represent the **attribute** of the corresponding columns x in the matrix T.*
Each column of the matrix has the same types, it can be: numerical, boolean or a string.

Note that with this definition is not possible to have the NULL value.

Definition 2 (User-set). *The **User-set** (US) is defined as following:*

$$US = \{\, x \mid x \text{ is a } \textbf{User} \,\}$$

where
***User** is the object representing a user of the system and is defined by an unique **userId**.*

Definition 3 (Query-set). *The **Query-set** (QS) is defined as following:*

$$QS = \{\, x \mid x \text{ is a } \textbf{AndQuery} \,\}$$

*where **AndQuery** is the object representing a query is a vector defined in this way:*

$$AndQuery_l = [queryId, ATx = Valx, ... = ..., ATy = Valy]$$

where
***queryId** is the unique identifier*
*$ATX \in A_n$, is an attribute from the **attribute vector***
Val is a compatible value to the corresponding attribute
$l \in [1 : n + 1]$

the query is consider in logic as:

$$AndQuery = (ATx = Valx) \wedge (... = ...) \wedge (ATy = Valy)$$

Definition 4 (Result-Set). *The **Result-set** (RS) of an **AndQuery** q in respect to a **relational table** T is defined as following:*

$$RS_{q,T} = \{\, T_{x,*} \mid x \text{ satisfy } \boldsymbol{q} \,\}$$

*The Result-set RS is the set of **tuple** of the relational table T that satisfy the constrains of q. This operation is handled by the RDBMS.*

Definition 5 (User-Query utility matrix). *Is a sparse matrix $U_{|US|,|QS|}$ where:*

$$\begin{aligned} &U = US \times QS \\ &U_{i,j} \in \{[0 : 100] \cup \{NULL\}\}, \quad \forall i \in |US|, j \in |QS| \end{aligned} \tag{1}$$

where
NULL, is a symbol to indicate an empty rating
*Each value in this matrix, is called **utility** and indicate how much the user liked the **Result-Set** of the specific query, if the value is NULL then the rating is undefined.*

2.2 Problem statements

2.2.1 Problem A. Given the utility matrix U which contains some NULL values, since some users haven't rated some queries, fulfill all the NULL value of the matrix, we call the complete matrix $\hat{U}$. That it, predict the missing ratings. After this operation create a procedure to retrieve the top-k queries, not already posed, for a specified user.

Fulfill the utility matrix is the core of every recommendation system, the idea is to predict for each users the utility of the Result-Set of the queries that are not already been rated by the specific user. Once the utility matrix is complete the *top-k* procedure is used to suggest or understand which query are of interest to a user of the system.

2.2.2 Problem B. Given the Utility Matrix $\hat{U}$ develop an algorithm to compute the rating u_{NQ} for a query NQ where:

NQ is an AndQuery
$u_{NQ} \in \{[0 : 100]\}$ is a value that indicate how much the Result-Set of the query NQ is liked in general by all users.

This problem, to be addressed after having completed the first one, is to define a way to compute the utility of a query in general for all the users. This means that the measure of utility should not be tailored on a specific user, but measured on the whole user set. And as before the utility of a query is bound to the users' interest in the set of tuples, namely the Result-Set.

2.3 Descriptive Definition

Consider a system with different users, a relational table with a lot of tuples and a query-log that contains a lot of queries that are already posed in the past. In addition some users have already rated some queries, more precisely, given a value between 0 and 100, the utility, that indicates how much the Result-Set of the query is liked. The problem A asks to predict the utility for each user for each query that is missing and a way to retrieve the top k query that a specific user would be interested in the corresponding Result-Set. Problem B asks to evaluate the utility of a new query considering the behavior and prediction of each user and each query on the system. The solution to the aforementioned problems leads to the construction of a query recommendation system.

3 RELATED WORK AND TECHNOLOGIES

3.0.1 Pyspark. The query recommendation system has been developed using the Pyspark framework, which is an interface for Apache Spark in the Python language that supports most of Spark's features. Pyspark enables the usage of Pyspark SQL DataFrame, a distributed collection of data grouped in named columns, used to aggregate, query, manipulate and transform data. In addition to this it enacts the possibility to split and parallelize the computation over different Resilient Distributed Dataset (RDD). Since a DataFrame is equivalent to a relational table in Spark SQL, Pyspark give the possibility to execute SQL query on the data thanks to the Spark SQL execution engine.

