Introduction to Artificial Intelligence – EECS 348
Assignment 5 – Sketch Recognition

This assignment can be completed in groups of up to 4 students.

The purpose of this assignment is to apply what you've 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.

On the assignments page, you'll notice that two files accompany this assignment:

- a5code.zip -- The code to start with for this assignment
- trainingFiles.zip -- Sketch training/testing data

You will need to submit three files:

- 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"), 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 an analysis of your results, description of your features, etc.

Please zip these 3 files into a file called netid.zip (where netid is YOUR netid).

The motivation behind this assignment comes from a paper in the sketch recognition/machine learning literature:
C. M. Bishop, M. Svensén, G. E. Hinton. Distinguishing Text from Graphics in On-line Handwritten Ink 2004 Proceedings International Workshop on Frontiers in Handwriting Recognition, IWFHR-9 F. Kimura and H. Fujisawa 142--147
The paper can be found here:
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.

Background

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 observation layer (the stroke node - whose value represents one or more stroke *features*). Our goal is, given a sequence of values for the stroke nodes (observation 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 (and described below). You will need to fill in only a small part (also described below).

Before diving into the code, let's look at the probability models that define the Hidden Markov Model and some problems that arise in these models, specific to our task.

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). This probability distribution is represented as a simple table, e.g.:

'drawing': 0.6

'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, $P(state_{t+1} \mid state_t)$. Because our state values are simple and discrete, this conditional probability distribution (CPD) can also be represented as a table, e.g.:

'drawing': {'drawing': 0.8, 'text': 0.2 }
'text': {'drawing': 0.5, 'text':0.5 }

where the first part is the 'Previous State Value' and the part in braces is the 'Probability Distribution for Current State Value.' 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. In fact, it's a list of points, represented as (x, y, time) tuples. For this reason, to reason about strokes in our Hidden Markov Model, we must convert each stroke into one or more features (much like we broke each document into a set of features in the Sentiment Analysis task).

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, $P(stroke_t \mid state_t)$ is: it is a distribution over the possible stroke lengths, given the classification (state) of that stroke.

Problem #1: Continuous features

Unfortunately, however, length is not a discrete feature, so our CPD cannot be represented with a simple table. There are two solutions to this problem. The first solution is to simply 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, e.g.

'drawing': {stroke=short: 0.2, stroke=long: 0.8}
'text': {stroke=short: 0.6, stroke=long: 0.4}

The second solution (much more complicated) is to leave the feature as a continuous valued feature, and fit a Gaussian distribution over the values of the features, so we use a Gaussian to represent the CPD rather than a table.

Problem #2: Multiple features

The second problem arises when we decide to use more than one feature to represent our strokes. If we consider all values of all features in combination, the feature space becomes very large and very complicated, and it will be hard to model in our CPD. To simplify the feature space, we will make the feature independence assumption. That is, 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(f1, f2, ..., fn | class) = P(f1 | class) * P(f2 | class) *...* P(fn | class)

Now that you understand the basic model, you can rest assured that all of the code to implement this model is already provided to you.

Provided Code

I have provided code that implements most of the functionality of this assignment in *StrokeHmm.py*, 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.

class HMM
Defines the basic functionality of a Hidden Markov Model. 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 need to know how to use these functions.

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

- 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 complete 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("../trainingFiles/") #../ means go back a directory
x.labelFile("../trainingFiles/0128_1.6.1.labeled.xml", "results.txt")
```

Your Job – the Actual Assignment

Part 1: Implement the Viterbi algorithm

You first task is to complete the label function in the HMM class. This function should simply 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).

IMPORTANT: To debug your algorithm, I suggest you DO NOT run it on stroke data. It will be impossible to tell if you are correct. You should instead add the necessary code to hand-construct an HMM that we have looked at in class examples (or one from the book, or the HMM example website posted on our resources page) and verify that your algorithm gives you the

same output. If you do not implement this algorithm correctly the rest of the assignment will simply 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 doesn’t 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 truth 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

Exploring More Features

Finally you should explore and include at least one more 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 you’ll keep your completed code from parts 1 and 2) so you can always run this original classifier. Put your modifications 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.)

You will submit your best classifier as your strokeHMM.py file.

Writeup

Finally, address each of the following in your results.txt file:

- Discuss the process you use to build your best classifier. Be sure to address each of the following:
  - What feature(s) did you try?
  - Were they continuous or discrete?
  - How did you determine thresholds for discrete features?
  - How well did it work?

Choose a set of training files and a set of test files (for this assignment it's OK if the sets overlap). Give the confusion matrix that results from running the naive (basic) classifier and the confusion matrix that results from running your best classifier.

Frequently Asked Questions
(Please read through the code first, then look at these.)

QUESTION: I think I need help understanding what "data" is that is being passed to the **label** function. I can't seem to get it to match up with how variables are being used down the line. At first I thought it was a list of evidence, then I said well evidence is a list of features.

ANSWER:

I would recommend reading in a file and then printing out the data and labels to see what they look like.

