University of New Hampshire

CS953

Prototype 3

Author:
Amith Ramanagar Chandrashekar

Author:
Pooja Himanshu Oza

Author:
Rachel E Cates

Author:
Ahmed M Alnazer

Supervisor:
Dr. Laura Dietz

April 25, 2019

Contents

1 Problem Definition

The demand for efficient and accurate methods of knowledge discovery is ever-increasing. With the explosion of data in last decade and the evolution in information needs, meeting this demand is becoming a greater challenge. Our project attempts to tackle this issue. The proposed system retrieves relevant information from a collection, combines this retrieved information, and presents it to the user.

2 Approaches

2.1 Spam Filter

In an attempt to improve precision and recall of document retrieval, a spam filter was applied before reranking them. This was done with the following commands/classes:

2.1.1 indexHamSpam

This is done once with the following command:

indexHamSpam -p dedup.articles-paragraphs.cbor -q manual.benchmarkY1test.cbor.hierarchical.qrels -hamTrain /path/to/hamTrainOutput -spamTrain /path/to/spamTrainOutput -hamSpamTest path/to/hamSpamTestOutput -hamTest path/to/hamTestOutput -spamTest path/to/spamTestOutput

Two training sets are created, one containing spam documents, and one containing ham documents. Three test sets are also created, one containing ham, one containing spam, and one containing both ham and spam. The separate ham and spam test sets are only needed if the user wants to get the raw F1 and MAP scores of their classifier. If they simply wish to apply the filter, they need not go through this step at all, but simply need their test set and the separate ham and spam train sets that I provide by running this command myself. Then, they can instantiate any of the classes that follow.

2.1.2 SpamClassifier.java

This class returns a HashMap of labels, classifying the documents passed as a parameter as either "ham" or "spam". The following API for the SpamClassifier is available to the user:

- readIndex(String path): Takes the path to the train or test data created by indexing the manual qrels in the previous step and returns a HashMap of paragraph ids mapped to their text. If the user wants raw F1 and MAP scores, then at least three separate calls to this method should be performed before any of the other methods in this class are called, as the

HashMaps created will be needed as parameters. Otherwise, they only need to call this method once to put their test data into a form that the classifiers will be able to understand.

- classifyWithUnigrams(HashMap<String, String> hamTrain, HashMap<String, String> spamTrain): Takes the training HashMaps and returns a trained NaiveBayesUnigramPredictor.
- classifyWithBigrams(HashMap<String, String> hamTrain, HashMap<String, String> spamTrain): Takes the training HashMaps and returns a trained NaiveBayesBigramPredictor.
- classifyWithTrigrams(HashMap<String, String> hamTrain, HashMap<String, String> spamTrain): Takes the training HashMaps and returns a trained NaiveBayesTrigramPredictor.
- classifyWithQuadgrams(HashMap<String, String> hamTrain, HashMap<String, String> spamTrain): Takes the training HashMaps and returns a trained NaiveBayesQuadgramPredictor.
- predict(LabelPredictor lp, HashMap<String, String> test): Takes the LabelPredictor trained in the previous step and the HashMap created from the test set and returns a HashMap of paragraph ids to labels of "ham" or "spam".
- getScores(LabelPredictor predictor, HashMap<String, String> test): Computes a ham and a spam score for each item in the test set. Return type is a HashMap where the key is a paragraph id, and the value is an ArrayList containing two doubles, the first corresponding to the ham score of that document, and the second corresponding to the spam score of that document.
- evaluate(LabelPredictor predictor, HashMap<String, String> hamTest, HashMap<String, String> spamTest, HashMap<String, String> testDocs): Takes all test HashMaps and the trained classifier and computes its F1 and MAP scores.

Example usage:

```
SpamClassifier sc = new SpamClassifier();
HashMap<String, String> hamTrain = sc.readIndex(hamTrainPath);
HashMap<String, String> spamTrain = sc.readIndex(spamTrainPath);
HashMap<String, String> test = sc.readIndex(docsForRerank);
LabelPredictor lp = sc.classifyWithUnigrams(hamTrain, spamTrain);
HashMap<String, String> labels = sc.predict(lp, test);
```

Any documents marked "spam" can then be ignored when reranking.

2.1.3 PythonSVM.java

Under the hood, this class uses Python's scikit-learn support vector machine. Because it has a Java wrapper however, it can easily be called by the other classes in the project. Moreover, the same training sets created from indexHamSpam can be used with it. The following methods are available for the user:

- readIndex(String path): Takes the path to the train or test data created by indexing the manual qrels. The return type is ArrayList<HashMap<Integer, Double>>, where each item in the list corresponds to a document, and the HashMap contains a mapping between a token's unique id to the frequency of that token. At least three separate calls to this method should be performed before any of the other methods in this class are called, as the ArrayLists created will be needed as parameters.
- prepareData(ArrayList<HashMap<Integer, Double>> hamData, ArrayList<HashMap<Integer, Double>> spamData, ArrayList<HashMap<Integer, Double>> test, String trainPath, String testPath): Takes the ArrayLists created in the previous method and uses them to create train and test csv files for easy manipulation by scikit-learn. Writes the csvs to the paths passed as the final two parameters.
- ArrayList<String> execLinearSVC(): Calls svm.py, which then trains a model based on the csvs created in the previous step. Prediction is handled at the same time, and an ArrayList of Strings is returned for each document in the test set, where a value of "-1" indicates that the document was classified as spam, and "1" indicates that the document was classified as ham.
- ArrayList<String> execPolynomialSVC(): Calls polynomial_svm.py, which then trains a model based on the csvs created in the previous step. Prediction is handled at the same time, and an ArrayList of Strings is returned for each document in the test set, where a value of "-1" indicates that the document was classified as spam, and "1" indicates that the document was classified as ham.
- ArrayList<String> execSigmoidSVC(): Calls sigmoid_svm.py, which then trains a model based on the csvs created in the previous step. Prediction is handled at the same time, and an ArrayList of Strings is returned for each document in the test set, where a value of "-1" indicates that the document was classified as spam, and "1" indicates that the document was classified as ham.
- ArrayList<String> execRbfSVC(): Calls rbf_svm.py, which then trains a model based on the csvs created in the previous step. Prediction is handled at the same time, and an ArrayList of Strings is returned for each document in the test set, where a value of "-1" indicates that the document was classified as spam, and "1" indicates that the document was classified as ham.

