COMPARISON OF MACHINE LEARNING ALGORITHMS IN SUGGESTING CANDIDATE EDGES TO CONSTRUCT A QUERY ON HETEROGENEOUS GRAPHS

by

ROHIT RAVI KUMAR BHOOPALAM

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT ARLINGTON

MAY 2016

Acknowledgements

I am grateful to my supervising professor Dr. Chengkai Li for giving me an opportunity to work under him. Without his excellent guidance, patience and continuous support this work would not have been possible.

I would like to thank Dr. Vassilis Athitsos and Dr. Dimitrios Zikos for serving as my committee members and for their invaluable comments, suggestions, and support.

I would like to especially thank Nandish for the constant guidance and comments he provided throughout this work. I would also like to thank the whole team of IDIR lab for all the suggestions they provided me.

Finally, I would like to express my gratitude to my parents for their encouragement and invaluable support through out.

April 21, 2016

Abstract

COMPARISON OF MACHINE LEARNING ALGORITHMS IN SUGGESTING CANDIDATE EDGES TO CONSTRUCT A QUERY ON HETEROGENEOUS GRAPHS

Rohit Ravi Kumar Bhoopalam, MS
The University of Texas at Arlington, 2016

Supervising Professor: Dr. Chengkai Li

Querying graph data can be difficult as it requires the user to have knowledge of the underlying schema and the query language. Visual query builders allow users to formulate the intended query by drawing nodes and edges of the query graph, which can be translated into a database query. Visual query builders help users formulate the query without requiring the user to have knowledge of the query language and the underlying schema. To the best of our knowledge, none of the currently available visual query builders suggest what nodes/edges to include into their query graph for users. We provide suggestions to users via machine learning algorithms and help them formulate their intended query.

No readily available dataset can be directly used to train our algorithms, so we simulate the training data using Freebase, DBpedia, and Wikipedia and use them to train our algorithms. We also compare the performance of four machine learning algorithms, namely Naïve Bayes (NB), Random Forest (RF), Classification based on Association Rules (CAR), and a recommendation system based on singular value decomposition (SVD), in suggesting the edges that can be added to the query graph. On an average,

CAR requires 67 suggestions to complete a query graph on Freebase while other algorithms require 83-160 suggestions. Moreover, Naïve Bayes requires an average of 134 suggestions to complete a query graph on DBpedia while other algorithms require 150-171 suggestions.

Table of Contents

List of Illustrations

List of Tables

Chapter 1

INTRODUCTION

The database community has recognized the importance of graphical query interfaces to the usability of data management systems. For querying graph databases, there exists a few systems which allow users to build queries by drawing nodes and edges of query graphs, which can be translated into underlying database queries. In these existing systems, a user has to go through an alphabetically sorted list of edge labels and select the edge label she is interested in adding to her query graph. This requires the user to know the exact edge label that she wants to add to her query graph, so that she can find her edge using binary search. Oftentimes, due to the lack of knowledge of the data and the schema, the user has to sequentially go through the list of edges.

This problem can be reduced by using a visual query builder system like Orion [1]. Orion supports two modes of operation. One is the passive mode, whereby adding an edge, a ranked list of edge types is shown in a pop-up. This ranking of edge types is based on the relevance to the user's query intent. Figure 1-1 shows Orion's passive mode of operation, where a ranked list of edge labels is suggested for a newly added edge between two nodes. These suggestions are displayed in a pop-up box. The other mode of operation is the active mode, in which the Orion suggests a top-k ranked list of edges based on user's query intent. Figure 1-2 shows the snapshot of a partially constructed query graph in an active mode, where the nodes and edges are suggested based on the partially constructed query graph. The white nodes and the edges incident on them are the suggestions given by the Orion.

The query construction process of an Orion user can be summarized as a query session, which consists of positive and negative edges that correspond to edge suggestions accepted and ignored by the user, respectively. This iterative process of suggesting edges to the user based on the partially constructed query graph can be viewed as a classification problem. Given a partially constructed query graph as an input to our algorithm, we are

interested in suggesting an edge or a group of edges to the user, and the edges are the classes that we want to find. Thus, we can model this problem of suggesting edges to the user based on partial query graph as a classification problem. Furthermore, we are only interested in using the classifiers that can provide scores to the set of edges (or classes) based on the partial query graph (or input data), so that we can sort the edges based on these scores and suggest a ranked list of edges to the user. In our problem, the number of classes is equal to the number of distinct positive edges.

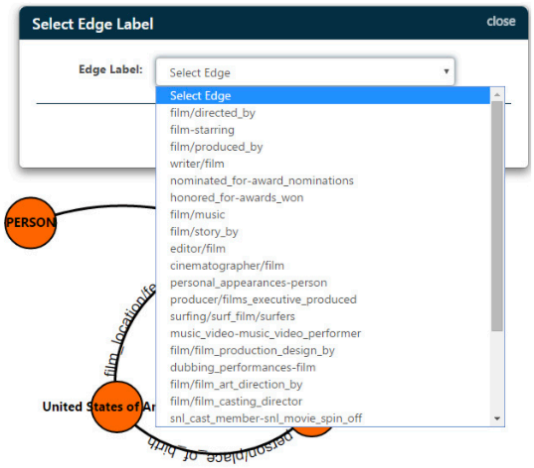


Figure 1-1 Adding an Edge in the Passive Mode

Orion's edge ranking algorithm is Random Decision Path (RDP), which ranks the candidate edges based on the query log of past query sessions. RDP ranks the candidate edges based on their correlation with the current query session. RDP uses different random subsets of the query session to construct multiple decision paths. This idea of RDP is inspired by the ensemble learning method of the random forests, which uses multiple random subsets of features to construct multiple decision trees. Chapter 3 of this thesis gives an overview of RDP.

We also implemented other edge ranking methods by adapting machine learning algorithms, namely Naïve Bayes (NB), Random Forests (RF), Recommendation System based on Singular Value Decomposition (SVD) and Classification based on Association Rules (CAR). Using NB or RF, we can find the probability that an edge e is a class associated with query

session Q, i.e., $P(e|Q)$. We can use these probability values of all the edges as a score to rank the candidate edges. Similarly, we can generate the score for each edge using association rules in CAR and cosine similarity in SVD to rank the candidate edges. Chapter 4 discusses in detail the implementation of these four machine learning classifiers.