```
>>> execfile("StrokeHmm.py")
>>> sl = StrokeLabeler()
>>> sl.loadLabeledFile("../trainingFiles/0128_1.7.1.labeled.xml")
([[Stroke 6e8e629a-a220-479c-aab7-28373d8c2402], [Stroke dbbd837a-13d6-49b2-addc-
6855bb370c86], [Stroke 3bdcd72d-0877-42c7-9c07-5fdd1c14dbe6], [Stroke 72f4bfda-1032-4c5d-
9c0c-4e2100f57ebc], [Stroke 32b710d0-7162-488d-8f58-d12dc0c19da1], [Stroke 60b8c7d0-5bf4-
4fc5-80b9-3b59536e4961], [Stroke d9ff886c-4090-4b67-8582-64df299767c2], [Stroke a1a630ac-
6587-498a-8b81-9a810e57eb27], [Stroke a1b0a86d-eea9-4037-8056-5d8582dff581]], ['drawing',
'drawing', 'drawing', 'drawing', 'drawing', 'text', 'text', 'text', 'text'])
```

Similarly, you can train the whole HMM and then print out what the transition, emission and priors look like.

```
>>> sl.trainHMMDir("../trainingFiles/")
Loading file ../trainingFiles//0128_1.6.1.labeled.xml for training
Loading file ../trainingFiles//0128_1.7.1.labeled.xml for training
.....lots of lines saying 'loading files'.....
Loading file ../trainingFiles//9171_3.6.1.labeled.xml for training
```

Loading file ../trainingFiles//9171_3.8.1.labeled.xml for training
Training the HMM...
HMM trained
Prior probabilities are: {'text': 0.5178571428571429, 'drawing': 0.48214285714285715}
Transition model is: {'text': {'text': 0.8208744710860366 2, 'drawing': 0.17912552891396333},
'drawing': {'text': 0.10572337042925278, 'drawing': 0.8942766295707472 7}}
Observation model is: {'text': {'length': [0.6340482573726541 3, 0.36595174262734587]},
'drawing': {'length': [0.31615925058548011, 0.6838407494145198 9]}}

From this you can see what priors, transitions and emissions look like (priors is a dictionary, transitions is a dictionary of dictionaries and emissions is also a dictionary of dictionaries where the values are lists).

With all that said, now look at the labelStrokes function:

```
def labelStrokes( self, strokes ):
    ''' return a list of labels for the given list of strokes '''
    if self.hmm == None:
        print "HMM must be trained first"
        return []
    strokeFeatures = self.featurefy(strokes)
    print "strokeFeatures (i.e. - data) look like this: " + str(strokeFeatures)
    return self.hmm.label(strokeFeatures)
```

You see that it takes a list of strokes, and calls 'featurefy' on it to get a list of stroke features then passes that off to hmm.label. So, if you put the print statement in that I inserted, you'll see that 'data' looks something like this:

