11-791 HW 4 Report

Chenyan Xiong
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
cx@cs.cmu.edu

1 Design and Implementations

In this homework, the requirements are to implement a vector space model in QA system. The vector space model is to express both the query sentence and answer sentence in a vector of tokens (could be bigram, or other features, but here it is just tokens). And then we are asked to use cosine similarity to rank the answers based on their textual similarity. This method could be called the 101 method of QA system, as it is very straightforward, easy to implement, and efficient. It should always be perhaps the first thing to try when facing a such problem.

The type system and architecture is designed and given. And they are sufficient to accomplish this homework. The places we need to implement are marked with to do, and mostly at function level, like tokenization and calculating the cosine similarity.

1.1 General Architecture

The general architecture is given, and not much change is done by me. There are mainly three components in the system:

1. DocumentReader: read document from raw text. The implementation given satisfies my needs.
2. DocumentVectorAnnotator: tokenize the given documents, and annotate them by TokenList.
3. RetrievalEvaluator: the main class in this task. Functions include cosine similarity calculation, rank assignment and MMR evaluation, together with final output.

In the rest part of this section, I will introduce the design and implementation of mine.

1.2 DocumentVectorAnnotator

This class will perform the following operation, in the function createTermFreqVector():

1. tokenize the sentence using StanforCoreNLP tokenizer.
2. for each sentence, add the tokenizes to a HashMap, as the vector space model for it.

1.3 RetrievalEvaluator

design ideas: Following the suggestion given by the design of qIdList and relList, in this class, all information is stored in ArrayLists, including: raw texts, vector models, scores and ranks. Each element in these ArrayLists stores the information of the corresponding document, and the alignment between these ArrayLists are completed by keeping their order all the same with input documents' order.

Thus all operations are done as filling the correct ArrayLists with right information. The pipeline is as follows:

1. PreProcessing: store list of query id, relevance score and raw string of all documents and build vector space model for them. Done in function processCas().
2. AssignScore(): Assign cosine score to answers. Input data: hVectorList(Vector models), relList(relevance score to check whether it is a query). Will calculate the cosine similarity between query and answer's vector space model, and output to scoreList.
3. CalcRank(): Calculate rank based on the scoreList generated by AssignScore(). It will first collect the scores for a qid, sort them ,and build a mapping from qid+score to rank (hScoreRank). After that, it will go through the scoreList again, and assign ranks based on built mapping.
4. compute_mrr(): compute MRR using vRank and relList, as described in assignment.

For more details in implementation, please check the Java docs.

2 Error Analysis and Possible Improvements

The very basic vector space model with cosine similarity on given 3 questions provide MRR of 0.833. If the vector space model is case sensitive, the MRR is actually 1. This is very counter-intuitive because usually vector space model cannot work so well, and the case information is not useful in lots of similar tasks. This observation shows that error analysis is hard in this task as too few training data is provided. It is hard to derive safe conclusions based on such small size of data. It is possible that the gain of one method over the other one is just by random, like the case sensitive vector space model versus insensitive vector space model. And a highly tuned system may result in over-fitting. Also, I expect that the difference of various methods cannot pass statistic significant test in this task.

But nevertheless, there are several possible direction to try in future work:

1. stop word removal. a widely used technique in NLP and IR.
2. use pos-tag as features, assign different weight to different matches based on their pos-tag. weights could be trained using training data.
3. use entity features. Named entity could be more important in matching.
4. ngrams. unigram loses order information, include ngrams of higher orders could help
5. dependency parsing: the parsing tree also contains information of sentence structures. It is possible that there exists several common patterns for correct answers in their parsing tree. Adding features in dependency parsing, either features from the tree structure, or different weighting for terms at different tree level could help.
6. different models/approaches for different kind of questions: one could build a question type recognizer to detect the question type, and treat questions of different types differently.