Assignment 1.

Q1[**2**]. What is the difference between document search and query answering?
Document search returns all of the documents that contain some/all of the words of the query in ranked order, as where query answering extracts the exact answer to a query from the provided documents.

Q2[**2**] What are the main problems of using stemming when preprocessing natural language documents before applying a document search?
The main problem of using stemming when preprocessing natural language documents before applying a document search is that both 'overstemming' and 'understemming' can occur.

Overstemming occurs when too many words share the same stem, causing the given algorithm to be over aggressive when discluding words. This leads to a loss of meaning and readability of the text, as well as incorrect retrieval of documents.

Understemming occurs when a stemmer fails to produce accurate root forms, causing the given algorithm to miss reducing some words to their base form. This leads to a loss of meaning and information, as well as a lack of total documents collected.

Q3[**8**]. Suppose that we have only three words in the language: `a`, `b`, and `c`, the documents:

`d1`: a a b a c a
`d2`: b b a a c c a
`d3`: a c c

and the query

`q`: a b.

a[2]) Show the text vectors for the query and the documents.
q: [1, 1, 0]
d1: [4, 1, 1]
d2: [3, 2, 2]
d3: [1, 0, 2]

b[2]) Normalize the vectors using the TF-IDF formulas. Make sure to use the 0.5 formula for the query.
Note: TF is the frequency of any "term" in a given "document"; IDF is constant per corpus that accounts for the ratio of documents that include the specific "term". The goal is to put more emphasis on the words that are less frequent in the document

TF-IDF = (ft,d / Σt ∃ d) x log2(m/df) for each word in the query, where:

- (*f*t,d / Σt ∃ d) : normalized value for the number of times the term appears in the document; basically (# times term appears in the document / total # terms that exist in the document)
- log2(m/df) : inverse document frequency (IDF), basically (total # of documents / total # of documents that contain the term)

q: TF-IDF for a = (0.5 + 0.5 * 1/1) * log2(3/3) = 0
TF-IDF for b = (0.5 + 0.5 * 1/1) * log2(3/2) = 0.584
TF-IDF for c = 0, word does not appear in query

vector -> [0, 0.584, 0]

d1: TF-IDF for a = (4/4) * log2(3/3) = 0
TF-IDF for b = (1/4) * log2(3/2) = 0.146
TF-IDF for c = (1/4) * log2(3/3) = 0
vector -> [0, 0.146, 0]

d2: TF-IDF for a = (3/3) * log2(3/3) = 0
TF-IDF for b = (2/3) * log2(3/2) = 0.390
TF-IDF for c = (2/3) * log2(3/3) = 0
vector -> [0, 0.390, 0]

d3: TF-IDF for a = (1/2) * log2(3/3) = 0
TF-IDF for b = (0/2) * log2(3/2) = 0
TF-IDF for c = (2/2) * log2(3/3) = 0
vector -> [0, 0, 0]

c[2]) Compute the cosine distance between the query and each of the documents using the normalized values.

Note: If we want to rank the documents relative to each query, we can do *cos (angle between query vector and document vector)*

- if angle is 0, similarity is 1 (i.e., identical)
- if angle is 90 degrees, similarity is 0 (i.e., don't share words in common)

Meaning, cosine(d,q) = <d dot product q>/(|d|*|q|), where:

- d dot product q = $d_1*q_1+...+d_n*q_n$
- |d| = size of vector = sqrt(sum d_i^2)

q: [0, 0.584, 0]; |q| = √0^2 + 0.584^2 + 0^2 = 0.584

d1: [0,0.146 , 0]; |d1| = √0^2 + 0.146^2 + 0^2 = 0.146
cos (θ) = ((0 * 0) + (0.584 *0.146) + (0 * 0)) / (0.584 * 0.146)
= 1

d2: [0, 0.167, 0]; |d2| = √0^2 + 0.390^2 + 0^2 = 0.390
cos (θ) = ((0 * 0) + (0.584 * 0.390) + (0 * 0)) / (0.584 * 0.390)
= 1

d3: [0, 0, 0]; |d3| = √0^2 + 0^2 + 0^2 = 0
cos (θ) = ((0 * 0) + (0.584 * 0) + (0 * 0)) / (0.584 * 0)
= 0

d[2]) Compute the distance between the query and each of the documents using the Okapi BM25 formula. Use `k1`=1.2, `b`=0.75, and `k2`=100.

Note: It is a ranking function used by search engines to find the best match / estimate the relevance of documents to a given query.

