November 9, 2023

Reciprocal Rank Fusion (RRF)

Results

k	map	P@5	P@10
10	0.4707	0.3332	0.2408
100	0.4780	0.3444	0.2455
150	0.4796	0.3464	0.2463
300	**0.4811**	0.3457	0.2460
400	0.4786	0.3457	0.2460

Algorithmic Details

The core of the algorithm runs on the following psuedocode :

```
for each query:
    initialise score[document] = 0 for all documents
    for each document:
        for each ranking_mechanism:
            rank = rank by this ranking_mechanism for this document for this query
            score += (1 + 1/ (k + rank)
    sort documents in decreasing order of scores
    write to results-file
```

For documents not retrieved by a particular ranking system, (for example 24:NULL), I tried assuming a high (ie, bad) rank of 500/1000 and incorporated it in the score as well. However, after experimenting with this default rank, I found empirically that the best results were achieved when this pair was not included in the scores altogether.

Bordacount

Results

Approach	map	P@5	P@10
I	0.3981	0.2911	0.2250
II	0.4773	0.3454	0.2457

Algorithmic Details

Approach I psuedocode :

```
for each query:
    per ranker, collect all documents and ranks given to all documents
    per ranker, give 1 point to the lowest rank document, and keep +1 to the end

    initialise score[document] = 0 for all documents
    for each document:
        for each ranking_mechanism:
            score += points to this document for this ranker
    sort documents in decreasing order of scores
    write to results-file
```

Approach II psuedocode, which performs better experimentally :

```
for each query:
    initialise score[document] = 0 for all documents
    for each document:
        for each ranking_mechanism:
            score += (1000 - rank of this document by this ranker)
    sort documents in decreasing order of scores
    write to results-file
```

The motivation behind the (1000 - rank) idea was that I wanted to promote a lower rank, and linearly , and this satisfies both clauses. For documents not retrieved by a particular ranking system, (for example 24:NULL), I assuming a high (ie, bad) rank of 2000 and incorporated it in the score as well. This effectively adds a score of -1000 for this document, and doing this made a significant improvement in results.

Condorcet

Results

NULL pairs included?	map	P@5	P@10
YES	0.4786	0.3449	0.2485
NO	0.3942	0.2888	0.2249

Algorithmic Details

The core of the algorithm runs on the following psuedocode :

```

def is_d1_better_or_d2(d1 , d2):
    score = 0
    for ranker in rankers:
        if ranker ranks d1 better:
            score += 1
        else :
            score -= 1

    if score > 0:
        return d1 ## More rankers support D1
    else:
        return d2 ## More rankers support D2

for each query:
    assemble list of documents for this query
    quicksort documents in decreasing order using above comparison of scores
    write to results-file
```

Following the theoretical suggestions of the paper linked in the assignment, I used a quicksort mechanism to break the ties (cycle) that may arise in a condorcet voter. Once random document is chosen and all documents are declared better or worse than it, and recursively, a sorting is achieved. Further, including NULL pairs as in the comparison of

two documents achieved better results. When a ranking mechanism retrieves d1 but not d2, not ignoring its' bias towards d1 adds to the retrieval quality.

Bayes Fuse

Results

NULL pairs included?	map	P@5	P@10
YES	0.4786	0.3449	0.2485
NO	0.3942	0.2888	0.2249

Algorithmic Details

The core of the algorithm runs on the following psuedocode :

```

for each ranker:
    collect the relevant documents and their ranks
    collect the irrelevant documents and their ranks

for each ranker:
    compute p(rank = r | relevant)
    compute p(rank = r | irrelevant)

# I have approximated these distributions by instead computing
# p(rank falls in range_i | R/NR) where range can be (1, 5) or (6, 10) etc.
# The normalisation factor cancels in the division below so need not be included


for each query:
    initialise score[document] = 0 for all documents
    for each document:
        for each ranker:
            score[document] += log( p(rank = r|R)/p(rank = r |NR))

    sort documents in decreasing order of scores
    write to results-file
```

To estimate the p(rank = R | relevant), I used a bucketing approximation. Counting all relevant documents retrieved by this model, I estimated with what probability it fell in the range (1, 10) or (50, 75) or (400, 450) etc and I assumefd that the probability of retrieval within a range was uniform (justifies since intervals small). Similar estimation was done for the irrelevant documents, and dividing these terms would cancel the length of the range factor, and is hence equivalent to the exact estimation itself. Further, I found experimentally that adding more precision (making ranges smaller) to the probability distribution gave better results until they saturated.

November 9, 2023

Borda Fuse

Results

Best Results :

map	P@5	P@10
0.4862	0.3485	0.2489

Some other combinations of optimisations of weights also produces comparable maps of 0.4835, 0.4855, and have been discussed below.

Algorithmic Details

The core of the algorithm runs on the following psuedocode :

```

for file in files:
    per ranker:
        count the number of relevant documents retrieved
        penalise non-retrieved relevant documents slightly

# Based on the score of retrieval quality computed above
# attach proportional weights to the rankers

for each query:
    compute the bordacount score, but for each ranker,
    weigh contributions by the reliability weights computed above

    sort documents in decreasing order of scores
    write to results-file
```

I experimented with the weights being allotted to the rankers over a permutation of heuristics. I tried promoting rankers which had low (ie, better) ranked relevant documents, and rankers which had high (ie, worse) or Null rankings for not relevant documents. I also tried penalising good rankings for documents not-relevant, and bad rankings for relevant documents. Experimentally, I found that best results were achieved when parameters were only tuned for relevant documents. Also, I found that incorporating data from the 5 fold files did not hurt performance, but did not aid in it either.

Helper Functions

reader()

```

## Housed in reader_file.py
def reader(filename):
    ..........
    return dictionary

# returns a dictionary with query ID as keys
```

```
# dict[queryID] -> list of (Document_id, relevance_label, rankings)
# where, rankings is a dictionary (ranker -> rank) with rank = -1
# if the RankingMechanism does not enlist this document
```

getscores.sh

Shell script that assumes a "trec_eval-9.0.7" directory for the trec_eval tool, and streamlines the computation of map and precision values.

getrels.py

For generating the qrels file from agg.txt, for computation of map and precisions.