Sample solutions to Homework 6, Information-Theoretic Modeling (Fall 2014)

Jussi Määttä

October 16, 2014

Question 1

For any n, we have $C_n^1 = 1$ (imagine we have a coin with two identical sides; there's only one possible outcome).

The case C_n^2 corresponds to the binomial model which we've already seen:

$$C_n^2 = \sum_{k=0}^{n} \binom{n}{k} \left(\frac{k}{n}\right)^k \left(1 - \frac{k}{n}\right)^{n-k}.$$

When $m \geq 3$, we can compute C_n^m using the recursion

$$C_n^m = C_n^{m-1} + \left(\frac{n}{m-2}\right) C_n^{m-2}.$$

The attached Matlab function `nml_constant.m` takes as parameters the numbers n and m and returns C_n^m computed using the formulae above.

Consider a sequence of 262 observations from a multinomial model with $m = 3$. Let the counts of the three possible outcomes be 115, 57 and 90.

Then multinomial NML gives us the code-length

$$\begin{aligned}
\ell &= -\log_2 \frac{p_{\hat{\theta}(D)}(D)}{C_n^m} \\
&= -\log_2 \frac{(115/262)^{115}\,(57/262)^{57}\,(90/262)^{90}}{C_{262}^3} \\
&= \log_2 C_{262}^3 - \left[115 \log_2 \frac{115}{262} + 57 \log_2 \frac{57}{262} + 90 \log_2 \frac{90}{262}\right] \\
&\approx 8.1445 + 400.7840 \approx 408.93.
\end{aligned}$$

The attached Matlab program `q1.m` computes this code-length using the function `nml_constant.m` discussed above.

Question 2

(a)

The given command produces the output

```
115 57 90
63 11 43
47 34 22
11 2 5
379 121
365 135
483 17
201 299
290 85 125
```

so the total codelength is

$$\begin{aligned}
\ell = &-\log_2 \frac{\left(\frac{115}{262}\right)^{115}\left(\frac{57}{262}\right)^{57}\left(\frac{90}{262}\right)^{90}}{C_{262}^3} \\
&- \ldots - \log_2 \frac{\left(\frac{483}{500}\right)^{483}\left(\frac{17}{500}\right)^{17}}{C_{500}^2} \\
&- \ldots - \log_2 \frac{\left(\frac{290}{495}\right)^{290}\left(\frac{85}{495}\right)^{85}\left(\frac{125}{495}\right)^{125}}{C_{495}^3} \\
\approx\ & 2899.86.
\end{aligned}$$

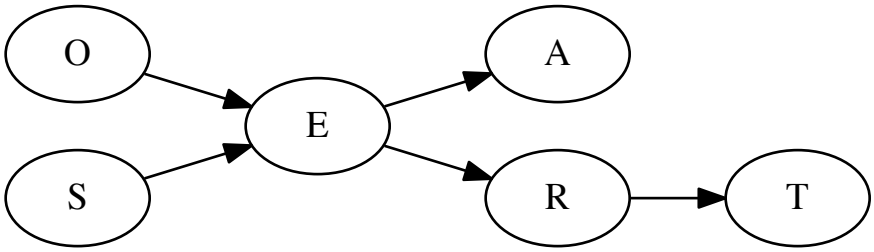

Figure 1: A Bayesian network with the variables A, R, E, O, S, and T.

Why is this the correct answer? For each variable, we have a separate multinomial distribution for each of the possible configuration its parents can take. We need to encode them all, so the final answer is the sum of the code-lengths of all these possibilities.[1]

The attached Matlab program `q2_a.m` computes the above code-length. It calls the Python program `splitcfg.py` that was provided with the homework problems and also uses the function `nml_constant.m` that was discussed in the above solution to Question 1.

(b)

The following solution is based on the hint (fix a total ordering of the variables) and the method described in a paper by Teyssier & Koller (2005)[2].

As suggested by the hint, it is useful to consider total orderings of the variables. Bayesian networks are directed acyclic graphs (DAG's), and any given DAG has a *topological ordering*. This means that we can arrange the variables in some order so that any given variable has all its parents on its left. For instance, the graph shown in Figure 1 has a topological ordering O, S, E, A, R, T (there are others as well).

Suppose now that we have fixed some ordering of the variables $X_1, X_2, \ldots, X_n$. Then:

1. X_1 cannot have any parents.

[1] There are some implicit independence assumptions here that are beyond the scope of this course. See the course *Probabilistic Models*.

