David Klaper (dklaper)

11-791 HW3 Report

Error Analysis

After running the baseline system I performed an analysis on the 19 examples that were not perfectly retrieved and split the kinds of errors in several categories:

Error Type	**Count**
Confounding Information	5
Synonym	5
Lemmatization/Tokenization Issue	6
Different Casing	2
PoS Morphing	1

Confounding Information means that the relevant gold standard answer also had additional unnecessary information that increased the length of the answer and thus reduced cosine similarity. For this error a sub-sentence analysis might be necessary and some of the errors are hard to correct. A different similarity measure might improve some of these errors.

Synonym means that the words in the query were replaced in the answer with synonyms or words that have a very similar meaning in the given context. Potentially a more sophisticated similarity model might catch this but probably only partly (the similarity measures I chose to not help this case though).

Lemmatization/Tokenization Issue reflects errors that remain because the current tokenization does not split punctuation, as well as errors with words in different forms, e.g. `die` and `died` should be the same in certain contexts. This also includes abbreviations that were not lemmatized to the correct form e.g. N.J. to New Jersey, as well as, stop words that boost the score of wrong answers. Most of these errors can be improved by preprocessing.

Different Casing is for cases where performing case-folding would improve the situation.

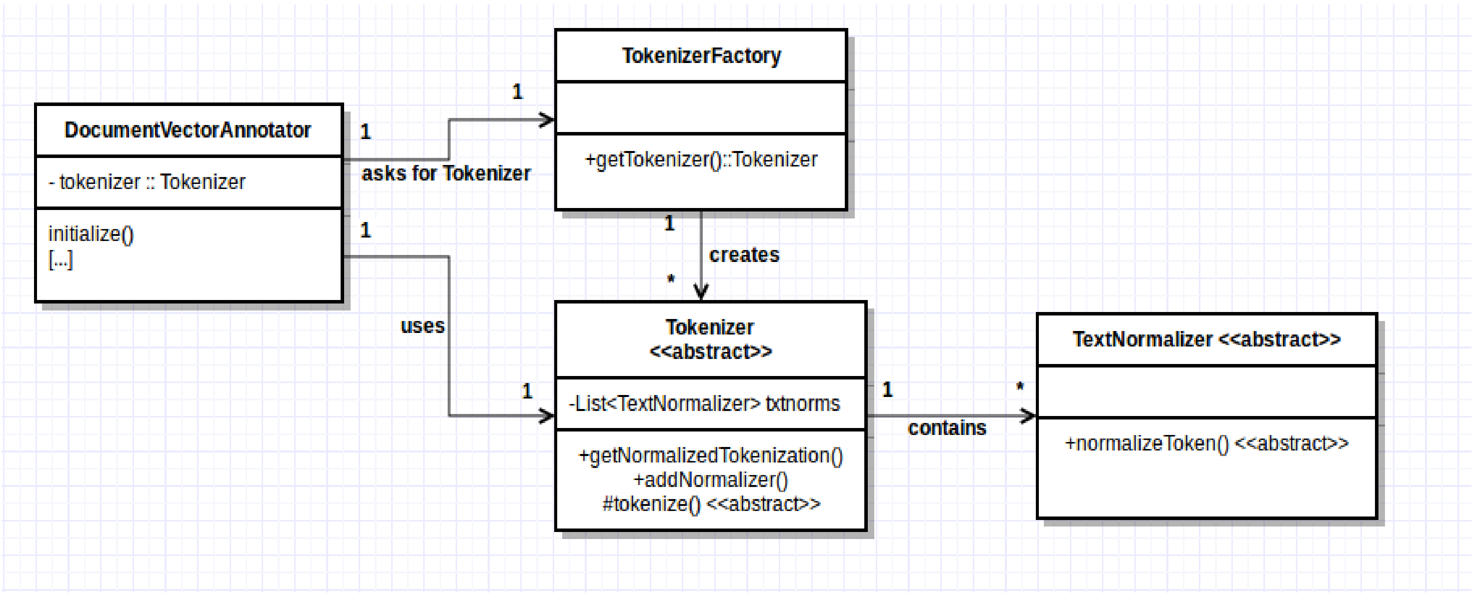
PoS Morphing means that a key word in the query was in a different Part of Speech than in the relevant answer. For instance, it could be the verb “*treat*” in the query and the noun “*treatment*” in the relevant answer. Lemmatization might not be enough for these cases.

System Architecture

This section explains the software engineering decisions and applied patterns for part 2 of the assignment. For text normalization, I used the factory pattern for creating text normalization objects that depending on the configuration parameters perform case folding, tokenization, and/or stemming.

