Appendix A

Exercise Project: Pattern Recognition System

The goal of this exercise project is to let you design and build a pattern recognition system capable of recognizing spoken words, simple melodies, or written characters using hidden Markov models (HMMs). The project broadly follows the standard approach to practical pattern recognition laid out in Chapter 4. Within the project, you will develop and test a MatLab pattern recognition toolbox based on HMMs. This toolbox is quite general and can be used for many other purposes after the course.

The structure of a general hidden Markov model is presented in Chapter 5. There are three basic HMM problems of interest that must be solved for the model to be useful in real-world applications:

- Given an observed sequence of feature vectors and a model λ, how do we efficiently compute the probability that the given model would generate the observed sequence?
- How do we adjust the model parameters $\lambda = \{q, A, B\}$, given a training sequence $\underline{\boldsymbol{x}} = (\boldsymbol{x}_1 \dots \boldsymbol{x}_t \dots \boldsymbol{x}_T)$, to maximize $P(\underline{\boldsymbol{X}} = \underline{\boldsymbol{x}}|\lambda)$?
- Given an observed sequence and a model λ, how do we determine which hidden state sequence best corresponds to the observations?

The exercise project consists of five assignment steps involving the first two problems. (The last problem type is not included.) The five steps are:

1. HMM signal source
2. Feature extraction
3. Algorithm implementation
4. Code verification and signal database
5. System demonstration

These assignments are presented in detail in the following sections.

A.1 HMM Signal Source

In this first project assignment you will discover how an HMM can generate a sequence with a kind of structured randomness that is typical of many real-life signals. You will do this by coding an HMM signal source in MatLab. You will also have to verify that your implementation is correct. You must submit your work before the deadline, as instructed on the course project web page.

Download the starting package `PattRecClasses` from the course web page, unpack it, and add the resulting directory `PattRecClasses` to your MatLab path. If you are not familiar with implementing classes and object-oriented programming in MatLab, you should consult the MatLab help system on this topic.

The `PattRecClasses` package contains definitions of a few classes that are intended to work together. Each class is defined in a separate sub-directory:

```
@DiscreteD %Implements discrete distribution
@GaussD %Implements a Gaussian scalar or vector distribution
@GaussMixD %Implements a GMM
@HMM %A general HMM class
@MarkovChain %Implements the state-generator part of an HMM
```

An `HMM` object consists of one object called `StateGen` of class `MarkovChain` and one `OutputDistr` object array of class `DiscreteD`, `GaussD`, or `GaussMixD`, to represent the state-conditional HMM output distributions. The purpose of this class structure is to allow extensions to any type of output probability distribution without changing the other classes. (You can easily define additional distribution types yourself, if needed.) Regardless of its type, the array of output distributions must include exactly one element for each of the possible `MarkovChain` states. It is very easy to define a simple HMM using the various class constructor methods:

```
%Example: Define and use a simple infinite-duration HMM
mc=MarkovChain([0.5;0.5], [0.9 0.1;0.05 0.95])%State generator
g1=GaussD('Mean',0,'StDev',1) %Distribution for state=1
g2=GaussD('Mean',2,'StDev',3) %Distribution for state=2
h=HMM(mc, [g1; g2]) %The HMM
x=rand(h, 100); %Generate an output sequence
```

Other examples are given in the sub-directory `testExamples`. Every MatLab class definition must include a *constructor* method that always has the same name as the class, e.g., `MarkovChain`. You will find that many of the class methods have already been implemented for you, although the code can probably be improved.

A.1.1 HMM Random Source

Your task is to code and verify MatLab methods to generate an output sequence of random real numbers $\underline{x} = (x_1 \ldots x_t \ldots x_T)$ from an HMM with scalar Gaussian output distributions. However, your code should be general enough to handle vector random variables as well.

An HMM output sequence is always the result of *two* separate random operations: First the hidden Markov chain must generate an integer state sequence $\underline{s} = (s_1 \ldots s_T)$, and then, for each element s_t in the state sequence, the corresponding state-conditional output distribution generates the random observable output X_t. In the `PattRecClasses` code package three different functions are involved in the process: `@HMM/rand`, `@MarkovChain/rand`, and for example `@GaussD/rand`, if the output distribution is an instance of the `GaussD` class.

1. Open `@DiscreteD/rand` and finish the code precisely as specified by the predefined function interface. Save your function with the same name in the same directory. You may need to use the help system, or look into the `@DiscreteD/DiscreteD` class definition to see exactly how the class properties are stored internally.

2. Open `@MarkovChain/rand` and finish the code precisely as specified by the predefined function interface. Save your function with the same name in the same directory. You may need to use the help system, or look into the `@MarkovChain/MarkovChain` class definition to see exactly how the class properties are stored internally.