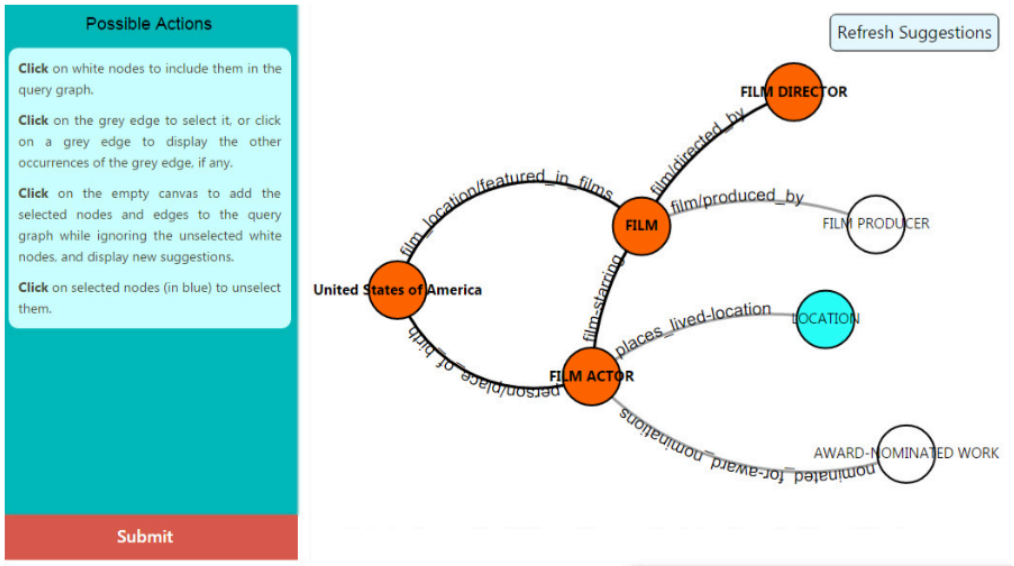


Figure 1-2 Automatic Edge Suggestions in the Active Mode

To the best of our knowledge, there exists no dataset that can be directly used to train our machine learning classifiers, so we simulated query logs for both Freebase [2] and DBpedia [3] data graphs using Wikipedia. The premise is that the various relationships between entities found in consecutive sentences of Wikipedia articles represent co-occurring properties that simulate the positive edges in a query session. Chapter 2 discusses the method used to simulate the query logs.

We conducted experiments to compare RDP with other edge ranking methods---RF, NB, CAR and SVD on both Freebase and DBpedia data graphs. On average, RDP requires 43 suggestions to complete a query graph, while other methods require 67 to 160 suggestions on Freebase data graph. Moreover, RDP requires 126.6 suggestions to complete a query graph on DBpedia data graph while other algorithms require 134 to 171 suggestions. Chapter 6 discusses the experimental setup and results in detail. We also found the accuracy of four

adapted machine learning algorithms by dividing the data into train and test set. Train set is used to build the model and the test set is used to find the accuracy of the models. Results of these experiments are discussed in Chapter 5.

Overall, the contribution of this research is as follows:

- We present a method to simulate the query log for both Freebase and DBpedia data graphs using Wikipedia articles.
- Four machine learning methods, namely NB, RF, SVD, and CAR, are adapted to suggest candidate edges that can be used to build a query graph.
- We compare Orion's RDP with the four machine learning algorithms we adapted.

Chapter 2

DATA PREPERATION

The query construction process of a user using a visual builder, where the user accepts or ignores the suggestions made by the system, can be summarized as a query session. A query session consists of positive and negative edges that correspond to edge suggestions accepted and ignored by the user, respectively. Numerous such query sessions, which captures user's query intent, can be called a query log. We need such a query log, which can be used to build our classifiers, so that given a user's partially built query session, our classifiers can find a class for the partially built query session (suggest an edge).

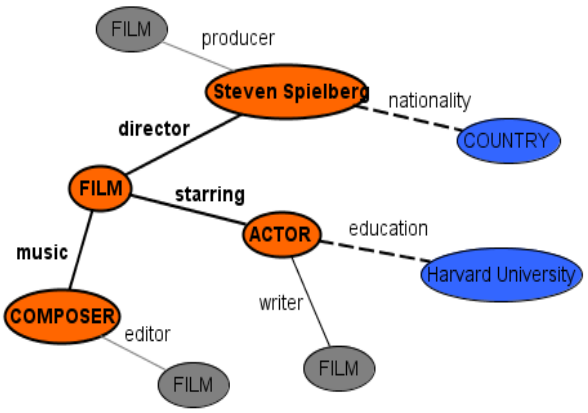


Figure 2-1 Sample Query Session

Figure 2-1 shows a query session, where the nodes in the orange color and the edges incident among them represents a partial query graph, nodes in the blue and grey color along with the edges incident on them represents the edge suggestions. Furthermore, the edges incident on the blue color represent the positive edges, they are the edges accepted by the user to be added to the current query session. Moreover, the edges incident on the grey nodes represent the negative edges, the edges the user has ignored. Figure 2-2 shows a query log, which has eight query sessions, representing the eight query intents of users. In the query session, $w1$ of

the query log, $education$ and $founder$ are the positive edges, and $\overline{nationality}$ is the negative edge.

Id	Query Session
w_1	education, founder, $\overline{\text{nationality}}$
w_2	starring, $\overline{\text{music}}$, director
w_3	nationality, $\overline{\text{education}}$, music, $\overline{\text{starring}}$
w_4	$\overline{\text{artist}}$, $\overline{\text{title}}$, writer, director
w_5	$\overline{\text{director}}$, founder, producer
w_6	writer, $\overline{\text{editor}}$, genre
w_7	$\overline{\text{award}}$, movie, director, $\overline{\text{genre}}$
w_8	education, founder, $\overline{\text{nationality}}$

Figure 2-2 Query Log

To the best of our knowledge, query log for large graphs are not publicly available which can be used to build our classifiers. In this chapter, we discuss the process of generating such query log.

2.1 Generating Positive Edges

We use Wikipedia articles to generate the positive edges of a query session. Each Wikipedia article describes an entity in detail and refers to other Wikipedia articles by wikilinks. Multiple entities mentioned in a window of consecutive sentences of a Wikipedia article can be considered to be related in some way. We find such pairwise relationships between all the entities mentioned in our window of consecutive sentences of an article. Our premise is that these consecutive co-occurring relationships simulate the positive edges of a user's query session. The intuition is that a user may also have such closely related facts as their query intent.

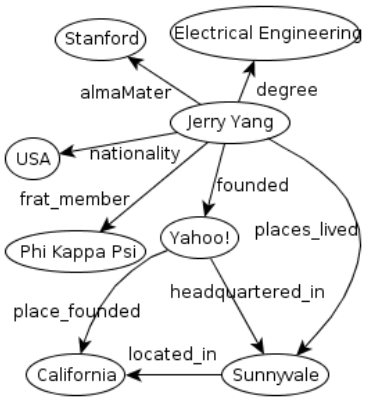
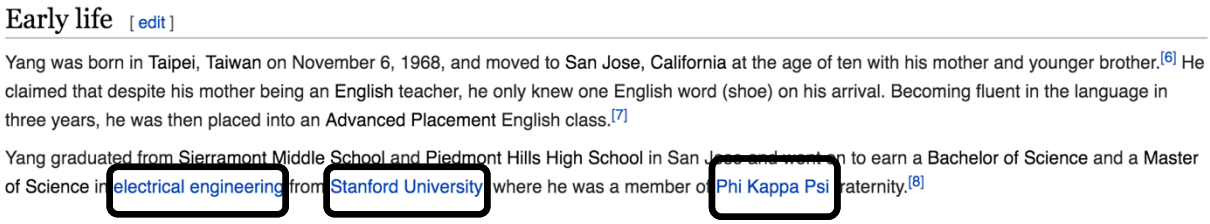