Example usage:

```
HashMap<String, HashMap<String, Integer>> freqMap = new HashMap<>();
ArrayList<String¿ words = new ArrayList<>();
PythonSVM pythonSVM = new PythonSVM(freqMap, words);
ArrayList<HashMap<Integer, Double>> spamTrain = pythonSVM.readIndex("spamIndex");
ArrayList<HashMap<Integer, Double>> hamTrain = pythonSVM.readIndex("hamIndex");
ArrayList<HashMap<Integer, Double>> test = pythonSVM.readIndex("testIndex");
String trainPath = "/traindata.csv";
String testPath = "/testdata.csv";
pythonSVM.prepareData(spamTrain, hamTrain, test, trainPath, testPath);
ArrayList <String> labels = pythonSVM.execLinearSVC();
```

2.1.4 Obtaining more spam training data

After discovering that the manual qrels contained very little spam, we found it necessary to add more data to the spam train set in order to better build our models. This was done by parsing the paragraph corpus, writing the text of each document to a file, and manually searching the file for passages that didn't contain any useful information. We were able to get 35 more spam data points through this method, which improved the performance of several classifiers. Performance improvements were especially observed in the NaiveBayesUnigramPredictor, NaiveBayesBigramPredictor, and SpecialCharPredictor classes.

2.1.5 Integration with the rest of the system

The spam filter was applied to documents before anything else in order to identify and eliminate irrelevant information and improve the final evaluation.

2.1.6 Evaluation

A evaluate method is provided to the user for all classifiers in the predictors package. This method returns both F1 and MAP scores. Evaluation of the PythonSVM classifiers is done by calling scikit-learn's built-in evaluation metrics, such as F1, precision, recall, and accuracy.

2.2 Query Expansion

The main motivation behind query expansion is to reformulate the user's query to improve retrieval performance. This is achieved by finding terms related to those of the original query, then constructing a new query by adding the semantically closest words.

All of our variations of query expansion use pseudo-relevance feedback. Thus the assumption is made that the top K documents retrieved using the baseline method (BM25) are relevant to the initial query. The main advantage of using pseudo-relevance feedback is its automation of the manual part of true relevance

feedback. 300-dimension GloVe word embedding is used to compute the K nearest neighbor words per query term.

2.2.1 Candidate set generation - method 1

The main motivation of candidate set generation is to reduce the computation domain to as few terms as possible. This first method runs the initial query using BM25 as a baseline and retrieves K documents. Next, it applies pseudo-relevance feedback, picks the top K documents as relevant, and process them to find the terms using the following steps:

- Concatenate the top K documents to a single string
- Process this string using the Standard Analyzer
- Remove stop words and any candidate terms that are only numerical in value

This produces a candidate term set

$$C = \{t_1, t_2, t_3, \ldots, t_n\}$$

2.2.2 Candidate set generation - method 2

This method follows the same approach as the former, but it uses the abstract of each entity in the document to generate the candidate set. The process is as follows:

- Concatenate the top K documents and abstract all of the entities as one string
- Process this string using the Standard Analyzer
- Remove stop words and any candidate terms that are only numerical in value

This produces a candidate term set

$$C = \{t_1, t_2, t_3, \ldots, t_n\}$$

2.2.3 Input data

Given the query

$$Q = \{q_1, q_2, \ldots, q_n\}$$

and the candidate set

$$C = \{t_1, t_2, \ldots, t_n\}$$

compute the K nearest terms

$$N = \{k_1, k_2, k_3, \ldots, K_n\}$$

2.2.4 Pseudo relevance feedback

This method computes the top K nearest words per query term. For example, given three query terms it would calculate the top ten nearest words for each, and would therefore return the thirty nearest words as well as the original query terms to expand the query. The following algorithm is used for this process:

```
terms = list()
for each Query term i in Q:
    if i is not in EmbeddingList || i is STOP_WORD:
        continue
    # word vector for the term i in Q
    v1 = i
    for each Candidate set term j in C:
        if j == i || j is not in EmbeddingList:
            continue
        # word vector for the term j in C
        v2 = j
        val = CosineSimilarity(v1,v2)
        HashMap.add(j,val)
    # sort the values of the HashMap
    HashMap.SortByValue()
    # Add the top K terms to the terms list
    terms.add(HashMap(topk))

# return the terms list
return terms
```

2.2.5 Query expansion with document frequency

This method computes the K nearest words per query term as described above. Terms are sorted in descending order by their document frequency and the top K are chosen for expansion.

2.2.6 Query expansion with inverse document frequency

The rationale of this approach is to expand the query on rare words. The K nearest words per query term are computed and sorted in descending order by their inverse document frequency. The top K are then chosen for expansion.

2.2.7 Query expansion with entity abstract

This method uses candidate set generation as described in section 2.2.2 and computes the K nearest words per query term as described above. The IDF and DF methods from sections 2.2.5 and 2.2.6 are used to find the expansion terms.

2.2.8 Relevance model 3

This is a probabilistic approach that models how useful a term is for expanding a query. Unlike word embedding, term importance is calculated by considering the probability of each term in the top K documents. The process is as follows:

- Retrieve the documents using BM25 and assume the top K are relevant
- Build document statistics and the term frequency vector for each document
- Get the candidate terms using candidate generation method 1.
- For each candidate term, compute its probability using

$$p(w|q) = \sum_{d_k} p(w|d)p(d|q)$$

- Choose the K highest probability terms for query expansion

2.2.9 Learning to rank

All of the query expansion approaches besides RM3 use learning to rank to combine the scores. Each method's output is considered one value for a query-document in a feature vector. The process is as follows:

- Generate the feature vector by combining all the run files into a feature file. (Run files from the train queries are used to create a feature vector)
- Z-score normalize each feature vector across the queries to create the weight vector
- Generate the feature file with normalization for the test queries
- Compute the dot product between the weight vector and the feature vector to generate the final scalar score for the query-document

2.2.10 Command line

The following command runs all of the query expansion methods described above.

```
java -jar -Xmx10g cs953-team1-1.0-SNAPSHOT-jar-with-dependencies.jar
search article -i <index_path> -q <query_path> -we <embedding_path>
-dim=300 --qe-exp-df --qe-exp-idf --qe-exp-entity --qe-exp-rm3 -bm25
--prf-val-term 200 --prf-val-k 200 --prf-val 10
```

2.2.11 Query expansion using top n entities

This method retrieves all of the entities in the document and counts how many time each one appears. The text of the highest frequencies are appended to the query.

