Solution sketch
Fall 2023

VECTOR SPACES

(a) Possible discussion points include:

(i) Use, e.g., TF-IDF scoring for each word in the document, to assess the value for that particular dimension. Value will be 0 for words that do not occur in the document, giving rise to a sparse vector. TF-IDF scoring for a word in a document boosts words that occur frequently in the document, and rarely in other documents. Representation (A) is simple, but no context is utilized and we are easily thrown off by superficial term differences (hence we try to address via stemming, synonym dictionaries, and other term normalization techniques), and for really large documents the vectors can grow to a lot of terms and become comparatively large for large documents unless we clip them. For classification tasks, as a rule of thumb we can more often get by with using linear techniques in extremely high-dimensional spaces, since it is often easy to find a separating hyperplane.

(ii) The guiding idea is that "a word is known by the company it keeps": Words that are often mentioned together will often be semantically related somehow, so we try to capture some context by considering word co-occurrence. Several techniques exist for computing embeddings, but we can imagine trying to, e.g., predict a blanked-out word in a sequence with a small and suitably trained neural network and then read out values at internal nodes in this network to produce the embedding. Embeddings for words can be used as building blocks to produce embeddings for longer texts, by forming weighted sums of word embeddings for the words in the document. More specialized techniques exist. Representation (B) always gives us dense vectors of a fixed, predefined size. The important property they have is that distances in embedding space correlate with semantics, e.g., the embedding for "car" would be closer to the embeddings for "ferry" and "airplane" than to the embeddings for "crocodile" and "flour". Doing vector arithmetic (addition and subtraction) and then a NN lookup on the result allows us to do simple reasoning by analogies (e.g., "king - man + woman = queen".)

(b) We have:
dot_product(x, y) = (0.6 * 1.0) + (0.2 * 0.1) + (0.8 * 0.9) = 0.6 + 0.02 + 0.72 = 1.34
length(x) = sqrt($0.6^2 + 0.2^2 + 0.8^2$) = sqrt(0.36 + 0.04 + 0.64) = sqrt(1.04)
length(y) = sqrt($1.0^2 + 0.1^2 + 0.9^2$) = sqrt(1.0 + 0.01 + 0.81) = sqrt(1.82)
cosine(x, y) = dot_product(x, y) / (length(x) * length(y)) = 0.974

(c) In an embedding space, "closeness" corresponds to "semantic similarity", i.e., proximity in embedding space correlates with having similar meaning or being otherwise semantically related. Thus, if we place a query buffer in the same embedding space as a large collection of documents, finding the *k* nearest neighbors (documents) to the query point corresponds to finding the *k* presumably best semantic matches for the query. An ANN index helps us to do exactly this: It's an engine that provides us the ability to do *k*-NN lookups for a given query, and that does so efficiently also when the dimensionality of the vectors grows and particularly also when the number of documents becomes very high. To realize this, the ANN index typically provides options to do this approximately and thus can sacrifice exactness and correctness for speed and performance. Applications built on top of an ANN index are numerous, and include both search and recommendation applications. See also here (and the example implementation.)

(d) See also here. Strategies include:

(i) Brute force. I.e., scan through all the vectors and assess their distances to the query vector as quickly as possible. Typically involves heavy use of SIMD instructions, possibly also GPUs. Brute force only works up to a certain point, so the strategies below all aim to narrow down the need to assess distances to a smaller subset of the vectors. (Within such a subset, we can apply brute force techniques.)

(ii) Tree-based algorithms: Build one or more trees by recursively partitioning the embedding space, until you have “few enough” vectors in each leaf node. To query, for each tree in the forest, traverse it guided by your query vector, and only compute distances to the vectors in the leaf node you arrive at.

(iii) Locality-sensitive hashing: Apply one or more hash functions to each vector to bucket them, where each hash function is designed such that if the distance *d(x, y)* is small then the probability of *x* and *y* being hashed to the same bucket is high. To query, apply each hash function to the query vector and only compute distances to the vectors in the buckets that the query vector hashes to.

(iv) Clustering-based algorithms, or quantization: Recode (cluster) the vectors to reduce the size of the dataset, and replace each vector with a leaner, approximate and quantized representation. To query, process this reduced and simplified data set. Can be combined with other strategies.

(v) Graph-based algorithms: For example, in HNSW, create a tower of graphs with successively more detail at each layer (similar to a hierarchy of skip lists.) The graph at each layer has nodes that represent vectors, and edges that represent vectors being nearby. To query, we start at some entry point and iteratively traverse the graph. At each step of the traversal, we examine the distances from a query to the neighbors of a current base node and then select as the next base node the adjacent node that minimizes the distance, while constantly keeping track of the best-discovered neighbors.

MEASURING RELEVANCE

(a) The $F_{\square}$-score is the (weighted) harmonic mean of precision and recall, i.e., it is a measure that collapses precision and recall into a single number. See here. Ideally we would like to have both great precision and recall, but might have to compromise. A measure that combines the two is thus often convenient to use. We can choose how much to emphasize precision versus recall, controlled by the parameter □. With □ = 1 we place equal weight on precision and recall, which mathematically works out to F_1 = 1 / (0.5 * (1 / P) + 0.5 * (1 / R)) = (2PR) / (P + R). In our case, we then have F_1 = (2 * 0.1 * 0.5) / (0.1 + 0.5) = 1/6 = 0.167.

(b) We have a ranked retrieval context, so we consider a ranked result set where we consider the top *k* results. For each choice of *k*, starting at the top and working ourselves down the list, we can calculate precision and recall among the top *k* results. This gives rise to a precision-recall curve, where we plot the recall values on the *x*-axis, and the corresponding precision values on the *y*-axis. This curve tells us how precision typically falls as a function of increasing recall. The precision-recall curve is bit sawtooth-like, and an interpolated precision-recall curve removes these by giving the curve a smoother staircase look. It does so by “interpolating” the precision values, by always drawing the maximum precision value to the right in the curve, i.e., P(R) = max(P(R’) | R’ ≥ R). See here.