Figure 2-3 Sample Data Graph



Early life [edit]

Yang was born in Taipei, Taiwan on November 6, 1968, and moved to San Jose, California at the age of ten with his mother and younger brother.[6] He claimed that despite his mother being an English teacher, he only knew one English word (shoe) on his arrival. Becoming fluent in the language in three years, he was then placed into an Advanced Placement English class.[7]

Yang graduated from Sierramont Middle School and Piedmont Hills High School in San Jose and went on to earn a Bachelor of Science and a Master of Science in electrical engineering from Stanford University where he was a member of Phi Kappa Psi fraternity.[8]

Figure 2-4 Screen shot of a Wikipedia article

To find co-occurring positive edges, we map entities mentioned in Wikipedia articles to nodes of the data graphs, such as Freebase and DBpedia. We consider all the edges present among these nodes as the positive edges of the query session. This will result in a lot of noise because the actual sentence of the Wikipedia articles might actually mention only a subset of these relationships. We remove this noise by filtering the co-occurring edges which have low support in our query log.

Figure 2-3 show a sample data graph consisting of few nodes and edges related to Jerry Yang. Figure 2-4 shows a few sentences of the Jerry Yang's Wikipedia article. The wikilinks present in the article, electrical engineering, Stanford Univeristy and Phi Kappa Psi can be directly mapped to three nodes of the sample data graph. All the edges among these three nodes, $degree$, $almaMatter$, and $frat\_member$ co-occur and they simulate the positive edges of a query session.

2.1.1 Freebase

Freebase is a graph database system designed to be a public repository of the world's knowledge [3]. There could be edges representing the relationship between two nodes in both the directions, which are equivalent to each other, we remove edges in one such direction, by keeping the other. We also remove numeric entities, as we do not handle queries with relational operations such as greater than or lesser than. Nodes of freebase can easily be mapped to entities of the wikilinks of Wikipedia.

2.1.2 DBpedia

DBpedia is a structured database created using the information from Wikipedia to make this information available on the web [4]. Similarly, we preprocess the DBpedia data and we can map DBpedia data to wikilinks of Wikipedia.

2.2 Injecting Negative Edges

The above mentioned method only generates positive edges of a query session. It is crucial to simulate edges that were not accepted by the users, which can represent the negative edges of the query session. A negative edge of the query session is an edge that does not occur in the current query session, but co-occurs with one of the positive edges in a different query session. Thus, we add all the negative edges for a query session. These negative edges and the positive edges simulated using the above method must be used to rank the candidate edges, since the candidate edges are correlated with both accepted and ignored edges of a query session.

P1: $<almaMatter,\ degree,\ founded>$

P2: $<almaMatter,\ founded,\ frat\_member>$

Let us consider the above two positive query sessions, P1 and P2, generated using the method described in section 2.1. Then, we inject the negative edges as shown in the below equations, New P1 and New P2.

New P1: $<almaMatter,\ degree,\ founded,\ \overline{frat\_member}>$

New P2: <$almaMatter,\ founded,\ frat\_member, \overline{degree}$>

P1 and P2 has at least one positive edge in common, so the edge present in P1 and not present in P2, $\overline{degree}$ will be added as a negative edge to P2. Similarly, $\overline{frat\_member}$ is a negative edge for P1 and it will be added to P1 to make the updated P1, New P1.

2.3 Generating Query log using Freebase and DBpedia

We generate query log for both Freebase and DBpedia using the Wikipedia articles as discussed in the secion 2.1 and 2.2. Query log generated using Freebase and Wikipedia is called Freebase query log, and query log generated using DBpedia and Wikipedia is called DBpedia query log. We use the query log generated using this method is used to build our machine learning classifiers.

Chapter 3

OVERVIEW OF RDP

Orion's edge ranking algorithm, Random Decision Paths (RDP), is a novel method for measuring the relevance of a candidate edge with a query session. Formulation of RDP is motivated by the random forests [5]. However, there are important differences between RDP and random forests in standard definition and application, and in our experimentation RDP outperforms the standard random forest.

To understand the similarities and differences between RDP and random forests, let us briefly review decision trees and random forests. A decision tree D can be seen as a classifier that maps patterns to classes: $D(x) = argmax_y P(y|x)$, where x is the pattern and y is the class of that pattern. A leaf node L stores the precomputed probabilities $P_L(y)$ of each class y while building a decision tree. In the classification phase, a pattern x is used to traverse the built tree D and when a leaf node L is reached, the tree outputs $P_D(y|x) = P_L(y)$. A random forest F is built using multiple decision trees, where a best feature among a random subset of features is used to split each node. In the classification phase, a random forest F defines a probability $P_F(y|x)$, as the average $P_D(y|x)$ over all trees $D \in F$.

Random forests can be applied to our problem, but they have a few undesirable properties. Each pattern is a query session consisting of a few positive and negative edges. Since query session contains a relatively lesser number of edges than the total number of edge types, while traversing a tree using the query session, a vast majority of the test for the presence of an edge will result in a "no." This means that the query session does not contain the edge specified in the test at that node of the tree. This leads to a highly unbalanced trees, where the path corresponding to all "no" results gets the majority of the training examples. At classification time, most of the time an input pattern x ends up at all-no paths, and thus the class probabilities $P_D(y|x)$ do not vary much from the prior probabilities.

RDP algorithm is mathematically equivalent to constructing a random forest on the fly, given a query session QS to classify. This random forest is explicitly built to classify the given query session Q, and this random forest is discarded afterwards; a new random forest is built for every Q. The tests at each tree node considers only the edges that appear in Q. This way, the probabilities stored at leaf nodes are computed from the training examples that are similar to Q, as they share at least some edges with Q. This way of building the random forest will have probabilities at leaf node more accurate than random forest constructed offline, without the knowledge of Q. Our experimental results also validate this belief of RDP.

At the same time, since the query session Q is known, constructing full random forest is not necessary, and the RDP exploits this fact to save significant computational time. For any given query session Q, RDP builds only one path of the decision tree D, since computing the output for any other paths of D is useless. Therefore, out of every tree of the random forest, RDP only needs to compute and store a single path. Consequently, the random forest built in this method is reduced to a set of decision paths, and this set of decision paths is called "random decision paths" (RDP) [1].

Chapter 4

MACHINE LEARNING MODELS

We have modeled the problem of suggesting candidate edges based on user's current query session as a classification task. Furthermore, we are interested in suggesting top-k candidate edges to user based on the query session, instead of suggesting only one candidate edge. Hence, we have to use machine learning classifiers which can provide an estimate of membership of a query session over a set of classes than predicting only one class for the given query session. Using these estimates of membership over a set of classes as a score we can sort the candidate edges and suggest top-k ranked edges to users. This category of machine learning classifiers which has the capability of estimating the membership of data over a set of classes is known as probabilistic classifier. [6]

We use four such probabilistic classifiers, namely Naïve Bayes (NB), Random Forest (RF), Classification based on Association Rules (CAR), and a recommendation system based on Singular Value Decomposition (SVD) to compare its performance with that of Orion's RDP algorithm. This chapter explains the method followed to build these four classifiers.