   Since the initial state of a Markov chain, and its transitions conditioned on the current state, can be seen as discrete random variables, you can use the `DiscreteD` class and the `rand` method you implemented in the previous step to simplify your work here.

   Note that your function must be able to generate output sequences for either an *infinite-duration* or a *finite-duration* Markov chain. Of course, your function should only produce sequences of finite length, even if the HMM itself could in principle continue forever.

3. Open `@HMM/rand` and finish the code precisely as specified by the predefined function interface. Save your function with the same name in the same directory. Once again you may need to look into the `@HMM/HMM` class definition to see how the class is implemented internally.

   To complete this method you need to call the `rand` method you just implemented for the state-generator component (of class `MarkovChain`) of the HMM. You also need to call another `rand` method for the `OutputDistr` array of the HMM. Just make sure to call this method for the correct `OutputDistr` element, depending on the value of the corresponding element in the state sequence. (However, the HMM should not generate any output for the final "state" that just indicates that the end of a finite-duration state sequence was reached.)

It is very important that your implementations always follow the interface specifications, both with regards functionality and input/output conventions.

When the `OutputDistr` is a `GaussD` array, the special `@GaussD/rand` method will be invoked automatically by MatLab. This method is very easy to implement for a scalar Gaussian distribution, using the standard MatLab function `randn`. However, it is a little more complicated for a Gaussian vector distribution. Therefore, the `@GaussD/rand` method has already been implemented for you.

A.1.2 Verify the MarkovChain and HMM Sources

To verify your code, use the following infinite-duration HMM $\lambda = \{q, A, B\}$ as a first test example:

$$q = \begin{pmatrix} 0.75 \\ 0.25 \end{pmatrix}; \quad A = \begin{pmatrix} 0.99 & 0.01 \\ 0.03 & 0.97 \end{pmatrix}; \quad B = \begin{pmatrix} b_1(x) \\ b_2(x) \end{pmatrix}$$

where $b_1(x)$ is a scalar Gaussian density function with mean $\mu_1 = 0$ and standard deviation $\sigma_1 = 1$, and $b_2(x)$ is a similar distribution with mean $\mu_2 = 3$ and standard deviation $\sigma_2 = 2$.

1. To verify your Markov chain code, calculate $P(S_t = j)$, $j \in \{1, 2\}$ for $t = 1, 2, 3 \ldots$ theoretically, by hand, to verify that $P(S_t = j)$ is actually constant for all t.

2. Use your Markov chain `rand` function to generate a sequence of $T = 10\,000$ state integer numbers from the test Markov chain. Calculate the relative frequency of occurrences of $S_t = 1$ and $S_t = 2$. The relative frequencies should of course be approximately equal to $P(S_t)$.

3. To verify your HMM `rand` method, first calculate $E[X_t]$ and $\text{var}[X_t]$ theoretically. The conditional expectation formulas $\mu_X = E[X] = E_Z[E_X[X|Z]]$ and $\text{var}[X] = E_Z[\text{var}_X[X|Z]] + \text{var}_Z[E_X[X|Z]]$ apply generally whenever some variable X depends on another variable Z and may be useful for the calculations. Then use your HMM `rand` function to generate a sequence of $T = 10\,000$ output scalar random numbers $\underline{x} = (x_1 \ldots x_t \ldots x_T)$ from the given HMM test example. Use the standard MatLab functions `mean` and `var` to calculate the mean and variance of your generated sequence. The result should agree approximately with your theoretical values.

4. To get an impression of how the HMM behaves, use `@HMM/rand` to generate a series of 500 contiguous samples X_t from the HMM, and plot them as a function of t. Do this many times until you have a good idea of what characterizes typical output of this HMM, and what structure there is to the randomness. Describe the behaviour in one or two sentences in your report. Also include one such plot in the report, labelled using `title`, `xlabel`, and `ylabel` to clearly show which variable is plotted along which axis. You should do this for every plot in the course project.

5. Create a new HMM, identical to the previous one except that it has $\mu_2 = \mu_1 = 0$. Generate and plot 500 contiguous values several times using `@HMM/rand` for this HMM. What is similar about how the two HMMs behave? What is different with this new HMM? Is it possible to estimate the state sequence $\underline{S}$ of the underlying Markov chain from the observed output variables $\underline{x}$ in this case?

6. Another aspect you must check is that your `rand`-function works for *finite-duration* HMMs. Define a new test HMM of your own and verify that your function returns reasonable results.

