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

Mid-Term Exam — Solutions

Solution for Task 1 (Entropy)

1.1 For B, the sum of the probabilities is

$$\Pr(B = 0) + \Pr(B = 1) = p + q = 1.$$

For BB, the sum of the probabilities is

$$\Pr(B = 00) + \Pr(B = 01) + \Pr(B = 10) + \Pr(B = 11) = p^2 + 2pq + q^2 = (p + q)^2 = 1,$$

where the second equality is just the binomial formula.

1.2 Let us write log instead of $\log_2$ for better readability. For H_B, we have

$$H_B = -p \cdot \log p - q \cdot \log q.$$

For H_{BB}, we have

$$\begin{aligned} H_BB &= -p^2 \cdot \log(p^2) - 2pq \cdot \log(pq) - q^2 \cdot \log(q^2) \\ &= -2p^2 \log p - 2pq \cdot \log p - 2pq \cdot \log q - 2q^2 \log q \\ &= -2p(p + q) \log p - 2q(p + q) \log q \\ &= 2 \cdot H_B. \end{aligned}$$

1.3 An example of a prefix-free code with the given code lengths would be

$$\begin{aligned} 00 &\rightarrow 0 \\ 01 &\rightarrow 001 \\ 10 &\rightarrow 010 \\ 11 &\rightarrow 0111 \end{aligned}$$

1.4 The expected code length for a single instance of BB is

$$3/4 \cdot 3/4 \cdot 1 + 2 \cdot 3/4 \cdot 1/4 \cdot 3 + 1/4 \cdot 1/4 \cdot 4 = (9 + 18 + 4)/16 = 31/16.$$

For a sequence of length n, we need to generate $n/2$ such codes (let's assume that n is even), hence the expected code length for such a sequence is $n/2 \cdot 31/16 = 31/32 \cdot n$, which is just a bit smaller than n.

Solution for Task 2 (List union and intersection)

2.1 When the two sets are disjoint, the union has size $n + m$ and it can't get any larger than that. Also, when the two set are disjoint, the intersection has size 0, and it obviously can't get any smaller than that.

Several people wrote that the intersection is smallest, when one of the sets is included in the other, and the minimal size is therefore $\min\{n, m\}$. This is wrong.

2.2 You have to do an exponential search, followed by a binary search here. Just a binary search is not good enough, because that will give you time complexity $O(\log |L|)$ and not $O(\log r)$. Here is the function in C++:

```
int rank(int x, const vector<int>& L)
{
  // First do an exponential search.
  int i = 0;
  while (i < L.size() && L[i] <= x) { i = 2 * i + 1; }
  // Then do a binary search.
  int l = 0;
  int r = i < L.size() ? i : L.size();
  while (l < r) {
    int m = (l + r) / 2;
    if (L[m] > x) { r = m - 1; } else { l = m; }
  }
  assert(l == r);
  return r;
}
```

A common mistake in the binary search was to stop it once we have `L[m] == x`. That is not correct, because there might be more elements `== x` to the right of `m` and we need to know how many in order to compute the rank.

2.3 Let ℓ_{first} and ℓ_{last} be the first and last element of L_1, respectively. Then all elements of L_1 are strictly inbetween two consecutive elements of L_2 (including the border cases) if and only if the ranks $\text{rank}(\ell_{\text{first}}, L_2)$ and $\text{rank}(\ell_{\text{last}}, L_2)$ are equal. This leads to the following simple function:

```
bool checkIfIntersectionIsEmpty(const vector<int>& L1, const vector<int>& L2)
{
  if (L1.size() == 0) return true;
  return rank(L1[0], L2) == rank(L1[L1.size() - 1], L2);
```

```
}
```

Note that this function also deals with the border cases correctly. If all elements of L_1 are strictly smaller than all elements of L_2, then both ranks will be zero. If all elements of L_1 are strictly larger than all elements of L_2, then both ranks will be $|L_2|$.

2.4 The probability that a fixed integer from $\{1, \ldots, N\}$ is present in the first list is n/N, and for the second list that probability is m/N. Therefore the probability that a fixed integer from $\{1, \ldots, N\}$ is present in both lists is $n/N \cdot m/N$. Hence the expected size of the intersection is $N \cdot n/N \cdot m/N = (n \cdot m)/N$.

Solution for Task 3 (k-grams)

3.1 The number of k-grams of a word x is $|x| + k - 1$. For example, for the word *exam* the $4 - 2 + 1 = 3$ 2-grams are $\{ex, xa, am\}$. For the word *midterm* the $7 - 4 + 1 = 6$ 5-grams are $\{midt, idte, dter, term\}$. The function is straightforward, here it is in C++:

```
void computeKgrams(const string& x, int k, vector<string>* kgrams)
{
  for (int i = 0; i + k < x.size(); ++i)
    kgrams->push_back(x.substr(i, k));
}
```

3.2 We have that $|A \cup B| = |A| + |B| - |A \cap B| = |x| + k - 1 + |Y| + k - 1 - \ell = |x| + |y| + 2k - 2 + \ell$. Hence

$$J(x, y) = \ell/(|x| + |y| - 2k + 2 + \ell.$$

The function is then straightforward, here it is in C++:

```
int jaccardDistance(const string& x, const string& y, int k, int l)
{
  return l / (x.size() + y.size() - 2*k + 2 + l);
}
```

