PGM Programming Assignment
Markov Networks for OCR

1 Overview

1.1 Introduction

In the last assignment, you used Bayesian networks to model real-world genetic inheritance networks. Your rival claims that this application to genetic inheritance underscores the limited applicability of graphical models, because one doesn't often find problems with network structures that clear.

To prove him wrong, you decide to apply the graphical model framework to the task of optical character recognition (OCR), a problem that is considerably messier than that of genetic inheritance. Your goal is to accept as input an image of text and output the text content itself.

The real-world applications of OCR are endless. Some examples include:

1. The Google Books project has scanned millions of printed books from libraries around the country. Using OCR, these book scans can be converted to text files, making them available for searching and downloading as eBooks.

2. There has long been research on OCR applied to handwritten documents. This research has been so successful that the US Postal Service can use OCR to pre-sort mail (based on the handwritten address on each envelope), and many ATMs now automatically read the checks you deposit so the funds are available sooner without the need for human intervention.

3. Research on OCR for real-world photos has made it possible for visually impaired individuals to read the text of street and building signs with the assistance of only a small camera. The camera captures an image of the sign, runs OCR on that image, and then uses text-to-speech synthesis to read the contents of the sign.

In this assignment, we will give you sets of images corresponding to handwritten characters in a word. Your task is to build a graphical model to recognize the character in each image as accurately as possible. This assignment is based on an assignment developed by Professor Andrew McCallum, Sameer Singh, and Michael Wick, from the University of Massachussetts, Amherst.

The full OCR system will be split across two assignments. In this one, you will construct a Markov network with a variety of different factors to gain familiarity with undirected graphical models and conditional random fields (CRFs) in particular. We provide code that will perform inference in the network you construct, so you can find the best character assignment for every image and evaluate the performance of the network. In Programming Assignment 7, you will extend the network to incorporate more features, use an inference engine that you built yourself, and learn the optimal factor weights from data.

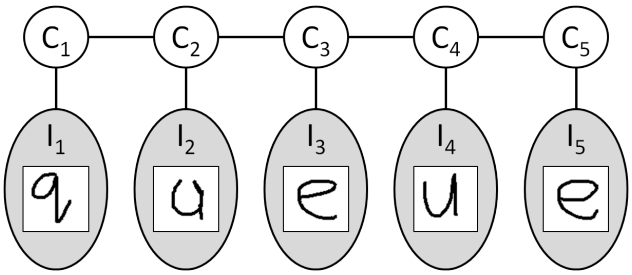
1.2 Markov Networks for OCR

Suppose you are given n total character images (corresponding to a single word of length n). There are two sets of variables in the model we will build: I_i and C_i for $i = 1, \ldots, n$. I_i is a vector-valued variable that corresponds to the actual pixel values of the ith image, and C_i is the character assignment to the ith image. Let K be the total number of possible character values. For this assignment, we only use lower case English letters, so $K = 26$. For example, $C_3 = 5$ means that the third character image contains the character 'e'.

It is important to note that:

- The I_i variables are always observed, while the C_i variables are never observed.
- We want to find the assignment to the C_i variables that correctly describes the images in the I_i variables.

In this assignment, we will use a Markov network to model the distribution over the C variables, given the images I. This is one example of a possible Markov network over an observed word of length 5:



In this example, we have factors $\phi_i^C(C_i, I_i)$ that represent how likely a given image is a particular character, and factors $\phi_i^P(C_i, C_{i+1})$ that represent the interactions between adjacent pairs of characters. Given the images I, we can come up with a prediction C^* by running MAP inference on the network to find the assignment $C^* = \text{argmax}_C \tilde{P}(C, I)$. For this particular example, we would hope that the factors we have chosen allow us to recover the word "queue" as the MAP assignment C^*.

Even with just these two types of factors, we can see that Markov networks allow us to use powerful models that not only take into account which character each image resembles, but also higher-order dependencies between characters (e.g., a 'q' is likely to be followed by a 'u'.) We will elaborate more on these factors and explore alternative Markov networks later in this assignment.

Since I is always observed, Markov networks like these can be seen as modeling the conditional distribution $P(C|I)$ in lieu of the joint distribution $P(C, I)$. They are therefore also known as conditional random fields (CRFs).