3.0.2 LSH. Locality sensitive hashing (LSH) is a widely popular technique used in approximate similarity search, having at its core an hashing function that allows to group similar items into the same hash buckets. This technique is used either in the nearest neighbors and data clustering problems.

3.0.3 Bucketed Random Projection for Euclidean Distance. Bucketed Random Projection is an LSH family for Euclidean distance. The Euclidean distance between two points is defined as follows:

$$d(x, y) = \sqrt{\sum_i (x - y)^2}$$

Its LSH family projects feature vectors x onto a random unit vector v and portions the projected results into hash buckets:

$$h(x) = \left\lfloor \frac{x \cdot v}{r} \right\rfloor$$

where r is a user-defined bucket length. The bucket length can be used to control the average size of hash buckets (and thus the number of buckets). A larger bucket length (i.e., fewer buckets) increases the probability of features being hashed to the same bucket (increasing the numbers of true and false positives).

3.0.4 MinHash for Jaccard Distance. MinHash is an LSH family for Jaccard distance where input features are sets of natural numbers. Jaccard distance of two sets A, B, is defined by the cardinality of their intersection and union:

$$d(A, B) = 1 - \frac{|A \cup B|}{|A \cap B|}$$

MinHash applies a random hash function g to each element in the set and take the minimum of all hashed values:

$$h(A) = \min_{a \in A}(g(a))$$

The input sets for MinHash are represented as binary vectors, where the vector indices represent the elements themselves and the non-zero values in the vector represent the presence of that element in the set

4 SOLUTION

The design of the query recommendation system described in this work and its development comprises the application of different recommender methodologies using a multiple linear regression model. The recommenders are hence combined in order to enhance the predictions by exploiting the results of the application of both content-based and collaborative filtering.

4.1 Problem A - Fulfill the utility matrix

Content-based filtering is used to exploit item features in order to produce a prediction based on the ratings of similar answer sets rated by the users. We consider items , for the purpose of building the item profiles and to compute similarity measurement, not as queries posed by users but the answer sets that are returned by those queries. The items are hence composed by a list of tuples, with every tuple being unique among others and composed by a list of attributes that makes them easily comparable. Comparison between tuples return only two possible values: a pair of tuples can only be equal - meaning that they share the same exact attributes' values and as a consequence they are the same record from the database - or not, without any intermediate notion of similarity. As a consequence the distance between two answer sets is given by the number of not equal tuples returned. Two answer sets with the same cardinality and returning the same tuples are considered to be equals. Due to the nature of the answer sets, the similarity between the items is calculated using the Jaccard distance with minhash, which is commonly used to measure the degree of similarity between two sets of comparable items. It is important to notice that, for the implementation of the MinHash algorithm in our framework of choice, empty sets cannot be transformed by minhash, which means that any item must have at least one tuple in the answer set. This is granted by the algorithm we adopted to construct the initial utility matrix from the original dataset: we decided, because of the way we synthetically construct the query set and in order to build a model as realistic as possible, to filter out and hence not report in the utility matrix any query for which the answer set was empty. After the training model, which computed the minhash for every single item and specifically on a vector containing the item's answer sets, the same model is fitted over the dataset to join items with a distance measure below a given threshold. As a result of this computation, we produced a dataset holding for each pair of similar queries their Jaccard distance. This enables us to predict the rate, for every item which was not yet rated in the input data, given by a user to a query based on the average rating given to similar queries, i.e. to query with a Jaccard distance lower than the desired threshold.

The computation of the item-item model, as a result of the content-based technique described above, is the first step in the realization of the query recommendation system. In order to enhance the prediction by lowering the average error, we decided to combine the previously described technique with collaborative filtering method, by calculating the prediction on the basis of rating given by similar users, where similarity is calculated on the basis of the vector of the ratings from each user in the utility matrix. Similarity between users is computed over the vector containing the ratings expressed by each user, in order to identify users that like or dislike the same items. Ratings are first normalized by subtracting to each rate the user mean rating. In accordance with what is done for the content-based filtering, after the user-user model preparation the latter is fitted over the original data and we obtain a dataset holding for each user its similar peers and the relative distance measure between them. The rate for a query by a user is then computed as the average rating for that specific item given by every similar user to that specific item.