3.3 Here is the function in C++. Note that, as already done above, the return type is void and the result is instead passed as a pointer argument. This is a standard convention that avoids potentially very expensive copy operations when returning the result (the returned result would have to be copied from a local variable to the variable of the calling function).

```
void buildKgramIndex(const vector<string>& vocabulary, int k,
                     hash_map<string, vector<string> >* index)
```

```
{
  for (int i = 0; i < vocabulary.size(); ++i)
  {
    vector<string> kgrams;
    const string& word = vocabulary[i];
    computeKGrams(word, k, &kgrams);
    for (int j = 0; j < kgrams.size(); ++j)
      index[kgrams[j]].push_back(word);
  }
}
```

3.4

Here is the function in C++.

```
int jaccardDistance(const string& x, const string& y,
                    int k, const hash_map<string, vector<string> >& kgramIndex)
{
  vector<string> xKgrams;
  computeKgrams(x, k, xKgrams);
  int l = 0;
  for (int i = 0; i < xKgrams.size(); ++i)
  {
    const vector<string>& words = kgramIndex[xKgrams[i]];
    assert(words.size() > 0);
    for (int j = 0; j < words.size(); ++j) if (words[j] == y) ++l;
  }
  return l;
}
```

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 *manner* and *banana* is 4.

	ε	b	a	n	a	n	a
ε	0	1	2	3	4	5	6
m	1	1	2	3	4	5	6
a	2	2	1	2	3	4	5
n	3	3	2	1	2	3	4
n	4	4	3	2	2	2	3
e	5	5	4	3	3	3	3
r	6	6	5	4	4	4	**4**

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 three different optimal paths from the upper left to the lower right. These paths are:

1. $R(1, b)$, $R(4, a)$, $R(5, n)$, $R(6, a)$
2. $R(1, b)$, $I(3, a)$, $R(6, a)$, $D(7)$
3. $R(1, b)$, $I(3, a)$, $D(6)$, $R(6, a)$

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 One way to see this is that each $R(i, x)$ can be replaced by a $I(i, x)$ immediately followed by a $D(i)$. If there were r replace operations before this increases the number of transformations by r.

Another way to see this is that we can always get from x to y by first deleting all characters from x and then inserting all characters from y. This gives a sequence of transformation of length $|x| + |y|$.

4.4 If we disallow insert operations, we can never get from a string x to a string y where $|y| > |x|$, since replace operations maintain the length of the string and delete operations only make it smaller. Therefore any x and y with $|y| > |x|$ are a counterexample.

Similarly, if we disallow delete operations, we can never get from a string x to a string y where $|y| < |x|$, since replace operations maintain the length of the string and insert operations only make it longer. Therefore any x and y with $|y| < |x|$ are a counterexample.

Solution for Task 5 (ranking)

5.1 + 5.2 The inverted lists + scores for the words *say*, *hello*, and *goodbye* are:

$$\begin{aligned} \text{say} &: (1, 1.0), (2, 1.0), (4, 1.0), (6, 0.5), (7, 0.5), (9, 0.5), (10, 0.5) \\ \text{hello} &: (4, 1.0), (5, 2.0), (7, 1.0), (8, 2.0), (10, 1.0) \end{aligned}$$

$$\text{goodbye} \quad : \quad (4, 1.7), (6, 1.7), (9, 1.7)$$

Explanation for the scores: the document frequencies (df) of the three words are 7, 5, and 3, respectively, which gives rise to inverse document frequencies (idf) of $\log_2(10/7) \approx 0.5$, $\log_2(10/5) = 1.0$, and $\log_2(10/3) \approx 1.7$. Mulitplying that with the term frequencies (tf) gives the scores shown above.

5.3 Computing the union of the two lists for *say* and *goodbye* and aggregating the scores by sum, we get:

$$(1, 1.0), (2, 1.0), (4, 2.7), (6, 2.2), (7, 0.5), (9, 2.2), (10, 0.5)$$

and sorting that by score (with ties broken by doc id), we get the ranking:

$$(4, 2.7), (6, 2.2), (9, 2.2), (1, 1.0), (2, 1.0), (7, 0.5), (10, 0.5)$$

5.4 For the top-k algorithm, we first have to sort the two list for *say* and *goodbye* by score (which they happen to already be, if sorted by doc id):

$$\begin{aligned} \text{say} \quad &: \quad (1, 1.0), (2, 1.0), (4, 1.0), (6, 0.5), (7, 0.5), (9, 0.5), (10, 0.5) \\ \text{goodbye} \quad &: \quad (4, 1.7), (6, 1.7), (9, 1.7) \end{aligned}$$

Now after the first round of top-k (reading the first pair from each list), we have

$$(1, [1.0, 2.7]), (4, [1.7, 2.7]), \text{all others: } \leq 2.7$$

After the second round (having read the first two pairs from each list), we have

$$(1, [1.0, 2.7]), (4, [1.7, 2.7]), (2, [1.0, 2.7]), (6, [1.7, 2.7]), \text{all others: } \leq 2.7$$

After the third round we have

$$(1, [1.0, 2.7]), (4, 2.7), (2, [1.0, 2.7]), (6, [1.7, 2.7]), (9, [1.7, 2.7]), \text{all others: } \leq 2.7$$

And after the fourth round we have

$$(1, 1.0), (4, 2.7), (2, 1.0), (6, 2.2), (9, [1.7, 2.2]), \text{all others: } \leq 0.5$$

Now we know that the top-ranked element can only be 4. If we wanted just anyone top-ranked document, we would have also stopped after round 3, but then we wouldn't have known whether there are other documents, and maybe with a smaller id, achieving a score of 2.7, too.