2 The Basic OCR Network

2.1 Provided Data

We provide you with all the data you need in the four files `PA3Data.mat`, `PA3Models.mat`, `PA3SampleCases.mat`, and `PA3TestCases.mat`. `PA3Data.mat` contains the following data structure:

- **allWords**: This is a cell array of every word from the dataset. In particular, `allwords{i}` returns a struct array containing the data for the ith word in the dataset. That struct array will have length(word i) entries, each with the following values:
  - **groundTruth**: A value 1 through 26 that provides the true value for that character.
  - **img**: The image pixel values for that character. It is a 16x8 matrix, where each entry is a pixel: 1 for black, 0 for white.

  For example, `allWords{i}(j).img` is the 16x8 matrix of pixel values for the jth character of the ith word.

`PA3Models.mat` contains:

- **imageModel**: This is a struct containing the following values:
  - **K**: The size of the alphabet, i.e., the number of possible values for each C_i. Set to 26.
  - **params**: The parameters used to calculate singleton factors (see next section).
- **pairwiseModel**: This is a 26×26 matrix that will be used to construct pairwise factors later on in the assignment.
- **tripletList**: This is a struct array of 2000 triplets, used to construct triplet factors later on.

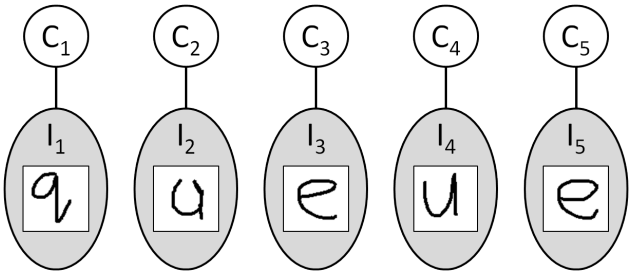
`PA3SampleCases.mat` contains the input and correct output for each of the six sample cases that we test for in the submit script. `PATestCases.mat` contains the test cases for each of the same tests. We do not provide the corresponding correct output.

While you work on the assignment, be sure to begin by loading both `PA3Data.mat` and `PA3Models.mat`. You can use the input / output pairs of `PA3SampleCases.mat` to check your understanding of each function and debug any issues. There is no need to ever load the test cases, but you are free to do so.

2.2 Singleton OCR Features

Before you begin to add more complex features to the model, it is important to establish a working baseline that is as simple as possible. That way, you can meaningfully evaluate the effectiveness of every change and improvement you make.

Your baseline will be a model that contains only the factors for singleton "OCR features," ϕ^C. There are n such factors, one for every character image. The scope of ϕ_i^C, the factor for the i^{th} image, is $\{C_i, I_i\}$; we call them singleton factors because I_i is always observed, so these factors essentially operate on single characters. This model looks like:



Using logistic regression on a separate dataset of OCR images, we have pretrained a model for you to compute a score for each image/character pair. (Don't worry if you haven't heard about logistic regression: the important part is that we have a function that scores the match between each individual image/character pair.) This model is implemented by `ComputeImageFactor(image, imageModel)`, which returns a vector of K values. The i^{th} entry in that vector is the "score" for character i on the provided image. You should write code to use this function to create the OCR factors ϕ^C. In particular, you should implement:

- **ComputeSingletonFactors.m**: This function takes a list of images and the provided image model. You should fill out the code to generate one factor for every image provided. The factor values should be given by the vector returned from ComputeImageFactor on each image.

Note that the factor values you compute here will only contain the character variable C_i. This occurs since the image variable I_i, on which the factor also depends, is always observed.

2.3 Performing Inference in the OCR Network

2.3.1 Constructing the OCR Network

As you have seen in the lectures, the Markov network is defined by a set of factors. Right now, you have only the singleton factors. As you progress in the assignment, you will implement functions to compute other sorts of factors. We have provided you the file `BuildOCRNetwork.m` to generate all of the different factors in one go, rather than calling each function separately.

It is a little tricky to specify which factors you want to include. Here are the details (what each of these factors mean is explained later):

- The singleton factors are always included.
- The pairwise factors are included if the `pairwiseModel` is passed in. To exclude these, pass in the empty array (`[]`).
- The triplet factors are included if the `tripletList` is passed in. To exclude these, pass in the emptry array.
- The similarity factors are included unless the `imageModel` has a field `ignoreSimilarity` set to true.

As an example, to include only singleton and triplet factors, you might do the following:

```
» imageModel.ignoreSimilarity = true;
» factors = BuildOCRNetwork(allWords{i}, imageModel, [], tripletList);
```

Since the default implementation of every function that creates factors is to return an empty array, you will never have to do these acrobatics unless you want to explore different combinations (the test cases will always call functions in the right way).