4.1 Naïve Bayes (NB)

Naïve Bayes is a supervised classification algorithm which works on the principle of Bayes theorem. NB assumes the conditional independence among all of its features.

4.1.1 Training NB Classifier

Supervised classification algorithms require training data to have labels. The labels are the classes that our algorithms are trained to classify. They learn to classify such labels using the features of the training data. To train the NB classifier, we have to convert each query session into a form where we will have the training instances and their corresponding labels.

4.1.1.1 Training Data Format

Let us consider a query session which consists of 5 edges, where $starring$, $director$ and $music$ are positive edges and $\overline{education}$ and $\overline{nationality}$ are negative edges.

$$<starring, director, music, \overline{education}, \overline{nationality}>$$

We want to train our NB classifier on multiple such query sessions, and we want our classifier to suggest an edge given a set of positive and negative edges. We have to convert our query session into multiple data instances, where for each data instance we take out one positive edge from the query session and we mark that positive edge as the label for the current data instance. Conversion of our sample query session into multiple training data instances is shown below, where the edge on the right hand side is the label for the set of edges on the left hand side of out data instances.

$$<director, music, \overline{education}, \overline{nationality}> \rightarrow starring$$

$$<starring, music, \overline{education}, \overline{nationality}> \rightarrow director$$

$$<starring, director, \overline{education}, \overline{nationality}> \rightarrow music$$

Each query session in our query log is converted into a number of training data instances, where the number of training data instances generated from a query session is equal to the number of positive edges in that query session. In the above example the query session has three positive edges, so three training samples can be generated from this query session.

4.1.1.2 Implementation

We used scikit-learn library of python, which is an open source library consisting the implementation of various machine learning algorithms. We used the GaussianNB algorithm which assumes the likelihood of the features to be a Gaussian.

4.1.2 Using the Built Model

After a NB model has been built using the training data, to find the class of a query session, we directly give the query session as an input to our model without any modifications to the data as in the training phase. The NB can provide the probability of data belonging to all the

classes. We can sort these classes based on the probability of the data belonging to their respective classes and return these as these ranked edges.

4.2 Random Forest (RF)

Random Forests are the ensembles of the multiple decision trees, where each tree is built using the data generated by bootstrapping and each node uses a random subset of features to find the best split.

4.2.1 Training RF Classifier

4.2.1.1 Training Data Format

We convert each query session into multiple training data instances as explained in Training Data Format section of Naïve Bayes.

4.2.1.2 Implementation

We used the 'RandomForestClassifier' algorithm implemented in sickit-learn, which is an open source library developed in Python programming language.

4.2.1.3 Ensemble

Ensembles are the combination of multiple classifiers with an intuition that this combination of classifiers will perform better than using an individual classifier. We have used bagging of decision trees with bootstrapping, which is one of the most widely used. [7]

4.2.1.4 Bootstrapping

To build ensembles of the decision trees, we have used random samples with replacement of the training data to train each decision tree. Using bootstrapping each tree will be trained on different training samples, number of original training samples will be equal to the number of training samples generated using the random sampling with replacement technique.

Bootstrapping will decrease the variance of the model without increasing the bias of the model. Using the same data to build an ensemble of decision trees, without bootstrapping will lead to a high correlation between the trees and they get highly sensitive to noise in the training

data. Bootstrapping will lead to a better model than training all the classifiers with the same data and the averaging of many the trees with bootstrapping will not be sensitive to noise. [8]

4.2.1.5 Splitting Each Node

To split each node of the tree we use random subset of features of length square root of total number of features to find the best split. Square root of total number of features is a recommended number of features to be used for splitting each node for a classification task and that of one third of number of features is recommended for a regression task. [8]

We are using Gini impurity as a criterion to find the best split among the random subset of features. The feature which gives the lower Gini impurity value will be chosen to split the node.

4.2.1.6 Number of Trees

We built the RF using 40, 80, 120, 160 & 200 trees. The accuracy of the random forest on test data improves with number trees used increases, but the time taken to train large number of trees increases with the increase in the number of trees.

4.2.2 Using the Built Model

After the RF model has been built, to classify the class of a query session, we can directly give that query session as an input to our model without any modifications to the data like we expand in training phase. We use the 'predict_proba' function implemented in scikit-learn random forest implementation to get the score for the belongingness of the query session for each class. We can rank all classes based on this score and return ranked list as the predicted edges for the given query session.

4.3 Classification based on Association Rules (CAR)

Classification Rule mining is finding rules from the database to build an accurate classifier with a predetermined target of finding the classes. Association rules are generated using minimum support and confidence cutoffs without any predetermined target. To the best of

our knowledge, combining these two concepts to build a classification model was first proposed by Liu, Hsu, and Ma. [9]

Inspired by this concept of combining classification with association rules, we built a classifier using the association rules.

4.3.1 Training CAR Classifier

We have to generate the rules of the form <$antecedent, class\_y$> which is represented as, $antecedent \rightarrow class\_y$. The antecedent is a set of edges and class_y is the label. Given a query session we want to suggest an edge, so the class labels are nothing but the all possible positive edges in the query log. Hence we can use the same method of generating the training samples from a given query session as discussed in the 4.1.1.1 Training Data Format.

Let us consider the query session <$starring$, $director$, $music$, $\overline{education}$, $\overline{nationality}$> the rules generated from the above query session are shown below.

< director, music, $\overline{education}$, $\overline{nationality}$ > → starring

< starring, music, $\overline{education}$, $\overline{nationality}$ > → director

< starring, director, $\overline{education}$, $\overline{nationality}$ > → music

The left-hand-side of the rules are the antecedents and the right-hand-side of the rules are the class labels. Class labels are positive edges of the query session.

We generated a query log by using Apriori algorithm with a minimum support value (minquerysup). Each query session present in the query log has the support more than or equal to the minquerysup used to generate the query log. The antecedent of the rules generated should have a minimum support value (minrulesup), so that the rule can be considered as an association rule. We have used the same value of minquerysup as the required value of minrulesup, hence all the rules generated satisfies the minrulesup value and can be considered association rules. We do not use a minimum confidence value for considering a rule to be an association rule, instead we use the confidence of the rule to predict the class. Hence, we consider all the rules generated by the above method to be the association rules.

We find the confidence values of all the association rules in the training phase of the algorithm. We use these confidence values in the classification phase of CAR.

4.3.2 Using the Built Model

In the training phase of CAR, we find all the association rules from the query log and the confidence of each rule is also found. In the classification phase of CAR, we first find all the matching rules of the given query session then we use the scoring function to rank the candidate edges, and we return a ranked list of candidate edges.

4.3.2.1 Finding Matching Rules

A matching rule is an association rule which has at least one edge of the query session in the rule's antecedent. Consider an association rule in the format $antecedent \rightarrow class\_Y$ and a query session (QS), then the length of the intersection of antecedent set & QS set should be at least one to consider the rule $antecedent \rightarrow class\_Y$ as a matching rule. Using this notion of matching rules, we find all the matching rules using the association rules & query session.

