Report for Homework3: Xingyu Yan(xingyuy@andrew.cmu.edu)

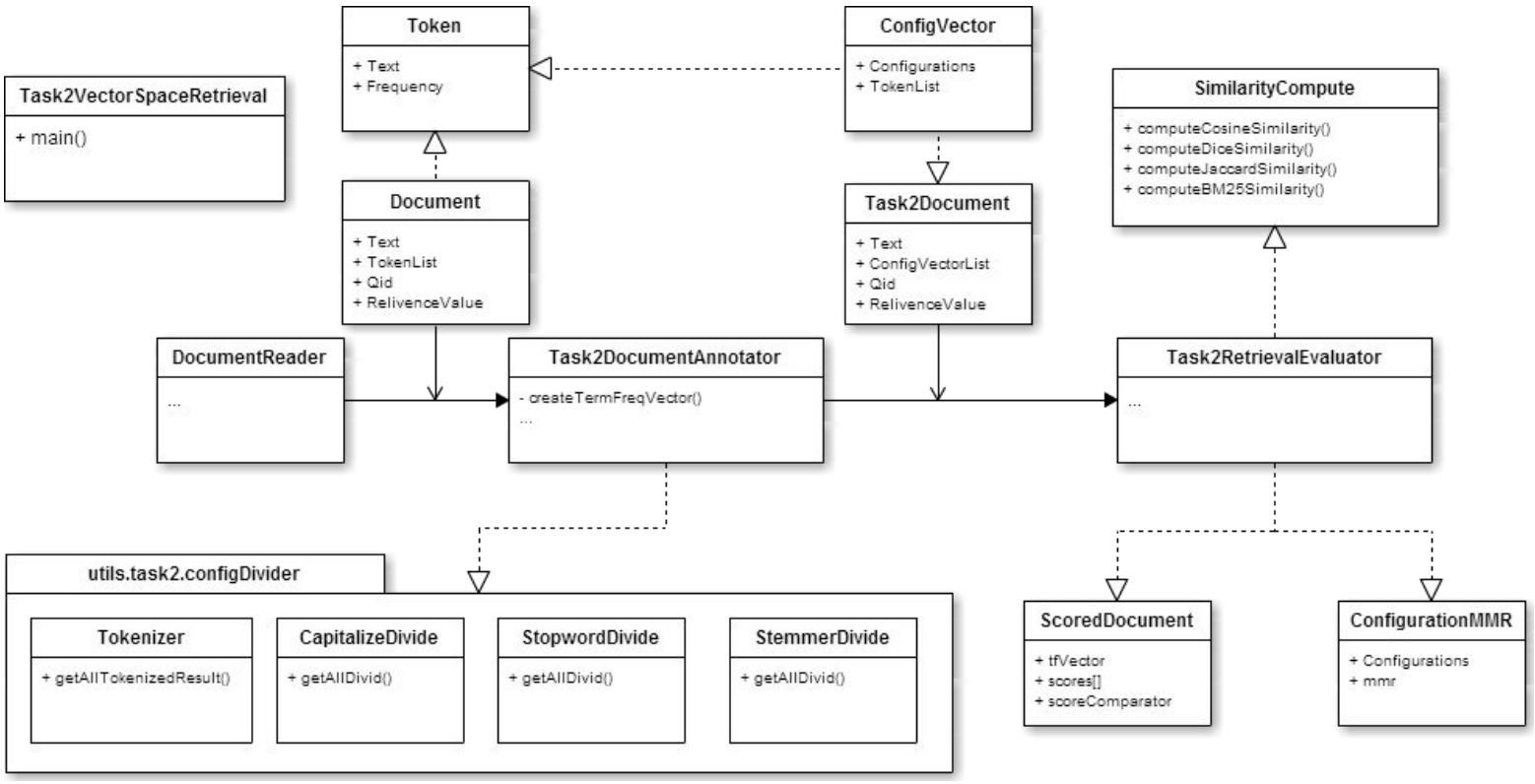
Note: As task1 don't really have so much space you can design, this report will mainly talk about my design and implementation of task2.

1) System design:

Objective: My system design is targeting to **automatic generate report** for every method that mentioned in the homework design, and find the best combination targeting this particular data-set. Although I know this design are still miles far from an elegant design, it now can provide enough help on debug and error analysis, and giving I have spent over 20 hours on this, I think I have to stop here.

Usage: Same way of triggering task1 (same path too) except set the mainClass = Task2VectorSpaceRetrieval.java. It's report is **task2_report.txt**, and I appended an excel file in the doc directory which is the mmr matrix for every method combination that extracted from report.

Current UML Diagram:



Brief Work Flow:

Using Task2VectorSpaceRetrieval as main class, the only different form the one provided in task 1 is different descriptor path.

I kept **DocumentReader**, and so as the original TypeSystem its using, but in **Task2DoocumentAnnotator**, instead of package the text in to one single tokenList, applying different tokenization setting, I generated a list of token list (which is essentially **Task2Document**). And in order to keep track of all their performance, I packed the configuration of how they are generated in to the type system, which is **ConfigVector**.