$r_{u_i}(q_j)$ is the rate for the query q_j given by user u_i
$p_{u_i}(q_j)$ is the predicted value for the query q_j and user u_i
$S_i = \{ux, uw, \ldots, uz\}$ is the set of similar users of u_i
$Q_i = \{ux, uw, \ldots, uz\}$ is the set of similar queries of q_i

$$p_{u_i}(q_j) = \frac{\sum_{x,y=S_i \times Q_i} r_{u_x}(q_y)}{|S_i \times Q_i|}, \quad r_{u_x}(q_y) \neq Null$$

The final prediction is the result of the average prediction returned by the two models described above. We observed that not all the values in the utility matrix are filled in this way, probably because of the sparsity of the original matrix or due to the relatively

small amount of data. We decided to calculate again the similarity between users considering the predicted ratings, since we add new data it is possible that different users result similarly, at this point we predict again the ratings as before, taking in consideration the item-item model and the new users similarity. We run this procedure two or three times. Since it's possible that some ratings are still missing For those values a baseline approach is adopted, computing the rating as the result of average of the ratings, either explicitly expressed by the user and predicted by the content-based and collaborative filtering techniques. At this point it is easy to find the top k queries for each user just by looping through each query for each user and returning the queries that have been predicted with the highest score.

4.2 Problem B - Rate a query in general for all the users

The problem B ask to compute the utility of a query q in general for all the users, the procedure that we propose is to calculate the result set of the query q, hash it with Minhash and find the most similar query q_{sim} considering Jaccard similarity in respect to all the queries saved in the utility matrix, that are already been hashed during problem A. As last step since we consider that the problem A is already solved, so the utility matrix is fulfill, we decided to average the ratings of all users of the query q_{sim}, since we didn't find any particular reason to give more prominence to one user rating than another.
An improvement to the described procedure could involve the use of the clustering techniques, to be applied on the answer sets returned by the queries posed by the users. Clustering would boost the performance of the procedure, since it would be much easier and quicker to compare the answer set returned by the investigated query - more specifically its signature - with a group of clusters instead that with every single result sets.

4.3 Pseudocode and comments

In this section the pseudocode with some comments of the core of our system is shown.

Algorithm 1: run_query

```
relational_table.createOrReplaceTempView("items");
result_set = {};
for q in query_set do
    q_id = get_q_id(q);
    q_string = get_query_attr(q);
    q_string.concatenateWith("AND");
    q_result_set =
     sql("SELECT id FROM items WHERE" +
     query_string);

    result_set.append(q_id, q_result_set);
end
return result_set;
```

The *run_query* function fetches the result set of each queries and return it as a *DataFrame*. As first step a View *items* is created from the *relational_table DataFrame*, then for each query the *query_id* and the query attributes are extracted, after the query is constructed and executed, the result is saved and appended with the q_id to the *result_set*.

Algorithm 2: calculate_similar_queries

```
result_sets = run_query(query_set, relational_table);
query_set.withColumn("vector", SparseVector(result_set));

query_set.withColumn("lsh", MinHashLSH("vector"));
calculated_similar_queries = approxSimilarityJoin(
        query_set, query_set, distance, JaccardDistance
);
return calculated_similar_queries;
```

To calculate the similar queries as first step the result set of each queries is computed, a new column *vector* is added to the *query_set*, it contains the *SparseVector* representation of the corresponding result set. Next the vector is hashed with *MinHashLSH* so it's possible to quickly calculate the Jaccard distance and lastly *approxSimilarityJoin* is done on the *query_set* in this way similar queries are paired.

Algorithm 3: calculate_similar_users

```
utility_matrix.withColumn("ratings",
                    DenseVector(Column – "user_id)
);
utility_matrix.withColumn("norm_ratings",
                Normalize("ratings"));
utility_matrix.withColumn("hashes",
        BucketedRandomProjectionLSH("norm_ratings")
);
calculated_similar_users = approxSimilarityJoin(
 utility_matrix, utility_matrix, distance, EuclideanDistance
);
return calculated_similar_users;
```

A similar methods is used to calculate similar users, in this case a *DenseVector* is created for each user with all his ratings stored in the *utility_matrix*, then each vector is normalized and *BucketedRandomProjectionLSH* is applyed. At this point the *approxSimilarityJoin* is calculated with the *EuclideanDistance* distance.