[2] Marc Teyssier and Daphne Koller: *Ordering-Based Search: A Simple and Effective Algorithm for Learning Bayesian Networks*, Proc. 21st Conference on Uncertainty in Artificial Intelligence (UAI2005), 2005. URL: `http://arxiv.org/abs/1207.1429`

2. X_2 can have either no parents or a single parent (X_1).

3. X_3 can have any of the following parent sets: $\emptyset$, $\{X_1\}$, $\{X_2\}$ or $\{X_1, X_2\}$.

4. X_n can have any parent set $\mathrm{Pa}_n \subseteq \{X_1, \ldots, X_{n-1}\}$.

Therefore, for a given ordering, there are $2^0 \cdot 2^1 \cdot 2^2 \cdot \ldots \cdot 2^{n-1} = 2^{n(n-1)/2}$ possible of parent sets. For $n = 6$, this equals $2^{15} = 32768$, which isn't that much. So we can simply try out all possibilities: compute the fNML score for each of them and pick the best one. (Note that this is a bit of a brute-force approach and is not usually possible in real life where you have more variables. In practice, we could e.g. limit the sizes of the parent sets to a maximum of three variables so that there would be less possibilities to consider.)

Now, of course, if we pick one particular ordering, it may not allow the best possible network structure in terms of the fNML score. So we should try many orderings. Since we have six variables, there are $6! = 720$ possible permutations. Now, if we were to do the above, we would have $6! \cdot 2^{15} \approx 24 \cdot 10^6$ computations, which would be a bit much (though actually still easily within the limits of what can be computed on a modern desktop computer). So let us do something more sophisticated. We start with a random ordering, say $X_1, X_2, \ldots, X_n$, and try out all $n-1$ orderings obtainable from it by swapping to adjacent variables. For each of these $n-1$ orderings, we find the optimal parent sets of each variable in terms of the fNML score. Then we choose the best of these and start again. To keep the algorithm exploring the space of orderings, we also keep a "tabu list" of orderings that we've already tried and that we're not allowed to enter again. This algorithm can be summarized as follows:

1. Select a random ordering π. Add it to the tabu list.

2. For each ordering π' obtainable from π by swapping two adjacent variables:

   (a) If π' is not in the tabu list, find the parent sets for each variable that are compatible with π' and that maximize the fNML score.

3. Replace π by the best π' discovered. Add the new π to the tabu list.

4. Goto 2.

Actually the algorithm above has no way to stop. We make two additions. First, we decide that if the score of π is not improved for, say, ten successive iterations, then we set π to a new random ordering. Second, we decide that after ten of these "random restarts", we stop the execution of the algorithm and return the best collection of parent sets found so far.

The attached Matlab program `q2b.m` does all this. It is not very fast (because trying all 32768 parent set collections isn't *that* simple, and because the code has not been optimized and would benefit from caching intermediate results), but it suffices for this problem. (The only thing the code caches is the NML normalization constants.)

The best result I found with the program is in fact displayed in Figure 1. It gives a code-length of 2875.96 bits.

Note 1. In fact, this is not the network from which the data was generated. We have only 500 data points and the number of possible network structures is much higher, so this is nothing to be surprised of. Had we more data, the chances of finding the "true" network would be better. The true network appears to be the one behind this link:
`http://www.bnlearn.com/bnrepository/#survey`

Note 2. If you're interested in Bayesian networks, do take the course *Probabilistic Models* lectured in the 3rd period!

Note 3. When working with Bayesian networks, we're actually interested in *equivalence classes* of networks. Two networks are said to be equivalent if they define the same probability distributions. For instance, the networks $A \to B \to C$ and $A \leftarrow B \leftarrow C$ are equivalent: in this framework, it makes no difference which one of them we use. However, $A \to B \leftarrow C$ is in a different equivalence class. Now, the fNML score is actually not *score equivalent*, that is, it may give a different score for two networks that are in the same equivalence class; but in practice, this isn't likely to make a big difference at least when we have enough data.

Question 3

The text was extracted from the web page so that the text of each definition is placed on a single (long) line. This gives a text file of 68 lines. Also note

that the numbers of the definitions are followed by a dot, *except* for the last definition.

I converted the text into a sequence of symbols. The set of symbols contains *(i)* all words appearing in the text (in lowercase), *(ii)* a symbol that marks the beginning of a new definition, and *(iii)* symbols for all punctuation marks appearing in the text (dot, colon and semicolon).

I sorted the word symbols in decreasing order of frequency and placed them in a Python list. If we further denote by -1 the beginning of a new definition and by $-2, -3, -4$ the punctuation marks, then we can use the numbers, $-4, -3, -2, -1, 0, 1, 2, 3, \ldots$ to encode the whole text. This is what I did.

