Solution sketch
Fall 2024

EVALUATION

(a) P = 8 / 15 = 0.533
R = 8 / 10 = 0.8
F_1 = (2PR) / (P + R) = 0.64

(b) MAP is the arithmetic mean of AP(*burrito*) and AP(*shrimp cocktail*). To compute the average precision AP(*burrito*) we compute the precision at the positions of the relevant documents, and average these values. Similarly for AP(*shrimp cocktail*). This gives us:
AP(*burrito*) = 1/5 * (P@1 + P@2 + P@4 + P@6 + P@8) = 0.81
AP(*shrimp cocktail*) = 1/2 * (P@1 + P@4) = 0.75
MAP = 1/2 * (AP(*burrito*) + AP(*shrimp cocktail*)) = 0.78
Clearly showing the correct steps without arriving at a final numerical result will give full marks.

(c) In an ideal search result, highly relevant documents should appear at the top of the list, as users are more likely to click on results they see first. NDCG supports graded (i.e., not binary) relevance scores. DCG sums the relevance scores of documents in the result list, but with a (logarithmic) discount factor for each position down the list. This discount reflects that documents appearing further down are less useful to users. NDCG equals DCG normalized by the ideal DCG, i.e., the DCG score of the perfect ranking. NDCG can be "fooled" by duplicates in the result set.

MIXED GRILL

(a) This is a rather open-ended question where the students can get creative. Reward a good thought process and discussion. A simple yet plausible option might be to modify the ranking function by adding a recency factor to the TF-IDF score. For instance:

score = tfidf_score + alpha * recency_score

Where recency_score is inversely proportional to the age of the document and alpha is a weight factor that controls the influence of recency. By weighting recent documents higher, the search engine can prioritize timely content, which is likely more relevant for news queries. To compute recency_score, we could, e.g., store the publication timestamp *t* as document metadata, and let recency_score = f(now() - *t*) where now() is the timestamp for when the query is evaluated.

(b) As these are sparse vector representations, when computing dot(A, B) make sure to compare the right terms.
length(A) = sqrt(0.4^2 + 0.8^2 + 0.2^2) = sqrt(0.16 + 0.64 + 0.04) = sqrt(0.84) = 0.92

length(B) = sqrt(0.5^2 + 0.3^2 + 0.3^2) = sqrt(0.25 + 0.09 + 0.09) = sqrt(0.43) = 0.66
dot(A, B) = (0.4 * 0.3) [for “science”] + (0.8 * 0.5) [for “search”] = 0.12 + 0.4 = 0.52
cosine(A, B) = dot(A, B) / (length(A) * length(B)) = 0.52 / (sqrt(0.84 * 0.43) = 0.87

(c) Cf. dynamic indexing. Don’t rebuild your entire index each time, this would be too time-consuming. For deletion, we could have a bit mask of the length of the number of documents in the index where a value of 1 indicates that the document has been logically (but not physically) removed while a value of 0 means that the document is logically present. Then filter by this when we search. For the addition of new documents, we could create a really small and memory-based auxiliary index holding the new documents that is searched if the query of the main index does not return anything (or, better, we query both in parallel and merge the results.) However, every once in a while we rebuild the main index, removing the old documents (and resetting our deletion bitmasks) and adding the new documents. If we do not do that, we risk having our auxiliary index being just as big, if not bigger than our main index. Another possible method is to use a logarithmic merge, this means maintaining multiple indexes that double in size each time. The smallest one is kept in memory while larger ones are on disk. Once the index gets too big either write it as a new index (if no indexes are on disk) or merge it with the smallest index on disk, if this is the only index, write to disk, otherwise continue merging until you have merged with every index on disk and then save it to disk.

(d) Using a compressed index has two big benefits, the first is that the total space required by the index is reduced. This might save us some money in terms of, e.g., less disk space. Also, data transfer needs/costs at various levels are positively impacted. But as a direct result of the smaller footprint, we can store more of the index in main memory which means that our system would be more performant. An important trade-off when choosing the compression scheme is space-savings versus decompression speed. E.g, you might very well opt for slightly worse compression results if the decompression speed is much better. This is especially true for posting list compression where you do linear traversal (compared to, say, some kind of tree-traversal which you might do for dictionary accesses.)

(e) *Gamma encoding*: We encode all gap values with a length and offset pair, where the offset is the value in binary with the leading bit removed and the length is the number of bits to make the offset in unary code.

Variable-byte encoding: Logically, convert the gap value to a number written in base 128 (7 bits) and how many digits you need and what those digits are. One byte per digit. One bit per byte is used as a “continuation bit” to flag if there are more digits in the number than the ones you have read so far.

Simple9 encoding: In a 32-bit integer, pack as many consecutive gap values as you can into 28 bits (“data bits”), and use the remaining 4 bits (“control bits”) to tell you how to interpret the data bits: As one 28-bit number, or as two 14-bit numbers, or as three 9-bit numbers (with one bit wasted), or as…

(f) Assume that we have already added to the filter a bunch of elements, including the element *x*. With *k* independent hash functions and no collisions, *k* of the backing bits were set when *x* was added. To delete *x,* we cannot simply unset these bits because at least one of these bits might also have been set when some other element $y \neq x$ was added. If we had unset these bits, we would effectively have deleted not just *x* but all elements that hash to at least one of the same backing bits as *x*.

