600.466 - ASSIGNMENT 2

Vector Models for Information Retrieval

(building a search engine)

Submission Date: Thursday, March 14, 2013

Overview

The goal of this assignment will be to build a vector-based IR engine, similar in underlying design to the Salton/Cornell SMART system. Some of the basic infrastructure for this project has been provided. You will explore the effects of several permutations and implement optional extensions to the model.

The data used for this assignment will be the standard CACM abstract collection, consisting of 3204 computer science journal abstracts from the years 1958-1979. In addition, you will have access to 33 queries for this collection and associated relevance judgements for training and development purposes. Additional queries and relevance judgements have been withheld as unseen test data. Although this collection has several weaknesses, its relatively small size and familiar subject area make it reasonably well suited for rapid experimentation and model development.

The suggested programming language for this assignment is Perl 5.0, useful for rapid prototyping and flexible text analysis.

Infrastructure

You have been provided with the following support programs and retrieval engine shells. The original input data to these programs are in the directory `$CS466/hw2`, in the files `cacm.raw`, `query.raw`, and `query.rels` (a list of the documents judged to be relevant to each query).

Corpus processing tools:

- `tokenize.lc` will tokenize the text and convert to uniform lower case.
- `nstemmer`, based on the Porter stemmer
- `common_words`, a suggest stoplist
- `make_hist.prl` calculates corpus frequencies and document frequencies for use in term weighting. The use of these programs is described in the `README` file.

Core functions:

You have also been given a program `vector1.prl` that forms the core of a vector-based IR model. You may freely modify and augment this set of example functions. Details are provided in comments within the program.

Part 1 - Implementation of key sections of the retrieval model

You will make several additions/modifications to the program `vector1.prl` to implement key components of the vector retrieval model. Additional detail is provided in this program.

You need only implement the functionality necessary to support the experiments in Part 2.

In particular, you will need to implement the term weighting methods and similarity measures outlined in Part 2, the precision-recall calculations (methods discussed in class) and functions for system performance feedback. Specifications and suggestions for implementation will be provided in the example program `vector1.prl`. For your reference:

$$\begin{aligned}
Cosine\_Sim(Doc_i, Doc_j) &= \frac{\sum_{t=1}^{T}(wt_{i,t} \cdot wt_{j,t})}{\sqrt{\sum_{t=1}^{T}(wt_{i,t})^2 \cdot \sum_{t=1}^{T}(wt_{j,t})^2}} \\
Dice\_Sim(Doc_i, Doc_j) &= \frac{2\ [\sum_{t=1}^{T}(wt_{i,t} \cdot wt_{j,t})]}{\sum_{t=1}^{T} wt_{i,t} + \sum_{t=1}^{T} wt_{j,t}} \\
Jaccard\_Sim(Doc_i, Doc_j) &= \frac{\sum_{t=1}^{T}(wt_{i,t} \cdot wt_{j,t})}{\sum_{t=1}^{T} wt_{i,t} + \sum_{t=1}^{T} wt_{j,t} - \sum_{t=1}^{T}(wt_{i,t} \cdot wt_{j,t})} \\
Overlap\_Sim(Doc_i, Doc_j) &= \frac{\sum_{t=1}^{T}(wt_{i,t} \cdot wt_{j,t})}{min(\sum_{t=1}^{T} wt_{i,t}\ ,\ \ \sum_{t=1}^{T} wt_{j,t})}
\end{aligned}$$

where T is the total number of terms, Doc_i and Doc_j are two vectors, and $wt_{i,t}$ is the weight of term t in document vector i.

Implementation of term vectors in Perl

Perl's associative arrays are a reasonable data structure for implementing term vectors given their inherent sparsity. Although vectors conceptually have a length equal to the vocabulary size (12,000 terms in this domain), on average only 10-200 of the terms have non-zero values in any given vector. Term weights in a vector (`$vec1{$term}`) are retrieved efficiently through hashing, with non-present terms returning a weight of 0.