A. Query : ‘a’

okapi(d1, qa) = ((ln(3 - 3 + 0.5)) / (3 + 0.5)) * (((1.2 + 1) * 4) / (1.2 * (1 - 0.75 + (0.75 (6/5.33))) + 4))) * ((100 + 1) / (100 + 1))
okapi(d1, qa) = - 0.328

okapi(d2, qa) = ((ln (3 - 3 + 0.5)) / (3 + 0.5)) * (((1.2 + 1) * 3) / (1.2 * (1 - 0.75 + 0.75 (7/5.33)) + 3))) * ((100 + 1) / (100 + 1))
okapi(d2, qa) = - 0.292

okapi(d3, qa) = ((ln (3 - 3 + 0.5)) / (3 + 0.5)) * (((1.2 + 1) * 1) / (1.2 * (1 - 0.75 + 0.75 (3/5.33)) + 2))) * ((100 + 1) / (100 + 1))
okapi(d2, qa) = -0.156

B. Query : ‘b’

okapi(d1, qb) = ((ln(3 - 2 + 0.5)) / (2 + 0.5)) * (((1.2 + 1) * 1) / (1.2 * (1 - 0.75 + (0.75 (6/5.33))) + 1))) * ((100 + 1) / (100 + 1))
okapi(d1, qa) = 0.155

okapi(d2, qb) = ((ln (3 - 2 + 0.5)) / (2 + 0.5)) * (((1.2 + 1) * 2) / (1.2 * (1 - 0.75 + 0.75 (7/5.33)) + 2))) * ((100 + 1) / (100 + 1))
okapi(d2, qa) = 0.205

okapi(d3, qb) = ((ln (3 - 2 + 0.5)) / (2 + 0.5)) * (((1.2 + 1) * 0) / (1.2 * (1 - 0.75 + 0.75 (3/5.33)) + 0))) * ((100 + 1) / (100 + 1))
okapi(d2, qa) = 0

okapi d1 = 0.81 -0.173
okapi d2 = 0.97 -0.087
okapi d3 = 0.14 -0.156

Thus, relevance: d2 > d3 > d1

Q4[**2**] Suppose that the user tells us that the `d3` document is relevant. Apply the *Rocchio Method* to change the query (assume α=0.9, β=0.1, γ=0). Show the new non-normalized vector for the query.

$$q_e = \alpha \cdot q + \frac{\beta}{|D_q^r|} \sum_{d_r \in D_q^r} d_r - \frac{\gamma}{|D_q^{irr}|} \sum_{d_i \in D_q^{irr}} d_i.$$

$$\alpha + \beta + \gamma = 1.$$

qe = 0.9 * [1, 1, 0] + 0.1 * (1/1) * [1, 0, 2]

qe = [0.9, 0.9, 0] + [0.1, 0, 0.2]

qe = [1, 0.9, 0.2]

Note: The query *q* is replaced with a new query *q*e that emphasizes the relevant documents and deemphasizes the irrelevant ones.

Q5[**2**] Consider a set of documents and the query "learn by doing". Suppose that you want to create a single Patricia Trie that stores only the documents associated with each leaf node (i.e, you don't store postings – that is, no word position information). What character sequences do you need to add to the trie in order to be able to answer this query? You can assume that all queries will contain 20 characters or less. Don't worry about capitalization.

"learn by doing"
"by doing"
"doing"

Note: Patricia Trie is used to store common prefixes of strings only once; each node in the trie represents a single character(s) of the string (in our case, using words, due to the 20 character limit), and the path from the root to a leaf. This allows for an efficient retrieval of documents based on the query (in our case, "learn by doing") or any of its substrings (given above).

Q6[**8**]. Consider the following scenario.

Documents: $\{d_1, \ldots, d_{10}\}$
Returned documents $\{d_1, d_2, d_3, d_4, d_5, d_6\}$
Relevant documents $\{d_2, d_4, d_6, d_8\}$

a[2]) What is the precision?

The precision is 3/6 or 0.5, meaning 3 out of the 6 returned documents are relevant.

b[2]) What is the recall?

The recall is ¾ or 0.75, meaning 3 out of 4 of the relevant documents were found.

c[2]) What is the f_1 score?

f1 = (1+1^2) * precision * recall / (1 * precision + recall)
f1 = 2 * (3/6) * (3/4) / (3/6 + 3/4)
f1 = 0.6 ; also known as 3/5

d[2]) What is the MAP score? (Assume that returned documents are returned in order of relevance)

Assuming only one query was called:

AP = ΣPc / # of relevant documents
AP = ((1/2) + (2/4) + (3/6)) / 4
AP = 0.375; also known as 3/8

Q7[**2**]. Consider an information retrieval system for document search.

a) Can returning more documents decrease the recall?

No; returning more documents will **increase** the recall. This is because, if the system returns all of the available documents, then all of the relevant documents are bound to be present within them.

For example, imagine a scenario where there are 100 documents to search from and 5 of them are relevant to the given query. If the system returns all 100, then the 5 relevant documents are guaranteed to be present, thus resulting in an extremely high precision score (1.0 in this case).

b) Can returning more documents decrease the precision?

Yes; returning more documents will **decrease** the precision. This is because, if the system returns all of the available documents, then the percentage of returned documents that are relevant will be extremely low.

For example, imagine a scenario where there are 100 documents to search from and 5 of them are relevant to the given query. If the system returns all 100, then the 5 relevant documents present within those documents will be drowned out amongst the rest, thus resulting in an extremely low precision score (0.05 in this case).