The decoder does the obvious: it increments the definition counter, adds whitespace as appropriate and capitalizes the first words of all sentences. It also takes care of the missing dot for Definition 23.

Since the words were sorted by decreasing frequency, more common words got a shorter encoding (single-digit numbers vs three-digit numbers). However, this did not result in a particularly good compression rate. I managed to remove lots of bytes from the program by converting the encoding of the text into a character string. By adding 36 to all indices, I got many printable ASCII characters.

In the end, the attached Python 2 program `q3.py` got squeezed down to 2150 bytes. It writes to standard output exactly the contents of the file `definitions.txt` whose size is 3247 bytes.

Here are some ideas on how one might achieve further compression (without using Huffman coding etc.):

- Store the individual words in a *trie* data structure.
- Build a directed graph where each word has its own vertex and there is an edge from one word to another if and only if these words appear in succession somewhere in the text. Try to store this graph in a small number of bytes. Then compress the actual text so that you always specify which of the edges one should traverse next.
- Use a separate file in which to store raw binary data.

Note 1. Take the project course if you thought this was fun!

Question 4

The hit counts that Google gave me produce the distances shown in Table 1.

Note that although NGD is supposed to be "normalized", it gives $\mathrm{NGD}(\texttt{Andrey}, \texttt{breakfast}) = 1.1557 > 1$. Possible reasons for this are that *(i)* the value $M \approx 5 \cdot 10^{10}$ may be incorrect, or *(ii)* the hit counts given by Google may be incorrect. If all the assumptions used in deriving the NGD are satisfied, the distances should lie within the unit interval.

Figure 2 shows a minimum spanning tree visualization of the distance matrix. We see that the words `breakfast`, `porridge` and `omelette` seem to be close to each other; they are connected to the other words via the edge `porridge`—`stochastic`, which has NGD ≈ 0.37. The rest of the words (except for `broccoli`, which seems out of place and is closest to `Andrey`) seem "sort-of" sensibly organized, except that the word `Andrey` behaves unexpectedly (it is most similar to the words `stochastic` and `complexity`, not `Kolmogorov`).

Another way to visualize the distance matrix is to use *multidimensional scaling* (MDS). I gave the distance matrix as an input to Matlab's built-in function mdscale, which implements a kind of non-classical MDS. The result is shown in Figure 3. The algorithm appears to perform somewhat satisfactorily (although not perfectly) in placing the words in a plane so that the relative distances are approximately preserved.

Looking at these visualizations and the matrix itself, it seems that the distances that involve the words `Andrey` and `broccoli` are not very convincing. It might be interesting to redo the visualizations without these words; this is left as an exercise for the reader.

Table 1 and Figure 2 were produced with the attached Python program `q4.m` which requires the packages `py-matplotlib` and `networkx`. Figure 3 is produced by the Matlab program `q4_mds.m` which uses the file `q4_distance_matrix.csv` generated by the Python program.

Note 1. See e.g. the paper by Cilibrasi and Vitanyi (2007) if you want to know where the formula for NGD came from. The paper is available online at `http://arxiv.org/abs/cs.CL/0412098`

Note 2. The answers you got may or may not depend on things such as whether you've logged on to your Google account or which localization of Google you're using.

	Andrey	Kolmogorov	complexity	stochastic	porridge	Rissanen	breakfast	omelette	broccoli
Andrey	0	0.6904	0.2372	0.2536	0.5202	0.3794	1.1557	0.4433	0.3599
Kolmogorov	0.6904	0	0.4932	0.3088	0.5391	0.1735	0.8673	0.7149	0.6892
complexity	0.2372	0.4932	0	0.2146	0.4058	0.5616	0.5470	0.4327	0.8662
stochastic	0.2536	0.3088	0.2146	0	0.3714	0.4738	0.6955	0.5024	0.4459
porridge	0.5202	0.5391	0.4058	0.3714	0	0.4883	0.4683	0.2840	0.4827
Rissanen	0.3794	0.1735	0.5616	0.4738	0.4883	0	0.7519	0.6605	0.4652
breakfast	1.1557	0.8673	0.5470	0.6955	0.4683	0.7519	0	0.4826	0.5266
omelette	0.4433	0.7149	0.4327	0.5024	0.2840	0.6605	0.4826	0	0.5010
broccoli	0.3599	0.6892	0.8662	0.4459	0.4827	0.4652	0.5266	0.5010	0

Table 1: Some normalized Google distances.

