Programming Assignment 3 - Inference Network Report

Submitted By: Ankita Mehta

November 10 , 2017

1 Inference Network

Solution:

1.1 Implementing structured query operators and belief operators

After creating compressed, uncompressed indices and Retrieval models (in the previous assignments) and next task is to implement inference network on a small set of documents. This has been done by implementing structured query operators and various belief functions as the scoring functions. One wrapper has been written to implement proximity expressions which can be either of term — unordered_window — ordered_window — boolean_and.
Following are the command line arguments :

1. ***-q or −−query_path*** : It is the mandatory argument, which specifies path to query file.
2. ***-m or −−manifest_file_path*** : It is also a mandatory argument, which specifies path to manifest file generated while creating indexes. This file will define whether Retrieval has to be done for compressed or uncompressed file.
3. ***-pe or −−prox_exp***: It specifies which proximity expression has to be run : term — unordered_window — ordered_window — boolean_and.

Just the codes have been written for Filters and Belief operators as running them would require query parsing (which was not supposed to be done in this assignment).

1.2 Design Tradeoffs:

1. In all the proximity expression models, a copy of Inverted lists of query words have been made for calculating the doc score. This has been done because, rather than taking pointers for every list and maintaining them, the temp set of inverted lists has been updated on the go.
2. Also, the document scores have been stored as negative numbers. Because in Python, in priority queue the element with minimum value comes first. So, to retrieve the document with maximum score at first, document scores have to be stored as negative integers.
3. In belief operator, rather than passing a single document, a list of document-score dictionary has been passed.
4. Filters have been implemented in the modular fashion and multiple dispatch has been used for evaluating its second argument. This has been done because of easy implementation and lack of time.
5. Second argument while implementing filter is the combination of any belief operator and proximity expression should be passed in the following format. It should be a structure/class with data members as : type (denotes the type of node : a belief_operator / ordered_window /unordered_window / term(s)) , children: which is of same structure as the query node and these can be a belief_operator / ordered_window /unordered_window / term(s), value : which will store the name of belief_operator or the query. This design has been chosen as compared to the tree format because of it is easy to understand and implement.

1.3 Various Questions and how they were approached:

Various difficulties that were faced while doing the project and I figured out these problems after having discussion with the professor.

1. The very fist question that came when I started working on this assignment was what all things to be tested and run ?
2. What should be the input to the belief operator : is it the query or list of documents or document ? and what is it returning ?
3. How does filters work and what documents does it score ?
4. How to perform dirichlet smoothening on the documents returned by ordered and uncordered window, like what should be the value of $|C|, |D|$?

2 Judgments.txt

Solution:

1. hamlet:2.0 3
2. two_gentlemen_of_verona:3.3 3
3. othello:3.1 3
4. henry_vi_part_2:1.0 3
5. othello:4.0 3
6. othello:4.1 3
7. titus_andronicus:2.1 3
8. henry_viii:2.0 3
9. twelfth_night:3.1 3
10. romeo_and_juliet:3.4 3
11. twelfth_night:1.1 3
12. two_gentlemen_of_verona:3.3 3
13. winters_tale:3.2 3
14. troilus_and_cressida:0.1 3
15. othello:3.0 3
16. cymbeline:1.0 3
17. titus_andronicus:2.1
18. much_ado_about_nothing:4.1 3
19. as_you_like_it:1.3 3
20. alls_well_that_ends_well:2.4 3
21. troilus_and_cressida:3.1 3
22. hamlet:4.0 3

Some documents are repeated because more than one query is occurring in that document. All the documents are given the score 3, because using the window either ordered or unordered, it returned only those documents having that query. So these documents are given high relevance.

For the given dataset and the query terms, no difference can be seen in the documents they retrieved. This can be because of the window used for both the operators. Window used for unordered is $|Q|$ and distance used is 1 for ordered operator. This means both the operators search for the exact phrase in the database and since our corpus doesn't have phrases with jumbled words, so both the query operators retrieved the same results. But there is difference in both the window operators in a way that both window operators will maintain the fact that all the query terms should occur in a document but ordered window will also maintain the sequence of all the query terms, whereas unordered will/will not maintain the sequence of query terms.
Also, there are no results retrieved by both the query operators for queries 1 to 5 and 10, since there are no exact or jumbled phrases present in the corpus.

3 Implement a structured query language

Solution:

A query is a sequence of words and to implement the structured query language for our retrieval system would require following steps to be taken :

1. Firstly, we would need to parse the query and store the query in the structure defined above. At first we would need to find what do we need to implement either filter require or filter reject.
2. Then, we would need to extract its two arguments i.e. proximity expression and other is the combination of belief operators.
3. After extracting the two arguments, we will evaluate each of them. 1st argument should be either of the proximity expressions, find the documents and their scores.
4. Then recursively implement the second argument by saving it into a specific structure as defined above. Evaluate the second argument and retrieve the document and their scores.
5. Then, for filter require, for the documents get from 1st argument retrieve their scores from the second argument. And for filter reject, reject the documents retrieved from the 1st argument.

4 Combination Functions and impact of normalization on and and weighted and operators ?

Solution:

Individual behavior:

1. **Not:** Not belief operator inverts the ranking of documents. The documents with the higher score will be given lower score and the documents with lower score will be given higher value.
2. **Weighted And:** This belief operator gives higher weights to the query terms which matters more than the other query terms. So while retrieving, those documents will be given more preference which has the higher weighted query term.
3. **OR:** This operator gives score to those documents which has atleast one of the query term occurring in them.

Implementing the combination functions in concert can refine the process of document retrieval for any query. But it could be possible that as we increase their combination, it could decrease the number of documents it will retrieve.
For example if we want to retrieve the documents which has some more weighted terms and some terms should not be present, Then in that case we can combine weighted and Not operator. In concert, these operators can prove to be very effective.

Normalising the scores on **and** and **Weighted and** can result in faster implementation of retrieval model. It will also help to implement combination of retrieval models, if we normalise them. I.e if we want to use one belief operator with one retrieval model and other belief operator with some other retrieval

model. Then in that case, we can apply normalisation on the scores before combining them.
However, combining the scores wouldn't create so much impact if we are using only one type of retrieval model. Since it will just scale everything between 0 and 1. Computation can be bit cheaper sometimes, but the documents will still remain rank equivalent.