Fall 2022
Solution sketch

PAIRWISE PREFERENCES

(a) See, e.g., here. The goal is not really to build pairs that parrot/replicate the current ranking (it is what it is, and it could even be quite bad), but rather to crowdsource user click behavior to identify pairs that can be used to improve on the current ranking and that are feasible to use as a "gold standard" for evaluation. For each *q*, we assume that the top 10 results are the same for all users. Accumulate the clicks for all users to produce a histogram of click-counts as a function of rank for *q*. Absolute click rates are unreliable due to strong position bias, so we have to adjust for this. E.g., compare observed (biased) click rates with expected click rates (or compare suitably de-biased values for both, but make sure to make an apples-to-apples comparison.) If there is a significant spike relative to what was expected for that position, that could imply that the document in that position should be ranked higher than it is, if it's not already at the top. If there is a significant drop relative to what was expected for that position, that could imply that the document in that position should be ranked lower than it is. This, together with the top 10 documents, allows us to create (d_i, d_j) pairs of interest: For example, if d_2 spikes through the roof compared to d_1, this gives rise to the pair (d_2, d_1). Or if d_2 sinks through the floor compared to d_3, this gives rise to the pair (d_3, d_2).

(b) The basic idea is that we want as many of the preference pairs as possible to be respected by the observed ranking. So we count how many of the preference pairs that are respected by the ranking (X), and how many that are not (Y). So X is the number of "agreements" and Y is the number of "disagreements".

(c) Here we have X = 5 and Y = 1, so K = (5 - 1) / (5 + 1) = 4/6 = 0.667.

(d) The range [-1, 1], endpoints included. A pair either contributes to X or Y (or neither if the document identifiers are mismatched), but not both. So in the extreme we have X + Y = |P| with (X = 0 and Y = |P|) or (X = |P| and Y = 0). Perfect agreement then yields K = (|P| - 0) / |P| = 1, and perfect disagreement yields K = (0 - |P|) / |P| = -1.

SUFFIX ARRAYS

(a) Choice (iv), because 5=eering, 0=engineering, 6=ering, 10=g, 2=gineering, 3=ineering, 8=ing, 4=neering, 9=ng, 1=ngineering, 7=ring.

(b) Choice (ii), false. It is possible to construct the suffix array in $O(n\log n)$ time (or even in linear time), but the naïve approach used in this course is actually $O(n^2 \log n)$: There are indeed $O(n\log n)$ comparisons that take place during sorting, but each comparison is actually $O(n)$ since on average the suffixes we compare have length $n/2$.

(c) Choice (iv), $O(m\log n)$. We can locate the right location in the suffix array with $O(\log n)$ comparisons using a binary search, but each comparison takes $O(m)$ time.

BASIC LINGUISTIC PROCESSING

a) See, e.g., here. The primary goal is to enhance recall, i.e., to properly deal with inflectional forms and sometimes derivationally related forms of a word. E.g., so that *country* will match *countries* and vice versa. Stemming usually refers to a crude heuristic process that chops off the ends of words in the hope of achieving this goal correctly most of the time, and often includes the removal of derivational affixes. Lemmatization usually refers to doing things properly with the use of a vocabulary and morphological analysis of words, normally aiming to remove inflectional endings

only and to return the base or dictionary form of a word, which is known as the lemma. Some points to note include:

i) Both are language-specific.
ii) A stemmer is a set of heuristic rewrite rules, i.e., code. A lemmatizer is basically a dictionary lookup, and requires an as complete as possible dictionary of words and all their different forms. Such sufficiently complete dictionaries can be hard to come by, and for some highly inflected languages (e.g., Finnish) they will usually be very incomplete or too large to practically deal with.
iii) A stemmer does reduction. (For example, *automation* maps to *automat*.) With a lemmatization dictionary you can either do reduction or expansion (For example, we could reduce *running* [or any other form of the word] to *run*, or we could expand *running* [or any other form of the word] to a disjunction over the full set {*run*, *runs*, *ran*, *running*}.)
iv) A stemmed word is not necessarily a real word. A lemmatizer will emit proper words. A stemmer is more likely than a lemmatizer to conflate different words, having different semantics: E.g., *automatic*, *automation* and *automata* might all reduce to the same stem *automat*, thus adversely impacting precision.
v) A lemmatizer can handle strong forms, e.g., all of {*are*, *am*, *is*} would in the dictionary map to *be*.
vi) A lemmatizer can be combined with morphological analysis such as a part-of-speech tagger. That way, we would know if a word like *answer* was used as a noun or a verb in a sentence, and process it as such. This is rarely done in practice, though.

