CSC 583: Programming Project
Building (a part of) Watson

Pratik Bhandari
MS, Computer Science
University of Arizona, Tucson, Arizona
pratikbhandari@email.arizona.edu

Abstract

This document reports the progress and result of the programming project for CSC 583: Text Retrieval and Web Search. The project is based on implementing a part of IBM Watson, a Question Answering system. We use Wikipedia pages as the text source and Jeopardy questions as the testing data whose answers are the titles of the Wikipedia pages. Two models, a Naive and an Improved one were developed where the improved model was built on top of the Naive model to get higher accuracy. By combining various information retrieval techniques and concepts of Natural Language Processing, a P@1 accuracy of 29% was obtained on the best model. The GitHub repository for the project can be found here.

1 Introduction

IBM Watson is a Question Answering (QA) system that was designed by IBM over the course of several years in an effort to implement the latest advancements in information retrieval and Natural Language Processing (NLP). It is famous for appearing in the hit US TV show, Jeopardy. In this project, we implemented a part of Watson where we took Jeopardy questions on a host of different topics and searched over an indexed collection of Wikipedia pages to find the correct answer from the document collection. The answer to each question is the title of the Wikipedia page itself.

2 Data Layout and Structure

For the project, we had two sources of data. First, the collection of Wikipedia pages and second the list of questions to be asked over these Wikipedia pages. Each type is described in detail below.

2.1 Wikipedia Articles

We are provided with a collection of approximately 280,000 Wikipedia pages covering an wide array of topics. This collection includes the answer to the provided questions. The pages are stored in 80 files which means each files has a large number of documents in itself.

Each page starts with a title, enclosed in double squared brackets. For example, a page about Google star with `[[Google]]`. Everything below that and before another title is the content of the page. Some of the pages start with a `CATEGORIES` field which denote which category the page falls under. Within the content, the text is divided into various sub-headings where the sub-headings take the form `==Sub Heading==`. A few of the pages redirected completely to other pages using the `#REDIRECT` tag. Since these pages were scraped from the web, the usual `tpl` tags along with the `http://` text, `Image` and `File` data are also observed. Details about how these lines were parsed and indexed will be discussed in later sections.

2.2 Questions File

The test question file is a single `questions.txt` file consisting of 100 questions from the Jeopardy game asked from shows that took place between 2013-01-01 and 2013-01-07. As mentioned in the previous subsection, the answer to these questions are the titles of the Wikipedia pages. Structurally, the questions are listed in a single file, with 4 lines per question, in the following format: `CATEGORY CLUE ANSWER NEWLINE`. Some of the answers come up in the form of `ANSWER 1 | ANSWER 2` where the answers are only a slight variations of one another and any one can match the Wikipedia article.

3 Indexing and Retrieval

The first step in the design of the model is to index the Wikipedia documents such that they can be easily queried and searched for at a later time. `Lucene` was used for indexing the Wikipedia pages in my project. Before being sent for indexing, each page had to be extracted and cleaned up as required by the user.

3.1 Indexing

The level of data processing and cleanup done to the document is different for the two models i.e. Naive and Improved. Since the improved model just builds on top of the Naive model, I'll start with the Naive model first. First, the title of the document which is enclosed within two squared brackets is identified and saved. This marks the start of the current document and end of the previous one. Then, the `CATEGORIES` field is extracted if present and stored in a different variable. Finally, the remainder is treated as content and processed as follows:

- All text within the subsection of `==REFERENCES==` and `==DISTINCTIONS==` are discarded.
- Empty lines are discarded.
- All lines starting with `REDIRECT` are discarded.
- All lines starting with == and ending with == (i.e. subsection names) are discarded.
- Everything other than the above mentioned text is considered as valid content and is converted to lowercase. Furthermore, all punctuation signs are removed as well.

In the code, functions `parseDocumentNaive` and `parseLineNaive` handle the reading of the files and parsing each file line by line respectively.

The above decisions for processing was made because these lines do not add any value to the pages in terms of information retrieval. So, removing these lines will decrease the size of the document and also improve the quality of matches by decreasing the probability of false positives. For the improved model, the processing performed in addition to the above steps are:

- All lines starting with `https://` were removed. Upon further research, they did not seem to be providing much value to the contents.
- All text within and including the [tpl] and [/tpl] tags were discarded. They too were present in large quantities in the text and seldom provided in any valuable information.
- In addition to the processed content, the unprocessed raw content was collected as well. This is necessary for the answer extraction process which will be explained later in the report.

In the code, functions `parseDocumentImproved` and `parseLineImproved` handle the reading of the files and parsing each file line by line respectively.