4.3.2.2 Scoring Candidate Edges

Algorithm 1 Calculating score for a Candidate Edge in CAR

Input: Candidate Edge, Matching Rules, confidence of all rules and Query Session ***Output***: Score of the given Candidate Edge

1: $candidate\ edge\ score = 0$
2: **for** $rule\ in\ Matching\ Rules$ **do**
3: $score\ of\ this\ rule = |antecedent\ intersection\ Query\ Session| * (confidence\ of\ this\ rule) \div |Query\ Session|$
4: $candidate\ edge\ score += score\ of\ this\ rule$
5: **end for**
6: **return** $candidate\ edge\ score$

Once we find all the matching rules for a given query session, we assign scores to all the candidate edges using the matching rules, query session, and the confidence of these matching rules as shown in Algorithm 1. We sort the candidate edges based on their assigned scores in the decreasing order and we return the sorted list of candidate edges as the output of

the CAR algorithm. In other words, we rank our candidate edges using the scores of the candidate edges calculated using the algorithm shown in Algorithm 1.

4.4 Recommendation System based on Singular Value Decomposition (SVD)

4.4.1 Training SVD Classifier

We use singular value decomposition to find the reduced the low-dimensional representation of the query log. Then we use this reduced space representation of the query log & the edges to find the neighborhood of a given query session, and we sort the candidates based on a score generated using the neighbors of the query session.

4.4.1.1 Training Data Format

We form a matrix from the query log, where the rows represent each query session of the query log and the columns represents the unique edges, both positive edges and negative edges of the query log. Let m×n be the dimensions of the matrix, m is the number of query sessions present in our query log and n is the number of unique edges present in the query log. For a query session of the query log, we mark all the columns corresponding to the all the edges present in this query session as one.

Let us consider a sample query session present in our query log to be, <$starring, director, music, \overline{education}, \overline{nationality}$>. This query session contains five edges, so the row corresponding to this query session will contain one in only five columns corresponding to edges $starring, director, music, \overline{education}$ and $\overline{nationality}$. We use the sparse matrix format representation, so we add one to the columns corresponding to the edges present in the row, values of the rest of the columns will be considered to be containing zeroes.

4.4.1.2 Singular Value Decomposition

Singular value decomposition is a matrix factorization technique that factors a m×n matrix A into three matrices as follows

$$A = U \times S \times V'$$

where U and V are two orthogonal matrices of dimensions m×r and n×r, respectively. r is the rank of the matrix A. S is a diagonal matrix of size r×r, values of S are stored by the decreasing order of their magnitude. We can use U, S and V matrices to get the dimensionality reduction of the matrix A, and to generate a lower rank approximation of matrix A.

Lower rank approximation is found by considering top-k values of matrix S and slicing the matrix S to get a matrix S_k of dimensions k×k. Similarly, matrices U and V can be sliced to get matrix $U_k$of dimensions m×k and matrix V_k of dimensions n×k respectively.[10]

4.4.1.3 Implementation

We use the 'svd' method implemented in python numpy library to find the singular value decomposition matrices of our query log.

Algorithm 2 Calculating score for a Candidate Edge in SVD

Input: Candidate Edge, top 100 similar Query Sessions, Query Session
Output: Score of the given Candidate Edge

```
candidate edge score = 0
for rule in top 100 Similar Query Sessions do
    if candidate edge in row then
        score of this rule = CosineSimilarity(Query Session, row)
        candidate edge score+ = score of this rule
    end if
end for
return candidate edge score
```

We find the matrices U, S and V and reduce them the dimensions and we use the value of k as 100.

Algorithm 2 Calculating score for a Candidate Edge in SVD

Input: Candidate Edge, top 100 similar Query Sessions, Query Session
Output: Score of the given Candidate Edge

```
candidate edge score = 0
for rule in top 100 Similar Query Sessions do
    if candidate edge in row then
        score of this rule = CosineSimilarity(Query Session, row)
        candidate edge score+ = score of this rule
    end if
end for
return candidate edge score
```

4.4.2 Using the Built Model

Given a query session we find the top 100 similar rows of the query log using the U_k, $S_k$and V_k matrices. Cosine similarity is the similarity method that we have used to find the top 100 query sessions from the query log.

4.4.2.1 Scoring Candidate Edges

Once we find the top 100 query sessions of the query log for a given query session, we assign scores to all the candidate edges using the Algorithm 2. Algorithm 2 uses the cosine similarity between the given query session and each query session of the top 100 query sessions to find the score of the candidate edges. We sort the candidate edges based on their scores in the decreasing order and we return this sorted list of candidate edges as the output of the SVD algorithm. In other words, we rank our candidate edges using the scores of the candidate edges calculated using the Algorithm 2.

Chapter 5

TESTING MODEL ACCURACY

In the previous chapter we discussed the approach of building four models, namely NB, RF, CAR and SVD. In this chapter we will discuss the approach we took to test our training method to build four machine learning classifiers.

5.1 Splitting Data into Train and Test Data

We divided our query log into two parts, namely training query log and test query log. We selected 80% of the query log randomly without repetitions as training query log and the remaining 20% was considered test query log. We built our classifiers using the train query log and we tested them on the test query log.

5.1.1 Training the classifiers

For each of the four classifiers, we used the same method of using the data as explained in the previous chapter respectively. We built four classifiers on both Freebase train query log and DBpedia train query log, thus generating eight different models.

5.1.2 Testing the classifiers

For testing the four classifiers trained using the train query log, we converted the test query session of test query log into multiple test query instances. The number of test query instances that can be created from a test query session is equal to the number of positive edges present in that query session. This process of converting the query session is very similar to the method we used to train our NB, RF and CAR. Let us consider a sample test query session of the test query log to be, $<starring, director, music, \overline{education}, \overline{nationality}>$.

We can generate three test query instances from the above test query session as shown below.

$$<director, music, \overline{education}, \overline{nationality}> \rightarrow starring$$

$$<starring, music, \overline{education}, \overline{nationality}> \rightarrow director$$

$$<starring, director, \overline{education}, \overline{nationality}> \rightarrow music$$

Even though the SVD classifier was trained without generating the training data as shown in the above method, while testing the SVD classifier we use this method of testing. We follow this method because after removing one positive edge from the query session and it as an input to the classifier, we expect the classifier to suggest us the positive edge that we removed from the query session. Since, this expectation holds true and is same for all the four classifiers that we have built, we use the same method to test all the classifiers.

5.2 Accuracy on Test Query Log

We tested our four classifiers on both Freebase Test Query log and DBpedia Test Query log using the above method of giving input to the classifiers.

