Chair for Algorithms
and Data Structures
Prof. Dr. Hannah Bast
Marjan Celikik

Search Engines WS 09/10

http://ad.informatik.uni-freiburg.de/teaching

UNI FREIBURG

Final Exam — Solutions

Solution for Task 1 (Ranking)

1.1 There are twelve different words and 5 documents. Here is the 12×5 term-document matrix. The entries are all zero or one, since each document contains each word at most once.

every	1	1	1	1	0
breath	1	0	0	0	0
you	1	1	1	1	1
take	1	0	0	1	0
move	0	1	0	0	0
make	0	1	0	0	0
bond	0	0	1	0	0
break	0	0	1	0	0
step	0	0	0	1	0
Ill	0	0	0	0	1
be	0	0	0	0	1
watching	0	0	0	0	1

1.2 Each document as well as the query have four non-zero entries which are all 1. By L_2-normalization these all becomes 0.5 (because $4 \cdot 0.5^2 = 1$). Here is the normalized matrix with the normalized query vector.

every	0.5	0.5	0.5	0.5	0	0.5
breath	0.5	0	0	0	0	0
you	0.5	0.5	0.5	0.5	0.5	0.5
take	0.5	0	0	0.5	0	0
move	0	0.5	0	0	0	0
make	0	0.5	0	0	0	0.5
bond	0	0	0.5	0	0	0.5
break	0	0	0.5	0	0	0
step	0	0	0	0.5	0	0
Ill	0	0	0	0	0.5	0
be	0	0	0	0	0.5	0
watching	0	0	0	0	0.5	0

The cosine similarities with documents 1, 2, 3, 4, and 5 are then 0.5, 0.75, 0.75, 0.5, and 0.25, respectively. With ties broken by doc id, we then obtain the document ranking $2, 3, 1, 4, 5$.

1.3 If all five documents are presented to the user the precision is 80% (4 documents out of the five shown are relevant) and the recall is 100% (all 4 relevant documents are in the ones shown). The F-measure is therefore $1/(5/4+1) = 8/9 = 88.89\%$. Recall 50% is achieved already by the first two hits (2 out of 4 relevant documents found), and the precision there is 100% because both of the these hits are relevant.

1.4 By doubling each word occurrence in each document, all entries in the term-document matrix get multiplied by 2. After L_2-normalization we obtain the exact same matrix as if we hadn't doubled each word. Therefore the cosine scores and the ranking for each query remain the same.

Solution for Task 2 (Compression / Entropy)

2.1 By simple counting we get $n_a = 2$, $n_b = 4$, $n_c = 8$, and $n_d = 2$. The corresponding relative frequencies are $n_a/n = 1/8$, $n_b/n = 1/4$, $n_c/n = 1/2$, and $n_d/n = 1/8$.

2.2 The entropy of the distribution is

$$1/8 \cdot \log_2 8 + 1/4 \cdot \log_2 4 + 1/2 \cdot \log_2 2 + 1/8 \cdot \log_2 8 = 3/8 + 2/4 + 1/2 + 3/8 = 7/4 = 1.75.$$

2.3 A prefix-free code would be $a : 110$, $b : 10$, $c : 0$, $d : 111$. Since the length of the code for letter σ is exactly $\log_2(n/n_\sigma)$, the expected code length is exactly the entropy computed in 2.2, that is, 1.75.

2.4 Let p_σ be the probability for letter σ. The probability of the string above being generated with these probabilities is $p_a^2 \cdot p_b^4 \cdot p_c^8 \cdot p_d^2$, and taking the logarithm we obtain the log likelihood $L = 2\ln p_a + 4\ln p_b + 8\ln p_c + 2\ln p_d$. We want to maximize L under the side constraint $p_a + p_b + p_c + p_d = 1$.

Writing the side constraint as $1 - p_a - p_b - p_c - p_d = 0$ and adding the LHS with a factor of λ to L we obtain:

$$2\ln p_a + 4\ln p_b + 8\ln p_c + 2\ln p_d + \lambda \cdot (1 - p_a - p_b - p_c - p_d).$$

Setting the partial derivatives with respect to each of the p_σ to zero, we obtain that $p_a/2 = p_b/4 = p_c/8 = p_d/2 = 1/\lambda$. Together with the side constraint this gives us $\lambda = n = 16$, which in turn gives us $p_\sigma = n_\sigma/n$, as claimed.