David Klaper (dklaper)

For this I replaced my tokenization in the DocumentVectorAnnotator with a call to a Tokenizer object that is constructed in the Annotator's *initialize* method by calling the factory with the given UIMA parameter value. The TokenizerFactory simply generates the appropriate Tokenizer based on the classnames provided in the *getTokenizer* method. Furthermore, the Tokenizer object can have an arbitrary number of TextNormalizer objects (configured via a multi-valued UIMA parameter) that normalize the token after tokenization (e.g., case folding etc.). For the specializations of the Tokenizer objects I use the template method pattern, so they only implement the *tokenize* method. The subsequent normalization as part of the tokenizer is called by the abstract Tokenizer class in *getNormalizedTokens* (the method called by the annotator). Since I combined factory and template method patterns here I drew the class diagram to give a quick overview of the pattern explained above (concrete implementations of Tokenizer and TextNormalizer were left out for brevity, they just each implement the corresponding abstract method an are what is *actually* returned by the tokenizerFactory):



In the RetrievalEvaluator, I decided to create the SimiliarityMeasure interface for simplifying the implementation of additional similarity metrics (Jaccard, and Dice coefficient). I created classes that inherit this method. In the code I submitted for part 1 only minimal changes are thus necessary replacing the call to *computeCosineSimilarity* in the RetrievalEvaluator to *computeSimilarity* of the SimilarityMeasure interface. Additionally, I used the factory pattern to configure which similarity measure to use by simply specifying a UIMA parameter. So the SimilarityMeasureFactory produces the appropriate similarity based on the parameter. This is a simpler variant of above and I think you get the idea without an additional diagram.

No changes in the type system were necessary for my implementation, cause the original text is stored separate from the text of the token which could be normalized

David Klaper (dklaper)

Algorithms and Resources

The algorithms I use are straight derivations from the formula. For both the tokenization and stemming (only stemming no lemmatization to avoid PoS-tagging) I used the Stanford CoreNLP tools in the improved version. In addition to cosine similarity I implement the Dice similarity coefficient. This is defined in vector terms (of a binary vector) as:

$$DC(Q, D) = \frac{2*|Q \cdot D|}{(|Q|^2 + |D|^2)}$$

Since the Dice coefficient is not a proper metric (does not satisfy triangular equality (source: Wikipedia)), I decided to also implement the Jaccard similarity coefficient as defined below:

$$JS(Q, D) = \frac{|Q \cdot D|}{(|max(Q, D)|)}$$ where max() is a vector function, performing an element-wise `max`

Both measures use set operations at its heart. Thus, the vectors **must be binary** vectors indicating the presence or absence of a type not the frequency! This helps with some of the mistakes where a word is used multiple times in a wrong answer and the wrong answer gets boosted based on the word count.

Experiments

After implementing the metrics, I run the AAE with 5 separate configurations (simple changes in the descriptors, no code changes necessary!). The results of this run and the necessary significance tests are below. There would have been more combinations but without using something like the Configuration Space Exploration Framework it gets very cumbersome to run so many different runs and keep a good overview. All experiments are executed on the provided data.

Text Normalization	**Similarity Metric**	**MRR**	**p-value**
None	Cosine Similarity	0.4375	--------
None	Dice	0.4500	0.5770
None	Jaccard	0.4500	0.6663
Stanford Tokens	Cosine Similarity	*0.5125	0.0828
Stanford Tokens	Dice	0.4875	0.3299
Stanford Tokens	Jaccard	0.4626	0.3299
Stanford Tokens/Stemming/Case Folding	Cosine Similarity	**0.6125	0.0174
Stanford Tokens/Stemming/Case Folding	Dice	**0.6292	0.0235
Stanford Tokens/Stemming/Case Folding	Jaccard	*0.5958	0.0705

* $p <= 0.1$ ** $p <= 0.05$ as compared to the first row (baseline). Confidence value calculated by paired

David Klaper (dklaper)

t-test as implemented in apache commons math3 (http://commons.apache.org/proper/commons-math/apidocs/org/apache/commons/math3/stat/inference/TTest.html#pairedTTest%28double[],%20double[]%29)

So the performance difference between the measures is rather small and depends a lot on the text normalization. This is mainly because all measures I implemented are simple measures that don't exploit global features like IDF or context vectors from a reference corpus.

The text normalization however provides huge gains. Interestingly, just using tokenization without stemming and case folding, gives the biggest advantage to cosine similarity. It outperforms the baseline with 90% confidence. When adding stemming and case folding all methods improve further and all methods significantly outperform the baseline. This might be explained by the large number of tokenization and similar errors in the error analysis. Both Dice and Cosine Similarity outperform the baseline with no preprocessing with more than 95% confidence. This clearly shows how important the preprocessing steps are. My effective use of patterns for using different normalizations could aid in finding even better combinations!

For the full code of my implementation, look at branch task2 in my repository: https://github.com/DKlaper/HW3-dklaper/tree/task2