The Perl construct "`foreach $term (keys (%vec1))`" may be used to enumerate the terms present in a vector, and the construct "`while (($term,$weight1) = each %vec1)`" will jointly enumerate the present terms and weights. Examples of the use of associative arrays in implementing the cosine similarity function are as follows:

```
=============================================
sub cosine_sim_a {

    my $vec1 = shift; my $vec2 = shift;

    my $num     = 0; my $sum_sq1 = 0; my $sum_sq2 = 0;
    my @val1 = values %{ $vec1 };
    my @val2 = values %{ $vec2 };

    # determine shortest length vector. This should speed
    # things up if one vector is considerable longer than
    # the other (i.e. query vector to document vector).

    if ((scalar @val1) > (scalar @val2)) {
        my $tmp  = $vec1;
           $vec1 = $vec2;
           $vec2 = $tmp;
    }

    # calculate the cross product

    while (my ($key, $val) = each %{ $vec1 }) {
        $num += $val * ($$vec2{ $key } || 0);
    }

    # calculate the sum of squares

    foreach my $term (@val1) { $sum_sq1 += $term * $term; }
    foreach my $term (@val2) { $sum_sq2 += $term * $term; }

    return ( $num / sqrt( $sum_sq1 * $sum_sq2 ));
}
=============================================

=============================================
sub cosine_sim_b {

    my $vec1 = shift;
    my $vec2 = shift;
    my $sum_sq1 = shift;
    my $sum_sq2 = shift;
    my $num     = 0;

    while (my ($key, $val) = each %{ $vec1 }) {
        $num += $val * $$vec2{ $key };
    }

    return ( $num / sqrt( $sum_sq1 * $sum_sq2 ));
}
=============================================
```

Perl 5 supports arrays of vectors, such as "`vecs[$docn]{$term}`".
Suggestion: the computation of cosine similarity can be made more efficient by precomputing and storing the sum of the squares of the term weights for each vector, as these are constants across vector similarity comparisons.

Implementation of recall/precision calculation

When comparing various permutations of the vector models, it is useful to compare precision at different fixed levels of recall, and also to compute a single measure of mean precision at different levels of recall. The methods used for computing precision and recall on small relevant sets will be discussed in class. For standard comparison of results, we will use the mean precision at 3 fixed levels of recall, averaged over 10 levels of recall, and two normalized measures:

$$\begin{aligned}
Prec_{mean1} &= \frac{Prec_{0.25} + Prec_{0.50} + Prec_{0.75}}{3} \\
Prec_{mean2} &= \frac{1}{10}\sum_{i=1}^{10} Prec_{(Rec=\frac{i}{10})} \\
Recall_{norm} &= 1 - \frac{\sum_{i=1}^{Rel} Rank_i - \sum_{i=1}^{Rel} i}{Rel\ \ (N - Rel)} \\
Prec_{norm} &= 1 - \frac{\sum_{i=1}^{Rel} logRank_i - \sum_{i=1}^{Rel} logi}{logN!/(N - Rel)!(Rel)!}
\end{aligned}$$

where Rel is the total number of relevant documents for the query, N is the total number of documents in the collection, and $Rank_i$ is the position of the ith *relevant* document in a rank ordered list of all documents sorted by their expected relevance to the query. These last two measures quantify the deviation from the optimal rank ordering where the relevant documents are the top Rel entries in the retrieved list.

Note that the formula for $Prec_{norm}$ (normalized precision) has several large factorials. A reasonable method of handling these is to use the approximation that $log(n!) = nlogn$ and compute the denominator $log(N!/((N-Rel)!(Rel)!))$ as $logN! - (log(N-Rel)! + log(Rel)!)$ which can be approximated as $NlogN - (N - Rel)log(N - Rel) - (Rel)log(Rel)$. Because the denominator is a constant across all different permutations of the algorithm (e.g. term weighting schemes, similarity measures, etc.), computing its value exactly is not necessary for a useful comparison of performance across methods.

Part 2 - Experiments in modifying parameters of the vector model

The goal of this part of the assignment will be to achieve both a quantitative measure and an intuitive feel of the effects of changing certain model parameters.