Solution for Task 3 (List intersection)

3.1 Here is the method in C++:

```
void intersect(const vector<int>& A, const vector<int>& B, vector<int>* C) {
  assert(C != NULL);
  assert(C->size() == 0);
  size_t i = 0;
```

```
  size_t j = 0;
  while (i < A.size() && j < B.size()) {
    if (A[i] == B[j]) { C.push_back(A[i]); ++i; ++j; }
    else if (A[i] < B[j]) { ++i; }
    else { ++j; }
  }
}
```

3.2 Here is an example of lists with skip pointers, where the skip pointers help a lot. For this example, $N = 100$, and $m = 2$. The skip pointer array is the two numbers given at the beginning of each line.

```
A:  0 1    57 98
B:  0 9    1 2 3 4 5 6 7 8 9
```

We start with the first element from list A. Skip pointer 1 for B tells us that the smallest element in list B that is $1 \cdot 100/2 = 50$ is at position 9, that is, one after the last element of the list, that is, there is no element in list B which is ≥ 50, and hence we know already at this point that the intersection must be empty.

3.3 Here is an example of lists with skip pointers, where the skip pointers do not help at all. Again, $N = 100$, and $m = 2$, and the skip pointer array is the two numbers given at the beginning of each line.

```
A:  0 4    10 20 30 40 50 60 70 80 90
B:  0 4    10 20 30 40 50 60 70 80 90
```

The only point where the skip pointers might help is when we reach element 50 in the first list. Then skip pointer 1 from list B tells us that we can skip to the element with value 50 in list B. But at that point we have reached the element just one before (with value 40) already anyway, and so we don't skip anything here.

3.4 Here is the method in C++:

```
void intersectWithSkipPointers(const vector<int>& A, const vector<int>& B,
                               const vector<int>& AS, const vector<int>& BS,
                               vector<int>* C) {
  assert(result != NULL);
  assert(result.size() == 0);
  size_t i = 0;
  size_t j = 0;
  size_t k = 0;
  size_t is = 0;
  size_t js = 0;
  while (i < A.size() && j < B.size()) {
    if (A[i] == B[j]) { C.push_back(A[i]; ++i; ++j; }
```

```
    else if (A[i] < B[j]) {
      ++i;
      while (is + 1 < AS.size() && A[AS[is+1]] <= B[j]) { ++is; }
      if (AS[is] > i) { i = AS[is]; }
    }
    else {
      ++j;
      while (js + 1 < BS.size() && B[BS[js+1]] <= A[i]) { ++js; }
      if (BS[js] > j) { j = BS[js]; }
    }
  }
}
```

Solution for Task 4 (Edit distance)

4.1 Here is the dynamic programming table, filled using the trivial equality that $ED(\varepsilon, z) = ED(z, \varepsilon) = |z|$ and the recursion from the lecture. According to the table, the edit distance between *orange* and *grape* is 3.

	ε	g	r	a	p	e
ε	0	1	2	3	4	5
o	1	1	2	3	4	5
r	2	2	1	2	3	4
a	3	3	2	1	2	3
n	4	4	3	2	2	3
g	5	4	4	3	3	3
e	6	5	5	4	4	**3**

4.2 The arrows in the table above show from which previous value a value may be derived in the recursion. According to these arrows, there are *two* different optimal paths from the upper left to the lower right. These paths are:

1. $R(1, g)$, $R(4, p)$, $D(5)$
2. $R(1, g)$, $D(4)$, $R(4, p)$

Here $R(i, x)$ means replace the character at position i by x, $I(i, x)$ means insert character x just after position i, and $D(i)$ means delete the character at position i.

4.3 Let T_1 be a sequence of $ED(x_1, y_1)$ transformations that transform x_1 into y_1. Let T_2 be a sequence of $ED(x_2, y_2)$ transformations that transform x_2 into y_2. Then T_1 will also transform $x_1 y_1$ into $x_2 y_1$, and then applying T_2 with all positions shifted to the right by $|x_2|$ will transform

x_2y_1 into x_2y_2. We have thus constructed a sequence of $ED(x_1, y_2) + ED(x_2, y_2)$ transformations that transform x_1y_1 into x_2y_2 and hence $ED(x_1y_1, x_2y_2) \leq ED(x_1, y_1) + ED(x_2, y_2)$ (there could be an even better sequence, and, as we show in 4.4, there indeed sometimes is).

4.4 Let z be an arbitrary string of length at least 1. Let $x_1 = z$ and $x_2 = \varepsilon$ and $y_1 = \varepsilon$ and $y_2 = z$. Then $ED(x_1, y_1) = |x_1| = |z|$ and $ED(x_2, y_2) = |y_2| = |z|$ and the sum of the two is $2|z|$. On the other hand x_1y_1 and x_2y_2 are both exactly z and so $ED(x_1y_1, x_2y_2) = 0$, which for non-empty z is strictly less than $2|z|$.

Solution for Task 5 (Classification)

5.1 By simple counting we get $\Pr(W = a|C = A) = 10/15 = 2/3$, $\Pr(W = b|C = A) = 5/15 = 1/3$, $\Pr(W = a|C = B) = 6/18 = 1/3$, $\Pr(W = b|C = B) = 12/18 = 2/3$, $\Pr(C = A) = 3/6 = 1/2$, and $\Pr(C = B) = 3/6 = 1/2$.

