IR programming hw1 report

B05902112
資工四 張中漢

Description of my VSM:

In utils.py, there's a function "buildMaps" that can generate a dictionary and two lists from inverted-file, which are Bigram2DF, Doc2Bigram and DocLens (and saved as files).

- With Bigram2DF, we can get all information as inverted-file provides and all terms' idf in form of dictionary.
- With Doc2Bigram, we can have a list of all bigrams that one doc contains.
- With DocLens, we can know every document's length (number of bigrams)

```
def buildMaps(invert):
    Bigram2DF = {}
    Doc2Bigrams = [[] for i in range(46972)]
    DocLens = [0 for i in range(46972)]
    line = invert.readline()
    while line:
        if line.count(' ') == 2:
            three_nums = (line[:-1]).split(' ')
            df = int(three_nums[2])
            bigram = (int(three_nums[0]), int(three_nums[1]))
            temp = []
            for i in range(df):
                line = invert.readline()
                two_nums = (line[:-1]).split(' ')
                temp.append((int(two_nums[0]), int(two_nums[1])))
                Doc2Bigrams[int(two_nums[0])].append((bigram, int(two_nums[1]))) # Doc2
                DocLens[int(two_nums[0])] += int(two_nums[1]) # DocLen = num of bigrams
            Bigram2DF.setdefault(bigram, (temp, np.log((46972-df+0.5)/(df+0.5)))    ) #
        line = invert.readline()
    return Bigram2DF, Doc2Bigrams, DocLens
```

Bigram2DF can serve as a linked list, so I don't need to make all vectors because there are too many zeros; thus, I can save much time and memories.

I use Okapi/BM25 method to define the weight score of a given feature, and I compute it with all parameters provided by the three maps mentioned above.

```python
def computeTF(count, d_len, avdlen):
    k = 1.5
    b = 0.4
    return ((k+1) * count) / (k * ((1-b)+(b*d_len/avdlen)) + count)


def computeWeight(query, DocLens, Bigram2DF, TopK): # query is a list of bigram tuple, ex: [((term_
    DocScores = [0 for i in range(46972)]
    avdlen = sum(DocLens) / 46972
    # print(query)
    for termNnum in query:
        Qterm = termNnum[0]
        Qtf =  computeTF(termNnum[1], len(query), avdlen)
        IDF = Bigram2DF[Qterm][1]
        QWeight = Qtf * IDF
        for docNtf in Bigram2DF[Qterm][0]:
            docID = docNtf[0]
            # print(docID)  # should be 10849
            tf = computeTF(docNtf[1], DocLens[docID], avdlen)
            DWeight = (tf * IDF)
            DocScores[docID] += (QWeight * DWeight)



    ScoreNDocIDList = ( sorted( [(x,i) for (i,x) in enumerate(DocScores)], reverse=True )[:TopK] )
    return ScoreNDocIDList
```

Description of my Rocchio Relevance Feedback:

ScoreNDocIDList stores the sorted tuples like (Score, DocumentID); in theory, those have the higher ranks or smaller indices are regarded as relevant docs.

```python
ScoreNDocIDList = computeWeight(QTermIDList,DocLens, Bigram2DF, 100)

# relevance feedback
if relevance_feedback == 1:
    rand = np.random.randint(10)
    DocID = (ScoreNDocIDList[rand])[1]
    QTermIDList = QTermIDList + QTermIDList + Doc2Bigrams[DocID]
    ScoreNDocIDList = computeWeight(QTermIDList,DocLens, Bigram2DF, 100)
```

I randomly pick one doc among the top 10s and append it to the original query, and double the original query to remain its significance.

Results of Experiments:

- MAP value under different parameters of VSM with train query set

Parameter b is somewhat an degree of penalty on long docs; after many times of tries, I observed b = 0.4 is the optimal and others are much worse than it.

For various k, it doesn't seem to exist an obviously best value, so I pick 1.5 (without Relevance Feedback)

k = 1.6, b = 0.4, MAP = 0.7961283155525578

k = 1.5, b = 0.4, MAP = 0.7968108174592483

k = 1.4, b = 0.4, MAP = 0.7953526678211351

k = 1.3, b = 0.4, MAP = 0.7951079228692393

k = 1.2, b = 0.4, MAP = 0.7887070000515493

- Feedback v.s. no Feedback

Without Relevance Feedback, the MAP scores usually can reach 0.796 or even 0.8 computed with train query set; it gets 0.771 on Kaggle.
With Relevance Feedback, my system doesn't always work well in train query set. Its MAP score on Kaggle ranges from 0.755 to 0.779, with 0.779 the best score on Kaggle.
A possible reason is that the correct docs scatter in the top of the ranking list, so it may miss it to append a non-relevant doc to the query.

Below are several MAP scores with relevance feedback and train query set:
0.7750711665513065, 0.778605841418991, 0.8142317438419575,
0.7763877553647258, 0.786564379289086, 0.8053472700739317
0.8030613413920913, 0.7995709705919201, 0.7971928951333541
0.7880000849990436
with the average is 0.7924033448656407

Discussion:

From the formula of MAP, I notice that outputting as many as possible can yield the highest score because the score only increases with non-negative numbers with the rank list get longer.

Comparing different variants of tf:
tf = (# of terms): 0.7454472559075696
Okapi/BM25: 0.7790181230560308
Okapi/BM25 with pivoted doc len normalizer: 0.7968108174592483
It makes more senses that giving penalty on terms with high tf and long docs improves the precision.