For each permutations of model parameters described below, compute the precision/recall measures described above, averaged over all 33 queries. It is useful to represent the output in tabular form:

Permutation Name	$P_{0.25}$	$P_{0.50}$	$P_{0.75}$	$P_{1.00}$	P_{mean1}	P_{mean2}	P_{norm}	R_{norm}
Raw TF weighting								
TF IDF weighting								
Boolean weighting								
Cosine similarity								
Dice similarity								
...								

1. Term weighting permutations:
   (a) Raw TF weighting ($wt_{t,d}$ = raw frequency of term t in document d).
   (b) * TF IDF weighting ($wt_{t,d} = TF_{t,d} \cdot log(\frac{N}{DF_t})$)
   (c) Boolean weighting ($wt_{t,d} = 1$ if term t present in doc d, 0 if absent)
2. Similarity measures:
   (a) * Cosine similarity
   (b) Dice, Jaccard or Overlap similarity (choose one)
3. Stemming
   (a) Use raw, unstemmed tokens (all converted to lower case)
   (b) * Use tokens stemmed by the Porter stemmer
4. Stopwords
   (a) * Exclude stopwords from term vectors
   (b) Include all tokens, including punctuation
5. Region weighting
   (a) Weight titles, keywords, author list and abstract words equally
   (b) * Use relative weights of titles=3x, keywords=4x, author list=3x, abstract=1x.
   (c) Use relative weights of titles=1x, keywords=1x, author list=1x, abstract=4x.

When comparing permutations along one dimension (such as similarity measures), use the starred (*) methods as the defaults for the other dimensions. Thus there are 8 unique combinations of methods (lines 1b, 2a, 3b, 4a and 5b are all the same, i.e. the default performance).

In addition, for a qualitative feel of parameter effects, compute and submit the following output for combinations 3a and 3b above. You are strongly encouraged to examine results for other permutations and other queries on your own.

1. List the top 20 retrieved documents for Queries 6, 9 and 22 by their number, title and similarity measure, with the "relevant" documents starred.

2. For the top 10 retrieved documents, show the terms on which the retrieval was based (those with non-zero weights for both query and retrieved document) along with these weights.

   It is interesting to see how little information actually contributes to identifying document relevance.

3. List the top 20 documents that are most similar to Documents 239, 1236 and 2740, giving number, title and similarity measure.

Part 3 - Extensions to the retrieval model

Implement extensions to the retrieval model selected from the set below, preferably original ones that improve performance over that observed in the given permutations above. Enter the results in precision/recall table above, labelled as permutations 6 (and 7 if necessary). Since different extensions have different complexity and expected time cost, rough estimates for this complexity are given below. Your total complexity score must be at least 3.

Note: If your total complexity score is 4 or greater you may significantly shorten Part 2 by only providing evaluation data for 2 permutations (the default (starred) permutation, and a second contrastive permutation 1a,2b,3a,4b,5c) plus results for any option you implement below, and you can skip the implementation of P_{norm} and R_{norm}.

If an extension does not have an obvious empirical evaluation, you do not need to provide one.

1. Implement a simple relevance feedback method, computing a new query centroid vector from a weighted linear combination of the original query, the relevant-labelled documents and the (negatively weighted) irrelevant-labelled documents. Note that the person giving the relevance feedback (you) may have different standard of relevance than the original query formulator.

   You can do an automatic (albeit biased) test of your performance by using the true relevance judgements (given in `query.rels`) on the top 20 documents initially ranked by your system as if these judgements were coming from the user. This feedback can then be used as above to re-reank all the data in the 2nd round. **[complexity: 2]**

2. Create a new corpus of documents from any document set of your choice. This can be web pages, news stories, email, files in your home directory, literally any collection of things that you may wish to search. The collection should have at least 100 documents, and preferably more than 200. You should provide at least 30 plausible queries on this document set and provide relevance judgements for these queries. **[complexity: 2]**

3. An improved term weighting strategy of your choice, possibly including a more refined weighting by region of document and negative term weighting for regions of

queries specifying subjects/sub-areas that should *not* be included, such as in query 18. **[complexity: 1]**

4. Augment the term set to include all *bigrams* in the document/query that do not contain stopwords. For example "*window managers and command interpreters*" would add the bigrams *window+managers* and *command+interpreters* to the term set, but not *managers+and*. This will help make searches sensitive to word order rather than unordered, bag-of-words co-occurrence.