- Command

```
    search --query-expansion Or -qe
        -i   "indexed_file"
        -q   "test cbor file"
        -top "number of top entity used in the expansion default 3"
        -qe-type "type of Query expansion
            * Query expansion with the entity text
            * Query expansion with the entity ID
            * Query expansion with the entity text and ID
            * Query expansion only Entity ID in the Entity ID field
        default entityText "
        values (entityText, entityID , entityTextID, entityIDInEntityField)
example:
    search -qe
        -i /home/team1/indexed_file
        -q /home/ama1003/DataScienceCS953/Prototype1/test.pages.cbor-outlines.cbor
        -top 3
        -qet entityTextID
```

- Validation: If the top number is less than zero, an error message is displayed, and the program stops.
- Results: After running successfully the results are saved in the "result" directory in /home/ama1003/DataScienceCS953/Prototype3/result

2.3 Using DBpedia

We decided to use DBpedia to get extra information about entities found in TREC-CAR, such as the relationship between entities and their type or category.

2.3.1 Query expansion using DBpedia

DBpeadia and SPARQL were used to expand queries by adding each entity retrieved from BM25 to a new query. DBpedia was searched for these new queries, and row counts were returned. In theory, high row counts would demonstrate an efficient expansion. Unfortunately, this method failed as no records were found for the expanded query.

2.3.2 Querying relations using DBpedia

We also used DBpedia to extract relations between entities. Our method takes two entities and uses jena.apache to construct a SPARQL query on the DBpedia API and extract their relation and type. Unfortunately, we encountered problems verifying the results since the outputs between the http://dbpedia.org/sparql tool and the API are not consistent.

2.3.3 Statistics from DBpedia

We were able to compute statistics on the amount of data from TREC-CAR that exists in DBpedia. First, we checked whether a given entity name exists in any part of the DBpedia label field. This returned a large number of records. Second, we looked for records that were exact matches of the entity name. Though fewer records were retrieved, they tended to be more relevant to the entity than those retrieved through the previous method. Train and test data from benchmarkY1 for section (outlines) level were used in this phase.

```
The comparison was run using the following command:
{"-dbpedia" or "--exist-dbpedia"}

To change the search from exact match to contrin use this command:
{"-dbpcontain" or "--dbpedia-contain"}


 Command example:
 search article
    -i /home/team1/indexed_file
    -q /home/ama1003/DataScienceCS953/Prototype1
        /test.pages.cbor-outlines.cbor
    -dbpedia
    (option) -dbpcontain

All results (including detail result for each entity search)
can found in /home/ama1003/DataScienceCS953/Prototype3/DBpedia
```

2.3.4 Extracting relations between entities from DBpedia

DBpedia was used to create a method that takes two entities and extracts all relations between them by retrieving all type similarities between them.

2.4 Entity Relations

The task focuses on identifying entities relevant to the query and leveraging the relevant entities to improve the passage retrieval relevant to the query. The mo-

tivation of the idea is that the relevant passages to the query contains relevant entities so identifying relevant entities would help in identifying relevant passages. To identify the relevant entities, the relations between the entity pairs is explored here. The below gives the features explored and the ranking methods used.

The candidate set of passages is retrieved using BM25 for every query which is used as baseline. For entity retrieval, the frequency of the entity in the BM25 result is used as the baseline.

2.4.1 Features

The feature vectors are built of the following features:

- 1 Hop Relation : For every entity mentioned in the retrieved passages of BM25 baseline, it is checked whether that entity is present in the either outlinks or inlinks of every other entities.
  - For every entity i.e. subject entity, create a pair with every other entity i.e. object entity i.e. (e1, e2), (e1, e3), (e2, e1), (e2, e3) etc.
  - Check whether the subject entity is present in either outlinks or inlinks field of the object entity.
  - Calculate an average for every entity as

$$e_i = \frac{\sum_{j=1}^{N}(e_i, e_j)}{N}, i \neq j \tag{1}$$

  - Create a feature vector where value of every entity is the above calculated average
  - Sort the vector based on the value and select top 100
- Entity Co-mention : For every entity mentioned in the retrieved passages of BM25 baseline, it is checked whether that entity is present in the same passage with other entity i.e. is the pair of the entity mentioned together in the same passage.
  - For every entity i.e. subject entity, create a pair with every other entity i.e. object entity i.e. (e1, e2), (e1, e3), (e2, e1), (e2, e3) etc.
  - Check whether the subject entity and the object entity is present in the same passage
  - If it is present then calculate the value of the pair as

$$(e_i, e_j) = \frac{1}{passage\_rank}, i \neq j \tag{2}$$

  - Calculate an average for every entity as given in equation (1) where the value of (e_i, e_j) is calculated using equation (2)

  - Create a feature vector where value of every entity is the above calculated average
  - Sort the vector based on the value and select top 100

- EcmX-EntityLinks : An external system of TREC-CAR-Methods developed by Dr. Laura Dietz is used to calculate the probabilities of entities based on Entity Context Model. The field EntityLinks is used to calculate the probabilities.

- EcmX-LeadText : An external system of TREC-CAR-Methods developed by Dr. Laura Dietz is used to calculate the probabilities of entities based on Entity Context Model. The field LeadText is used to calculate the probabilities.

- BM25-Frequency: For every entity, calculate the count of the entity in the top 100 BM25 passages. This is used as baseline for entity retrieval part of the task.

