CLASSIFICATION

a) The support vectors are the points/examples in the training set that are the closest to the SVM's decision boundary. The weight vector $\boldsymbol{w}$ is the sum of $a_i y_i \boldsymbol{x}_i$ for all training examples $\boldsymbol{x}_i$ where a_i are the Lagrange multipliers produced by the SVM training procedure and y_i is the training example's label. As it turns out, we have $a_i = 0$ for all training examples except for the support vectors. Hence, we only need to know the support vectors and their corresponding Lagrange multipliers in order to compute the weights $\boldsymbol{w}$. The weight vector is the vector that is normal/perpendicular to the linear SVM's decision boundary/plane.

b) (1) Keep the SVM linear but introduce slack variables. I.e., we compute the best linear plane that minimizes the cost of misclassifications. And misclassifications will occur since the problem is not linearly separable. We can tweak and tune how much cost we assign to misclassifications versus margin width.
(2) Introduce some suitable non-linear kernel function. This means "lifting" the data points into a higher-dimensional space such that the problem becomes linearly separable in that higher-dimensional space. The kernel function deals with the inner product in this space, we don't actually have to preprocess/transform each data point.

c) Assessing a model's performance on the same data with which it was trained is cheating. We therefore randomly split our pool of data into a training set and a testing set. But such a particular split could be very "lucky" or "unlucky" depending on how difficult cases are distributed in your split, and we don't know how reliable our performance/quality assessment is . With *k*-fold CV we take a systematic approach to this, to try to average out the "luck" part mentioned above: We randomly split the pool of data into *k* disjoint blocks of (approximately) equal size, and then do *k* iterations: In iteration *i* we assign block *i* as the testing set and the union of the remaining blocks as the training set. This gives us some performance/quality metric of the model for iteration *i*. By doing this *k* times and averaging our metric, we arrive at a more correct and reliable performance/quality assessment than for just one random split, and the spread of metric values across iterations also gives us a sense of its variability and uncertainty. The higher the choice of *k*, the more "correct" our assessment gets. Note the extreme case where *k* equals the number of examples we have in our pool of data. This is called leave-one-out CV.

EVALUATION

a) Assuming binary relevance, let RET and REL denote the sets of retrieved and relevant documents, respectively. Precision is defined as the ratio $|RET \cap REL| / |RET|$, i.e., the number of relevant documents that were retrieved divided by the total number of documents retrieved. Recall is defined as the ratio $|RET \cap REL| / |REL|$, i.e., the number of relevant documents that were retrieved divided by the total number of relevant documents. A situation where you'd prioritize recall is if there's a big cost of missing out on a relevant document, e.g., imagine a lawyer doing research and preparing for a case that requires that he has a complete overview of all relevant case law. A situation where you'd prioritize precision is if the presence of an irrelevant document has a big cost, e.g., imagine

that there is only a single relevant document to your query and that any irrelevant document thus constitutes "noise".

b) An example of a feature designed to increase recall would be, e.g., lemmatization: By capturing more variants of the same search term (e.g., a search for "car" matches both "car" and "cars") we increase RET and thus the factor |RET∩REL| will hopefully increase. (The denominator |REL| is a constant for a given query.) An example of a feature designed to increase precision would be a feature that conjunctively added good disambiguation terms to the query (e.g., a search for "jaguar" would be expanded to "jaguar" AND "animal"). Assuming that I am looking for information about the animal and not the car, we'd reduce the denominator |RET| while hopefully keeping the counter |RET∩REL| constant.

c) Ideally we'd want to have both high precision and high recall, but often we have to strike a compromise between the two. Yet, it's often convenient to have a metric that captures both. The *F*-score is a metric that combines precision (*P*) and recall (*R*) into a single number, and is defined as the weighted harmonic mean of the two: $F(\alpha) = 1 / [(1 - \alpha)(1/P) + \alpha(1/R)]$, where α determines the relative emphasis placed on *P* and *R*, respectively. This expression is sometimes rewritten into an alternate and equivalent form $F(\beta) = (1 + \beta^2)(PR / (\beta^2 P + R))$ where $\alpha = 1 / (1 + \beta^2)$. The choice of value $\alpha = 0.5$ is equivalent to the choice of $\beta = 1$.

d) MAP assesses the quality of a retrieval algorithm that returns ranked result sets, computed over a set of queries. Binary relevance judgments are assumed. To compute the average precision (AP) value for a query, compute the average value of *P@k* over all positions *k* where a relevant document is found. To compute the mean average precision (MAP) for a set of queries, compute the mean AP value over all queries.