   The associative-array representation of vectors should make this easy to implement. You may use `make_hist.prl` to compute overall document frequencies by giving it a stream of bigrams (easy to generate). **[complexity: 2]**

5. Redefine the term set to include all 5-character n-grams in the text (including spaces (as `"_"`) and hyphens, excluding other punctuation), and including padded spaces at the beginning and end of queries. For example, a query (or document) with the string *window managers* would contain the terms `"_wind"`, `"windo"`, `"indow"`, `"ndow_"`, `"dow_m"`, `"ow_man"`, etc. This technique tends to be robust in the face of OCR errors and other noise, and effectively captures some multi-word terms. You may remove stopwords first at your discretion, but please state if you do so. **[complexity: 2]**

6. Implement query expansion through the use of thesaurus classes. **[complexity: 1.5]**

7. Cluster the top k returned documents (or those above similarity s) using the basic greedy Salton clustering method described in class. List documents in each cluster separately, with a line separating the clusters and ordered by similarity within clusters. The number of clusters you choose can be based on any reasonable criteria. Label each cluster with the most salient terms found in each cluster (for example, high frequency and high TF-IDF terms in the cluster). This would be useful when a query such as "windows" returns clearly different partitions of documents on (Microsoft) windows and (glass) windows. **[complexity: 2]**

8. Reduce the dimensionality of the term vectors using SVD. (Only for the very ambitious). **[complexity: 4]**

9. Modify your program to accept queries directly from the keyboard, rather than requiring pre-computation of a query corpus. **[complexity: 2]**

10. Implement (or re-implement) your full vector IR program in C, C++ or Java. **[complexity: 3-4, depending on scope]**

11. Create a basic web interface to your search engine. You should read in a query from an HTML form, call a modified vector.pl program as a CGI script, and generate output as HTML, with simple HTML links to the documents (you can use `"#"` offsets into the full abstract file, no need to split into separate document files). It is not necessary to implement persistent state for your program or any special loading efficiency, although implementing some of the large data structures as rapid-load DBM files would help speed performance. **[complexity: 1.5 or 2 (with DBM)]**

12. Create the web interface as described above, but implement your search engine as a persistent server that does not need to be invoked for each new query. You should support multiple socket connections and a mechanism for session tracing. Ideally you should also support feedback buttons for "more like this" relevance feedback. This should effectively act as a full web-based search engine. This option should only be chosen for those who have the background to implement such socket connections, etc. on their own. [**complexity: 2 or 2.5 (with relevance feedback interface)**]

When appropriate, your two extensions will also be evaluated using a second set of held-out queries. Thus methods should not be tailored specifically to idiosyncrasies in the given query set. The test queries and relevance judgments are from the same source as the training material, however, and have similar properties.

Evaluation:

Submissions will be evaluated as follows:

- 30% - Part 1: Implementation of the vector-based retrieval engine
- 15% - Part 2: Exploration of permutations
- 50% - Part 3: Quality and creativity of your extensions to the model
- 5% - Part 3: Performance of your extensions on held-out test data

What to Turn in:

Please hand in your assignment **both** in hard copy and electronically. For hard copy submissions, please provide printouts of your code, commented sufficiently so that I can understand what you did solely from reading the comments. It is not necessary to include printouts of any large secondary data files, such as new class members.

NOTE: *It is essential that the first page of your hard-copy submission contain a clear and very short (2-5 line) description of both of the Part 3 extensions you implemented, and also any non-standard approaches you used or questionable assumptions you made.*

To simplify the electronic submission of code and data files, please create a single tar file and mail that. For example, you are encouraged to place all of your code/data in a subdirectory called `HW2/src`. Before creating a tar of this directory, remove all unessential files in the `src` subdirectory and issue the following command from the `HW2` directory:

```
tar -cvf - src | gzip > jdoe.hw2.tgz
```

where `jdoe` is replaced with your username.

This tar file should be submitted electronically via the web interface found at `http://www.ugrad.cs.jhu.edu/cgi-bin/cgiwrap/cs466/submit.pl`.