COL762 Information Retrieval Assignment 2

Pratyush Pandey (2017EE10938)

2^{nd} November, 2020

Contents

1 Part 1 - Probabilistic Retrieval Query expansion

We start by defining the following terms

1.1 Collection Frequency

Terms that occur in only a few documents are often more valuable than ones that occur in many. Collection frequency weights (also known as inverse document frequency weights) are defined as follows, for term t(i):
Given
`n = the number of documents term t(i) occurs in`
`N = the number of documents in the collection`
the Collection Frequency Weight for a term is then
`CFW (i) = log N - log n`

1.2 Term Frequency

The second source of weighting is a term's within-document frequency: the more often a term occurs in a document, the more likely it is to be important for that document. Thus while a term's collection frequency is the same for any document, its document frequency varies. The term frequency for term t(i) in document d(j) is:
`TF (i,j) = the number of occurrences of term t(i) in document d(j)`

1.3 Document length

The third input to weighting is the length of a document. A term that occurs the same number of times in a short document and in a long one is likely to be more valuable for the former. We therefore have the length of a document d(j) thus:
`DL (j) = the total of term occurrences in document d(j)`
The use of document length described below actually normalizes the measure by the length of an average document:
`NDL (j) = (DL (j)) / (Average DL for all documents)`
This has the advantage that the units in which DL is counted do not matter much. A very simple measure such as number of characters in d(j) can be quite adequate as a substitute for number of term occurrences.

1.4 Combining the evidence

The three kinds of data for each term need to be combined together and with those for other terms from the request, to give a matching score for the particular document against the request. For our case, we use the one for BM25:
For one term t(i) and one document d(j), the Combined Weight is
`CW (i,j) = [ CFW (i) * TF (i,j) * (K1+1) ] /[ K1 * ( (1-b) + (b * (NDL (j)) ) ) + TF (i,j) ]`
K1 and b are tuning constants (see below). The formula ensures that (a) the effect of term frequency is not too strong (doubling TF does not double the weight), and (b) for a term occurring once in a document of average length, the weight is just CFW.
The overall score for a document d(j) is simply the sum of the weights of the query terms present in the document. Documents are ranked in descending order of their scores, for presen- tation to the user.

1.5 Parameter Tuning for BM25 Model

The tuning constant K1 modifies the extent of the influence of term frequency. Ideally, it should be set after systematic trials on the particular collection of documents. (Higher values would increase the influence of TF; K1=0 eliminates the influence altogether. The value K1=2 was found to be effective; it is probably a safe value to start on.
The constant b, which ranges between 0 and 1, modifies the effect of document length.1 If b=1 the assumption is that documents are long simply be- cause they are repetitive, while if b=0 the assumption is that they are long because they are multitopic. Thus setting b towards 1, e.g. b=.75, will reduce the effect of term frequency on the ground that it is primarily attributable to verbosity. If b=0 there is no length adjustment effect, so greater length counts for more, on the assumption that it is not predominantly attributable to verbosity. We have found (in TREC) that setting b=.75 is helpful.

1.6 Relevance weights

The basis for relevance weighting is simply the relation between the relevant and non-relevant document distributions for a search term modulated by its collection frequency, since a term could be in relevant documents just because it is in a lot of documents (collection frequency weights are indeed a special case of relevance weights). It is also necessary to allow for the fact that one is predicting matching behaviour on rather little relevance data, so one should in particular not assume that because a term has not been in any relevant documents so far, it never will be.

