PGMs week 6 Inference: Max-Sum Inference

1 Study material

- **WATCH** the Koller videos on MAP Inference:
  - *PGM43 - Inference MAP: Max-Sum Exact Inference*
  - *PGM44 - Inference MAP: Finding a MAP Assignment*
  - *PGM45 - Inference MAP: Tractable MAP Problems*
- READ Barber chapter 5.2 - 5.4, but we will mostly focus on the Koller videos this week.
- Important points:
  - Using log factors changes factor products to factor summations.
  - We can find the MAP values (instead of the normal marginals) by replacing sum-marginalisation with max-marginalisation.
  - The combination of the above two yields the max-sum algorithm. Apart from the change in the nature of the result, just about all the other properties and techniques we have seen before still holds. Those familiar with HMMs will recognise the the max-sum algorithm as the general case of the Viterbi algorithm used there.
  - Not discussed in the above material: This can be very useful for continuous variables where the integrals required by the marginalisation do not have analytical solutions. By instead opting for the MAP solution in this manner, we are exchanging the need for integration for that of differentiation – in general a rather easier task (the derivative is zero at the maximum point).
  - One can also build hybrid systems where certain variables are being marginalised via summation/integration, and others by maximisation.

2 Task 1: Hamming code (once again)

Repeat the exercises of the last two weeks (both loopy belief propagation, as well as the junction tree version) using the max-sum algorithm. Possible issues to watch out for:

- Koller's examples only shows sep-sets with single variables whereas we here have multiple variables in the sep-set.
- Similarly to the Viterbi algorithm decoding can be done via a backtracking step. This is normally done via "back-pointers". As before, marginalisation is the last step in constructing a new message. However, for each of the potential values in the message, you will now need to record the specific values of the variables you marginalised over to obtain this max. Investigate the `pgmpy` code to see if support for this is provided.
- The easier alternative is to add a small amount of perturbation noise to all of the factors. This makes the maximal value in each cluster unique, and these maxima also synchronises over clusters.

3 Task 2: Removing impulse noise from an image (optional)

This one is to exercise your ability to design a useful PGM. The design is much more open ended and therefore might take you somewhat longer to do. The resultant system will be loopy and, although still small (probably a few thousand factors), it will be *much* larger than the previous systems in this course. You need to:

- Design a set of factors that will reflect the semantics for the task at hand.
- Build the structure that will combine these factors into a graphical model with valid RIP.
- Do loopy inference (belief propagation) on this structure until it is calibrated.
- Do queries on the calibrated graph to determine whether a particular pixel position should be part of the foreground or background in the given image.

In the `instructions/data/noisy10.gz` file you will find the noisy version of a binarised 28x28 pixel hand-written character. It was obtained by randomly flipping 10% of the bits of the original clean image (available in `clean.gz`). Plot both (via `matplotlib` or similar) to get a feeling for the damage done. Your task is to build a PGM that will clean up this noisy image. For this you will have to think about the general properties of such handwritten characters (of course you should refrain from using the explicit detail of `clean.gz`). Do consider using factors larger than those of standard pairwise MRFs.

After you have designed and implemented the factors reflecting the semantics of this situation, you will have to consider how they can be compiled into a graphical model, and how inference will be done subsequently.

Using `pgmpy`: Although a `ClusterGraph` object is available, my first perusal indicates that this software does not support any mechanism for automatically determining its structure. (Please confirm.)

The only alternative seems to be to build a factor graph. In this you have a simple choice:

- Do things pretty much as you did for the earlier tasks. This will probably imply that you implement your own factor graph and then do your own loopy message passing on it. In principle this should be very do-able, my only uncertainty is whether the inference will be fast enough in practical terms.
- Alternatively you can make use of the built-in `FactorGraph` and its corresponding inference code. Since this structure is implicitly unambiguously determined from the set of factors, this should in principle be simple. However, my brief attempt at this was somewhat frustrating. This might be due to my ignorance as well as the limited time I spent on this. If you want to go this way, I'd suggest you first attempt this with the Hamming-code task. If you should succeed, please let me know.

Using `emdw`: This code copes effortlessly with the task at hand. The vector of `Factors`, as well as the various observations (i.e pixel values from the actual image as well as any other knowns) are used to automatically configure a valid `ClusterGraph`. This graph is then calibrated by calling a loopy belief inference function. The calibrated graph is then queried to find the probability of any particular pixel position to form part of the foreground character in the image.

For a reference I have also included the resulting foreground probabilities after convergence in `cleaned_with_pgm.gz`. To obtain the resulting binarised image one simply have to check whether these probabilities exceed 0.5 (foreground) or not (background). The clustergraph had 1248 clusters in it and convergence was obtained after passing about 47k messages between them. The `time` instruction reported about 1/3 of a second total execution time on my somewhat ordinary Linux laptop.