Assignment 1

Student ID：z5192086　Name: PAN LUO

Q1:

/k,/s,/P means the searching objects are limited in the number of k words, are limited in the same sentence, are limited in the same paragraph respectively.

For the normal positional inverted index, it contains “term”, ”numbers of docs containing term”, ” docs ID” and “term’s position”.

So, to implement /k,/s,/P these three searching function with the modification of positional inverted index. My thought is adding additional dimension called “Paragraph ID” and “Sentences ID”.

You know, paragraph is made by sentences, and sentence is made by words. So I create two relevant connections, one is “sentences to words”, another is “paragraphs to sentences”. These two connections can store in the one positional inverted index, or just store separately in order to save storage and can running in a more efficient way for different kinds of queries. For example, when just using /k,/s in one particular paragraph, then we just need to use algorithm to match the same “Sentence ID” and look for the distance of the objective words. To be more specific, if the search is “a/s b/k”, then what we do is searching for the “Sentences ID” which contains both “a” and “b”. It means for word “a” and “b”, their “Sentences ID” is the same. Then for the “Words ID”, we calculate each distance of posting and return the result which the distance of “a” and “b” is smaller or equal to k. In this case, we assume that we are searching on a “paragraph level”. So we do not need to build the connection between “Paragraph” and “Sentences”.

But to be more general, we can build three connections of “Sentences to words”, “Paragraph to sentences” and “Documents to Paragraphs”. When searching in different level, we just link two or three searching. For example, when using “/p,/s,/k” these 3 proximities between “paragraph level” and “sentences level”, then what we do is building two search. First is searching for the same paragraph ID and calculate the distance of sentences, if the distance is zero, then return the results. Second, we use these results as Sentences ID, and calculate the distance of words, if the distance is smaller or equal to k, then return the final results. If we are searching between different documents, then just add another search between “Documents to Paragraphs” as the first step so on and so forth.

What’s more, if we want to put the “Document ID”、“Paragraph ID”、“Sentences ID” into one positional inverted index, then we can just change every posting into three dimension vector like (Document ID, Paragraph ID, Sentences ID). Then if we do /s,/k,/P search, we can just read every dimension and do the same calculate like before. But I think storing them together will cause problem while read them together into memory, maybe it is a little heavy. Because one paragraph may contain too many words, so saving them together means we should build many mappings.

All above is my design for this question. I have thought some additional cases, for example, I am not sure if the last word in one sentence or paragraph is also “with in k words” between the first word of next sentence or paragraph. If so, when we are doing search in “Sentence level” and “Paragraph level”, then we should calculate the distance of “Sentence ID” and

"Paragraph ID" which is close enough that can cause two words "within k words" in different paragraphs.

Q2:

(1) Assume that we use the number of "n" pointers. Then, in the Step 1, if we do sequential search in the skip pointers array, then the cost will be O(n/2). Next, in the Step 2, if we do sequential search in the target segment, then the cost will be O(L/2N). So the total cost is O(n/2 + L/2n). If we want to minimize the total cost, then we calculate the minimum of F(n) = n/2 + L/2n. Calculate the derivative of F(n) and let it equal to zero. Then we can get "1/2(1 – L/n^2) = 0", and the result is "L = n^2", and the final result is "n = $\sqrt{L}$". In the assumption, n is the number of skip pointers. And use the number of $\sqrt{L}$ pointers lead to the minimum total cost. In other words, using the number of $\sqrt{L}$ pointers has the best worst case performance. That's proved.

(2) Assume that we use number of "n" pointers. In the step 1, if we do binary search in the skip pointers array, then the cost will be O(log(n)). Next, in the Step 2, if we do binary search in the target segment, then the cost will be O(log(L/n)). So the total cost is O(log(n)+log(L/n)). If we want to minimize the total cost, then we calculate the minimum of F(n) = log(n) + log(L/n) = log L. So the total cost is a constant which equal to log(L). So the total cost is not relevant to the number of skip pointers. And no matter how many skip pointers are used, the performance is the same.

(3) Assume that we use number of "n" pointers. In the step 1, if we do binary search in the skip pointers array, then the cost will be O(log(n)). Next, in the Step 2, if we do sequential search in the target segment, then the cost will be O(L/2N). So the total cost is O(log(n) + L/2n). If we want to minimize the total cost, then we calculate the minimum of F(n) = log(n) + L/2n. Calculate the derivative of F(n) and let it equal to zero. Then we can get 1/nln(2) – L/2n^2 = 0, and the final result is "n = L·ln(2)/2 ". In the assumption, n is the number of skip pointers. And use the number of "L·ln(2)/2" pointers lead to the minimum total cost. In other words, using the number of L·ln(2)/2 pointers has the best worst case performance.