Relevance weights are therefore estimates, which accounts for the 0.5s in the formula below.
Given, for term t(i),
`r = the number of known relevant documents term t(i) occurs in`
`R = the number of known relevant document for a request`
`n = the number of documents term t(i) occurs in`
`N = the number of documents in the collection`
Note: We take n = Number of relevant documents term t(i) occurs in * Size of the document collection / 100 to avoid indexing on the entire document space (22 GB file). It was found that this heuristic didn't affect the performance, at the same time, it made the code scalable by avoiding loading the entire document collection and working with it.
The Relevance Weight is
`RW (i) = log [ ( (r+0.5)(N-n-R+r+0.5) ) / ( (n-r+0.5)(R-r+0.5) ) ]`
This formula can be used instead of CFW (formula (1)) for all terms used in a second or subsequent iteration. It can also be used for the first iteration by setting r=R=0; the resulting formula is normally a very close approximation to CFW as in (1). Note, however, that this approximation does not work with very frequent terms (such as some terms that might usually be on a stopword list). A term that occurs in most documents in the collection will be given a very small positive weight by (1), but a large negative weight by the approximation ((3) with r=R=0). A rough and ready strategy for avoiding this difficulty is simply to force negative weights to a small positive value, which is what we use for our case.

1.7 Query expansion

The power of relevance feedback comes not so much from reweighting the original query terms, as from expanding the query by adding new search terms to it. Essentially, terms may be taken from the documents assessed as relevant; however, some selection should be performed (it is not usually desirable to include all such terms).
All terms taken from relevant documents are ranked according to their Offer Weight
`OW (i) = r * RW (i)`

1.8 Pre-Processing the Data

This takes place in multiple stages due to the structure of this data. On an average, each document takes 0.03 second to go through the text and add them to the posting list. This is achieved by careful analysis of the bottlenecks and using C based hashing functions for filtering and removal.

- **Parsing:** We use the python library data = data.translate(str.maketrans('', '', '[]0() ,;:+*"?$`.')) to get rid of all the symbols. Note that this approach is faster than most 're' or 'regex' based pattern matching methods by at least 10x times since they are written in cPython and use native C compilers. The only way to obtain more speedup is by writing custom C functions yourself.
- **Stop Words:** Stop words are the most commonly occurring words which don't give any additional value to the document vector. in-fact removing these will increase computation and space efficiency. nltk library has a method to download the stopwords, so instead of explicitly mentioning all the stopwords ourselves we can just use the nltk library and iterate over all the words and remove the stop words. However, this operation proves to be a bottleneck since there are about 50 stopwords being removed from nearly 10-12 lakh tokens. A way to optimise this process is by caching the stopwords in memory:

```
cachedStopWords = stopwords.words("english")
def check_valid_token(token):
    return token not in cachedStopWords and not re.search(
    '[0-9]+', token)
```

  caching stop words instead of writing return token not in stopwords.words("english") results in a 10x speedup in stopword removal. Since stopwords were already removed from the doc, we used this method to remove stopwords from queries.

- **Apostrophe:** When we remove the apostrophe punctuation first, it will convert words like don't to dont, and it is a stop word which wont be removed. so what we are doing is we are first removing the stop words, and then symbols and then finally stopwords because few words might still have a apostrophe which are not stop words.
- **Single characters:** Single characters are not much useful in knowing the importance of the document and few final single characters might be irrelevant symbols, so it is always good be remove the single characters. Just like the stop words, these characters are overly present and dont add much value to the retrieval quality. If anything, they poison the TF-IDF scores of other words due to their relatively high frequency.
- **Numbers:** are removed. Numbers might have relevance in certain cases (eg: years - 1984 etc) but these cases are relatively lesser. Experimentation was done initially by converting numbers to words using `num2words`, but that did not increase retrieval quality while increasing runtimes and requiring further processing

1.9 Storing the data

All data was stored in dictionary format and only an inverted index with frequencies was created. The choice for dictionary data structure was based on a hashing based lookup that would work in $O(1)$ time even for a large corpus collection, which is what we deal with here. So the data is stored in the format:

```
{<token 1>: {<doc-id 1 1>: <frequency in the doc>, <doc-id 2>: ....} }
```

Here is the actual first line of the dict for query 162662:

```
"{'$doc_len$': {'$avg_doc_len$': 13.03030303030303, 'D2981241': 16, 'D2727318': 26,..}
'call': {'D2981241': 1}, 'group': {'D2981241': 1, 'D699018': 2}, 'lions': {'D2981241':
2}, 'vocabulary': {'D2981241': 1}, 'english': {'D2981241': 1}"
```