2.4.2 Ranking

- Entity Ranking
  - Learning-To-Rank - All Features
    * A combined feature vector file is generated as per ranklib format
    * Normalize each feature with zscore (This is done using a seperate Python script written by Pooja Oza)
    * Train the RankLib model using zscored training feature vectors and generate the weights
    * Calculate the final score of the entity as dot product between the feature vector and the weights
  - Average Centroid Vector - All Features
    * Calculate the centroid vector by calculating the average for every feature
    * Calculate the final score of the entity as dot product between the feature vector and the centroid vector

- Passage Re-Ranking
  - Learning-to-Rank Degree Centrality
    * Re-rank the passage by adding the entity score calculated by Learning-to-Rank approach mentioned above of every entity present in the passage with passage score of BM25
  - Average Centroid Degree Centrality
    * Re-rank the passage by adding the entity score calculated by Average Centroid approach mentioned above of every entity present in the passage with passage score of BM25

- Entity Degree
    - Re-rank the passages by adding the frequency of the entities present in the passage with the passage score of BM25

2.4.3 Commands

- 1 Hop Relation & Entity Co-mention:

```
java -jar target/cs953-team1-1.0-SNAPSHOT-jar-with-dependencies.jar search
--entity-relation section --entity-index /home/team1/entity.lucene/
-q /home/team1/query_data/benchmarkY1-test/test.pages.cbor-outlines.cbor
-i /home/team1/indexed_file/
-qrel /home/team1/query_data/benchmarkY1-test/
test.pages.cbor-hierarchical.entity.qrels
```

  The output files are generated in the **result** folder with the name *output_ranking_1hoprelation_feature_vector_section_test.txt* and *output_ranking_comention_feature_vector_section_test.txt*

  Please note that this command takes a huge time to execute and generate feature files. These features files are then zscored using python script.

- BM25-Frequency

```
java -jar target/cs953-team1-1.0-SNAPSHOT-jar-with-dependencies.jar search
--entity-default-freq section --entity-index /home/team1/entity.lucene/
-q /home/team1/query_data/benchmarkY1-test/test.pages.cbor-outlines.cbor
-i /home/team1/indexed_file/
-qrel /home/team1/query_data/benchmarkY1-test/
test.pages.cbor-hierarchical.entity.qrels
```

  The output files are generated in the **result** folder with the name *output_ranking_entityBM25Freq_section_test.txt*

- Learning-to-Rank

```
java -jar target/cs953-team1-1.0-SNAPSHOT-jar-with-dependencies.jar search
--entity-ranklib section
-f /home/team1/prototype3/pooja_data/
output_ranking_feature_vector_section_test_zscored_python.txt
-model /home/team1/prototype3/pooja_data/momodel_section_train_prototyp3.txt
-q /home/team1/query_data/benchmarkY1-test/test.pages.cbor-outlines.cbor
-i /home/team1/indexed_file/
```

The output files are generated in the **result** folder with the name *output_ranking_entity_ranklib_section_test.txt* and *output_ranking_paragraph_ranklib_section_test.txt*

- Average Centroid

```
java -jar target/cs953-team1-1.0-SNAPSHOT-jar-with-dependencies.jar search
--entity-centroid section
-f /home/team1/prototype3/pooja_data/
output_ranking_feature_vector_section_test_python.txt
-model /home/team1/prototype3/pooja_data/momodel_section_train_prototyp3.txt
-q /home/team1/query_data/benchmarkY1-test/test.pages.cbor-outlines.cbor
-i /home/team1/indexed_file/
```

The output files are generated in the **result** folder with the name *output_ranking_entity_avg_centroid_section_test.txt* and *output_ranking_paragraph_avg_centroid_section_test.txt*

- Entity Degree

```
java -jar target/cs953-team1-1.0-SNAPSHOT-jar-with-dependencies.jar search
--entity-degree section
-i /home/team1/indexed_file/
-q /home/team1/query_data/benchmarkY1-test/test.pages.cbor-outlines.cbor
--entity-index /home/team1/entity.lucene/
```

The output files are generated in the **result** folder with the name *output_ranking_entityDegree_section_test.txt*

3 Results

3.1 Spam Filter

The SpecialCharPredictor gets a document's ratio of non-alphabetic characters to total tokens, and multiplies this ratio by a weight parameter if other conditions are met. For instance, if the text begins or ends with a special character, it is very likely to be spam, so the previously-computed ratio is multiplied by a lambda value. Due to the nature of the spam data, this classifier outperformed all other classifiers in the predictors package, with F1 and MAP scores of 0.797 and 0.319 respectively **(Figures 1 and 2)**.

Though the LinearSVM achieved the best performance **(Table 1)**, an F1 score of 1.0 is perhaps too optimistic, and further validation of this result would be ideal before taking it at face value. The language models that performed the best were the Naive Bayes unigram and bigram classifiers. Further n-grams only decreased performance. It also turns out that stop words are a very poor

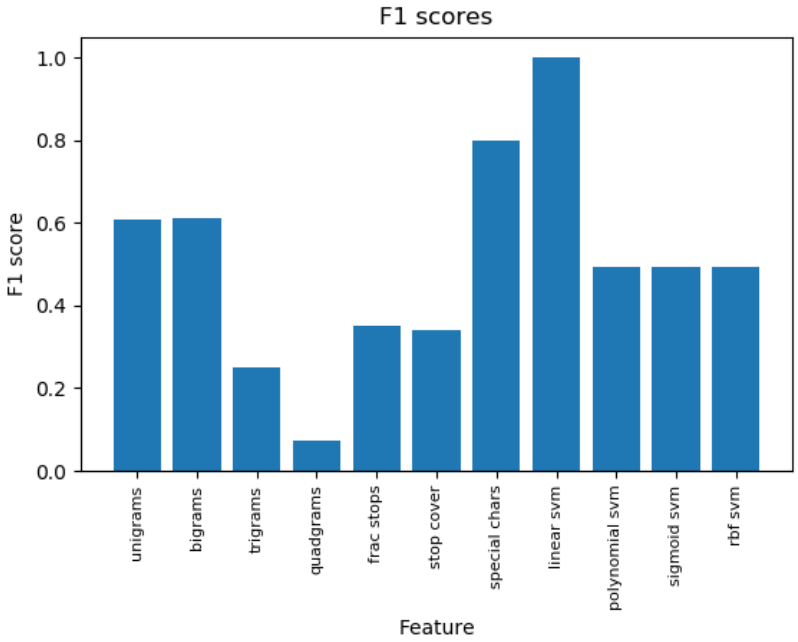

Figure 1: F1 scores for each classifier. Though the LinearSVM performs best, the SpecialCharClassifier also achieved good performance.

