11-791 HW3 Report
Nikolas Wolfe (nwolfe)

Introduction

In this paper we will analyze the results of an exercise in question-answering information retrieval. To start we will use a primitive method known as cosine similarity to compute a ranking for a set of candidate answers to a given question. After analyzing the errors of our initial approach, we will deduce a new strategy to improve performance and apply object oriented principles to extend the functionality of our system while leaving intact the core functions which do not change. Finally, we will implement our new strategy and report on our results.

Analysis of Errors in Retrieval Exercise

Using a simple space-delimiting tokenizer and cosine similarity as our metric, the example retrieval exercise returns only 1 correct answer out of 20, i.e. it was only able to rank the correct answer first in one instance. Analyzing the mistakes, we find some recurring issues:

1. **Answer Length**: Cosine similarity measures are sensitive to the length of a document. If a correct answer is too wordy, it may be ranked lower than it ideally should be. For instance, in answer to the question "*What has been the largest crowd to ever come see Michael Jordan?*", the correct answer from our document corpus is as follows:

   "When Michael Jordan--one of the greatest basketball player of all time--made what was expected to be his last trip to play in Atlanta last March, an NBA record 62,046 fans turned out to see him and the Bulls."

   Using a cosine similarity measure, the following was deemed a more likely answer:

   "*A supposedly last play of Michael Jordan gathered some of the largest crowd in history of NBA."*

   The correct answer was obscured by superfluous information. If the document had been shorter, it would have been ranked higher.

2. **Tokenization**: Splitting a document only on spaces and not stripping punctuation can lead to identical words being tokenized as different words because they are appended by punctuation mark(s). For example, given the question *"Give us the name of the volcano that destroyed the ancient city of Pompeii"*, we see that in every answer

the word *Pompeii* is appended by a punctuation mark, e.g. "*Pompeii.*" and *"Pompeii;"* An obvious problem here is that Pompeii is being tokenized as three different words, rather than just one.

3. **Stop Words**: Words like "the," "an," "a" and other commonly used words carry little semantic information. Counting these in our word vectors is clearly introducing unnecessary noise.

4. **Stemming and Morphology**: The most important aspects of improving cosine similarity are finding ways to remove unnecessary information and making sure similar words are counted as such. When words change due to English inflectional and derivational morphology, e.g. "*China's Sorrow*" vs "*Sorrow of China*", or "*Alaska*" vs "*Alaska's*", "*inform*" vs "*information"* it is helpful to be able to collapse these words into a single type. Verb conjugation can also affect classification, e.g. "*bit*" vs. "*bite*" vs "*biting*". Using stemming algorithms such as Porter Stemming may help to reduce this type of redundant dimensionality. Using the `StanfordLemmatizer` class (see next section) is also a good way to achieve some collapsing of words derived from common roots.

5. **Casing**: Another way to collapse similar words is to get rid of casing differences, which can also lead to identical words being counted as different types. For instance, in Question 9, we see that the word "*Moon*" is capitalized in the correct response whereas in the question it is spelled with a lowercase *M* as "*moon*". Unsurprisingly, in the top ranked (wrong) answer, the word "*moon*" is uncapitalized. Simply forcing everything to lower or upper case may help this problem.

System Architecture

In order to account for most of the issues identified above (and do a little better), we need to basically do two things:

1. Intelligently pre-process the text
2. Explore different similarity metrics

In order to allow for this, I have created some extensible functionality to allow for various experiments to be conducted and compared in a "plug and play" fashion. Essentially, the CollectionReader and Annotation steps in my pipeline will remain (mostly) the same. I also do not change or extend the existing type-system. To organize annotation data in my CAS Consumer, I have created two data types, a **Question** object (to house relevant data and statistics for a given question), and an **Answer** object. Question objects are straightforward and I will not elaborate much on them here. Answer objects are more complicated. They house relevant data members, and are furthermore responsible for knowing how to print a report about themselves and how to compare themselves to another Answer object.

Answer objects have no notion of cosine similarity. We abstract the notion of similarity to simply refer to a double value which represents an Answer's similarity to another Answer object. This allows for different similarity measures besides cosine similarity to still employ the use of Answer objects.

For utility's sake, I have also created two HashMap extensions which are useful for bag-of-words representations of data and accumulating Objects of a particular type. The first extension is called a **FrequencyCounter** and it is fashioned after a Python FrequencyCounter object which returns a frequency map of words given a string. My FrequencyCounter class acts as a **Decorator** over a HashMap parameterized as a mapping from Java Strings to Integers, and handles the counting of map keys such that whenever an item is added to the map, it is appended to the keyset or the count is updated. This class can be further "decorated" to allow for pre-processing of text such that a more intelligent frequency map can be produced, and that is precisely what I do to improve the performance of the cosine similarity measure.