Those different configurations included:

a. Using naive tokenizer (white space), replacing some given symbols to space or even eliminate symbol and single characters(like change China's to China);
b. Do nothing or converting all characters to lowercase;
c. Do nothing or using stopword list to filter stopwords.
d. Do not stem, using given Stanford stemmer or using apache snowball stemmer.

All of these configurations are implemented in **utils.task2.configDivider**, which contained a series of tool class and provided 36 different output tokenList from one single document.

In the **Task2VectorSpaceRetrieval**, I reconstructed the data structure from the original grouped by document to a structure grouped by configuration so that it can calculate the MMR of each algorithm of each configuration. TF-IDF is the alternative term frequency so I implemented inside **Task2VectorSpaceRetrieval**, and packaged all the similarity algorithm to a util class named **SimilarityComput**.

In **SimilarityComput,** I implemented cosine similarity, Dice similariy, Jaccard similarity and BM25. And for the BM25 algorithm, I tried several times about changing both k1 and b in it but it doesn't make so much difference, and eventually I choose **k1=1.2** and **b = 0.75** as the final algorithm configuration for BM25.

And because all other algorithms required original term frequency, I only used calculated TF-IDF to calculate the naive cosine similarity, so that for each configured tokenized document, I'll generate 5 MMR so that its totally 180 MMR result. I stored them in **ConfigurationMMR** along with all their configurations including tokenization and similarity calculation.

At last, I sorted the **ConfigurationMMR** list and print out the top 10 configurations of this data-set on the console, and kept all the processing data in the report file.

2) Result Analysis:

In the result matrix, we can see that the best results are almost all coming from the naive cosine similarity, but in the overall tabs cosine similarity didn't have an substantial advance over other algorithms (only 0.5% over TFIDF, 1.5% over Dice), which in my interpretation we can not be sure that's the better way to do it due to the limited test sample size.

And as for tokenization methods, we can see that **filter symbols** got an noticeable improvement over the white space tokenizer (7%), and so is **using stemmer** over not using any stemmer(8%). **Filter stopwords** provided only 0.8% improvement on average, which is not quite satisfying, I guess maybe a better stopword list may help. **Unified lower case** contributed 3% on average.

The most interesting part is, the top 5 results are presenting a very promising pattern: they didn't care about case unify(false:true = 2:3), they are more likely to **avoid** using **stoplists** which advanced on average(false:true = 4:1), most of them are coming from **naive cosine similarity** (naive cos : tf-idf cos = 4:1), and most importantly, all of them are stemmed with **Stanford stemmer**. Looking like those Stanford libraries are really handy....

Again, the detail of the result are provided in an excel table in the doc directory. As for the results are not entirely what I expected, just hope that it didn't went wrong.

1) Inherited from HW1: Collection Reader, Manual AEE(where just called AAE in HW1), OutputConsumer and BenchmarkComsumer only made some little changes for new data structure.
2) Main AEE: Because I designed a aggregate analysis engine nesting structure, this is the main AAE to control the basic work flow of all analysis engines.
3) Lingpipe AE: Used Lingpipe as an answer system.
4) Abner AAE: I implemented Abner with 3 different configurations. Because tokenization of Abner is the most slow part and I got 2 AE needs tokenized result, I generated the tokenize part as a single AE, which significantly shortened the running time.
5) ResultCombine AE: I got 3 part of result, each part may have different idea about how to extract data. This AE is made for combine those extraction and provide a final result.

2) Outsource Using:
   a) Manual AE:
      i. ftp://ftp.ncbi.nih.gov/gene/DATA/GENE_INFO/All_Data.gene_info.gz Gene database;
      ii. Stanford NLP tool
   b) Abner AAE:
      i. Abner v1.5
   c) Lingpipe AE:
      i. Lingpipe v4.1.0, HMM model.

3) Problems when implementing:
   a) Abner's tokenize crises:

      Abner is a good tool of extracting gene name, but it have a really strange characteristics: when you are using it's tokenize tagger, the result will be tokenized as

well instead of being the original. Even though that's really not so common, it is disturbing the location extracting for the words, which it failed to provide as well.

In order to fix that, I coded a special class to change the recover the tokenized String to the original one, and search location/recover gene name at the same time.

b) .Combination problem:

Originally, my plan for the combination function is just vote: for the same area, 2 or more vote can be considered assured, and decide the shorter one be the final output if their length is not identical. But when I tested it, although it got a really high precision, it's recall is only about 1%, which is totally unacceptable. And when I looked at different combination, it's really embarrassing that Lingpipe did the best work on it's own. (If you want to see the result of the original combination, just config combination descriptor pointing to ResultCombineAnnotator_Original)

That's really not a good sign because in that case, any add-on will damage the output. So finally my method is just "avoid Lingpipe". When any other method overlapped with Lingpipe, I give it away and only add extra tag when lingpipe did not cover.