Kernal Type	**F1**	**Accuracy**	**Precision**	**Recall**
Linear	1	1	1	1
Polynomial	0.492	0.9711	0.485	0.5
Sigmoid	0.492	0.9711	0.485	0.5
Rbf	0.492	0.9711	0.485	0.5

Table 1: F1 score, accuracy, precision, and recall for the four different kernal types availabel in the PythonSVM class.

indicator of whether a document will be spam, at least for this particular data set.

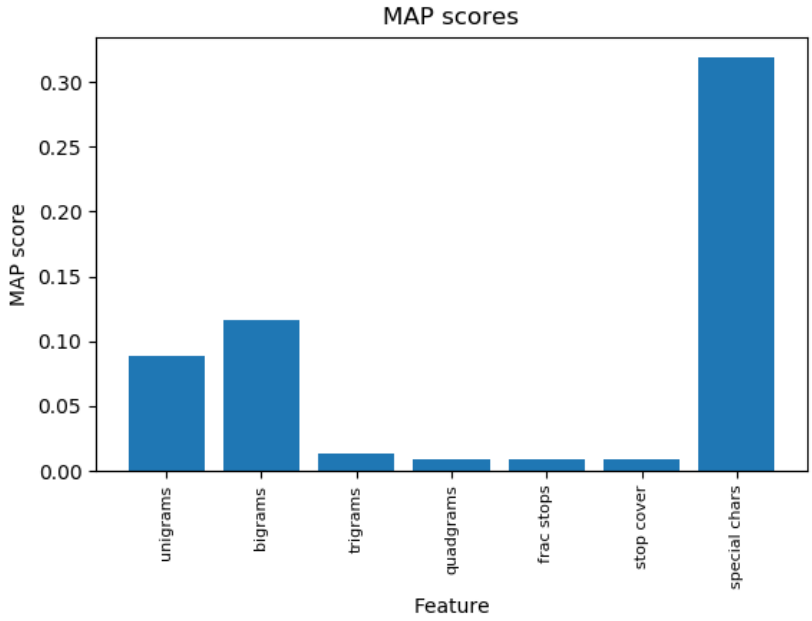

Figure 2: MAP scores for each classifier in the predictors package. The SpecialCharClassifier outperformed the other classifiers by a wide margin.

3.2 Query Expansion

The methods in the below table are:

- BM25 - Lucene default implementation
- DF - Query expansion using Document Frequency 2.2.5
- IDF - Query expansion using Inverse DF 2.2.6
- ENT-DF, ENT-IDF - Query expansion using Entities abstract 2.2.7
- RM3 - Relevance model 3 2.2.8
- Combined - Learning to rank for the query expansion methods 2.2.9

3.2.1 Page Level

Method Name	**Map**	**Rprec**
BM25	0.0950	0.1579
DF	0.0822	0.1375
IDF	0.0822	0.0822
ENT-DF	0.0126	0.0334
ENT-IDF	0.0292	0.0640
RM3	0.0749	0.1363
Combined	0.1105	0.1645

Table 2: Page-No Spam Filter enabled-Train

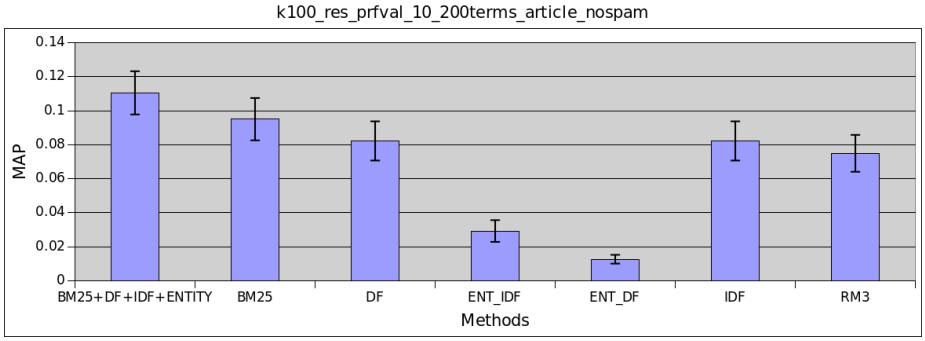

Figure 3: Page-No Spam Filter enabled-Train.

Method Name	Map	Rprec
BM25	0.0868	0.1447
DF	0.0723	0.1225
IDF	0.0723	0.1225
ENT-DF	0.0092	0.0272
ENT-IDF	0.0183	0.0471
RM3	0.0630	0.1148
Combined	0.1003	0.1531

Table 3: Page-No Spam Filter enabled-Test

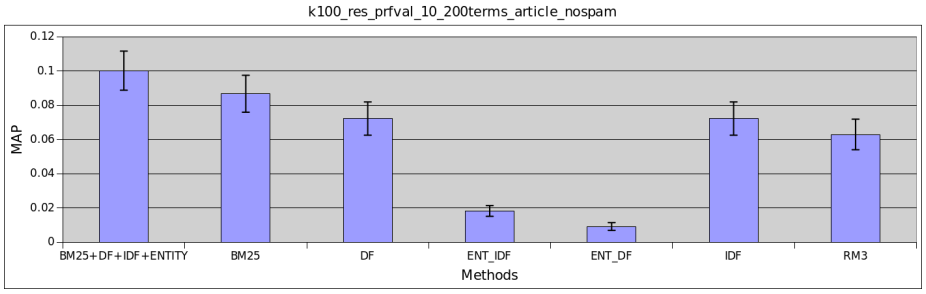

Figure 4: Page-No Spam Filter enabled-Test.