Note that the first entry in the inverted index is Length of Documents, stored along with the other entries to avoid extra operations in the retrieval stage (like iterating over the list and retrieving the number of unique docs and their lengths). The average length is store in the dict itself for convenience and faster reranking. A couple of other things to note here:

- Due to the large size of the doc file, all docs are not loaded in the memory together. Only the relevant documents, ie 100 documents are loaded while re-ranking each query at a time. For the next query, the previous 100 documents are rewritten by the next 100 documents.
- We take n = Number of relevant documents term t(i) occurs in * Size of the document collection / 100 to avoid indexing on the entire document space (22 GB file). It was found that this heuristic didn't affect the performance, at the same time, it made the code scalable by avoiding loading the entire document collection and working with it.

1.10 Performance and Scoring

2 Relevance Model based Language Modeling

$P(w|R)$ the relative frequency with which we expect to see the word w during repeated independent random sampling of words from all of the relevant documents. Since we have available training data in the form of relevance judgments, estimating $P(w|R)$ is as simple as counting the number of occurrences of w in the relevant documents and appropriately smoothing the counts.

$$P(w|R) \approx P(w|q_1.....q_k) \tag{1}$$

$$P(w|R) \approx \frac{P(w, q_1.....q_k)}{P(q_1.....q_k)} \tag{2}$$

We now have the following methods to approximate these probabilities

2.1 I.I.D. sampling

Let's assume that the query words q1....qk and the words w in relevant documents are sampled identically and independently from a unigram distribution C, the collection of documents given to us. Let M_R represent some finite universe of unigram distributions from which we could sample. The sampling process proceeds as follows: we pick a distribution $M \in M_R$ with probability P(M) and sample from it k+1 times. Then the total probability of observing w together with q1....qk is

$$P(w, q_1 \ldots q_k) = \sum_{M \in \mathcal{M}} P(M) P(w|M) \prod_{i=1}^{k} P(q_i|M)$$

Here we assume that w and q1...qk are independent once we pick a distribution M.

2.2 Conditional sampling

$$P(w, q_1 \ldots q_k) = P(w) \prod_{i=1}^{k} \sum_{M_i \in \mathcal{M}} P(M_i|w) P(q_i|M_i)$$

It is informative to contrast the assumptions made in the two methods we proposed for estimating the joint $P(w, q_1.....q_k)$. By Bayes' rule, Method 1 can also be viewed as sampling of $q_1.....q_k$ conditioned on w. However, for Method 1, this would add an additional constraint that all query words qi are sampled from the same distribution M, whereas in Method 2 we are free to pick a separate Mi for every qi. We believe that Method 1 makes a stronger mutual independence assumption, compared to a series of pairwise independence assumptions made by Method 2. In empirical evaluations, we found that Method 2 is less sensitive to the choice of our universe of distributions M. Method 2 also performs slightly better in terms of retrieval effectiveness and tracking errors.

2.3 Final Estimation Details

To ensure proper additivity of our model, we set the query prior to be

$$P(q_1....q_k) = \sum_{w} P(w, q_1....q_k) \tag{3}$$

where the summation is performed over all words w in the vocabulary. Similarly, we set the word prior $P(w)$ to be

$$P(w) = \sum_{M \in M_R} P(w|M) P(M) \tag{4}$$

We restrict our M_R to only contain 100 models corresponding to the top-ranked documents retrieved by the query. For retrieval, we use a baseline language modeling approach to IR. We use Dirichlet Smoothing to smooth our maximum likelihood document models with the background model of English.