b) Some points to note include:
i) If you do reduction to base form, you must synchronize query processing with content processing. Updates to lemmatization/synonym dictionaries (or the stemming algorithm, but that's exceedingly rare) can be very problematic, and might require a full reindex.
ii) If you do query expansion, you don't do it on the content processing side. The queries will become more complex, and this might have an adverse effect on search performance. Updates to lemmatization/synonym dictionaries are easy, just update the dictionary.
iii) If you do content expansion, you don't do it on the query processing side. Updates to lemmatization/synonym dictionaries can be very problematic, and might require a full reindex. Queries remain simple and search performance can be better. How content expansion happens: Unless the inverted index supports word stacking (i.e., it's a positional index that allows more than one word to have the same position identifier) we cannot serve lemmatized phrases. In that case all the expansions typically get appended to the original text, so at least we retain the ability to do phrase searches over the original text. Content expansion increases the index footprint.
iv) Hybrid setups are possible. It's not uncommon to do content expansion for standard words found in everyday language, but query expansion for dealing with synonyms and domain-specific terms: The latter dictionaries are much more likely to need updates than the former.

c) A crude heuristic process that chops off the ends of words, using language-specific rules. Porter's algorithm consists of five phases of word reductions, applied sequentially. Within each phase there are various conventions to select rules, such as selecting the rule from each rule group that applies to the longest suffix. For example, in the first phase, the suffix *sses* is replaced with *s* (so *caresses* becomes *caress*), the suffix *ies* is replaced with i (so *ponies* become *poni*), and so on.

MIXED GRILL

a) What it is: A Bloom filter is a space-efficient probabilistic data structure for checking set membership. If the filter answers “no” then it is definitely no. If the filter answers “yes” there is a certain probability that it is wrong. The probability of false positives can be controlled by how you design the filter and equals a simple function *f(m, k, n)* where *m*, *k* and *n* are described below. How it works: To be able to store up to *n* membership values, we keep a bit vector of *m* bits for storage and combine this with the use of *k* independent hash functions. Initially, the set is empty and all *m* bits are 0. To add an item to the set, we apply the *k* hash functions to the item to identify up to *k* positions in our bit vectors, and set these bits to 1. To look up an item in the set, we apply the *k* hash functions to the item to identify up to *k* positions in our bit vectors, and answer “yes” if and only if all these bits are 1. Elements can only be added to a vanilla Bloom filter and not deleted, since the same bit might be set by having added different items. (There exists extended versions of Bloom filters that can also handle deletions.)

b) Numerous good examples exist. See, e.g., here or here. Basically, Bloom filters can be used as a quick pre-check to avoid doing work. You might end up doing wasted work (assumed benign and controlled by the filter’s false-positive probability), but won’t miss out on doing necessary work (since the filter’s false-negative probability is zero.) The work we want to avoid needs to be weighed up against the cost of the pre-check. E.g., do I think I need to communicate with a remote machine (and would like to avoid that), or do I think I need to access a slow disk drive (and would like to avoid that), or..?

c) For example, you could encounter dead-end nodes and get stuck. Or, your graph could contain disconnected components, and you could never reach nodes in components other than the component you happened to start in. The end result of this is that there doesn’t exist a steady-state probability distribution, i.e., a unique long-term visit rate for each node. Teleportation makes the Markov chain ergodic, and that ensures that we do actually converge to a steady-state probability distribution where the starting point doesn’t matter. (Ergodicity basically means that it’s possible to get from one state to any other state in a finite number of steps.)

d) If a single document contains the word *foo* 14 times, that document would contribute 14 to the collection frequency count (since that deals with word counts and it occurs 14 times in our document) but only contribute 1 to the document frequency count (since that deals with document counts and we’re talking about a single document.) For example, Zipf’s law uses collection frequency, whereas the IDF part of TF-IDF scoring uses document frequency.

e) The support vectors are the data points that lie the closest to the decision boundary hyperplane. This hyperplane maximizes the margin: That is, the distance between the hyperplane and the support vectors is as large as possible. The support vectors uniquely influence the position and orientation of the hyperplane: If you remove data points from the training set that are not support vectors then the SVM classifier’s hyperplane will remain the same. The SVM training procedure gives us a weight (a Lagrange multiplier) back per training data point, these weights being non-zero only for the support vectors. To use the SVM for classification, we only need the support vectors and their weights, as these alone define the hyperplane.

f) *foo* → [R, R, N, N, R]
bar → [N, R, N, R, R]
AP(*foo*) = 1/3 * (1/1 + 2/2 + 3/5) = 13/15 = 0.867
AP(*bar*) = 1/3 * (1/2 + 2/4 + 3/5) = 8/15 = 0.533
MAP({*foo*, *bar*}) = 1/2 * (AP(*foo*) + AP(*bar*)) = 21/30 = 7/10 = 0.7

FIELDS

a) E.g., see Section 6.1 here. The “fat dictionary” approach, or the “fat postings” approach. Should ideally provide examples/discussion related to, e.g., query performance, index size, operational flexibility, and more: Given the amount of points that are up for grabs, a high-scoring answer is expected to have a well-rounded discussion and display some creativity on applying different facets of what they’ve learnt in the course.
b) E.g., see Section 6.1.1 here, or consider tiering your index based on field importance. Should ideally provide examples/discussion of how to apply the field weights in practice, and how we might best determine what the weight values should be. See also comment above.