After the pages have been read and stored in variables, I indexed them on a document by document basis i.e. one document for each page. The function `loadDocument` is responsible for this. In the *Naive* model, the category and content are not lemmatized while in the *Improved* model they are lemmatized using CLU Lab's wrapper for CoreNLP. This is why a `StandardAnalyzer` was used for the Naive model and a `WhiteSpaceAnalyzer` was used for the Improved one. For the improved model, the indexing process also involved removing stop words from the content. This did not end up improving the model very much and was discarded. More about this will be discussed in coming sections. It is important to note that the title of the pages are not modified in any way and are saved as it is. This is done for matching with the answers to be easier and less fault-prone.

3.2 Retrieval

The indexing process takes up around 3 hours to complete. After this, the retrieval component was implemented. The `questions.txt` file was read and each question, category and answer set was extracted to be queried one at a time. The parsing of the query too is different on the Naive and Improved model. For the Naive model, while the file is read line-by-line, special 'tags' are used to identify the content of each line. The processing done to each line is:

- Converted the question and category text to lowercase.

- Removed all punctuations from the question and category texts.
- The answer was left as it is since this will match with the title of a Wikipedia page which too has not been processed.
- The question was taken as a whole during querying and the category was not used as part of the query.

For the Improved model, the following additional processing were performed:

- All text between and including small brackets ($(.*)$) was removed from the category.
- The category was lemmatized.
- The query was processed as follows:
  - The parts of speech of the query terms were calculated and only relevant parts of speech were preserved and others discarded. POS tags such as adjectives, nouns, verbs, pronouns and their variations were chosen while conjunctions, interjections and Wh-pronouns were removed. This selection of POS tags is naive and will require further reading and research to get better results.
  - This result was then lemmatized.
  - Finally, all the stop words, if present, were removed as well.
- With the query now being a subset of words of the original question, the category was added to the query as well during searching. So, while searching this query was used to match with both the Wikipedia title and category field using Lucene's $MultiFieldQueryParser$.

4 Measuring Performance

With the two models of the Naive and Improved Jeopardy System developed and in place, the performance of the system was measured. For this, I chose the precision at 1 (P@1) measure to calculate the performance of the system. I chose this metric because it made sense to consider only the top 1 matched pages as correct. Furthermore, the P@1 measure is simple to calculate. By only looking at the document with the largest matched score to a query, we can create a strict precision rule. There is no specific reason for choosing P@1 over P@10 though. This is why for the later part of the project, I also consider the more relaxed P@10 performance measure to find out the top 10 best matches for a given query.

The performance for the system on various scenarios is given in the table below:

	Category	
Model	Without	With
Naive	19	N/A
Improved	22	29

Table 1: Table showing the P@1 scores for the two models of the system compared to when category was used and not. N/A in the table refers to *Not Implemented*

.

The queries are created and accuracy is calculated in the `checkAccuracyNaive` and `checkAccuracyImproved` functions for the Naive and Improved models respectively.

5 Changing the scoring function

For this section of the project, I changed the scoring function to use another measure of scoring. The default scoring function of Lucene is BM25. The system was tested on two scoring function:

- Scoring based on *tf.idf* weighting
- Scoring based on Boolean Similarity

The scoring functions can be changed by setting the `setSimilarity` method of the `IndexWriter` and `IndexSearcher` to the respective similarity function name. In this case, `tf.idf` based scoring can be done by setting `.setSimilarity(new ClassicSimilarity)` and Boolean similarity based scoring can be performed by setting `.setSimilarity(new BooleanSimilarity)`.

I changed the scoring function for the Improved model and the results were as expected.

- `tf.idf` weighting: 27
- Boolean similarity: 20

A decrease in performance i.e. accuracy was expected in the system because the default BM25 scoring function works best compared to the other two scoring methods used here. So I expected a

decrease in performance by a small value which was corroborated by the result. Similarly, Boolean similarity performs worse than `tf.idf` weighting which results in the system having the least score when this function was used to measure performance.

6 Error Analysis

After a standard IR model was implemented, I performed an error analysis of the system. In practice, this analysis went hand-by-hand during development since every new feature was a result of an analysis and the conclusion from the analysis.

Overall, 29 questions were answered correctly and 71 questions were not. Some of the questions that were correctly predicted by my system are given below:

1. Question: Daniel Hertzberg James B. Stewart of this paper shared a 1988 Pulitzer for their stories about insider trading
   Answer: The Wall Street Journal

2. Question: This woman who won consecutive heptathlons at the Olympics went to UCLA on a basketball scholarship
   Answer: Jackie Joyner-Kersee

3. Question: Several bridges, including El Tahrir, cross the Nile in this capital
   Answer: Cairo

4. Question: On May 5, 1878 Alice Chambers was the last person buried in this Dodge City, Kansas cemetery
   Answer: Boot Hill

This gives a general idea to what kind of question and data were predicted correctly and what kinds were not. The system predicted the above answers correctly because of the following reasons:

1. Unique appearance of a person together with an event/object and a date seems to consistently give positive results. In the case of `The Wall Street Journal` and `Boot Hill`, this combination in the query was only uniquely present in these documents leading them to have higher sores.