5.2 For a given record, we predict that class c which maximizes $\Pr(C = c) \cdot \prod_w \Pr(W = w|C = c)$. For the string *aaa* this is $1/2 \cdot (2/3)^3$ for class A and $1/2 \cdot (1/3)^3$ for class B and so the prediction is A. For the string *bbb* this is $1/2 \cdot (1/3)^3$ for class A and $1/2 \cdot (2/3)^3$ for class B and so the prediction is A.

5.3 Consider an arbitrary string and let n_a the number of as in it, and n_b the number of bs. Then $\Pr(C = c) \cdot \prod \Pr(W = w|C = c)$ is $p_A := 1/2 \cdot (2/3)^{n_a} \cdot (1/3)^{n_b}$ for class A and $p_B := 1/2 \cdot (1/3)^{n_a} \cdot (2/3)^{n_b}$ for class B. Now $p_A/p_B = 2^{n_a}/2^{n_b}$ and this is > 1 if $n_a > n_b$ and < 1 if $n_a < n_b$ (and $= 1$ if $n_a = n_b$, that is, in that case Naive Bayes does not help tp decide between A and B).

5.4 We observe that for the values of the priors in 5.1 it does not matter how the letters are distributed over the strings that are assigned to class A (and the same holds for class B), it is just the counts that matter. Therefore our priors remain exactly the same if we replace the first string *aba* by *abb* and the second string *baabaaa* by *aaabaaa*. With the same priors, the statement from 5.3 also holds unchanged, and so, based on these priors, *abb* will be predicted to be from class B, although in the training set it was assigned to class A.

Solution for Task 6 (Clustering)

6.1 In the first round, we start with cluster centroids 1 and 21. The points closer to 1 are $1, 4, 7, 8$ and so they will form a cluster in the first round. The points closer to 21 are $12, 15, 21$ and so they will form the other cluster in the first round. Recomputing the cluster centroids gives us $(1 + 4 + 7 + 8)/4 = 5$ and $(12 + 15 + 21)/3 = 16$.

On the second round, we hence start with cluster centroids 5 and 16. The points closer to 5 are again $1, 4, 7, 8$, and the points closer to 16 are again $12, 15, 21$. This gives the same clusters as in the first round, and so a recomputation of the cluster centroids gives again 5 and 16.

Any round after this one will give the same clusters and centroids.

6.2 We denote our n points by $x_1, \ldots, x_n$ and without loss of generality we assume that they are in sorted ascending order, that is, $x_1 \leq x_2 \leq \cdots \leq x_n$. The $n - 1$ differences which are the

basis for ONE-DEE's clustering decision are then $d_i = x_{i+1} - x_i$, for $i = 1, \ldots, n-1$. Now if we multiply each x_i by the same constant factor α, then all differences d_i also get multiplied by that same constant factor (simply because $\alpha \cdot x_{i+1} - \alpha \cdot x_i = \alpha \cdot (x_{i+1} - x_i)$). Therefore the set of i of the k largest d_i does not change, and hence we obtain the exact same clustering.

*N*ote: We tacitly assumed here that in the case of ties (equal differences) we break the tie in a way that does not depend on the absolute value of the d_i. A natural way to break ties would be to prefer the difference with smaller id (that is, the difference of the leftmost points), and that rule indeed does not depend on the value of the differences.

6.3 Observe that by the way ONE-DEE works, whenever two points x and y are in the same cluster, than also all points with values in the interval $[x, y]$ are in the same cluster. This implies that *not* all partitionings are possible. For example, consider the input sequence $1, 2, 3$. Then there is no way that we get the clustering $\{\, \{1, 3\}, \{2\} \,\}$.

*N*ote: This depends on how we assume the input is given. If we demand the input is given in sorted order, than the above argument is correct. If the input can be given in any order, than ONE-DEE in fact satisfies richness. For the example above, if we want the clustering $\{x_1, x_3\}$ and $\{x_2\}$, we can just set $x_1 = 1$, $x_3 = 2$, and $x_2 = 5$.

6.4 Here is the method in C++. Note that the number of clusters is implicitly given by the number of vectors in the result vector of vectors (the clusters).

```
void oneDeeClustering(const vector<int>& X, vector<vector<int> >* result) {
  assert(clusters != NULL);
  int k = clusters.size();
  int n = X.size();
  assert(n > 0);
  // Compute the n-1 differences and sort them.
  vector<int> diffs(n - 1);
  sort(X.begin(), X.end());
  for (int i = 0; i < n - 1; ++i) { diffs[i] = X[i+1] - X[i]; }
  sort(diffs.begin(), diffs.end());
  // Now scan the points from left to right and fill the result.
  int j = 0;
  for (int i = 0; i < n; ++i) {
    if (j > 0 && j + 1 < k && i > 0 && X[i] - X[i-1] >= diffs[k-1]) { ++j; }
    (*result)[j].push_back(X[i]);
  }
}
```