Method Name	Map	Rprec
BM25	0.0954	0.15797
DF	0.0817	0.1379
IDF	0.0817	0.1379
ENT-DF	0.0111	0.0272
ENT-IDF	0.0278	0.0620
RM3	0.0728	0.1370
Combined	0.1132	0.1643

Table 4: Page-Spam Filter enabled-Train

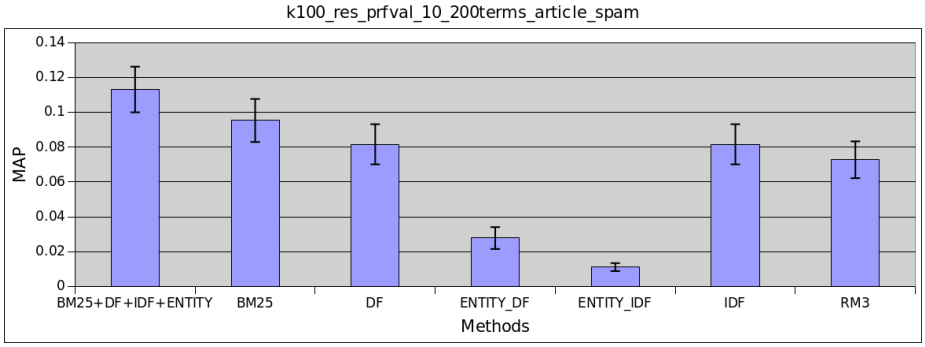

Figure 5: Page-Spam Filter enabled-Train.

Method Name	**Map**	**Rprec**
BM25	0.0885	0.1451
DF	0.0704	0.1224
IDF	0.0704	0.1224
ENT-DF	0.0098	0.0292
ENT-IDF	0.0193	0.0490
RM3	0.0624	0.1166
Combined	0.1029	0.1552

Table 5: Page-Spam Filter enabled-Test

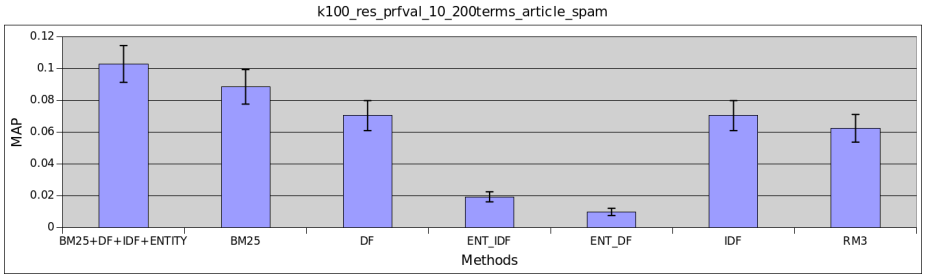

Figure 6: Page-Spam Filter enabled-Test.

3.2.2 Section level

Method Name	Map	Rprec
BM25	0.1476	0.1115
DF	0.0986	0.0615
IDF	0.0986	0.0615
ENT-DF	0.0138	0.0098
ENT-IDF	0.0347	0.0222
RM3	0.0948	0.0530
Combined	0.1584	0.1203

Table 6: Section-No Spam Filter enabled-Train

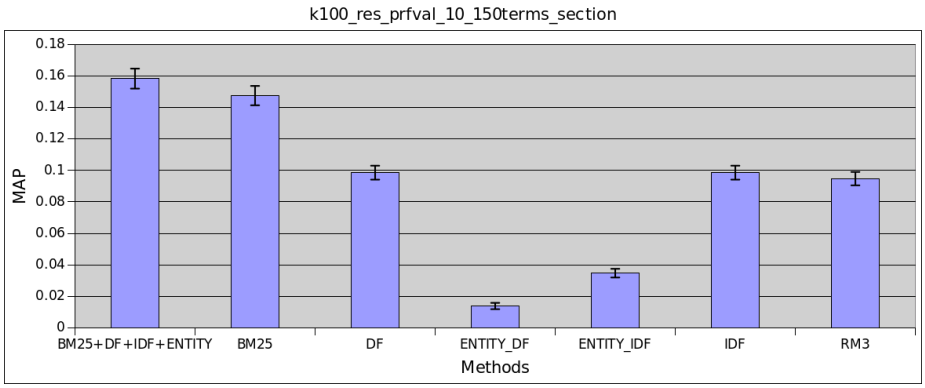

Figure 7: Section-No Spam Filter enabled-Train.

Method Name	Map	Rprec
BM25	0.1392	0.1062
DF	0.0997	0.0691
IDF	0.0997	0.0691
ENT-DF	0.0142	0.0105
ENT-IDF	0.0389	0.0266
RM3	0.0906	0.0590
Combined	0.1476	0.1128

Table 7: Section-No Spam Filter enabled-Test

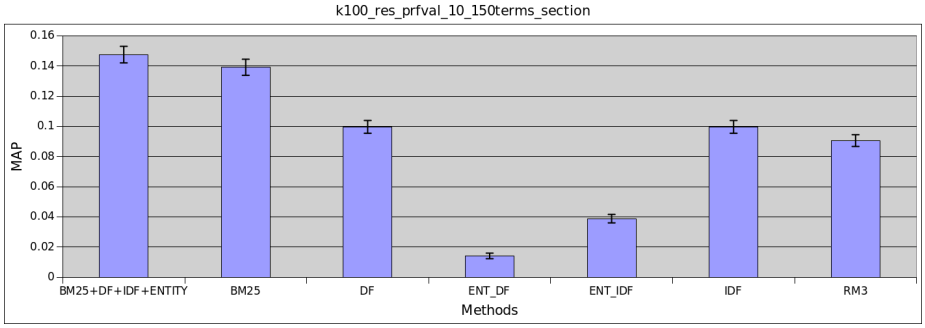

Figure 8: Section-No Spam Filter enabled-Test.

3.2.3 Note and Observations

Note:

- Spam filters is applied only for the page level queries.
- Image caption tells whether spam filter is applied or not.