2. Query + Category matching helped boost the scores by a good margin. This can be seen in pages such as `Cairo` where the relevant information already has a high hit in the categories section.

3. Appearance of relevant terms in high frequency can lead to correct matches. If a unique word or name appears for a certain number of times, capped by a threshold, such pages get ranked higher compared to other ones.

4. Same word sequences match the questions with the documents whose text appear in the same sequence as the question text. In the match of page `Mayim Bialik`, the terms `Amy Farrah Fowler` and The Big Bang Theory appear as same word sequences since they are in the same order in question and in the page text.

Similarly, there are a considerable number of questions which have not been answered correctly. The problems observed on such question-answer scenarios are:

1. Some questions do not hold enough information for a single page to rank very high for it. For example, one of the questions is `Fisher-Price toys` and the category is `Name the parent company`. Here, the query itself is very general and the term Fisher-Price ends up appearing in a lot of different Wikipedia pages in the collection.
   Even with the category as a hint, the query is not strong enough for the correct answer to be displayed by my system. Instead the result is the page `Little People` where the term "Fisher-Price" is very high but is not related to what the question is asking for. Similar is the case for a question `Crest Toothpaste` where the answer is supposed to be "Procter Gamble", it's parent company but the result is "Crest (Toothpaste)", a page with the same text as the question itself.

2. Some of the questions are very general in nature. The text in the question along with the category does not contain enough information to point to a single answer that stands out. For example, a type of question that is asked a few times is the song of a name along with its year of release. Such as `1980: "Rock With You"` and `1988: "Man In The Mirror"`. These questions are not descriptive enough to give a correct matching with the system that I have implemented.

3. The question format for some questions were rather complex for my system to handle. For example, there are question whose categories contain text like `(Alex: We'll give you the museum. You give us the state.)`. My standard IR system discards all text within small brackets in the category field. So, all of the above text is removed. Without that, the questions becomes meaningless without a proper thing to search for, eventually leading to a false answer.

4. Some of the questions were not directly related to the Wikipedia pages and had to be indirectly linked to them. One example of this would be the `News Flash` category for `Ottoman Empire`. The question is in the form of a news flash which has not no information about any of the text that is present in the Wikipedia page for Ottoman Empire. Such questions were answered wrong by my system.

The above were some of the problems observed in the question-answer scenarios in my system. Based on this, the errors can be classified into the following classes:

1. **Errors due to generalized questions**: As discussed above, the errors in this class was due to the questions being to vague and general for any one document to match it with a reasonable score. The highest rank document would then be merely on the basis of word count regardless of context.

2. **Errors due to lack of direct matching**: Some of the questions did not directly link to a Wikipedia page of relevance. The question and category would contain almost no tokens that would match with any document of relevance and the correct answer could only be matched by connecting information from various Wikipedia pages. This might be less of an error and more of an increased complexity that my system currently does not handle. Similar case is for questions that look for song titles when song lyrics is given.

3. **Errors due to misleading and sometimes complex category text**: Some of the category in the questions contains text like `(Alex: Not "domination.")` and `(Alex: We'll give you the church. You tell us the capital city in which it is located.)`. This adds extra complexity to parse the category which sometimes also results in false positives. On the other hand, discarding this extra text will result in a query that does not hold any information about the target/output of the query.

4. **Errors due to short queries with output target in category field**: In questions which had the field `Name the parent company`, the question is just the name of a company and the system is supposed to find the Wikipedia page which refers to its parent company. The question is too short in this case and matches a large number of Wikipedia pages based on word count of the question text. Most of the times this is irrelevant to the parent company we are searching for resulting in a wrong answer.

The above are only four classes that seem evident when researching the matches and mismatches on a deeper than surface level. A more deeper look by going through each question will definitely show more errors in the implementation of the system as well as in the question and Wikipedia data format.

On my system, stemming and lemmatization had positive impact by around 6-7%. The best configuration is using lemmatization on the queries as well as the Wikipedia content. The Wikipedia page titles are not modified in any way and are just converted to lower-case. Everything else, including the categories are lemmatized. This works best for my system because by lemmatizing, most of the words are converted to their root form. Since both the query and content is lemmatized, the chances of "same word sequencing" will increase since both text will be in the same form. Hence, grounding the text into a common form and removing inflections helped my system's performance by a good margin.

7 Improving Retrieval

For the graduate section of the project, I aimed at improving the standard IR system developed up to that point. I tried three ways in which the system could be improved using natural language processing and information retrieval techniques.