Q3:

(1) According to the Score(d, Q) function, for each word in the query, the first part "idf_t" is the a constant, it is equal to the normalization of log(N/df). And for each word, the "idf" part is the same. For the second part of Score(d, Q) function, given by the value of k1、k3、b, we can compute that the second part is a incremental function about the variable tf(t,d). For the third part of Score(d, Q) function, it is also a constant because we often treat tf(t, Q) as 1.

Overall, to calculate maxscore of each term, we only need to concentrate on the tf(t,d). In the score function, if we can get the highest tf(t,d) of one term, then we can compute the maxscore of that term. So we do not need to examine the posting list. What we should do is just pre-computing the highest tf(t,d) of one term before building the posting list. In other words, maxscore for each keyword can be computed *without* examining the postings list. That's proved.

(2) For term A, the Maxscore(A)= 6 x [(3x10) / (2+10)] x (3/3) = 15

For term B, the Maxscore(B) = 2 x [(3x4) / (2+4)] x (3/3) = 4

For term C, the Maxscore(C) = 1 x [(3x7) / (2+7)] x (3x3) = 2.33

For the query{A,B,C}:

d1 has Score(d, Q) = 6 + 2 + 1 = 9 . From here, we can kick out B,C lists and continue calculate the score of documents which occur in A list. Because 9 > (4+2.33), so if we want to score more than 9, we must find documents which occurs in A list. So now we could just do SkipTo(x) on list B,C.

d2 has Score(d, Q) = 14.4 + 1 =15.4

d5 has Score(d, Q) = 10. 8 + 4 + 1.5 = 16.3

d8 has Score(d, Q) = 15 + 1 = 16

So the top 2 result is 16.3 and 16. The document is d5 and d8. After calculating Score of d1, only list the documents occur in list A will be scored.

Q4:

(1) The precision = 6 / 20 = 0.3

(2) The recall = 6 / 8 = 0.75

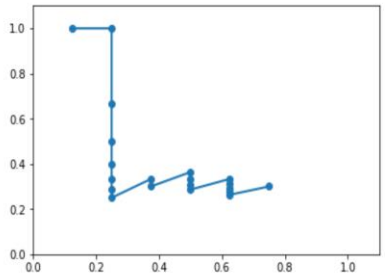
So the F1 = 1/ { (1/2)x [(1/0.3) + (1/0.75)] } = 3/7 = 0.42857

```
In [5]: %matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

B_raw_recalls    = np.array([1/8, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 3/8, 3/8, 4/8, 4/8, 4/8, 4/8, 5/8, 5/8, 5/8, 5/8, 5/8, 6/8])
B_raw_precisions = np.array([1/1, 1/1, 2/3, 2/4, 2/5, 2/6, 2/7, 2/8, 3/9, 3/10, 4/11, 4/12, 4/13, 4/14, 5/15, 5/16, 5/17, 5/18, 5/19, 6/20])

plt.xlim(0, 1.1)
plt.ylim(0, 1.1)

line, = plt.plot(B_raw_recalls, B_raw_precisions, 'o-', linewidth=2)
plt.show()
```



(3) As is shown in above picture and figure. The uninterpolated precision(s) of the system at 25% recall are 1、2/3、2/4、2/5、2/6、2/7、2/8

(4) As is shown in above picture and figure. When 8< k < 11, the maximum precision is 4/11. And 2/8 < 33% < 4/8 (2/8 is the recall which k = 8 and 4/8 is k =11). So the interpolated precision at 33% recall is 4/11 = 0.364

(5) The MAP = 1/6 x (1 + 1 + 3/9 + 4/11 + 5/15 + 6/20) = 0.555

(6) The largest MAP means the top k's result is relevant. In the top 20 results, there are 6 Relevant documents, so to maximize the MAP, the 21th and 22th documents should be Relevant. So the largest MAP of this system is

1/8 x (1+1+3/9+4/11+5/15+6/20+7/21+8/22) = 0.503

(7) The smallest MAP means the last 2 Relevant documents occurs in the last two documents of the whole collection. So the 9999th and 10000th documents should be relevant. So the smallest MAP of this system is

1/8 x (1+1+3/9+4/11+5/15+6/20+7/9999+8/10000) = 0.416

(8) 0.555 - 0.503 = 0.052

0.555 – 0.416 = 0.139

So the error range of calculating (5) instead of (6) and (7) for this query is [0.052, 0.139]. The maximum error is 0.139, the minimum error is 0.052, and the error can be a number between these two.