Observations: Applying SpecialCharSpamFilter as spam filter helped us to improve the result of the page level queries from .1003 to .1029. Out of all methods, the best performing method score for the page level is **0.1029** and for the section level the score is **0.1476** on the Benchmark-y1-test.Here is the comparison with baseline on the Benchmark-y1-test, L2R indicates Learning to rank score.

Method Name	**Map**	**Rprec**	**Level**
Baseline-BM25	0.0885	0.1451	page
L2R	0.1029	0.1552	page
Baseline-BM25	0.1392	0.1062	section
L2R	0.1476	0.1128	section

Table 8: Baseline improvements

The results can be found here "/home/team1/prototype3/amith_data"

3.3 DBpedia

The following shows the results of the TREC-CAR and DBpedia comparison. We used two TREC-CAR entity datasets, one for training, and the other for testing.

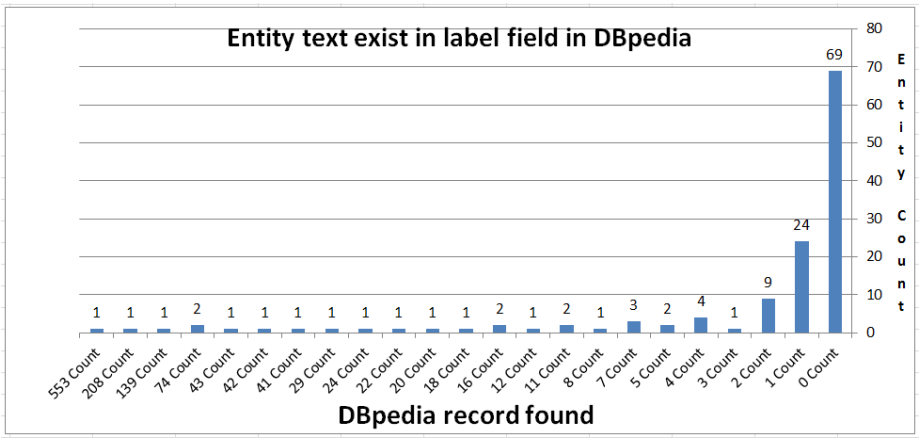

Figure 9: Results from the test dataset. The Y axis shows the number of entities, and the X axis shows the number of records found that contain the entity.

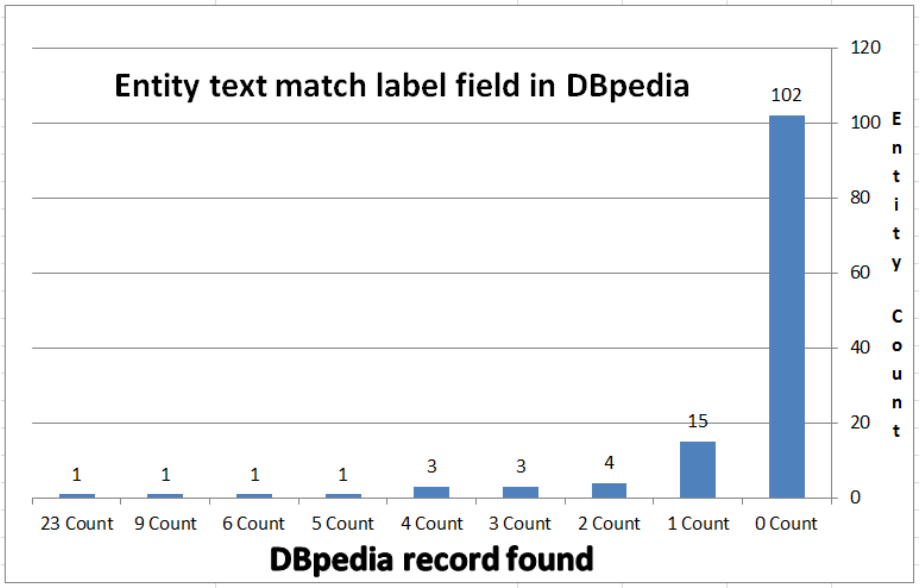

Figure 10: Results from the test dataset. The Y axis shows the number of entities, and the X axis shows the number of records found that match an entity's text. Notice that there are a higher number of records with zero than in the previous figure.

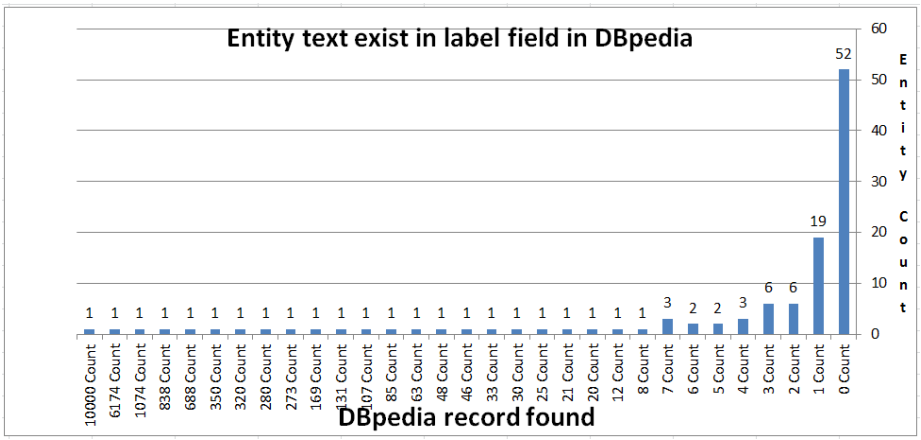

Figure 11: Results from the train dataset. The Y axis shows the number of entities, and the X axis shows the number of records found that contain the entity.