7. Finally, your `rand`-function should work also when the state-conditional output distributions generate random vectors. Define a new test HMM of your own where the outputs are Gaussian vector distributions, and verify that this also works with your code. (Note that a single instance of the `GaussD` class is capable of generating vector output; stacking several `GaussD`-objects is not correct.) At least one of the output distributions should have a non-diagonal covariance matrix such as

$$\Sigma = \begin{pmatrix} 2 & 1 \\ 1 & 4 \end{pmatrix}.$$

Your assignment report should include

- Copies of your `@DiscreteD/rand`, `@MarkovChain/rand`, and `@HMM/rand` functions, either attached as separate m-files, or in a zip archive. Merely pasting the code into a pdf or doc file is not sufficient. Also, please avoid the proprietary rar format.

- Your theoretically calculated $P(S_t = j)$ for the first infinite-duration HMM, and your corresponding measured relative frequencies.

- Your theoretically calculated $E\left[X_t\right]$ and $\mathrm{var}\left[X_t\right]$, and your corresponding measured results.

- A plot of 500 contiguous values randomized from the first infinite-duration HMM, with a description of typical output behaviour.

- A discussion of the output behaviour of the second infinite-duration HMM, with answers to the associated questions.

- The definition of your *finite-duration* test HMM, together with the lengths of some test sequences you obtained, and relevant code. Discuss briefly why you think those lengths are reasonable.

- The definition of your vector-valued test HMM, and the code you used to verify that vector output distributions work with your implementation.

A.2 Feature Extraction

The goal of this project is to produce a complete MatLab system for pattern recognition in sequence data. In this particular assignment, you will be working on feature extraction for your recognizer. This is the initial step that takes an input signal and processes it into a form useful for applying standard pattern recognition techniques, such as the HMMs used here.

A.2.1 Feature Extraction in General

The feature extraction process is very important in any pattern recognition system. The input often includes too much data and we need to focus on the signal aspects we are interested in. This depends, of course, on the task.

The parameters defining the HMM must be adapted to match the selected classification features, which is done by the training procedure. Note that it is the HMM, and *not* the feature extractor, that supplies all of the learning and "intelligence" here—feature extraction is just a mechanical process to make it easier for the HMM to do its job by removing unnecessary, confusing, or otherwise unhelpful information. However, it is nevertheless important that the features are relevant to the problem and allow different classes to be distinguished.

Designing a suitable feature extractor is a significant part of the art of successful pattern recognition. Common sense, knowledge about the data, and a bit of thinking are all important components of the design process. A longer discussion of feature extraction is provided in Section 4.2.

In this course project, the specifics of the training data and the feature extraction process depend on the task. There are three different tasks to choose from:

Speech Recognition Design a system capable of recognizing spoken words from a small vocabulary. Instructions for this specific project task are provided in Section A.2.2.

Song Recognition Design a system capable of recognizing short hummed, played, or sung melodies. Instructions for this specific project task are provided in Section A.2.3.

Character Recognition Design a system capable of recognizing letters or characters drawn with the mouse (on-line character recognition). Instructions for this specific project task are provided in Section A.2.4.

You are required to select one of these three tasks that you would prefer to work on. Please make a choice as soon as possible and e-mail it to the course project assistant!

For the sake of variety, we will try to maintain an even distribution of groups over the three different tasks, and will not let everyone work on the same topic. Tasks are therefore assigned on a first come, first serve basis. If your preferred problem is full, you will be assigned to another task where slots are available. Once the course assistants confirm which problem you will be working on, you may read the appropriate section below and solve the associated problems.

Pattern recognition and hidden Markov models can be used for many other sequence classification problems, in addition to the tasks proposed above. Other applications, to name a few, include recognizing bird species from their songs, music genre detection, distinguishing between speech and singing, distinguishing between different voices, DNA sequence classification, identifying the language of a text, and spam protection. More examples and inspiration might be found among the challenges at kaggle.com.

If you want to, you are very welcome to try your own application for the course project! In that case, please contact the course assistants, so we can help you develop your idea, make sure it fits with the course and with HMM classification, and that you won't have to do too much work.

A.2.2 Speech Recognition

The goal of this project task is to design a pattern recognition system capable of distinguishing between spoken words from a small vocabulary. This is a simple example of the more general area of speech recognition. Speech recognition technology is considered to have great potential, but has proven to be a very challenging practical problem. Speech recognizer performance remains significantly inferior to humans, especially in adverse conditions, and is an area of much active research.

For this particular assignment you will be working on understanding speech signals and the features used in speech recognition. The feature extraction technique commonly used in speech recognition today is known as *mel frequency cepstrum coefficients* (MFCC). It has been designed to mimic several aspects of human hearing and speech perception. The steps involved in this feature extraction technique are outlined below. However, since this is a complex procedure, a ready-made MFCC implementation called `GetSpeechFeatures` has been provided for you on the course project homepage.

The goal of this assignment is to give insight into the ideas behind feature extraction in modern speech recognition. Further information about these topics can be gained from the course EN2300 Speech Signal Processing given by the Electrical Engineering department at KTH.