STRINGORAMA

a) A trie is a prefix tree where the root is the empty element and each branch represents one of the possible first elements of the dictionary entries. For strings, elements would be characters. The next depth level would be the second character, and so forth: For a trie node we can associate the string composed of the characters that take us from the root to the node. This allows traversals in the same style as with a B-tree, and prefix sharing avoids some data redundancy. Testing membership for a query string is cheap as we don't even have to read the full string in the case of non-membership.

b) Cf. assignment B-2 and the EditSearchEngine class. To be able to traverse it efficiently we use the fact that each trie path is a prefix shared by all terms in the subtrie identified by the path, in other words, we do not need to recalculate the number of edits to get to the prefix for each further element we consider. We also make use of that once all the next elements from a prefix generate an edit distance larger than *k*, then we can prune that branch and not calculate the remaining suffixes.

c) Large values of *k* imply that we cannot prune branches as quickly, which implies having to traverse a larger portion of the trie, which implies a longer running time. The time complexity decreases very quickly with *k* (it is $O(k|\Sigma|^k)$, specifically), thus, typically, *k* is just 1, 2, or 3.

d) To create the suffix array for the string *superduper* we first consider the set of positions where a query is allowed to start, which is [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]. Logically, these positions correspond to the suffixes *superduper*, *uperduper*, *perduper*, *erduper*, …, *r*. We then sort this array of positions, comparing two positions by the lexicographical comparison of their implied suffixes. This yields the resulting suffix array [5, 8, 3, 7, 2, 9, 4, 0, 6, 1], which logically corresponds to the strings:

duper (value 5)
er (value 8)
erduper (value 3)
per (value 7)
perduper (value 2)

r (value 9)
rduper (value 4)
superduper (value 0)
uper (value 6)
uperduper (value 1)

e) Cf. assignment B-1 and the SuffixArray class. Do a binary search for *up* in the suffix array. When doing comparisons, we lexicographically compare *up* with the implied strings corresponding to the positional indexes listed in the suffix array. We'd start with the middle of the suffix array (value 2, corresponding to *perduper*). The comparison with *up* then tells us that we should consider the last half of the suffix array. Repeating this process lands us at the penultimate element of the suffix array (value 6, corresponding to *uper*) where the binary search ends. We now need to discern if this location in the suffix array is where (i) actual matches for *up* start, or (ii) matches for *up* should have started if they had existed (but don't exist.) Since *uper* starts with *up* we have case (i). We can then simply scan down the suffix array from our start location until either the array ends or until the implied strings no longer start with *up*. This yields the starting positions for both matches.

CLASSIFICATION

(a) (i) Using Euclidean distance, the three labeled examples closest to *xtest* are *x2*, *x1,* and *x4*, in that order. (Their respective distances to *xtest* being 1.0, sqrt(2), and 2.0, respectively.) Two of these have the label "o", one has the label "+". Hence *xtest* would be classified as "o" with simple unweighted voting.
(ii) Note that the closest example *x2* is labeled "+", not "o". The weight placed on that "+" will therefore be greater than the weight placed on any of the two "o" objects. If the weight drops sharply as a function of the distance, the sum of the two "o" weights might not exceed the weight for the single "+", and it could be that *xtest* gets classified as "+" and not "o". Whether this happens or not would depend on how sharply the weight drops as a function of the distance.

(b) First compute centroids for "+" and "o":

centroid("+") = [(-2, 1) + (-2, -1) + (1, -1) + (2, 1)] / 4 = (-1, 0) / 4 = (-0.25, 0)
centroid("o") = [(-2, 2) + (1, 1)] / 2 = (-1, 3) / 2 = (-0.5, 1.5)

Then find out which centroid *xtest* is the closest to:

distance(*xtest*, centroid("+"))
= sqrt((0.75)^2 + (1)^2) = sqrt(9/16 + 16/16) = sqrt(25/16) = 5/4 = 1.25
distance(*xtest*, centroid("o"))
= sqrt((0.5)^2 + (0.5)^2) = sqrt(1/4 + 1/4) = sqrt(2) / 2 = 1 / sqrt(2) = 0.707

Hence, *xtest* is classified as “o”.

(c) The plot of the data clearly reveals that the classification problem is not linearly separable. John should therefore use a linear SVM with slack variables (“soft margin classification”, to allow for misclassifications). He could also use a non-linear SVM with some suitable kernel function: This might make the problem linearly separable in some higher-dimensional space and obviate the need for slack variables.

(d) The transformed *xtest* is represented as (0, 1). Since we are using naive Bayes, we assume that *b1* and *b2* are independent conditioned on the label. Thus, with the specified smoothing, we have:

P(“+”) = 4 / 6 = 2 / 3
P(“o”) = 2 / 6 = 1 / 3
P(*b1* = 0 | “+”) = (2 + 1) / (4 + 2) = 1 / 2
P(*b1* = 0 | “o”) = (1 + 1) / (2 + 2) = 1 / 2
P(*b2* = 1 | “+”) = (2 + 1) / (4 + 2) = 1 / 2
P(*b2* = 1 | “o”) = (2 + 1) / (2 + 2) = 3 / 4

P(“+” | *xtest*)
$\propto$ P(“+”) * P(*b1* = 0 | “+”) * P(*b2* = 1 | “+”) = (2 / 3) * (1 / 2) * (1 / 2) = 1 / 6
P(“o” | *xtest*)
$\propto$ P(“o”) * P(*b1* = 0 | “o”) * P(*b2* = 1 | “o”) = (1 / 3) * (1 / 2) * (3 / 4) = 1 / 8

Hence, *xtest* is classified as “+”.

Note that the text is open to some interpretation on how the (b1, b2) space could be thought of as “words” in a “document”, which might impact the exact calculations above. The students should justify their reasoning.