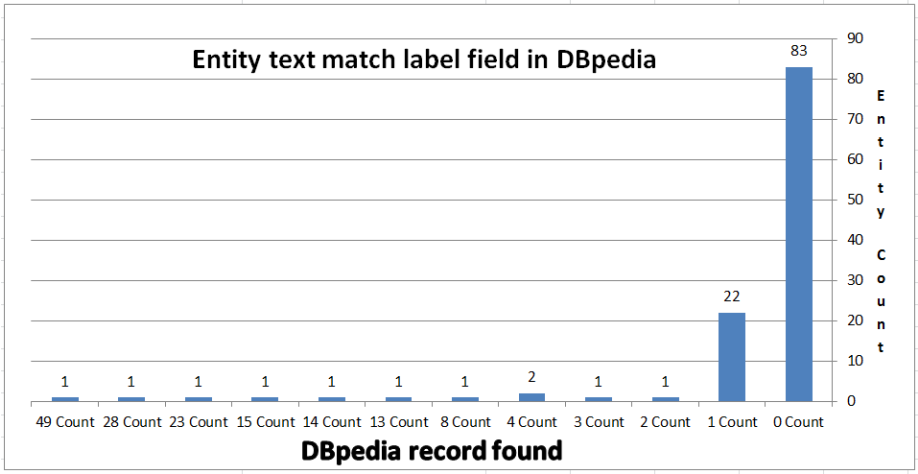

Figure 12: Results from the train dataset. The Y axis shows the number of entities, and the X axis shows the number of records found that match an entity's text. Notice that there are a higher number of records with zero than in the previous figure.

3.4 Entity Relation

Method Name	Map
BM25-Freq	0.1108
1hopRelation	0.0788
Entity Co-mention	0.1485
EcmX-EntityLinks	0.0525
EcmX-LeadText	0.0479
Learning2Rank – All Features	0.1577
AvgCentroid – All Features	0.1159

Table 9: Entity Retrieval BenchmarkY1-Test Section Level

Method Name	Map
BM25	0.1392
Learning2Rank Degree Centrality	0.1198
AvgCentroid Degree Centrality	0.0422
Entity Degree	0.1103

Table 10: Passage Retrieval BenchmarkY1-Test Section Level

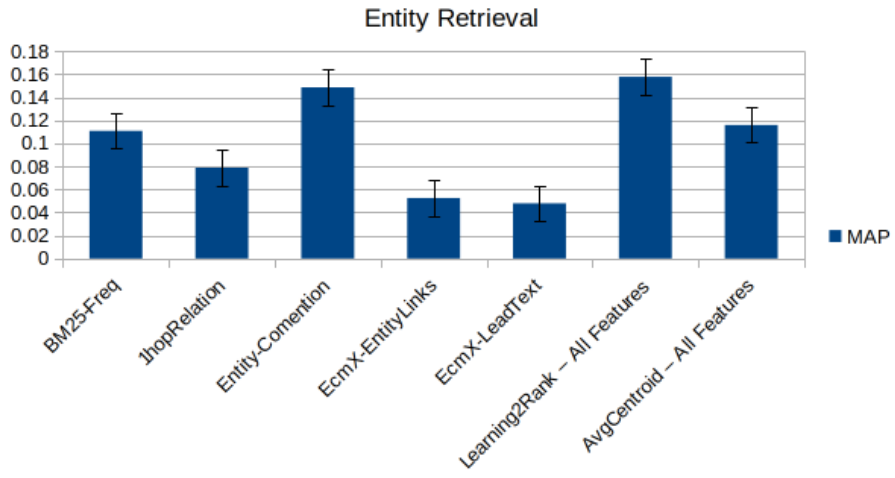

Figure 13: MAP scores at Section Level for BenchmarkY1-Test for each feature described in section 2.4 and for all features combined. The learning-to-rank of all features works the best followed by Entity Co-mention feature.

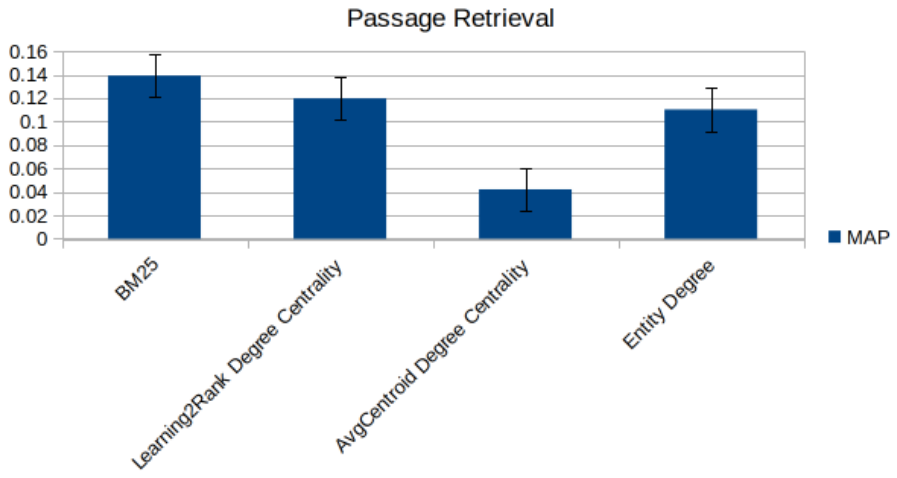

Figure 14: MAP scores for each Passage Re-Ranking method given section 2.4 at Section Level for BenchmarkY1-Test.

4 Contributions

- **Rachel:** Responsible for the command line option to index the manual qrels for the creation of ham and spam training and test sets. Implemented a series of spam classifiers in the predictors package, as well as the F1 and MAP classes in the evaluators package to assess their performance. Also implemented a wrapper class for Python's sklearn support vector machine with four different kernal options. This can be found in the svm package.
- **Amith:** Responsible for implementing all new query expansion methods such as Document Frequency, Inverse Document Frequency, Entity-Abstract Candidate Generation, and Relevance Model 3. Also implemented a program which automates the process of combining the scores for query-document from different methods, learn a weighted vector from the train queries and use it on test queries to create a combined run file, install scripts, install script document, merging codes
- **Pooja:** Implemented the entity relation methods described in Section 2.4 as well as the RankLib feature vector file generation in Python. Created and maintained the Mongodb collection for entities.
- **Ahmed:** Responsible for querying DBpedia and extracting useful information and statistics related to TREC-CAR. Also implemented query expansion using entity text and/or ID. Helped integrate the spam filter with the rest of the system.