CS151 - Assignment 5 (last one!)
Due: Wednesday, Nov. 10 at midnight

The purpose of this assignment is to apply what youve learned about Hidden Markov Models to a real world problem, sketch recognition. In particular, you will implement the Viterbi algorithm to determine whether or not an individual pen stroke is a part of a drawing or a letter and you will play with some different feature possibilities.

Read through this entire handout before getting started.

You may work with a partner on this assignment,. We have three partnered assignments (including this one). Remember that you may NOT work with the same parter on all three of these assignments. If you're looking for a partner, e-mail me sooner than later.

Background

The motivation behind this assignment comes from a paper in the sketch recognition/machine learning literature:

> C. M. Bishop, M. Svensn, G. E. Hinton. Distinguishing Text from Graphics in On-line Handwritten. 2004. In *Proceedings of the International Workshop on Frontiers in Handwriting Recognition*, 42–147.

The paper can be found at
http://research.microsoft.com/en-us/um/people/cmbishop/downloads/Bishop-IWFHR-04.pdf

The authors of this paper propose to use a Hidden Markov Model to distinguish between text and graphics in a sketch. Their model is more complex

than ours in that they use the output of a neural network as input to the observation layer to the HMM. This paper is recommended, though not required reading.

Our model is a standard Hidden Markov Model, with one node at the hidden layer (the state node - either drawing or text) and one node at the observed layer (the stroke node - whose value represents one or more features of the stroke). Our goal is, given a sequence of values for the stroke nodes (observed layer) from $t = 1, ..., n$, to estimate the most likely sequence of state values for time $1, ..., n$. This is a most-likely-explanation problem, as we discussed in class, and you can solve it using the Viterbi algorithm. Most of the code to solve this problem is provided for you.

Before diving into the code, let's look at the probability models that define the HMM and some problems that arise in these models, specific to our task. I will give specific number here **only as example probabilities**. The actual conditional probability tables are *trained* based on the labeled training data.

Prior Probability Model

The first model to consider is $P(state_0)$, the prior probability over the first stroke's state (either text or drawing). Many of the examples we've looked at in class have had a *uniform* prior where we don't have any bias towards either case. In our case, we have a slight bias towards it being a drawing:

$p(state_0 = drawing)$	0.6
$p(state_0 = text)$	0.4

Meaning that there is a 0.6 probability that the first stroke is a drawing stroke, and a 0.4 probability that the first stroke is a text stroke.

Transition Model

The next model is our transition model, $P(state_t|state_{t-1})$. Because our state values are simple and discrete, this conditional probability distribution (CPD) can also be represented as a table:

$p(drawing\|drawing)$	0.8
$p(text\|drawing)$	0.2
$p(drawing\|text)$	0.5
$p(text\|text)$	0.5

This means that there is a 0.8 probability that a 'drawing' stroke will be followed by another 'drawing' stroke, and a 0.5 probability that a 'text' stroke will be followed by a 'text' stroke.

Evidence Model

Finally, we must consider the evidence model. The evidence model is a bit more complicated. A "stroke" is not a nice, discrete value like we've seen in the examples so far. In fact, for us, it's a list of points, represented as (x, y, time) tuples. For this reason, to reason about strokes in our HMM we must convert each stroke into one or more features (much like we broke each document into a set of features in Naive Bayes assignment).

For example, let's consider only one feature, the stroke's length. In this case, the value of $stroke_t$ is simply the length of the stroke drawn at time t, i.e., just a number. So, $P(stroke_t|state_t)$ is a distribution over the possible stroke lengths, given the classification (state) of that stroke.

This representation brings up a few problems:

- Problem 1: Continuous features

  Length is not a discrete feature, so our CPD cannot be represented with a simple table. There are two solutions to this problem, both of which you may choose to explore. The first solution is to turn length into a discrete feature by "binning" or "bucketing" it. For example, we might define all strokes less than 300 units in length to be "short" and all other strokes to be "long". Given this discretization, we can now represent this CPD (the Evidence Model) as a table, for example:

$p(short\|drawing)$	0.2
$p(long\|drawing)$	0.8
$p(short\|text)$	0.6
$p(long\|text)$	0.4

  The second solution is to leave the feature as a continuous valued feature, and fit a Gaussian (or other) distribution over the values of the features. In this case, rather than using a table, we represent $P(length|state)$ using a parameterized Gaussian. Either approach is fine, though I'd suggest starting with the binning approach since this is more similar to what we have explored so far.