The prediction algorithm works as follow. First get the users-query as a tuple to predict and the one already predicted. Then the similar queries are calculated, the variable that specify the maximum iteration is set and the user-query to predict are count. In the while loop the similar user with the current *utility_matrix* are calculated and the predictions are done by first joining the similar users and similar queries to the ones to predict, then the join to the user-query predicted is done to search the similar user-query already predicted, at the end this *DataFrame* is *groubBy* the query-user and the average is calculated. In other words the last step for every user-query to predict collapses all the similar query-user with the average. As discussed later a better solution could be the one that take in consideration the Euclidean and Jaccard distance in the calculation. Then some values are updates, in particular in the *utility_matrix* are added the prediction, in this way new

Algorithm 4: predict_rating

```
u_q_to_predict = get_u_q_to_predict(utility_matrix);
u_q_rated = get_u_q_rated(utility_matrix);
Q_i =
 calculate_similar_queries(query_set, relational_table);
maxiter = 3;
n_to_predict = u_q_to_predict.count();
while maxiter ≠ 0 or n_to_predict ≠ 0 do
    S_i = calculate_similar_users(utility_matrix);

    pred = u_q_to_predict.join(S_i, "user_id")
                         .join(Q_i, "query_id"));
    pred = pred.join(u_q_rated,
          (pred["user_id_sim"] == u_q_rated["user_id"])&
    (pred["query_id_sim"] == u_q_rated["query_id"]))
    .groupBy("query_id", "user_id")
    .agg(avg("rating").alias("predicted_rating"));

    utility_matrix.add(pred);
    u_q_to_predict.subtract(pred);
    u_q_rated.add(pred)
    n_to_predict = u_q_to_predict.count();
    maxiter = maxiter − 1;
end
user_rating_mean = u_q_rated.groupBy("user_id")
       .agg(avg("rating").alias("predicted_rating");
pred = u_q_to_predict.join(user_rating_mean,"user_id");

utility_matrix.add(pred);

return utility_matrix
```

information are considered in the next cycle when the similar users are calculated. At the end if some ratings are yet to be predicted they are calculated with the mean of corresponding user rating.

5 EXPERIMENTAL EVALUATION

In order to evaluate the proposed solution and the produced prediction, there are different methodologies that could be put in place.

5.0.1 Dataset. The dataset used to train, run and evaluate the recommendation system has been derived from the data made available by the yelp data set, consisting of different files. The data consists of the list of registered businesses, the list of yelp users and the ratings given by a user to a business. For each business and for each user a unique identifier is assigned, which acts as a key reference in the ratings table. In addition to the unique id, the businesses are provided with a list of attributes, most of which have been then extracted and loaded into the relational table, after a process of data cleansing that comprised the assignment of default values to null fields and the homogenization of the data values. The resulting table enacts the creation of the answer sets in response to the queries posed by users, which have been synthetically created by randomly selecting from one to five attributes from the available table fields and, for each different attribute, a value in accordance to the attribute's value set. We obtained the utility matrix, to be given in input to our solution, from this data by applying the following steps. For each user we assigned a random number of queries in a chosen range. We then computed the rating of each query as the average of the ratings given by that user to the businesses returned in the query's answer set, scaling up then the average, which was a number between 0 and 5, to a value in the range 0-100. The sparsity of the utility matrix hence obtained has a value of 6%. Since the dataset is built in this way it is possible to reconstruct the rating of a particular query and user to compare the result with the rating of the recommendation system.

5.0.2 Evaluation. The predictions could be evaluated explicitly by the same users, by rating queries and the corresponding result sets, confirming or rejecting in this way the prediction. This solution is not considered to be feasible due to the impossibility of contacting all the users to collect their feedback.
As a consequence, the chosen evaluation methodology is to compare performance of the developed query recommendation system - whose technical details have been presented in the previous chapters - with the predictions produced via other techniques, such as for example the global baseline estimate or the Alternating Least Square (ALS) algorithm. The comparison is realized by computing the and comparing the Root Mean Square Error (RMSE) of the different algorithms. The RMSE is a measure of the accuracy of prediction algorithms, in which predicted ratings for a test dataset of user-item pairs are compared with the rating values that are already known. The difference between the known value and the predicted value would be the error. The RMSE is computed by squaring the sum of all the error values for the test set, finding the average, and then taking the square root of that average. The utility matrix is then splitted in order to feed the algorithms with a training set and a test set. The latter is not dimensionally fixed but randomly chosen as a subsection of the utility matrix given only the desired dimensions, namely the number of users and queries to be considered within. For all the utility matrix entries in the test set the corresponding values are obfuscated - i.e. replaced by a null value in the input dataset - in order to let the recommendation system predict them and to be able later to compare the rating given by the user, not used to train the recommendation algorithm, with the predicted one.

The obtained RMSE indicates how far is in average the prediction from the real values, evaluating in this way the performance of the solution. We observed that our solution could be tuned by manipulating the parameters used in the collaborative and content-based filtering, in particular the thresholds to the distance measures set for the search of nearest neighbors and the identification of similar items.

