Reading text

To show the versatility of HMMs, let's try applying them to another problem; if you're careful and you plan ahead, you can probably re-use much of your code from Parts 1 and 2 to solve this problem. Our goal is to

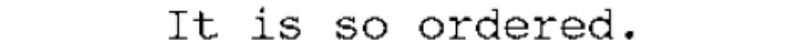

Figure 3: Our goal is to extract text from a noisy scanned image of a document.

recognize text in an image – e.g., to recognize that Figure 2 says "It is so ordered." But the images are noisy, so any particular letter may be difficult to recognize. However, if we make the assumption that these images have English words and sentences, we can use statistical properties of the language to resolve ambiguities, just like in Part 2.

We'll assume that all the text in our images has the same fixed-width font of the same size. In particular, each letter fits in a box that's 16 pixels wide and 25 pixels tall. We'll also assume that our documents only have the 26 uppercase latin characters, the 26 lowercase characters, the 10 digits, spaces, and 7 punctuation symbols, `(),.-!?'"`. Suppose we're trying to recognize a text string with n characters, so we have n observed variables (the subimage corresponding to each letter) $O_1, ..., O_n$ and n hidden variables, $l_1 ..., l_n$, which are the letters we want to recognize. We're thus interested in $P(l_1, ..., l_n | O_1, ..., O_n)$. As in part 1, we can rewrite this using Bayes' Law, estimate $P(O_i|l_i)$ and $P(l_i|l_{i-1})$ from training data, then use probabilistic inference to estimate the posterior, in order to recognize letters.

What to do. Write a program called `image2text.py` that is called like this:

```
python3 ./image2text.py train-image-file.png train-text.txt test-image-file.png
```

The program should load in the train-image-file, which contains images of letters to use for training (we've supplied one for you). It should also load in the text training file, which is simply some text document that is representative of the language (English, in this case) that will be recognized. (The training file from Part 1 could be a good choice). Then, it should use the classifier it has learned to detect the text in test-image-file.png, using (1) the simple Bayes net of Figure 1b and (2) the HMM of Fig 1a with MAP inference (Viterbi). The last two lines of output from your program should be these two results, as follows:

```
[djcran@tank]$ python3 ./image2text.py train-image-file.png train-text.txt test-image-file.png
 Simple: 1t 1s so orcerec.
    HMM: It is so ordered.
```

Hints. We've supplied you with skeleton code that takes care of all the image I/O for you, so you don't have to worry about any image processing routines. The skeleton code converts the images into simple Python list-of-lists data structures that represents the characters as a 2-d grid of black and white dots. You'll need to define an HMM and estimate its parameters from training data. The transition and initial state probabilities should be easy to estimate from the text training data. For the emission probability, we suggest using a simple naive Bayes classifier. The train-image-file.png file contains a perfect (noise-free) version of each letter. The text strings your program will encounter will have nearly these same letters, but they may be corrupted with noise. If we assume that $m\%$ of the pixels are noisy, then a naive Bayes classifier could assume that each pixel of a given noisy image of a letter will match the corresponding pixel in the reference letter with probability $(100 - m)\%$.