The accuracy of a classifier is found using the formula,

$$Accuracy = \frac{Number\ of\ correctly\ classified\ test\ instances}{Total\ number\ of\ test\ instances}$$

Accuracy of CAR and SVD was not very high as NB and RF, but the edge that we were expecting the model to predict was usually one among the top-5 or top-10 ranked edges returned by the CAR and SVD. So to capture this fact that the CAR and SVD were able to rank our expected edge in top-5 or top-10, we are finding the top-5 accuracy and top-10 accuracy as defined by the following two formulae,

$$Top5\ Accuracy = \frac{Number\ of\ cases\ where\ expected\ edge\ was\ in\ top5\ ranked\ candidates}{Total\ number\ of\ test\ instances}$$

$$Top10\ Accuracy = \frac{Number\ of\ cases\ where\ expected\ edge\ was\ in\ top10\ ranked\ candidates}{Total\ number\ of\ test\ instances}$$

5.2.1 Accuracy of Classifiers on Freebase Test Query Log

Algorithm	Test Accuracy
NB	97.90 %
RF 40	98.00 %
RF 80	98.29 %
RF 120	98.33 %
RF 160	98.40 %
RF 200	98.41 %

Table 5-1 Accuracy of NB and RF on Freebase Test Query Log

The accuracy of a classifier is found using the NB is 97.9%, which means in 97.9% of the total number of test instances, the NB has predicted the missing positive edge of the test query session. RF was trained with 40, 80, 120, 160 and 200 trees and in Table 5-1, list them as RF 40, RF 80, RF 120, RF 160 and RF 200 respectively.

As shown in Table 5-2, the accuracy of CAR and SVD were relatively lower than NB and RF, when we consider the Top5 Accuracy and Top10 Accuracy the performance of SVD and CAR are comparable with NB and RF classifiers.

We also generated CAR model with a variation of using only positive edges in the antecedent and ignoring all the negative edges, called CAR-POS.

CAR-POS model built using only positive edges has performed better than CAR model built using both positive edges and negative edges on Freebase.

Algorithm	Test Accuracy
CAR Accuracy	11.24 %
CAR Top5 Accuracy	48.22 %
CAR Top10 Accuracy	63.69 %
SVD Accuracy	15.92 %
SVD Top5 Accuracy	69.18 %
SVD Top10 Accuracy	97.57 %

Table 5-2 Accuracy of CAR and SVD on Freebase Test Query Log

Algorithm	Test Accuracy
CAR-POS Accuracy	19.02 %
CAR-POS Top5 Accuracy	63.27 %
CAR-POS Top10 Accuracy	81.44 %

Table 5-3 Accuracy of CAR-POS and SVD-POS on Freebase Test Query Log

5.2.2 Accuracy of Classifiers on DBpedia Test Query Log

The accuracy of a classifier is found using the NB is 98.66%, which means in 98.66% of the total number of test instances, the NB has predicted the missing positive edge of the test query session. RF test accuracy has consistently improved with an increase in number of trees in both Freebase test query log and DBpedia test query log.

Algorithm	Test Accuracy
NB	98.66 %
RF 40	95.50 %
RF 80	96.19 %
RF 120	96.37 %
RF 160	96.53 %
RF 200	96.54 %

Table 5-4 Accuracy of NB and RF on DBpedia Test Query Log

Algorithm	Test Accuracy
CAR Accuracy	18.80 %
CAR Top5 Accuracy	43.79 %
CAR Top10 Accuracy	64.26 %
SVD Accuracy	16.72 %
SVD Top5 Accuracy	58.21 %
SVD Top10 Accuracy	96.4 %

Table 5-5 Accuracy of CAR and SVD on DBpedia Test Query Log

Similar to Table 5-2, in Table 5-5, the accuracy of CAR and SVD were relatively lower than NB and RF, when we consider the Top5 Accuracy and Top10 Accuracy the performance of SVD and CAR are comparable with NB and RF classifiers.

Algorithm	Test Accuracy
CAR-POS Accuracy	15.75 %
CAR-POS Top5 Accuracy	55.63 %
CAR-POS Top10 Accuracy	78.38 %

Table 5-6 Accuracy of CAR-POS on DBpedia Test Query Log

CAR Accuracy is better than CAR-POS accuracy value on DBpedia. The CAR-POS Top5 Accuracy and CAR-POS Top10 Accuracy is better than that of CAR model on DBpedia.

Chapter 6

EXPERIMENTAL SETUP AND RESULTS

In the previous chapter we discussed the approach of testing our four classifiers, namely NB, RF, CAR and SVD. In this chapter we will discuss the approach we took to test our classifiers on manually generated queries.

6.1 Building the model on the query log

As discussed in the chapter 4, we build four classifiers on the full query log for both Freebase and DBpedia.

6.2 Testing the Models on Target Query Graphs

Once we build our models using the Freebase and DBpedia query log, we test the model to generate the target queries designed for both Freebase and DBpedia.

6.2.1 Target Query Graphs

Target query graphs are a group of edges that forms a query consisting of only the positive edges present in the query log. We generate such real world target queries manually that could be of interest to our users, and compare our classification algorithms to check how many suggestions are required by our machine learning algorithm to complete the target query graph starting from a single edge as an input.

We believe that comparing our classifiers using this method will give us an idea about the performance of our classifiers on real world queries. Each time our model returns the ranked edges we select only the top ranked edge as the suggested edge of our machine learning classifiers.

6.2.1.1 Freebase Target Query Graphs

We generated 43 target query graphs for freebase and these target query graphs are manually generated. We believe that they fairly represent the real world scenario of user generated queries.

Number of Edges in Target Query Graphs	Number of Graphs created
Two edged query graphs	6
Three edged query graphs	10
Four edged query graphs	9
Five edged query graphs	17
Six edged query graphs	1

Table 6-1 Number of target queries generated for Freebase

Table 6-1 shows the number of target queries and the number of edges present in each target queries. If $director$ and $producer$ are the two edges present in a two edged target query, we can start to build the query from the $director$ edge and we can also start to build the query using the $producer$ edge. Thus, if there are two unique edges, we will have two starting points to build that particular query graph. Similarly, from 43 target query graphs, we can form 167 input instances from which the query graph construction can be started.

6.2.1.2 DBpedia Target Query Graphs

Similar to the Freebase, we generated 33 target query graphs for DBpedia data. Table 6-2 shows the number of target queries and the number of edges present in each target queries. From 33 target query graphs, we can form 130 input instances from which the query graph construction can be started.

Number of Edges in Target Query Graphs	Number of Graphs created
Three edged query graphs	2
Four edged query graphs	29
Five edged query graphs	2

Table 6-2 Number of target queries generated for DBpedia

6.2.2 Testing on Target Query Graphs

Given a target query graph, we give the classifier only one edge as the input query session, all possible candidate edges and the classifier returns a ranked list of candidates edges, we then select the topmost ranked candidate edge as the edge suggested by the classifier. If the topmost ranked candidate edge is present in our target query graph, we add this edge as a positive edge to our query session. If the topmost ranked candidate edge is not present in our target query graph, we then add this edge as a negative edge to our query session. Then, we remove the topmost ranked candidate edge from the list of candidate edges and we give this as an input along with updated query session to the classifier. This process is repeated until we find all the edges present in our target query graph to be part of query session.

Using the above approach, we find the number of suggestions required for the classifier to suggest the target query graph, starting with one edge of the target query graph in the query session to having all the edges in the target query graphs in our query session.

