A.5 System Demonstration

Now that you have verified your toolbox, the final project assignment is to use it to implement a classification system for spoken words, hummed, played, or whistled melodies, written characters, or whatever other data you are using. Use the function you worked on in assignment A.2 to extract features. Also create a set of HMMs, one for each class, with a matching output distribution from either the `DiscreteD`, `GaussD`, or `GaussMixD` class, depending on the nature and characteristics of your feature data.

1. Figure out a way to partition your dataset into two parts: a training set and a test set. The training set should be the largest of the two and is used for learning the parameters of the HMMs. The examples in the test set are then used to assess the performance of the resulting recognizer for your particular task. Both sets should contain several examples of every class your system is designed to recognize.

2. Initialize and train your classification system using your training set only, creating one left-right HMM for each class.

3. Apply the resulting classifier to your test data and note the number of misclassifications. This is a form of cross-validation; see Section 4.5.1. To get a more accurate estimate of classifier performance, you can repeat the cross-validation process many times, using different partitionings, and compute the average error rate over all trials.

4. Take a look at the misclassified examples. Sometimes one can find a simple reason why these have been particularly difficult to classify. Other times, this is less clear.

5. Build a simple live demo in MatLab which accepts data from the user, either using `audiorecorder` or by calling `DrawCharacter`, and then shows the classification of the input and plots the log-likelihoods of the various classes for comparison. If the demo involves audio, it should play back each recorded sound, so that one can be sure there is no distortion or other sound issues.

6. Play around with your demo and try to estimate how it performs in practice for your own voice, playing, or handwriting.

7. Prepare to describe and demonstrate your complete classification system at one of the scheduled presentation seminars. Do not forget to register for the seminar in advance!

8. At the presentation you should introduce your problem, describe your system design, report the test set performance, and do a live demonstration of your system. You are encouraged to use your own laptop

for the presentation, if possible. A microphone, speakers, and a video projector will be available. Always bring all the files your require on a USB memory stick or CD, as a backup in case your particular laptop does not work with the projector. If necessary, also bring an appropriate adapter to connect to the VGA cable on the projector.

Practical Hints

If your project involves an audio database, it might be easier to work with the data in MatLab if you use a consistent pattern for storing and naming your recordings. For instance, you could create one sub-directory for each class, and number the example recordings in each folder as "`1.wav`", "`2.wav`", etc. The MatLab command `dir` could potentially also be helpful.

To create your models you can often use the function `MakeLeftRightHMM`, but first you must decide on a suitable *number of states* in your models. This need not be the same for all classes, but it could be. A useful rule of thumb is that each model should have at least one state for each distinct regime you expect in your feature series, including silences for audio. While HMMs are good at describing sharp switches from one segment to another, they assume that the mean output value is constant within each segment (HMM signals are piecewise i.i.d.). Gradual transitions between different behaviours, such as diphthongs, pitch slides, or many pen paths, will likely require more than one state to be modelled well. It is probably a good idea to try several different numbers of states, and try to see what works best in practice.

You also need to think about the HMM output distributions, which should match your features and their behaviour. For continuous feature values, you need to consider if you expect the state-conditional feature distributions to be approximately normal or not. Continuous output distributions that are multimodal (have more than one peak) or skewed (asymmetric) are likely better described by multi-component GMMs. For multidimensional feature vectors, you need to think about whether or not you should model possible correlations between the vector components. In the word recognition task, be sure to use normalized MFCCs, and consider including dynamic features. You may have to experiment with the analysis window length and the number of coefficients to find something that works well.

For the testing, it may be convenient to store all your trained HMM instances in a single array, e.g., as

```
hmms(1)=h1; %HMM for the first class
hmms(2)=h2; %HMM for the next class
%And so on
```

As your `@HMM/logprob` method was designed to work also with an HMM array, you can then obtain probabilities of features `xtest` from a given example with all your HMMs in one single call, simply as

```
lP=logprob(hmms,xtest)
```

When you are done you can save your trained system to disk using the MatLab command `save`.

Solutions to Common Problems

Do not expect everything to work well immediately. It is not uncommon to see some log-probabilities being infinite, or even `NaN` ("not a number"), at first. These problems are often due to problematic features or models, or a combination of features and models that do not work well together, as the EM-algorithm can be quite sensitive to such issues.

If you see unreasonable or undefined probabilities or parameter values, you must work out what the causes are and address them before your demonstration. After all, problem solving is an important part of pattern recognizer development! Some of the most common issues, and possible resolutions, are described below.

If `NaN`s appear at any point during training, this often signifies a big problem somewhere, and may lead to models where some or all parameters are `NaN`. Such models cannot be used at all, and should absolutely be avoided.