For Method 1 we arbitrarily choose to use unigram distribution priors P(M). For Method 2, we choose to estimate the conditional probabilities of picking a distribution Mi based on w as follows:

$$P(M_i|w) = P(w|M_i) P(w) / P(M_i) \tag{5}$$

2.4 Unigram Model with Dirichlet Smoothing

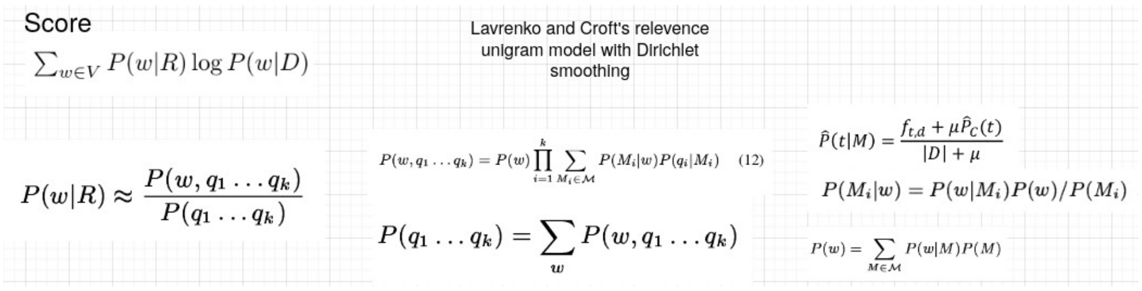
Score
$\sum_{w \in V} P(w|R) \log P(w|D)$
Lavrenko and Croft's relevence unigram model with Dirichlet smoothing
$P(w|R) \approx \frac{P(w, q_1 \ldots q_k)}{P(q_1 \ldots q_k)}$
$P(w, q_1 \ldots q_k) = P(w) \prod_{i=1}^{k} \sum_{M_i \in \mathcal{M}} P(M_i|w) P(q_i|M_i) \quad (12)$
$P(q_1 \ldots q_k) = \sum_w P(w, q_1 \ldots q_k)$
$\hat{P}(t|M) = \frac{f_{t,d} + \mu \hat{P}_C(t)}{|D| + \mu}$
$P(M_i|w) = P(w|M_i)P(w)/P(M_i)$
$P(w) = \sum_{M \in \mathcal{M}} P(w|M)P(M)$

Figure 1: Calculation workflow based on above discussion

In the above workflow for document reranking, we make the following assumptions -

- μ = Average Document Length (in the collection of 100 documents)
- $P_C(t) = tf(t)/100$ where $tf(t)$ = term frequency of the term t
- $P(M_i) = 1/100$
- $P(q_1....q_k)$ can be ignored since it is the same for all documents so it wont affect ordering. We dont compute it.
- M is the document collection of the 100 relevant documents
- V is the entire vocabulary of the full document collection. In our implementation, however, we approximate it by taking only the 100 relevant documents.
- We further remove all stop words in queries and lowercase the query tokens, since these are already removed as per the document preprocessing steps mentioned in the assignment statement

2.5 Bigram Model with Dirichlet Smoothing with Unigram Backoff

In bigram models, we make the approximation that the probability of a word depends only on the identity of the immediately preceding word, giving us

$$p(s) = \prod_{i=1}^{l} P(w_i|w_1....w_{i-1}) \approx \prod_{i=1}^{l} P(w_i|w_{i-1}) \tag{6}$$

This can be replaced by sum of log models and ordering of documents will be preserved.

$$Score = \sum \log(P(w_i|w_{i-1})) \tag{7}$$

To make $P(w_i|w_{i-1})$ meaningful for i = 1, we can pad the beginning of the sentence with a distinguished token $< BOS >$; that is, we pretend w_0 is $< BOS >$. In addition, to make the sum of the probabilities of all strings $\sum_s p(s) = 1$, it is necessary to place a distinguished $< EOS >$ at the end of sentences.

$$P(w_i|w_{i-1}) = c(w_{i-1}, w_i) / \sum_w c(w_{i-1}, w) = C(w_{i-1}, w_i)/C(w_{i-1}) \tag{8}$$