2.3.2 Running Inference

The Markov network which uses only the singleton factors from the previous section is an entirely valid, though simplistic, Markov network. To establish our baseline performance metrics, we need to perform inference in the network. In particular, we need to compute values for $C_1, \ldots, C_n$ that maximize their joint probability. Since we only have singleton factors right now, this is straightforward - in fact, adjacent characters have no effect on each other, so it suffices to find the MAP assignment to each of $C_1, \ldots, C_n$ independently. But to prepare for the introduction of more complex factors in the latter part of this assignment, we will use a more general mechanism.

In the next programming assignment, it will be up to you to implement efficient inference in a Markov network. For this assignment, however, we have provided you the function `RunInference.m` which calls a binary (C++) implementation of an inference engine to find the MAP assignment given a set of factors. We have also provided the `BuildOCRNetwork.m` function, which strings together all of the factors for a single word as described above.

We provide you with a set of higher level functions to use along with inference:

- **ComputeWordPredictions.m**: This function accepts the cell array of word images (i.e., `allWords`) and returns a cell array such that `wordPredictions{i}` is an array of the predicted (MAP) assignments to the i^{th} word's characters. For example, if I believe the first word is 'cat', then I have `wordPredictions{1} = [3 1 20]`.
- **ScorePredictions.m**: Given `allWords` and the predictions computed by `ComputeWordPredictions.m`, returns the character and word level accuracies. The character level accuracy is simply the number of correctly identified characters divided by the total number of characters. The word accuracy is the number of words in which *every* character is correctly identified divided by the total number of words.
- **ScoreModel.m**: A wrapper around the above two function, this computes the character and word accuracies for the current model.

If the inference binary called by `RunInference.m` works correctly for you (see next section), you should now be able to run:

```
» [charAcc, wordAcc] = ScoreModel(allWords, imageModel, [], []);
```

to compute the character and word accuracies of the singleton model on the 100 examples in `PA3Data.mat`. You should see values of 76.7% and 22.0%, respectively.

2.3.3 The Inference Binary

The provided file `RunInference.m` depends on a small binary utility `doinference`, located in the `inference/` subfolder of the assignment package. This is a small program that leverages libDAI[1] to perform inference in many varieties of graphical models. We are giving you a binary instead of an Octave implementation, as you will have the opportunity to implement these inference routines yourself in later assignments.

[1] `http://cs.ru.nl/ jorism/libDAI/`

For most people, we expect `RunInference.m` to work fine, and even if it does not, none of the tests depend on its correct functioning. So you are free to ignore everything that follows in this section.

Getting a binary program to run can be much more difficult than running Octave code. We have provided versions of the file for Windows, Mac, and Linux (`RunInference.m` automatically calls the right one), and each is statically linked to avoid system dependencies. The Windows version was built for 32-bit. The Mac version was built for 64-bit on OSX 10.6. The Linux version was built for Ubuntu 10.04 64-bit.

Even still, the variation in processor architectures and operating systems is too great to guarantee that no one will have difficulty. Thus, we have included `inference-src.zip`, which contains the relevant source files to build the binary for yourself if one of ours does not work. In the case that the provided versions do not work, you shoud:

- Follow the instructions in README for building libDAI.
- We have modified the Makefile to automatically build doinference along with libDAI.
- doinference will be in examples/doinference
- If for any reason you need to rebuild just the binary, you can run `make doinference`.

Unfortunately, we cannot promise that this will work for everyone, nor can we promise to provide bug fixes or support. No required component of the assignment depends on running the inference procedures, and we encourage you to use the forums to help each other get the binary built.

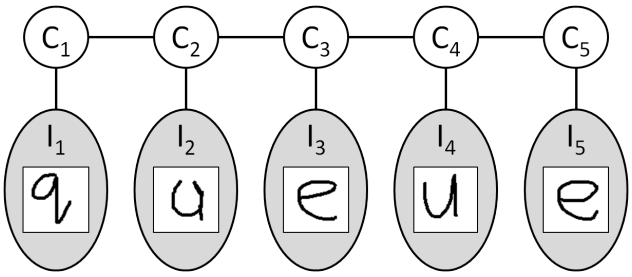
3 Extending the Network

You now have a model that performs reasonably well on the OCR task, but there is a good deal of room for improvement. One issue with the singleton OCR factors is that they do not consider any interactions between multiple character assignments. This is, after all, the point of the Markov network: it gives us a structured way to represent these interactions. We will consider two sorts of interactions. The first are linguistic interactions that we will call the pairwise and triplet factors, ϕ^P and ϕ^T. The second are image interactions that we will call similarity factors, ϕ^S.