(c) We have two queries *foo* and *bar*, so MAP is the arithmetic mean of AP(*carrot*) and AP(*chocolate*). To compute the average precision AP(*carrot*) we compute the precision *at the positions of the relevant documents*, and average these values. Similarly for AP(*chocolate*). This gives us:
AP(*carrot*) = 1/4 * (P@1 + P@2 + P@4 + P@8) = 26/32 = 0.8125

AP(*chocolate*) = 1/3 * (P@1 + P@3 + P@4) = 29/36 = 0.8055
MAP = 1/2 * (AP(*carrot*) + AP(*chocolate*)) = 466/576 = 0.8090

(d) The list *L* and a given pair in *P* might either be in agreement, or in disagreement. Considering all pairs in *P*, we simply count the number of agreements (X) and the number of disagreements (Y), and then consider their difference (X - Y) as a fraction of the total number of pairs (X + Y). This gives us a number in the range [-1, 1] where the extremes signify either perfect disagreement or perfect agreement. In our case, we then have 5 agreements and 1 disagreement, arriving at the score (5 - 1) / (5 + 1) = 4/6 = 0.667.

MIXED GRILL WITH CARROTS

(a) See, e.g., here. For the estimated prior probabilities we have:
Pr(*healthy*) = 3/4 = 0.75
Pr(*unhealthy*) = 1/4 = 0.25
For the estimated smoothed conditional probabilities we have:
Pr(*carrot* | *healthy*) = (5 + 1) / (8 + 6) = 6/14 = 3/7 = 0.429
Pr(*toffee* | *healthy*) = Pr(*jellybean* | *healthy*) = (0 + 1) / (8 + 6) = 1/14 = 0.071
Pr(*carrot* | *unhealthy*) = (1 + 1) / (3 + 6) = 2/9 = 0.222
Pr(*toffee* | *unhealthy*) = Pr(*jellybean* | *unhealthy*) = (1 + 1) / (3 + 6) = 2/9 = 0.222
We then get:
Pr(*healthy* | *d*) ∝ 3/4 * $(3/7)^3$ * 1/14 * 1/14 ≈ 0.0003
Pr(*unhealthy* | *d*) ∝ 1/4 * $(2/9)^3$ * 2/9 * 2/9 ≈ 0.0001
Conditional probabilities for some words are not shown, since these words do not appear in *d* and thus do not appear in the last two expressions.

(b) There are 6 unique words and we thus have 6 posting lists. Let (*x*, *y*) be shorthand for a logical posting {*document_id*: *x*, *term_frequency*: *y*}. We then have:
carrot → [(1, 2), (2, 2), (3, 1), (4, 1)]
broccoli → [(1, 1)]
spinach → [(2, 1)]
mango → [(3, 1)]
toffee → [(4, 1)]
jellybean → [(4, 1)]

(c) We have:
  (i) The posting list for *carrot* contains 8 integers in total. Gap-encoding or not, all the values are below 128 so a single byte per integer will suffice, for a total of 8 bytes. See here.
  (ii) With gap-encoding of the *document_id* field we are compressing [(1, 2), (1, 2), (1, 1), (1, 1)]. The integer value 1 can be gamma-encoded using a single bit and occurs 6 times, and the integer value 2 can be gamma-encoded using 3 bits and occurs twice. See here. We can thus compress the entire posting list using only 12 bits.

(d) We have:
  (i) Only listing the positions of the 1-bits, we have:
  Before inserting anything: [], i.e., no bits are set.
  After inserting *carrot*: [7, 12, 15], i.e., 3 bits are set.
  After inserting *carrot* and *toffee:* [0, 3, 7, 12, 15], i.e., 5 bits are set.
  (ii) Querying for *carrot* will say “member” since all bits at positions 7, 12, and 15 are set. Querying for *steak* will say “not member” since the bit at position 11 is not set. (Bits at positions 7, 11, and 15 would all have to be set for the filter to say “member” for *steak*.)
  (iii) To reduce the probability of false positives, add more storage bits and/or tweak the number of hash functions. Deciding on a false positive rate that we accept, we can

compute exactly how many more storage bits and/or hash functions we need to have to achieve the desired false positive rate.

APROXXIMAT MATHCING

(a) See, e.g., here (and the example implementation.) Two key insights include:
   (i) If two strings share a prefix of length N, the first N columns of their edit tables against *q* will be the same and can be reused.
   (ii) A trie organizes the strings by prefixes, so as soon as the edit distance exceeds *k* we can abort our search/exploration along that branch and effectively prune down our search space.

(b) We can keep the exact same search algorithm as above! But (i) extend the trie data structure slightly, if needed, to hold metadata along with each final state, and (ii) add some pre- and post-processing. Note that (i) is an enabler for (ii). For example, we could then:
- To index: Instead of storing all the original strings in *D* in the trie (e.g., *richards*), we store their phonetic hashes (e.g., *R263*). With each final state in this trie we add metadata that allows us to trace back to the original string (e.g., *richards*) either through the original string itself or some key that allows us to look it up in an external data structure.
- To query: Compute the phonetic hash of *q* (e.g., *lichardson* might yield *L263*.) Then find matches that are within *k* (a very small value, e.g., *k* = 1) edits of *L263* (e.g., *R263*), using the algorithm from (a). Utilize the metadata in the final state for *R263* to report back the original string(s) that map to *R263*, e.g., *richards*.