IR Project Summary

Roee Tarab
University of Haifa
Haifa, Israel

Yoni Weidenfeld
University of Haifa
Haifa, Israel

ABSTRACT

In this paper, we describe the process of building our project, top to bottom. Specifically, we discuss the type of data processing used, retrieval approach implemented and the academic paper we relied upon. In addition, we state some of the difficulties and problems we had to confront with, and some solutions and ways of coping.

KEYWORDS

Question Answering, Learning To Rank, Solr, Stanford Dependency Parser.

1 INTRODUCTION

In this project, we choose to implement and try to re-achieve the outcomes of the main study chosen for this project [1]. Our goal is to retrieve the top 5 rated answers out of a set of possible answers for a given query out of a known data set. Our strategy for achieving such a goal, is by first creating a pool of candidates out of the given data set (Yahoo! Webscope L6 collection [2]). Then, we wish to extract some feature values for each question-answer couple in each block of the data set. These feature values will be given as an input to the RankLib library, which in turn will build us a re-ranking model to be given to Solr program. This model will then act as a retrieving algorithm upon submitting a query, and retrieve a 5-highest-ranked list out of the entire answers data set.

2 Data processing and analysis

2.1 Query's grammatical structure

The first step of the project, was merging the Stanford Dependency Parser[3] in our project, "Stanford-corenlp-3.9.1.jar". The purpose is to obtain the grammatical structure of a sentence and overcome the problem of verbosity of the given user queries. The Stanford Parser builds connections between couples and threesomes of phrases with linguistic connections between them. In addition, the parser outputs a connection tree.

After many tests made, we found the following connections were the ones most relevant to us, and reflect the true meaning of the sentence in the best way:

- **"nmod"** – The nmod relation is used for nominal modifiers of nouns or clausal predicates. nmod is a noun functioning as a non-core (oblique) argument or adjunct. Used for prepositional complements and for ‘s genitives.

Example: "The office of the chair" *nmod*(office, chair).

- **"case"** - case is usually an inflectional feature of nouns and, depending on language, other parts of speech (pronouns, adjectives, determiners, numerals, verbs) that mark agreement with nouns.

Example: "The office of the chair" *case*(of, chair).

- **"compound"** - noun compounds.

Example: "Phone book" *compound*(book, phone).

- **"amod"** - An adjectival modifier of an NP is any adjectival phrase that serves to modify the meaning of the NP.

Example: "Sam eats red meat" *amod*(meat, red).

- **"nsubj"** - A nominal subject is a noun phrase which is the syntactic subject of a clause. The governor of this relation might not always be a verb: when the verb is a copular verb, the root of the clause is the complement of the copular verb, which can be an adjective or noun.

Example: "Clinton defeated Dole" *nsubj*(defeated, Clinton).

- **"advmod"** - An adverb modifier of a word is a (non-clausal) adverb or adverb-headed phrase that serves to modify the meaning of the word.

Example: "Genetically modified food" *advmod*(modified, genetically).

2.2 Analyzing data-set and query

The output of step 2.1(meaningful words which were extracted from the user query in real time) as well as our data set move on to this step.

The data processing itself is all included in the Schema file of our project[4]. We created the "nbestanswers", "question" and "main_category" fields for convenience and defined their field types to be *text_en_splitting*, which also defines the tokenizers and filters in the Schema file.

The *text_en_splitting* consists of two analyzers. One for the indexed data and one for the output of step 2.1. The filter classes were added to examine a string of tokens, and decide whether to keep them, transform them or discard them depending on the attributes given. In this paper, Before computing each of the features, all terms from query and candidate answers were stemmed using Porter (solr.PorterStemFilterFactory). Some of the other classes that were used for this purpose: solr.WhitespaceTokenizerFactory, solr.StopFilterFactory, solr.WordDelimiterGraphFilterFactory,

solr.LowerCaseFilterFactory, solr.KeywordMarkerFilterFactory, solr.FlattenGraphFilterFactory.

At this point we would like to mention that most of the job done for this purpose was research oriented, in order to find and include the correct elements specified in the paper, so to recreate its success in the final retrieval scores, and less program oriented. With that being said, most of the job done using the Stanford Dependency Parser was of programmatic nature.