`NaN`s during training are particularly common with Gaussian or GMM output distributions. The mathematical basis of the issue is often that the parameters of a state-conditional Gaussian distribution or GMM component may be inferred from a subset of the training data which has virtually no variation, for instance a single sample. If the material used to estimate some particular Gaussian parameters shows no variation, the variance is estimated to be zero. This leads to a division by zero in the Gaussian density function, and causes training to break down.

Sometimes, these issues are related to an overabundance of GMM components, or due to using too many HMM states. If you are using GMMs and experience this problem, you might be able to recover by decreasing the number of GMM components and trying again. The most common cause, however, is inappropriate features with one or more elements that (sometimes or always) take on discrete values with no variation. Such features are just not suitable to model with Gaussian distributions. In theory, it is possible to define advanced models that can describe such features, but this is not something the current `PattRecClasses` code can handle.

If your features are generally scalar and discrete in nature, it is probably advisable to switch to use `DiscreteD` output distributions instead; these have no variances to estimate, and are conceptually a much more appropriate choice for integer features. Otherwise, your best bet is to redesign your features to ensure that there always is some variation present. A simple but somewhat inelegant workaround is to add a bit of random noise to the features: small enough so that the features remain substantially the same in the big picture, but big enough to ensure there always is some variation.

Another issue altogether is that models may train just fine, but produce infinite log-probabilities during testing, and sometimes even `NaN`s. This is typically because data sequences that are deemed impossible under a given model will have probability zero, the logarithm of which is minus infinity.

Numerical problems with very small probabilities in the Forward Algorithm can also lead to a division by zero, and produce `NaN`s. If `NaN`s only occur for out-of-class examples, and you are using *continuous*, not discrete, output distributions, you can probably ignore these values.

Infinitely negative log-probabilities are particularly common with discrete distributions. As discussed in Section 4.8.4, the reason is that, to maximize likelihood, probability mass should only be allocated to those outcomes actually observed during training. Any discrete symbol or event not observed in the training data, or not observed for a particular hidden state, will typically be assigned zero probability mass. If previously unobserved events or combinations of events happen to occur in the validation data, for instance due to errors or noise, the entire data sequences where they occur may be deemed impossible by the model. This is surprisingly common in practice. It may even happen that *all* models assign zero probability to the validation data, which would be highly undesirable.

We see that the maximum likelihood parameter symbol probability estimates produced by EM training are too extreme and overconfident, in a sense. This is very similar to the situation described in Example 8.1. It can also be resolved similarly by using a Bayesian approach, where a prior is introduced for the discrete distribution parameters (symbol probabilities), and the parameter estimates become MAP rather than maximum likelihood.

If used correctly, the prior ensures that all outcomes will be assigned nonzero probability, but that uncommon events still are associated with suitably low probabilities. The influence of the prior depends on how much data that is available: the more applicable training data that is available, the smaller the effect of the prior becomes.

Because the prior effectively acts as fractional observations that are added to all empirical frequency counts, the approach is sometimes known as *pseudocounts*. Since this procedure evens out the differences between big and small probabilities a bit, it is an example of so-called *smoothing*. Various kinds of smoothing are common post-processing steps in many practical applications of pattern recognition.

The `DiscreteD` class has facilities for using pseudocounts in HMM training. It is recommended that you use these in your project if you are using discrete output distributions.

At the project presentation you should

- Have all project and presentation files with you, easily accessible on a USB memory stick, CD, or similar.

- Introduce yourself to the class (maximum one minute per person). What is your Masters and specialization? If you are here on exchange, what is your home university, where is it located, and what do you study there?
- Talk about your application and your data. Which pattern recognition application inspired your choice of examples? Play or display an example or two from your database.
- Briefly describe your feature extraction scheme. Are the features discrete or continuous? Scalars or vectors? Name a number of ways the data can vary, along with the innovations or techniques used by the feature extractor to be more robust against these kinds of variation.
- Talk about your HMM design. How many states did you use, and why? What about the output distributions?
- Outline how you trained and tested your system. In particular, explain how you partitioned the data into training and test sets and describe the reasoning behind your choice.
- Plot or illustrate some example training sequences from one class, and compare these against random output sequences generated by the corresponding trained HMM using `@HMM/rand`. In what ways are they similar, and how do they differ? Discuss what aspects of the data the HMM has learned to describe.
- Report the average classification error of your recognizer over the test set. Also report the error rate for the most commonly misclassified class.
- Play back or show some misclassified instances to illustrate the errors. You may present a *confusion matrix* C, a table with elements c_{ij} showing how often examples from class i were classified as class j.
- Do a live demonstration of how the classifier works with your own voice, playing, or handwriting.
- Present your conclusions: How did the choices you made in the design process affect your classifier? What are the strengths and weaknesses of your system? What have you learned?

Presentations should preferably be at most ten minutes. No written report is necessary for this assignment. However, please be sure send us your presentation slides in an e-mail. Also be prepared to provide the code and data for your classification system, as we may ask for this to check your work in case anything is unclear.