1. Consolidated search using category and content separately

2. Content filtering using Part of Speech

3. Re-ranking of Top 10 using Answer Extraction

7.1 Consolidated search using category and content separately

My first approach at improving the current IR system was to implement a different method of using category together with the question to form the final query which would be used to search for the correct documents. Here, I performed two separate searches on each Wikipedia page, once using the question and second using the category text from each question in `questions.txt`. Both searches were performed over the document content which did not have a separate $Category$ section but instead all the content was indexed within the same $Content$ section.

The Top 10 for each search was calculated and the result along with its score was saved in a HashMap. After both searches for each question was complete, the two HashMaps were concatenated in such a way that matching documents had their scores added in the final list of matches.

The accuracy of this system turned out to be 2 less than that accuracy of the standard IR system i.e. the P@1 score was 27%. This decrease in score was not intended and hence I moved on to the other implementation approach.

7.2 Content filtering using Part of Speech

Here, I tried to filter the document content by using Part of Speech tags. Just like how the query was shortened using relevant POS tags, I extracted the POS tags of all terms in the document content and filtered the words whose Part of Speech tag was relevant to Question Answering. This greatly shortened the index size as well.

The choice of important POS tags are the ones that provide more meaning to the content just as nouns, pronouns, verbs, adjectives, determiners, adverbs, among others. This shortened content was then lemmatized like before and then the stop words were removed as well. The list of POS tags included is:

```
["JJ", "NN", "NNS", "DT", "JJ",
"CD", "NNS", "CC", "JJR", "JJS",
"LS", "MD", "NNP", "NNPS", "PDT",
"RB", "RBR", "RBS", "UH", "VB",
"VBD", "VBG", "VBN", "VBP",
"VBZ"]
```

The function `lemmatize`, `partOfSpeech` and `removeStops` perform the lemmatization, POS filtering and Stop word removal respectively. So now both the query and content has undergone POS tag filtering. This system performed with the same accuracy as my standard IR system. I also calculated the P@10 score for this system and the accuracy are:

- P@1: 29
- P@10: 56

So, the P@1 accuracy did not change. On hindsight this does make sense because my method of POS filtering removes words such as conjunctions, Wh-determiners, interjections, etc. These are common words in the document which do not hold a lot of information to match the actual title of the document. All they did was decrease the size of the document but did not add any value to the scoring function since the actual matching and relevance is still mainly because of the unique nouns and pronouns which have been preserved in my approach.

I believe extending this method to implement Named Entity Recognition and using syntactic dependencies between words in a sentence might help improve the performance.

7.3 Re-ranking of Top 10 using Answer Extraction

In this approach, I referred to Mihai's paper on Design of a Factoid Question Answering System(1) which mentions how candidate answers can be ranked based on the properties of the context where they appear in the retrieved passages i.e. the answer context. I used this approach to re-rank the Top 10 results from my standard IR system. I used the following four heuristics among the seven that the paper describes.

1. **(H0)** *Same word sequence* - computes the number of words that are recognized in the same order in the answer context;

2. **(H1)** *Same sentence* - the number of question words in the same sentence as the candidate answer;

3. **(H2)** *Matched keywords* - the number of question words found in the answer context;

4. **(H3)** *Answer span* - the largest distance (in words) between two question keywords in the given context.

I chose these four heuristics because they felt the most relevant to my system among all the seven heuristics. *Punctuation flag* and *Comma words* seemed less relevant for this system. The functions *calculateH0*, *calculateH1*, *calculateH2* and *calculateH3* calculate these above heuristics respectively. The final score was calculated as:

$$H0/ConCnt + H1 + H2 - 1/4\sqrt{H3}$$

Here, $ConCnt$ is the total number of answer contexts in a document.

I treated each document separately and calculated their score using the method described above. In our case, the answer context is the sentences in the Wikipedia page that contain the title of that page itself. The candidate answers are the top 10 titles and by using this scoring method, these titles are re-ranked into a new Top 10 where the correct title, if present in the old top 10, is hopefully bumped into the top of the list. The function `reRankScore` calls the above four functions and calculates the new score for each Wikipedia page.

The results showed that the accuracy did not change and remained the same as that of the standard IR system. This means that the re-ranking, while shuffled the top 10 around each other, did not propel the actual document to the top of the list.

This result can be partially because of the final scoring formula of the four heuristics. The weight for each heuristic has been selected on the basis of the paper and the same weights might not necessarily apply when only a subset of the heuristics are taken. A careful study into each heuristic along with which one would be most useful and relevant is required to get the weights correct.

8 Execution

The project code needs two index directories to run. You can download the index directories from this link.

It is important to place the downloaded directories in the same folder where `build.sbt` resides. From there, run the following command to start the program:

```
sbt 'runMain edu.arizona.cs.QueryEngine'
```

References

[1] Mihai Surdeanu and David Dominguez-Sal and Pere Comas, *Design and performance analysis of a factoid question answering system for spontaneous speech transcriptions*. INTERSPEECH, 2006.