3 Description of the retrieval approach implemented

3.1 Pool of candidates

In order to create the pool of candidates described in the paper, we ran the Stanford Dependency Parser's output in Solr's retrieval mechanism, along with a default similarity measure (BM25): "solr.BM25Similarity".

3.2 Reranking

After successfully building a pool of candidates, the next step was to generate a re-ranking algorithm upon the pool of candidates. For this we used the "Lemur Project RankLib" library, with the Learning to rank based LambdaMart algorithm, as described in the paper.

RankLib receives a unique feature file format:

<line> .=. <target> qid:<qid> <feature>:<value>
<feature>:<value> ... <feature>:<value> # <info>
<target> .=. <positive integer>
<qid> .=. <positive integer>
<feature> .=. <positive integer>
<value> .=. <float>
<info> .=. <string>

For example (taken from the SVM-Rank website. Note that everything after "#" is ignored):

3 qid:1 1:1 2:1 3:0 4:0.2 5:0 # 1A
2 qid:1 1:0 2:0 3:1 4:0.1 5:1 # 1B
1 qid:1 1:0 2:1 3:0 4:0.4 5:0 # 1C
1 qid:1 1:0 2:0 3:1 4:0.3 5:0 # 1D
1 qid:2 1:0 2:0 3:1 4:0.2 5:0 # 2A
2 qid:2 1:1 2:0 3:1 4:0.4 5:0 # 2B

In order to create this file format we had to build a feature file in Solr[5] from scratch, so we could retrieve the correct feature values to be entered to RankLib as an input. The features we built represent specific properties between question and answer, for each question-answer couple, such as: number of overlapping terms, standard BM25 score, Number of words in an answer etc.

In order to generate the features for each of the 80000 couples, we built a C# script[6] to extract the feature results using the following query:

```
string url =
string.Format("{0}localhost:8983/solr/techproducts/query?q=nbestanswers:{1}&fl=id,answer,nbestanswers,[features%20store=my_efi_feature_store%20efi.text_a={2}%20efi.text_b={3}%20efi.text_c={4}]&start=0&rows=1000", "http://", questionToSolr, dominantWordsInQuestion[0], dominantWordsInQuestion[1], dominantWordsInQuestion[2]);
```

remark: The parameter dominantWordsInQuestion[i], is used as an argument to the "number of overlapping terms" feature extracted from the question programmatically.

In order to compute the relevance label, which is the <target> in the RankLib file format, and is necessary to it, we had to come up with a sufficient scoring measure for relevancy, and normalize it to a scale of 1 to 4. The measure chosen was Solr's original score and the scaling was done upon it, in the following manner:

```
current_relevance = (((3 * (currentPairoriginal - minoriginalScore)) / (maxoriginalScore - minoriginalScore)) + 1);
```

minoriginalScore – min original score of all pairs.

maxoriginalScore - max original score of all pairs.

currentPairoriginal – current pair original score

current_relevance – current pair relevance label

The feature file was uploaded to Solr using the following command:

```
curl -X PUT -H "Content-Type: application/json" -d "@MyfeaturesFile.json" "http://localhost:8983/solr/techproducts/schema/feature-store"
```

We then had to process the output given to us by Solr in order to extract the feature values from the HTML output. This was also done with the aid of a C# script[6] we built.

Only at this stage, we could have built the feature file needed for RankLib[7]. We used the RankLib library to build the model using our feature file with the following command:

```
java -jar bin/RankLib-2.1-patched.jar -train featurefile.txt -ranker 6 -metric2t NDCG@10 -metric2T ERR@10 -round 600 -lr 0.000005 -tree 100 -save mymodelToSolr.txt
```

LambdaMart algorithm's output was an XML file, including a trained model which we could have then used in order to retrieve our data. But there was still a problem with the output's format. Solr's input was a JSON file. So we had to do the

conversion (XML to JSON) manually. We used a python script[8] for that purpose.