5.1 Suggested optimizations and improvements derived by the evaluations

5.1.1 Execution speed. Considering the problem of fulfilling the utility matrix that for its nature is quadratic, if we consider the utility matrix quadratic and sparse, the prediction of a single rating should be fast to be performed on a large dataset, even if spark helps with its ability in the parallelization of the execution.
We saw that our solution is almost twice as slow compared to the

ALS solution, the idea to reduce the execution time of our algorithm is to use clustering to reduce the time spent to search similar query and similar items. That is, for example: instead of searching for similar users between all users, search in which cluster the user is closer and return the users in that cluster.

5.1.2 Improve the prediction with weighted average. Our system when it retrieves similar users and similar queries to a specific user and query it averages each value of the cross product similar_user-similar_query. An improvement could be to take in consideration the distance, in this case Jaccard for similar queries and Euclidean for similar users, and weight the average accordingly, for example by applying the log function to the distance and then calculate the average.

5.1.3 Technique to improve the prediction for this specific problem. Since we assume that the rating of a query is the average of the rating of the single items in the result set we can exploit this rule to improve the prediction of our recommendation system trying to predict the correct rating of every single item for each user. The technique works as follow:
The first step is to retrieve the users and queries that are already rated in the utility matrix (user_query_rated), then fulfill the utility matrix with the recommendation system. At this point take in consideration a single user, we loop through each query and for each item on the corresponding result set we assign the rating of the query of the user. Now it is possible that a single item has multiple ratings since an item can be in different result sets, we average all the ratings of every single item. Now for each user_query_rated of the user we linearly rescale the rating of the items in the result set in a way that the sum of each item in the result set divided by the number of items in the result set equals the rating of the corresponding query. Now for each query of the user not in user_query_rated we assign the value of the query as the sum of each item in the result set divided by the number of items in the result set. This procedure can be executed multiple times and the rating of every single item should converge.
Then execute the procedure for all the other users.
This procedure tries to reconstruct the dataset that we used to construct the utility matrix, but with a rating between 0 and 100 instead of 0 and 5, and in some sense can be considered a different problem but the idea is to use this knowledge to improve the predictions of our problem.
If this technique is applied is possible to solve the problem B in a different way, recall the problem B statement: compute the utility of a query q in general for all the users. The idea is to calculate the result set of the query q and then reconstruct the rating with the average of the rating of every single item (that we reconstruct previously) for every single users. To notice that this is a different methods than the one describe before, here we didn't use the user-query utility matrix to do the calculation but the derived user-item utility matrix.

5.1.4 Construct the user profile in a different way. Another idea that we have to find similar users is to construct a user profile in as follow: Retrieve the most liked query (for example rating>70) of the user and perform a frequent item set algorithm on the result set of these queries, do the same for the queries that are ok (for example (40<rating<70), and for the ones that the user does not like (for example rating<40). Now the user profile consists of these three frequent itemset and the idea is to use them to find similar users. Even in this case it is possible to use clustering techniques to speed up the search of similar users.

5.1.5 Find the right similar threshold. Another point that we didn't explore is finding the perfect threshold when we search for similar user and queries. We think that our algorithm can perform much better with the optimal threshold that can be found with training models and machine learning techniques.

6 CONCLUSION

In conclusion with this work we were able to build a recommendation system that is able to predict the missing values and fulfill the utility matrix. The core idea of our algorithm is to take in consideration similar users and similar queries to predict the rating of the specific query for the specific user. We used LSH of Euclidean distance of the user rating of all queries to find similar users and LSH of Jaccard distance on the result set of every query to find similar queries. We constructed our dataset from a real dataset to be able to work on data as real as possible, and our construction technique also facilitates and permits a concrete evaluation since it gives the possibility to fulfill the entire utility matrix with the information of yelp dataset, that obviously we didn't take in consideration during the prediction step. Talking about execution performance we use Locality sensitive hashing that improves the system execution speed a lot, and we propose different techniques based on clustering to further improve the performance. Regarding the prediction of our system we saw in the evaluation process that common algorithms such as Alternating Least Squares (ALS) matrix factorization still perform better according to the RMSE evaluation metrics but further tests can be done using the optimal threshold for the distance metrics of our system. After the experiment evaluation we propose different ideas to further improve the query recommendation system. For developing our solution we used the pyspark framework mainly because it is designed to parallelize the computation and is based on the map reduce concepts, another important point is that it integrates a query engine that we used to build the result set for the queries.