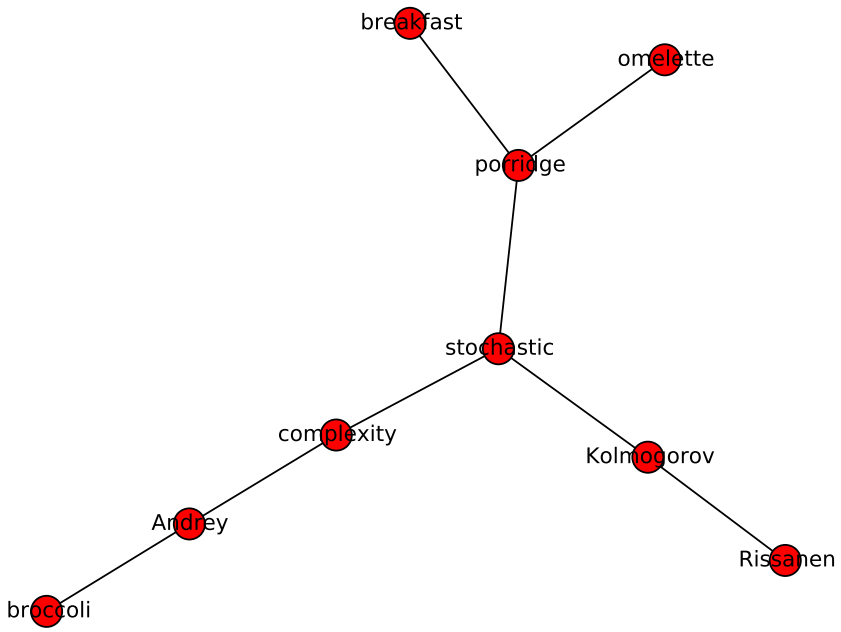

Figure 2: Minimum spanning tree for the distance matrix.

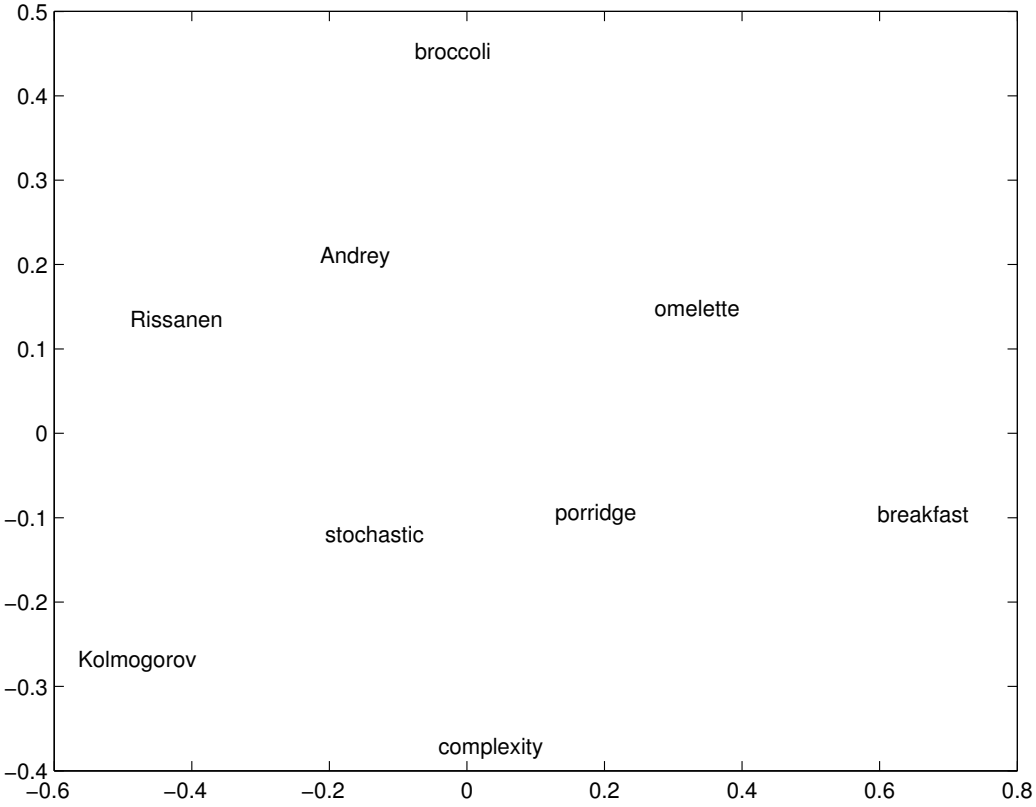

Figure 3: The distance matrix visualized by a non-metric multidimensional scaling (MDS) algorithm built-in to Matlab.