Let us consider an example target graph to be,

Target graph: <director, producer, award>

We start with one edge in our query session and we find all possible candidate edges and we give this as an input to our classifiers. Let us consider the edge 'director' to be the input instance as shown below.

Query Session – [director]

Candidate Edges – [writer, producer, award …]

The classifier would return a ranked candidate edge, let us consider the ranked candidate edges to be as shown below.

Ranked Candidate Edges – [award, ...]

The 'award' is the top ranked edge, we check if this edge is present in our target query graph, and it happens to be present in our target query graph, so we update our query session and we remove the 'award' edge from the candidate edges and we give this as input to our classifier to rank the candidate edges.

Query Session – [director, award]

Candidate Edges – [writer, producer, music_composer …]

Ranked Candidate Edges – [music_composer, …]

'music_composer' edge is not present in our target query session, thus this node is rejected and we add this as a negative edge to our query session, we remove this edge from the candidate edges, and we give updated query session and candidate edges as input to our classifier.

Query Session – [director, award, $\overline{music\_composer}$]

Candidate Edges – [writer, producer, …]

This process is repeated until all the edges present in our target query becomes part of the query session and we record number of suggestions required by our classifier to complete this process, we call this process as completing the target graph. We have a limit of 200 suggestions, if the classifier is taking more than 200 suggestions, we stop the process of generating the target query graph.

6.2.2.1 Experimental Setup for Finding Number of Suggestions

We developed all the modules which ranks the candidate edges using the classifiers in python programming language and the target query graphs generation method we developed in java programming language. We used a Python-Flask web framework to connect these two programs and all the ranking functions were exposed as a web API. The Java program makes a POST request to the ranking function with candidate edges and query session through the Flask web service and the ranking function ranks the candidate edges and returns the ranked candidate edges. Figure 6-1, shows the depicts the process of these web API calls.

Figure 6-1 Web Service Setup

6.2.3 Experimental Results

We ran the above explained experimental setup using all the four classifiers that we have built on both Freebase and DBpedia target queries. An important measure of the efficiency of a query graph completion system is the number of suggestions required to successfully complete a target graph using the initial query session. A system that takes lesser number of suggestions can help the user to build a query graph more accurately with lesser number of suggestions.

6.2.3.1 Results on Freebase Target Queries

Figure 6-2 shows the average number of suggestions required to complete each of 167 target query graph instances for Freebase. The algorithm used in Orion, RDP performs better than all other algorithms. RDP requires 43.5 average number of suggestions to complete the

target query graph, while CAR requires 67.8 average number of suggestions and the CAR-POS algorithm requires 52.5 average number of suggestions.