– Problem 2: Multiple features

The second problem arises when we decide to use more than one feature to represent our strokes. If we consider a full table with all values of all features in combination, the feature space becomes very large and very complicated. To simplify the feature space, we will make the feature independence assumption (similar to our Naive Bayes assumption): we assume that given the state, all features are independent from one another. This assumption allows us to represent the CPD for multiple features as:

$P(stroke|class) = P(f_1, f_2, ..., f_n|class) = P(f1_|class) * P(f_2|class) * ... * P(f_n|class)$

Where $f_1, f_2, ..., f_n$ are different features, for example, f_1 could the the length feature, f_2 could be a drawing speed feature, etc.

Provided Code

I have provided code that implements most of the functionality of this assignment. to make your job of playing with different features a little easier and more fun. Here's an overview of the provided code. Please see the comments in the code for more details.

Like last time, I have put the code and data files on the cs NFS at:

`/common/cs/cs151/assignment5/`

If you're having trouble grabbing it from here, I've also posted a link to the data on the course web page.

There are only two python files: `StrokeHMM.py` and `guid.py`. The only one you will need to be concerned with and look at is `StrokeHMM.py` which contains definitions for the following classes:

- class `HMM`

  Defines the basic functionality of an HMM. Algorithms for learning the CPDs for discrete and continuous features are *already implemented*. You only need to write the label function, which applies the Viterbi

algorithm to return the most likely sequence of states, given a sequence of stroke data (features).

- class `strokeLabeler`

  Defines a stroke labeler class. There are utility functions to load and save files in an XML format that can be opened in the labeler (see below). There are also functions to train the HMM provided for you. You will need to know how to use these functions.

  The functions in strokeLabeler that you will need to modify are:

  - The constructor: you need to modify `self.featureNames`, `self.contOrDisc`, and `self.numFVals` as you play with different features
  - `featurefy`: This function converts a stroke into a dictionary of features. You'll need to add to it as you play with different features
  - `featureTest`: This function is handy for testing and debugging your feature code and you may want to modify it as you play with different features

  If you want you can modify others. You will also need to add more functions to compute recognition statistics.

- class `Stroke`

  Defines a stroke class. A stroke is a list of points each with an x, y and time value. It is in this class that you should add your function(s) to calculate new feature(s).

  Note: After competing the assignment, you should be able use the labeler in the following way:

```
execfile("StrokeHMM.py")
x = StrokeLabeler()
x.trainHMMDir("sketchData/")
x.labelFile("somedatafile.xml", "results.txt")
```

The Data

Also included in the starter folder is a `data` directory containing labeled sketches. You don't need to actually know how the data is stored, but feel

free to look at the files if you're curious (the data is stored as points in xml format). There are two files in the data directory. `sketchData.tar.gz` contains some labeled sketches to be used for training and testing your HMM. `sketchViewer.jar` is a program that can be used to view the actual sketches (again, if you're curious). You can run it by either typing "open sketchViewer.jar" or double clicking on it. Inside the program, you can open up the different sketch files to see what they actually contain.

What to do

Part 1: Implement the Viterbi algorithm

Your first task is to complete the `label` function in the `HMM` class. This function should implement the Viterbi algorithm that we discussed in class to find the most likely sequence of labels, given a sequence of strokes (represented as features).

One of the skills I want to work on for this assignment is building on top of code that you did not write. We have implemented most of the code, so the two challenges for this first part will be implementing the Viterbi algorithm, but also understanding what is getting passed around. If the documentation isn't clear about what parameters are, etc., then *insert some code to print them out* and look at what it there.

You can load an individual training example using the `loadLabeledFile` function in the `StrokLabeler` class. Print it out to see what is stored. Similarly, you can train an HMM and see what all of the probabilities are using the `trainHMMDir` function. Finally, if you look at the `labelStrokes` method, this is where your `label` method will be called.

CAUTION: Python code for the Viterbi algorithm is available on Wikipedia. Therefore, referring to the Wikipedia Viterbi site is off limits (as is looking at code anywhere else on the web).

IMPORTANT: To debug your algorithm, you MUST create a sample test example from one of the examples we've looked at in class or the book. DO NOT try and debug/test it on stroke data. It will be impossible to tell if you are correct. You should add the necessary code to hand-construct an HMM that we have looked at in class examples (or one from the book) and verify that your algorithm gives you the same output. If you do not implement

this algorithm correctly the rest of the assignment will be frustrating.

Viterbi is a complex algorithm, and as such, your comments should be elaborate. Include the code for the simple example that you tested with (label this simple example with a comment that says "Part 1 Viterbi Testing Example" AND COMMENT IT OUT so that it doesnt run when you run the sketch recognition part). Your submission for this portion will include the Viterbi code with extensive comments, and code that tests your implementation with a simple example (commented out).