Where $C(w_1, w_2)$ is the count of w_1 occurring before w_2 in the sentence

To extend this strategy to bigram modeling, we similarly smooth the empirical bigram estimate with hyperparameter μ_1 pseudo-counts distributed fractionally according to the collection prior bigram model, $P(w_i|w_{i-1}, C)$:

$$P(w_i|w_{i-1}, D, C) = \frac{f_{w_{i-1},w_i} + \mu_1 P(w_i|w_{i-1}, C)}{f_{w_{i-1}} + \mu_1}$$

Alternatively, we can use the following model with $\lambda = \mu/(\mu + |D|)$ and μ = average length of the document collection C

- Unigram ML model:

$$p_{ML}(w_i) = \frac{c(w_i)}{\sum_{w_i} c(w_i)}$$

- Bigram interpolated model:

$$p_{interp}(w_i|w_{i-1}) = \lambda p_{ML}(w_i|w_{i-1}) + (1 - \lambda) p_{ML}(w_i)$$

Unigram and bigram models can then be easily mixed by treating our smoothed unigram distribution $P(w|D, C)$ as an additional prior on the bigram model and adding in μ_2 pseudo-counts drawn from it:

$$P(w_i|w_{i-1}, D, C) = \frac{f_{w_{i-1},w_i} + \mu_1 P(w_i|w_{i-1}, C) + \mu_2 P(w|D, C)}{f_{w_{i-1}} + \mu_1 + \mu_2}$$

Unigram Backoff is implemented using the following rule:
If $C(w_{i-1}, w_i) = 0$ then $P(w_i|w_{i-1}) = \alpha P(w_i)$ where $\alpha = 0.6$

2.6 Implementation details of Bigram Model

Here, we keep a track of the positions of all tokens in the document set, to identify if a bigram is present or no.

We store the position list in the following format: `{<doc-id 1>: {<token 1>: <List of positions in the doc>, <token 2>: ....} }`

Here is a sample of the position list on a small subset of the given documents
`{'D2981241': {'<BOD>': [0], 'call': [1], 'group': [2], 'lions': [3, 4], 'vocabulary': [5], 'english': [6], 'language': [7], 'word': [8], 'definitions': [9], '<EOD>': [10]}, 'D2727318': {'<BOD>': [0], 'laser': [1, 9, 13, 20]....`

- We append all query lists with $< BOD >$ at the start and $< EOD >$ at the end to account for the first and last query tokens. We further remove all stop words in queries and lowercase the query tokens, since these are already removed as per the document preprocessing steps mentioned in the assignment statement.

- Instead of using the product form of score calculation in Eq 6, we add the logs of all probabilities to score the documents as shown in Eq 7

2.7 Performance and Scoring

3 Parameters: Summary

Parameter	Value
BM25: k1	2
BM25: b	0.75
$P(M)$	1/100
$P_c(t)$	tf(t)/100
Dirichlet Smoothing: μ	Average Length of Doc Collection
Unigram Backoff: α	0.6

Table 1: Summary of all the hyperparameters and assumptions used in this assignment

4 Hardware details

The following is the machine information on which the execution parameters were calculated
model name : Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
Ram : 16GB
Number of cores : 4

5 References

[1] Victor Lavrenko and W. Bruce Croft. 2001. Relevance based language models. In Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval (SIGIR '01). Association for Computing Machinery, New York, NY, USA, 120{127. DOI:https://doi.org/10.1145/383952.383972

[2] Robertson, Stephen Jones, K.. (1997). Simple, Proven Approaches to Text Retrieval.

[3] Matthew Lease and Eugene Charniak. 2008. A Dirichlet-Smoothed Bigram Model for Retrieving Spontaneous Speech. Advances in Multilingual and Multimodal Information Retrieval: 8th Workshop of the Cross-Language Evaluation Forum, CLEF 2007, Budapest, Hungary, September 19-21, 2007, Revised Selected Papers. Springer-Verlag, Berlin, Heidelberg, 687{694. DOI:https://doi.org/10.1007/978-3-540-85760-0$_8$7