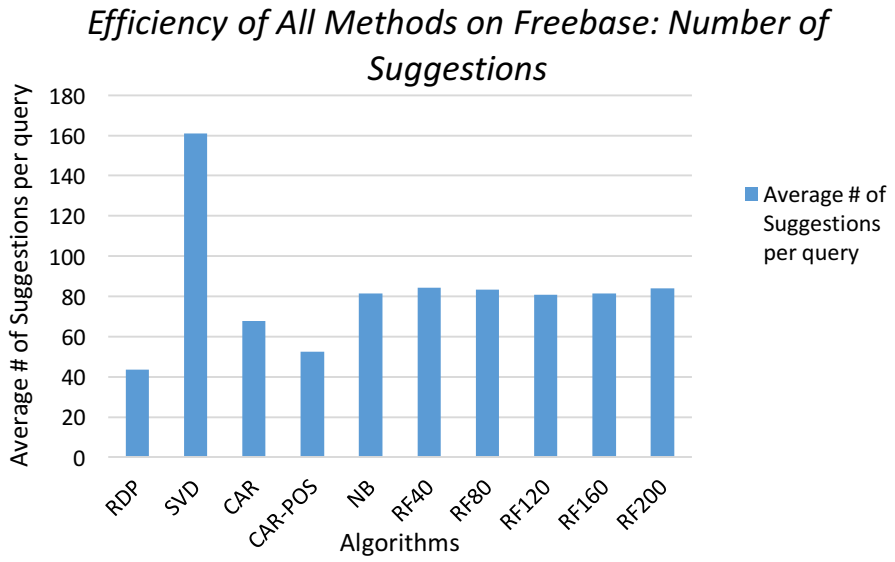


Figure 6-2 Efficiency of All Methods on Freebase: Number of Suggestions

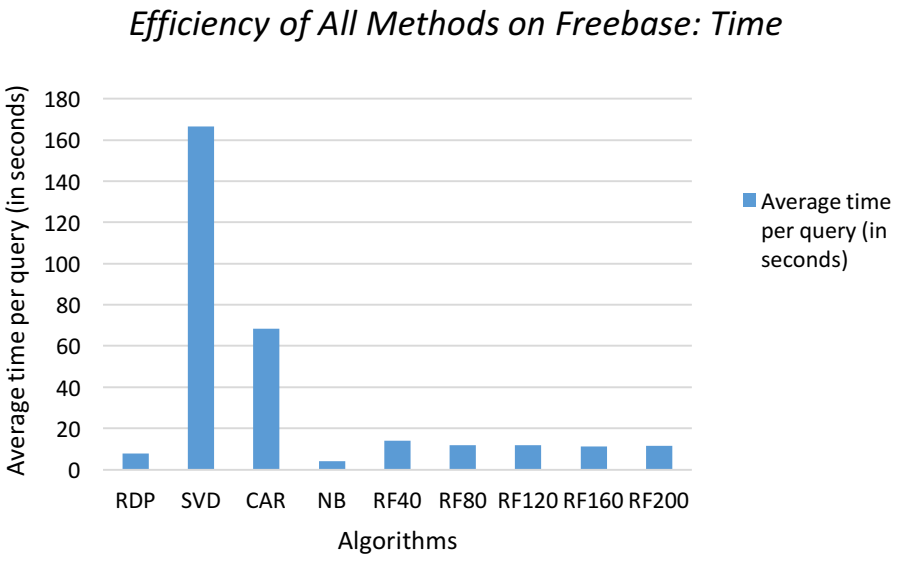


Figure 6-3 Efficiency of All Methods on Freebase: Time

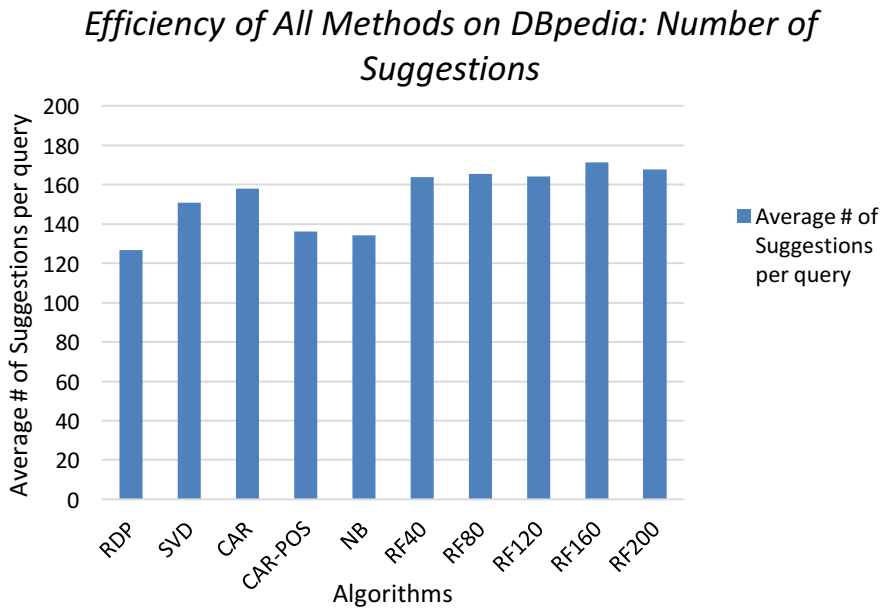


Figure 6-4 Efficiency of All Methods on DBpedia: Number of Suggestions

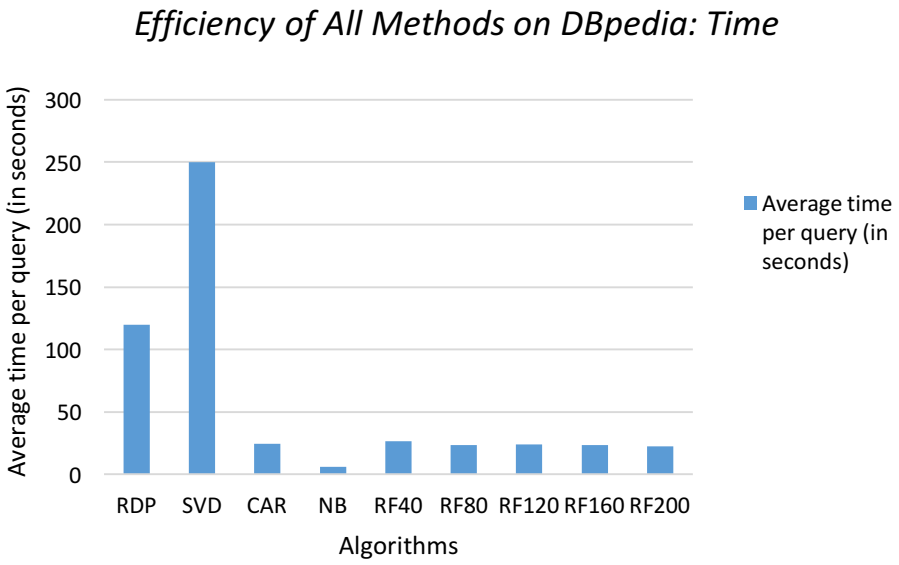


Figure 6-5 Efficiency of All Methods on DBpedia: Time

Figure 6-3 shows the average number of time required to complete each of 167 target query graph instances on Freebase. RDP, NB and RF significantly outperforms SVD and CAR.

RDP requires 7.7 seconds, NB requires 3.9 seconds and different RF implementations require time in the range of 11.2 – 14.08 seconds.

6.2.3.2 Results on DBpedia Target Queries

Figure 6-4 shows the average number of suggestions required to complete each of 130 target query graph instances for DBpedia. RDP performs better than all other algorithms. RDP requires 126.6 average number of suggestions to complete the target query graph, while NB requires 134.3 average number of suggestions and the CAR-POS algorithm requires 136.1 average number of suggestions.

Figure 6-5 shows the average number of time required to complete each of 130 target query graph instances for DBpedia. RDP, NB and RF outperforms SVD and CAR. RDP requires 119.7 seconds, NB requires 5.9 seconds and different RF implementations require time in the range of 22.2 – 26.7 seconds.

Chapter 7

CONCLUSION AND FUTURE WORK

In this work, we have compared RDP with NB, RF, CAR and SVD. We found that to complete a target query graph, Orion's RDP method requires an average of 43.5 suggestions per query graph on Freebase, while the closest performing method to RDP on Freebase is CAR-POS, which requires an average of 52.5 suggestions to complete a query graph. On DBpedia, RDP requires an average of 126.6 number of suggestions and the closest performing method to RDP is NB, which requires 134.3 average number of suggestions to complete a query graph.

We have presented a method for generating query sessions using Wikipedia articles and knowledge graphs. We also presented a method to test our classifiers using the concept of completing a target query graphs.

We plan to carry on our future work in the following directions,

- User study should be performed by integrating the four algorithms adapted to rank the candidate edges with Orion's user interface. This should be compared with RDP to find the best algorithm according to the user's feedback.
- Query log should also be generated using other websites than using only the Wikipedia articles.
- The actual query sessions of the users of Orion should be logged and should be added to the query log used to build the classifiers.

References

[1] Jayaram, N. (2016). Towards Better Usability of Query Systems for Massive Ultra-Heterogeneous Graphs: Novel Approaches of Query Formulation and Query Specification (Doctoral dissertation, University of Texas at Arlington, 2016). Arlington, TX

[2] Bollacker, K., Evans, C., Paritosh, P., Sturge, T., & Taylor, J. (2008). Freebase: A collaboratively created graph database for structuring human knowledge. ACM, 1247-1250. doi:10.1145/1376616.1376746

[3] Bizer, C., Lehmann, J., Kobilarov, G., Auer, S., Becker, C., Cyganiak, R., & Hellmann, S. (2009). DBpedia - A crystallization point for the Web of Data. Web Semantics: Science, Services and Agents on the World Wide Web, 7(3), 154-165. doi:10.1016/j.websem.2009.07.002

[4] L.Breiman.Randomforests.*MachineLearning*,45(1):5–32

[5] Kim, K. I., & Simon, R. (2010). Probabilistic classifiers with high-dimensional data. Biostatistics, 12(3), 399-412. doi:10.1093/biostatistics/kxq069

[6] DIETTERICH, THOMAS G. "An Experimental Comparison of Three Methods for Constructing Ensembles of Decision Trees: Bagging, Boosting, and Randomization." An Experimental Comparison of Three Methods for Constructing Ensembles of Decision Trees: Bagging, Boosting, and Randomization (1999): 1-22. Kluwer Academic Publishers. Web.

[7] Hastie, T., Tibshirani, R., & Friedman, J. H. (2001). The elements of statistical learning: Data mining, inference, and prediction: With 200 full-color illustrations. New York: Springer.

[8] Liu, B., Hsu, W., & Ma, Y. (1998). Integrating Classification and Association Rule Mining. KDD-98 Proceedings. Retrieved from https://www.aaai.org/Papers/KDD/1998/KDD98-012.pdf

[9] Sarwar, B., Karypis, G., Konstan, J., & Riedl, J. (2000). Application of Dimensionality Reduction in Recommender System - A Case Study (Master's thesis, University of Minnesota, 2000). Minneapolis.

Biographical Information

Rohit Ravi Kumar Bhoopalam received his Bachelors in computer science and engineering from Visvesvaraya Technological University, India, in 2011. After completing his Bachelors degree, he worked in a couple of start-ups in Bangalore for two and a half years. He completed his Masters degree in computer science and engineering in Spring 2016. His areas of interests include machine learning, data mining and web development.