Part 2: Evaluating your classifier

Once you have Viterbi running, take a minute to train and test your classifier using the feature provided in the code (stroke length, binned into two discrete values). I suggest you use only a subset of the provided files for training. Start small (5 or so), and you'll play around with the number of training files below.

This classifier does OK on some files, and not so hot on others. But how can you tell exactly how well it does?

To quantify the classification accuracy in this case, we will use a structure called a "confusion matrix". This is simply a matrix that lists how different strokes were classified, given their true labels. For example:

True Label	Classified as Drawing	Classified as Text	Percent Correct
Drawing	30	10	75%
Text	5	20	80%

That is, out of 40 'drawing' strokes total, 30 were correctly classified and 10 were not. Out of 25 'text' strokes, 20 were classified correctly, and 5 were not. Note that this is very similar to 'precision' and 'recall' – just another way of looking at it.

In the `strokeLabeler` class, write a function called `confusion(trueLabels, classifications)` that takes a list of true labels and a list of labels output by the classifier for the same set of strokes (in order). It should return a dictionary with the following structure (based on the table above):

```
{'drawing': {'drawing': 30, 'text': 10}, 'text': {'drawing': 5, 'text': 20}}
```

That is, the entries in the main dictionary should be the true labels, and each

true label should map to another dictionary giving classification numbers for those strokes.

Part 3: Improving your classifier

Finally you should explore and include at least one additional feature.

WARNING: You are going to have to compare the basic classifier to your best classifier. So make a COPY of your `strokeHMM.py` file and call it `strokeHMMbasic.py` (this is where youll keep your completed code from parts 1 and 2) so you can always run this original classifier. Put your modifications for part 3 in your `strokeHMM.py` file.

Note that I have already provided code to calculate the curvature of a stroke, so you might choose to try to use curvature to improve your classifier. However, you must add another (or more than one) feature. Possible features you might want to implement include:

- Distance from side of sketch (or top/bottom of sketch) - Bounding box area - Bounding box height/width ratio - Drawing speed (max, min, average) -Proximity to nearest neighbor

This list is just a suggestion. Feel free to get creative.

You also might want to experiment with making features discrete or continuous, and you will certainly want to be methodical about how to discretize your features. If you get REALLY ambitious you might try modeling continuous features with a more complex distribution than a simple Gaussian or relaxing the independence assumption between features (this will require modifying a substantial amount of the provided code, though.)

Part 4: The writeup

Choose a set of training files and a set of test files. As always, your training set and your test set should be disjoint!

In a file called `results.txt`, address each of the following:

- Give the confusion matrix from running the basic classifier and your best classifer

- Briefly (i.e. a couple of sentences) discuss these results.
- Discuss the process you use to build your best classifier:
  - Why did you choose the feature(s) you did?
  - How did you decide continuous vs. discrete?
  - How did you determine thresholds for discrete features?

Extra Credit

You can get extra credit of up to 10 points in one of two ways. First, you may get extra credit by providing some additional experimental analysis for your approaches. For example, you could investigate the impact of training set size on the performance of the two models. This will be worth up to 5 points, depending on the quality of your analysis.

Second, you may get extra credit for a particularly ambitious improved HMM model. This will be based on your description in your writeup! You can receive up to 10 points for a very ambitious model (though note still only 10 points total of extra credit).

When you're done

When you're all done, follow the directions on the course web page for submitting your work. Make sure that your code compiles, that your files are named as specified and that all your functions have the same name and number of parameters. If you get an error, try changing the name of the folder to include a version number and resubmit.

If you worked with a partner, put both people's last names on the submitted directory, but only submit one copy.

What to submit

- `strokeHMMbasic.py`: Your modified version of the HMM file provided including your Viterbi implementation from part 1, the simple example

that you tested it with (label this simple example with a comment that says "Part 1 Viterbi Testing Example" and comment it out), as well as your code to calculate the confusion matrix (part 2).

- `strokeHMM.py`: Your modified version of the HMM file from parts 1 and 2 including all of your additions from part 3
- `results.txt`: A text file with the analysis of your results, the description of your features, etc.

Commenting and code style

Your code should be commented appropriately (though you don't need to go overboard). The most important things:

- Your name (or names) and the assignment number should be at the top of each file
- Each class and method should have a short "docstring"
- If anything is complicated, put a short note in there to help the graders out if there are any issues.

There are many possible ways to approach this problem, which makes code style and comments very important here so that the grader and I can understand what you did. For this reason, you will lose points for poorly commented or poorly organized code.

Grading

Part	points
Part 1	
example	10
`label`	40
Part 2	15
Part 3 and part 4 (writeup)	35
style/commenting	10
extra credit	10
total	110 + 10 extra