3.1 Pairwise Factors

We begin with the pairwise factors. The intuition behind these factors is as follows. Suppose I am a sloppy writer, and my u's are so round that they look the same as a's. In isolation, then, it can be difficult for the OCR factors to distinguish between these two characters. But suppose that the OCR factor for the *previous* character assigns a very high score to the letter 'q'; i.e., we are fairly certain that the previous letter is a 'q'. In that case, the 'u' vs. 'a' ambiguity disappears. The odds of seeing 'qu' in English text are so much higher than seeing 'qa' that I can be nearly certain that the letter is a u even if the OCR factor assigns them nearly equal scores.

More formally, suppose we have a word with n characters, $C_1, \ldots, C_n$. We will introduce $n-1$ factors, $\phi_i^P(C_i, C_{i+1})$ for $i = 1, \ldots, n-1$, giving us the first model we saw in the assignment:



What values should we use for these factors? We consider two possibilities. First, implement:

- **ComputeEqualPairwiseFactors.m**: This function (and the following one) accepts an array of the images in one word and the value K, the size of the alphabet (for our purposes, it is 26). You should compute the $n-1$ pairwise factors described above. For this function, you should assign a value of 1 to every single possible factor value.

Use `ComputeWordPredictions` and `ScorePredictions` (or just `ScoreModel`) to obtain the MAP assignment and compute character and word accuracies as you did before. You should see the same values as you saw with only the singleton factors. Why is this?

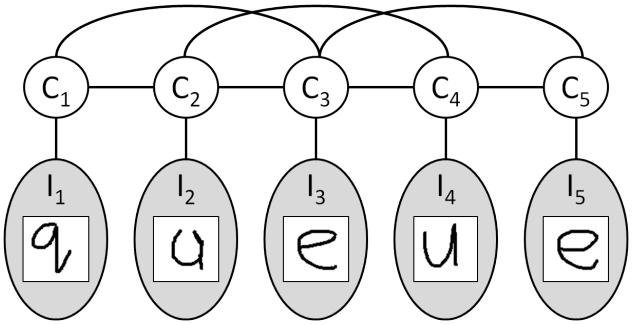
Instead of having uniform factors, we want to assign higher probabilities to common pairs of letters, and lower probabilities to rare ones. In `PA3Models.mat`, we have provided the pairwiseModel matrix, where `pairwiseModel(i,j)` is proportional to the probability that the j^{th} letter appears after seeing the i^{th} letter (these probabilities were extracted from a separate corpus of English text). With this, implement:

- **ComputePairwiseFactors.m**: This function is just like `ComputeEqualPairwiseFactors`, but it also uses the `pairwiseModel` we provided. For the factor values, an assignment of character i to the first character and character j to the second should have a score of `pairwiseModel(i,j)`.

You need to modify `BuildOCRNetwork.m` to use these new, improved pairwise factors instead of the uniform-valued ones (there is a comment at line 45 describing the change). Compute the character and word accuracies as before. You should get values of 79.16% and 26.0%.

3.2 Triplet Factors

Triplet factors serve a similar purpose as pairwise factors: we want to encourage common triplets, such as "ing", while discouraging uncommon ones. Adding in triplet factors to our model results in a network like this:



Formally speaking, we will introduce an additional set of $n-2$ factors where the ith factor is $\phi_i^T(C_i, C_{i+1}, C_{i+2})$ for $i = 1, \ldots, n-2$. Note that edges in a Markov network do not correspond directly to factors, in the sense that there are no hyperedges that connect triplets of nodes at once. Instead, our set of factors needs to factorize over the Markov network, which means that the scope of each factor needs to be a clique in the Markov network. As such, we have added extra edges between every C_i and C_{i+2} in the above network.

There are $26^3 = 17,576$ possible states in a factor over triplets, making this factor large and unwieldy. Instead of assigning a different value to each of them, we will initialize all values to 1, and only change the values of the top 2000 triplets. We will set $\phi_i^T(C_i = a, C_{i+1} = b, C_{i+2} = c)$ to be proportional to the probability that we see the letter c after seeing a and then b in succession, and normalize the factor such that the value of the 2000^{th} most common triplet is 1. These normalized frequencies can be found in `tripletList` (in `PA3Models.mat`).