The second HashMap extension is also a **Decorator** called **BetterMap**, which allows me to generically use a key to map to an ArrayList of values. The class is generic with respect to keys and values, but it does require that the value of any key Object be a parameterized ArrayList of the type of the value Object. In my implementation I make an integer mapping of question numbers to ArrayList of Questions and Answers. This is useful when I just want to collect items matching a particular type and not worry about collisions with existing keys.

Implementation Details

For the sake of allowing different similarity measures to be computed, I have created a Type called **Similarity** which requires implementing classes to adhere to a function:

```
public Double computeSimilarity(Question query, Answer ans);
```

This is a way to abstract the notion of similarity computation such that different strategies may be employed. The first and most obvious client of this Interface is the `CosineSimilarityStrategy` class, which computes the cosine similarity between a Question and and Answer. This is what was used in Part 1, and in order to compute Dice Coefficients and Jaccard Coefficients, I have simply implemented classes `DiceSimilarityStrategy` and `JaccardSimilarityStrategy` accordingly. As the name intentionally implies, this is an application of the **Strategy** pattern.

Keeping with this pattern, it is possible to combine and compare strategies which draw upon several existing Similarity implementations. I did this by creating one class called `DiceJaccardStrategy`, which draws upon both the Dice Coefficient and the Jaccard similarity measures and returns the harmonic mean of the two. I also created three final classes which try to combine all existing Strategy implementations by returning either the highest similarity measure, the average of all similarity measures, or a weighted sum of all the measures. The best results came from using a weighted sum of the similarity measures and

In order to pre-process the text for cosine similarity computations, I have subclassed the **FrequencyCounter** class in order to allow for word FrequencyCounters which process input before computing a bag-of-words vector. The first attempt in this direction was the creation of the `LemmatizeFrequencyCounter` and the `StemFrequencyCounter` subclasses, which do as their titles would suggest. They lemmatize and/or stem the words using the Stanford Lemmatizer before adding them to a BoW vector. This resulted in significant improvements in performance (see next section). A final elaboration on this was the `CleanStemFrequencyCounter` class which used a regular expression to remove all punctuation marks and whitespace characters prior to splitting and stemming all of the words in the documents. This resulted in the greatest improvement vis a vis pre-processing the text. Stop word removal was also tried in an ill-fated `CleanStemStopWordFrequencyCounter` subclass but this proved overall to be worse in terms of performance.

Due to the large number of implementing classes involved in this design, it begins to make sense to use Factories to generate the objects, so that we do not have to change code in a bunch of constructors every time we want to try a new algorithm implementation. The things that change the most in this case are the means of preprocessing the data, and then the strategy for similarity computation. I therefore use a `SimilarityFactory` and a `FrequencyCounterFactory` to defer the creation of the objects being used to external classes which are aware of the implementing classes and subclasses. This is an example of the Factory pattern.

The implementations I used for Jaccard and Dice similarity were culled from a variety of sources including Wikipedia and MIT open courseware. For the weighted sum I simply played with the weights of the various measures of similarity (Cosine, Dice, Jaccard) until I got the best results. These results are documented on the next page.

Results

Overall I implemented several different FrequencyCounters and Similarity measures. A comparison of their effect on MRR is given below. I simply altered the parameters of the Factories involved to get the different combinations of FrequencyCounters and Similarity strategies.

Pre-Processing	**Similarity**	**MRR**
Space Tokenization	Cosine	**0.438 -- BASELINE**
Stop Word Removal	Cosine	0.525
Lemmatization	Cosine	0.458
Stemming	Cosine	0.550
Stemming + Stop Words	Cosine	0.596
Stem + Remove Punctuation	Cosine	0.663
Stem + Remove Punctuation	Dice	0.591
Stem + Remove Punctuation	Jaccard	0.508
Space Tokenization	Weighted Sum	0.446
Stop Word Removal	Weighted Sum	0.516
Lemmatization	Weighted Sum	0.500
Stemming	Weighted Sum	0.629
Stemming + Stop Words	Weighted Sum	0.604
Stem + Remove Punctuation	Weighted Sum	**0.679 -- BEST**

Analysis

We see that the best results come from using a weighted combination of Cosine, Dice, and Jaccard coefficients, combined with a pre-processing algorithm that removes all punctuation marks and does word stemming. Ironically, however, the greatest gains are made from simply doing cosine similarity on pre-processed text. In almost every case, finding a way to reduce the dimensionality of the documents via pre-processing helped in the similarity measurements. Jaccard and Dice similarities are only reported for the Stem + Remove Punctuation case because this was the overall best result in both cases.

Regarding design, while this system is every extensible and allows for literally any kind of measurements or tokenization to be swapped in or out at any time (including runtime, with some minor modifications), it would be a big improvement if some of the features of UIMA aggregate analysis engines were brought to bear here. This solution relied mostly on the use of design patterns and built off of the basic UIMA pipeline which existed for the exercise in Part 1.