Next, we uploaded the model to Solr using the Curl command:

```
curl -X PUT -H "Content-Type: application/json"  -d
"@mymodelFormat.json"
http://localhost:8983/solr/techproducts/schema/model-store
```

Finally, we used the following URL query:

```
string url =
string.Format("http://localhost:8983/solr/techproducts/query?q=
nbestanswers:{1}&rq={{!ltr%20model=my_efi_feature_store%20re
RankDocs=1000%20efi.text_a={2}%20efi.text_b={3}%20efi.text_c={
4}}}&fl=id,score,question,nbestanswers&start=0&rows=1000",
parameterstoQuery[0], parameterstoQuery[1],
parameterstoQuery[2], parameterstoQuery[3]);
```

in order to retrieve the final 5 top ranked answers.
Let us explain the mentioned URL:

- **q=nbestanswers:** means we search on the "nbestanswers" field.
- **ltr%20model=my_efi_feature_store:** means we use the ltrModel named "my_efi_feature_store".
- **reRankDocs=1000:** means we rerank on the first 1000 candidates.
- **efi.text_a, efi.text_b and efi.text_c**: these are the first, second and third parameters respectively.
- **fl=id,score,question,nbestanswers:** defines which properties will be returned.
- **start=0&rows=1000:** guarantees having 1000 canditates to rerank upon.
- **parameterstoQuery[i]**: these are the parameters to the "number of overlapping terms" feature given in the feature file.

4 difficulties and problems

In the process of this project, we encountered a very large set of difficulties, and came up with quite a few solutions in the end of a very long journey. We would like to enumerate some of these problems:

1. The main issue was that we faced this project with zero familiarity with Machine Learning subject. Every step we took in this world was new to us, and we had to learn a lot of things "on the go". Some basic knowledge would have served us a great deal with some fundamental concepts of learning to rank methods.
2. Most of the project was research oriented. We had to understand in what way do all the tools and libraries we were exposed to work exactly. We had to understand the rationale behind each and every one of them, and how to make the connection between them. There was very little information about the way to implement an ltr algorithm through RankLib and assimilating the output model to Solr, and even the basic fact that RankLib is not a retrieval engine, and does not work directly with the data set was not clear to us from the beginning.
3. RankLib feature file format was also a puzzle for us. The format was very difficult to understand, as we were always thinking about a way to incorporate the data set into RankLib. We then understood that the algorithm needs only numbers to learn, and the actual strings in the data set are irrelevant.
4. How can we generate the feature file format? Even after understanding we must work with numbers, we had to extract the values for the features somehow. Most of the actual code in this project evolved from the need to extract the values of the features for RankLib.
5. There were a lot of technical issues with Solr – from problems with the Curl command, to non-informative errors while uploading the model file (7 had to be casted to "7" in order to avoid a non-informative error) and many more.

This is only the tip of the iceberg regarding the difficulties we had to overcome in this project, and the hours spent on each of these seemingly minor issues, and major issues together.

5 Summary

Overall, this was a very educating and satisfying journey through the world of information retrieval and we are very glad to be familiar with it's fundamentals now. We hope to someday have our paths cross this subject again.

REFERENCES

[1] Khvalchik M., Kulkarni A. (2017) Open-Domain Non-factoid Question Answering. In: Ekštein K., Matoušek V. (eds) Text, Speech, and Dialogue. TSD 2017. Lecture Notes in Computer Science, vol 10415. Springer, Cham. L IR-Project/based paper.pdf at master · yoniwe1/IR-Project · GitHub
[2] Yahoo! Webscope L6 collection https://ciir.cs.umass.edu/downloads/nfL6/
[3] Stanford Dependency Parser https://stanfordnlp.github.io/CoreNLP/download.html
[4] Schema File https://github.com/yoniwe1/IR-Project/blob/master/managed-schema
[5] Solr's feature file https://github.com/yoniwe1/IR-Project/blob/master/myFeatures.json
[6] Script to extract feature values https://github.com/yoniwe1/IR-Project/blob/master/FeaturesForRank-C%23/ExtractFeatures.cs
[7] RankLib feature file https://github.com/yoniwe1/IR-Project/blob/master/formatToRankLib2.json
[8] Xml to JSON python converter https://github.com/yoniwe1/IR-Project/blob/master/convert_lambdamart_xml_to_json.py