There are two reasons for only selecting the top 2000. Firstly, having "sparse" factors like this makes inference faster (if the inference engine exploits it); secondly, rare triplets (e.g., "zxg") are unlikely to be accurately represented in the corpus of English words that we extracted the triplet frequencies from, so we would prefer not to rely on their measured frequencies (in particular, rare triplets are likely to be several orders of magnitude less frequent than their common counterparts, which can severely throw off the graphical model in some scenarios). In Programming Assignment 7, we will also investigate how this kind of sparsity can help us gain better performance by reducing the extent of overfitting when learning parameters from data.

Fill in the following function:

- **ComputeTripletFactors.m**: This function is just like `ComputePairwiseFactors`, except that it uses tripletList.mat to compute factors over triplets.

Compute the character and word accuracies as before. You should get values of 80.03% and 34.0%. You will also notice an appreciable slow-down in the time it takes to calculate the factors and perform inference.

3.3 Image Similarity Factors

In the previous section, we exploited our knowledge about English text and relationships between adjacent characters. In this section, we will look at longer range factors based on relationships between character images. In particular, we would like to exploit the following insight: two images that look similar should be likely to be assigned the same character value, even if we can't say what that character value should be.

Formally, suppose we choose two characters i and j ($i \neq j$) from a word with images I_i and I_j. We will define a similarity factor $\phi_{ij}^S(C_i, C_j)$ over their character assignments. We want to select factor values such that the following is true: If I_i and I_j are similar (in a way we must specify), then $\phi_{ij}^S(C_i, C_j)$ will take on a large value for $C_i == C_j$ and a small value for $C_i \neq C_j$. If I_i and I_j are *not* similar, then the reverse is true: $\phi_{ij}^S(C_i, C_j)$ should take on a larger value when $C_i \neq C_j$.

We provide a function `ImageSimilarity.m` that computes the "similarity score" between two images. In our basic implementation, `ImageSimilarity(`I_i, I_j`)` returns the cosine distance between the two images I_i and I_j with some scaling thrown in so the factor values work out well[2]. We set $\phi_{ij}^S(C_i = C_j) =$ `ImageSimilarity(`I_i, I_j`)`, and $\phi_{ij}^S(C_i \neq C_j) = 1$. Feel free to experiment with more advanced image similarity features to maximize performance!

[2]This is an example of where parameter learning is crucial, as you'll see later in the course. You might also notice that the provided data have similar or sometimes identical characters. We modified these to help the similarity factors in face of no parameter learning.

Given this, you should implement the following functions:

- **ComputeSimilarityFactor.m**: This function accepts a list of all the images in a word and two indices, i and j. It should return one factor: the similarity factor between the i^{th} and j^{th} images.
- **ComputeAllSimilarityFactors.m**: This function should compute a list of every similarity factor for the images in a given word. That is, you should use ComputeSimilarityFactor for every i, j pair ($i \neq j$) in the word and return the resulting list of factors.

At your own risk, use `BuildOCRNetwork` to build a network using all the similarity factors. We recommend that you save your files first, and only experiment on words that are no more than 5 characters in length. (The first word in the dataset has 9 characters, for a full joint distribution of more than 5 trillion numbers. If you are feeling brave and adventurous, try running this code on the dataset, and look at `inf.log` to see how much memory the inference engine estimates it will need!)

Clearly, adding every similarity factor is a bad idea from a computational point of view: inference takes forever, and we're only running inference on single words. Why is this the case? Think about what the network would have to look like (and which edges we would need to draw) to support a similarity factor between every pair of variables.

This does not mean, however, that the similarity factors are a bad idea in general; we simply need to be more selective in adding them to the model. To do so, you should implement the final function:

- **ChooseTopSimilarityFactors.m:** This function should take in an array of all the similarity factors and a parameter F, and return the top F factors based on their similarity score. Ties may be broken arbitrarily.

Modify `BuildOCRNetwork` appropriately, choosing only the top 2 similarity factors, and compute the character and word accuracies as before. You should get values of 81.62% and 37.0% - a significant improvement over our starting word accuracy of 22.0%! Play around with the similarity factors and triplet factors - can you further improve on this level of accuracy?

4 Summary

Hooray! In this assignment, you have successfully used Markov networks (and more specifically, conditional random fields) to do optical character recognition. We saw how Markov networks allow us to model interactions between characters, leveraging linguistic models (e.g., frequencies of pairs and triplets of characters) as well as long-range image similarities for better performance. These are examples of how probabilistic graphical models can capture complex interactions in a natural way. We also explored the trade-offs between complexity, accuracy, and computational efficiency, themes that we will return to repeatedly over the rest of the course.