```
>>> execfile("StrokeHmm.py")
>>> sl = StrokeLabeler()
>>> sl.trainHMMDir("../trainingFiles/")
......lots of things printed.....
>>> strokes, labels = sl.loadLabeledFile("../trainingFiles/0128_1.7.1.labeled.xml")
>>> print strokes
[[Stroke 6e8e629a-a220-479c-aab7-28373d8c2402], [Stroke dbbd837a-13d6-49b2-addc-
6855bb370c86], [Stroke 3bdcd72d-0877-42c7-9c07-5fdd1c14dbe6], [Stroke 72f4bfda-1032-4c5d-
9c0c-4e2100f57ebc], [Stroke 32b710d0-7162-488d-8f58-d12dc0c19da1], [Stroke 60b8c7d0-5bf4-
4fc5-80b9-3b59536e4961], [Stroke d9ff886c-4090-4b67-8582-64df299767c2], [Stroke a1a630ac-
6587-498a-8b81-9a810e57eb27], [Stroke a1b0a86d-eea9-4037-8056-5d8582dff581]]
>>> sl.labelStrokes(strokes)
strokeFeatures (i.e. - data) look like this: [{'length': 1}, {'length': 1}, {'length': 1}, {'length': 1},
{'length': 1}, {'length': 1}, {'length': 1}, {'length': 1}, {'length': 0}]
label function not yet implemented
```

QUESTION:
But 'Problem #1' makes a rather big deal about how our lengths are continuously valued and that we have to resolve this by binning or creating a continuous distribution.
... but all the lengths are 0 and 1?
ANSWER:
That's right - 0 means short and 1 means long. If you check out the featureFy function, you'll see that length less that 300 is considered 'short' or 0, and length greater than 300 is 'long' or 1.

QUESTION:I guess I'm also having difficulty with creating emissions too (when testing my Viterbi implementation on an example other than the stroke recognition problem). It looks like a triple dictionary now, but I'm not sure how to make it one.
ANSWER: So if you're using the Sunny/Cloudy/Rainy example to test viterbi (which is a good example to use)...

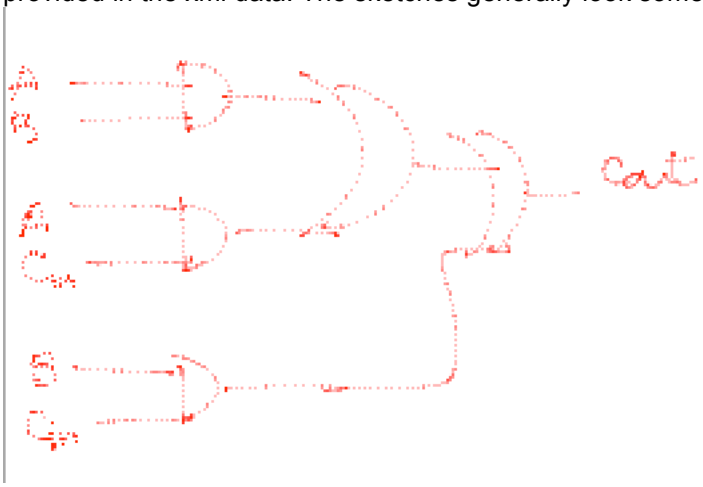
Think of the 'groundcondition' as a feature. So, for each possible state (sunny, cloudy or rainy), you'll need to link 'groundcondition' to a list of probabilities in some order that you choose. For example, the order could be [dry, dryish, damp, soggy] so the list [0.2, 0.3, 0.2, 0.3] would indicate a probability of dry being 0.2.

To construct this, notice that you can simply copy the printed observation model at the interpreter:
>>> emissions = {'text': {'length': [0.63404825737265413, 0.36595174262734587]}, 'drawing': {'length': [0.31615925058548011, 0.68384074941451989]}}

So, since you can do that, just edit that and put in the right states, observations and values and drop it into your code.
>>> emissions = {'sunny': {'groundcond': [0.3, 0.2, 0.1, 0.4]}, 'cloudy': {'groundcond': [0.3, 0.2, 0.1, 0.4]}, 'rainy': {'groundcond': [0.3, 0.2, 0.1, 0.4]}}

Note that the values above are made up and that you should get the real ones from the tables in the example online. Also, that would be self.emissions instead of just emissions.

QUESTION: How can I look at a sketch other than looking at the data in the xml file?
ANSWER:
It actually isn't necessary to view the sketches since all of the labels (text or drawing) are provided in the xml data. The sketches generally look something like this:



If you really want to view the sketches, see below…

Viewing the Sketches

Windows or Mac

If only have access to a Mac, you can take a look at the data files using the program mentioned below.

- sketchViewer.jar – Another tool to allow you to view the sketches (works on Macs and Windows)

Windows only

If you want to look at the sketches you are classifying, and the classification your algorithm produced, you can use the labeler software posted along with this file (listed following this paragraph). Unfortunately, this software only runs on Windows, and you need to be able to install some libraries to make it run, so you won't be able to run it on lab computers. If you do not have access to a Windows machine (yours or a friend's), there is another option below.

- text_drawing.txt -- The domain file for the labeler
- labeler.zip – A tool to allow you to view the sketches (only runs on Windows; see instructions below)

To run the labeler, unzip the file you downloaded, and download the domain.txt file from above. If you are running on standard Windows (not a tablet), you will also need to install the following: http://www.microsoft.com/downloads/details.aspx?familyid=B46D4B83-A821-40BC-AA85-C9EE3D6E9699&displaylang=en

Once you have unzipped the software, go into the 2008-07-30->Programs->Labeler folder and run Labeler.exe (the shortcut from the outer directory does not work).

Once inside the labeler, click "Load Domain" and choose the domain.txt file you downloaded above (probably actually called text_drawing.txt or something similar). Then click "Open Sketch" and load any xml sketch file. If you load a file that has been labeled by your HMM, you should see text strokes in blue and drawing strokes in red.