e) NDCG allows us to go beyond binary relevance judgments and quantify the quality of a ranked result set . Each document has a gain (G) score, e.g., 0 for an irrelevant document, 2 for a mildly relevant document, 8 for a very relevant document, and so on. The cumulative gain (CG) across a result set would be the sum of gains over the result set. However, we want to discount the gains so that we reward high gains occuring at the top of the ranked result set, i.e., we want position 1 to account for more than position 2, and so on. Typically, the discounting profile we apply is logarithmic. If we discount the gains as we compute the sum we get the DCG measure. To get the NDCG we normalize DCG by the DCG score of the "perfect" ordering, i.e., the ordering where all high-gain documents come before all lower-gain documents. The NDCG measure can be fooled and artificially inflated by having duplicates of high-gain documents in the result set, a situation which would not contribute positively to perceived user value.

STRINGS

a) The suffix array is the array of integer indices into the string, not the implied substrings/suffixes of the string. For *hakkebakke* we have [6, 1, 5, 9, 4, 0, 8, 3, 7, 2]. Students should show how this is constructed. To search for *ak*, do a binary search to locate where in this array we might insert *ak* and then scan forward from there until entries no longer start with *ak*. If we just wanted to find out how many occurrences of *ak* there are and we had reason to suspect that there are lots, we could do two binary searches (to identify the start

and the end) instead of a binary search (to identify the start) and a scan (to identify the end.)

b) To see how a suffix array can help us, let's construct the suffix array of the three example strings, here showing the suffixes and not the string indices and with a magic delimiter symbol [$ < # < @ < a] added to indicate from which of the three substrings the suffix originates:

a$
a@
aaababca@
aababc#
aababca@
abababca$
ababc#
***ababc**a$*
***ababc**a@*
abc#
abca$
abca@
bababca$
babc#
babca$
babca@
bc#
bca$
bca@
c#
ca$
ca@

The longest common substring corresponds to the longest common prefix of the three suffixes *ababca$*, *ababc#* and *ababca@*, indicated in bold above. We can see that the solution is a sequence [*i*, .., *j*] in the array of sorted suffixes, with the property that it contains at least one suffix from every string and the longest common prefix of the first and last suffix in the suffix is maximum. We can locate such a longest common prefix by a scan through the array.

HEAPS' LAW

a) See 2014 exam.

b) See 2014 exam. In log-log space Heap's law is a straight line. We are given data for two points, which is sufficient information to deduce the line's slope and intercept and thus the values *k=30* and *b=0.5*. Students should show how to do this, not simply guess or magically

stipulate values for k and b. With k and b in hand and the given value of T, we can easily compute that M = 60 million.

POTPOURRI

a) In Rocchio relevance feedback the user submits an original query q_{orig}, which generates some results of which the user point to some being relevant and some being irrelevant. (Think of the relevant ones indicating "more like these" and the irrelevant ones indicating "less like these". The relevant/irrelevant feedback sets D_{rel} and D_{irrel} can be empty.) To form a refined query that will hopefully yield even better results than the original query, we first compute the centroids (averages) of the vectors for D_{rel} and D_{irrel}. Denoting these centroids as c_{rel} and c_{irrel}, respectively, the refined and improved query q_{ref} is then computed as the weighted vector sum $q_{ref} = \alpha q_{orig} + \beta c_{rel} - \gamma c_{irrel}$. We clip negative values in q_{ref}, if any, to 0. We can select and tweak the weights α, β *and* γ to suit our application needs.

b) A Bloom filter is a probabilistic data structure for checking set membership for a set having n elements: We keep a bit vector with m bits (all initially false) and have k independent hash functions h_i. To add an element x to the filter, we set all bits $h_i(x)$ to true. To check if an element y is a set member, we check if all k bits $h_i(y)$ are set to true. If the Bloom filter says "not member", we can be sure that this is correct. I.e., the Bloom filter has no false negatives. If the Bloom filter says "member", we can only be sure that this is correct with some probability. I.e., the Bloom filter might have false positives. The probability of a false positive is a simple function $f(n, m, k)$. Knowing n, we can choose values of m and k such that the false positive probability is "acceptable" for our application needs. In a traditional Bloom filter we can only add elements, not delete elements. Extensions exist (e.g., cuckoo filters or counting Bloom filters) that support deletions.