Sound Signals

On CDs and in the computer, sound is usually represented by discrete samples of fluctuations in air pressure, forming "sound waves." These sampled signals typically have a very high rate such as 1 411 200 bits per second for CD-quality stereo sound.

While this representation is good for high-fidelity sound playback, it is not very similar to how humans interpret sound, and contains lots of information that is of little use for speech recognition.

When we humans listen to music or speech we do not perceive each of these fluctuations, rather we hear different *frequencies* such as musical chords or the pitch of a voice.

To get a better impression of this, download the sound signal package `Sounds.zip` from the course project homepage. Each separate sound file in the package can be loaded into MatLab using the function `wavread`. You can use the function `sound` to listen to the sounds; use the sampling frequency returned by `wavread` to get the correct playback rate. Plot the female speech signal and the music signal. Label your plots to show which variable is plotted along which axis, and make sure the time axis has the correct scale and units. Then zoom in on a range of 20 ms or so in many different regions of the plots. You will find that the signal in most sections has a "wavy" appearance. It is the frequency of these oscillations that humans perceive.

Audioread

The Fourier Transform

Information about the frequency content can be extracted through *Fourier analysis*, which you probably are familiar with from other courses.[1] In brief, it is a way to represent a signal not as a succession of distinct sample values at different times, but as a sum of component sine waves with different frequencies, amplitudes, and phases. For discretely sampled signals as discussed here the discrete Fourier transform (DFT) is used. This can be calculated in a computationally efficient manner using a technique known as the fast Fourier transform, FFT.

The downside of the Fourier transform is that, while it can extract the frequency contents of a sound, it discards all timing information in the process; it cannot tell us where in the sound these frequencies appear. Human hearing, meanwhile, is a compromise between time and frequency resolution.

To obtain a similar compromise in sound processing the entire signal is not Fourier analyzed as a whole. Instead, the DFT is applied separately to many short segments, or *frames*, of the signal from different times. When the spectral slices from each and every segments are lined up side by side to show the intensity of each frequency over time, the resulting "heat map" representation of the sound is known as a *spectrogram*, see Figure A.1.

Mathematically, this analysis is typically accomplished by multiplying the signal with a *window function*, which is zero everywhere except on a

[1] If you have not heard of Fourier analysis before, the topic is covered in many signal processing textbooks and on Wikipedia. A nice interactive demonstration of Fourier series approximations in Java is available at `http://www.jhu.edu/~signals/fourier2/`.

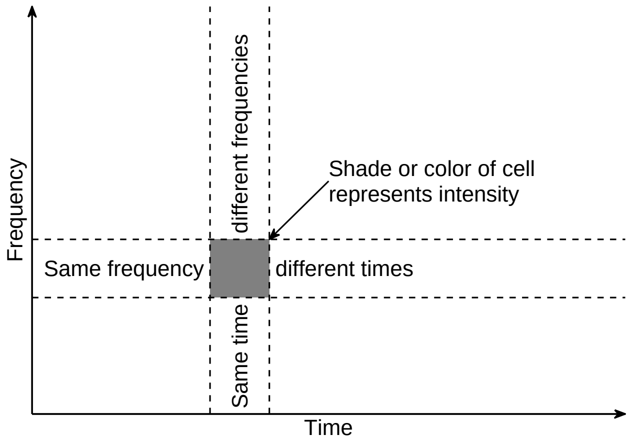

Figure A.1: Schematic illustration of the structure of a spectrogram.

short interval (often on the order of 10–20 ms), before applying the DFT. The window function is then translated and applied to the signal at several equidistant locations as shown in Figure A.2. As demonstrated, analysis windows often overlap. Note that the windows have the shape of a reasonably smooth single hump, since wavy or abruptly changing window functions will introduce artifacts (spurious frequencies) into the analysis. One common window function is the *Hann window* or *raised-cosine window*, which is given by

$$W(n) = \begin{cases} \frac{1}{2}\left(1 - \cos\left(\frac{2\pi n}{N_w}\right)\right) & \text{if } 0 \leq n < N_w, \\ 0 & \text{otherwise,} \end{cases}$$

for discrete n and window length N_w.

The function `GetSpeechFeatures` can compute short-time (windowed) spectrograms. Use it to find spectrograms for the music sample and the female speech sample you downloaded, and then plot the results using the command `imagesc`. Use a window length around 30 ms. You can run **`help GetSpeechFeatures`** before you start so you know how the function works. Make sure to put the time variable along the horizontal axis and use the additional outputs from `GetSpeechFeatures` to get correct time and frequency scales (and units) for your plots. Again, label your plots and their axes.

You will get an easier-to-interpret picture if you take the logarithm of the spectrogram intensity values before plotting. This corresponds to the decibel scale and the logarithmic properties of human intensity perception. The `colorbar` function and the command `axis xy` could also come in handy.

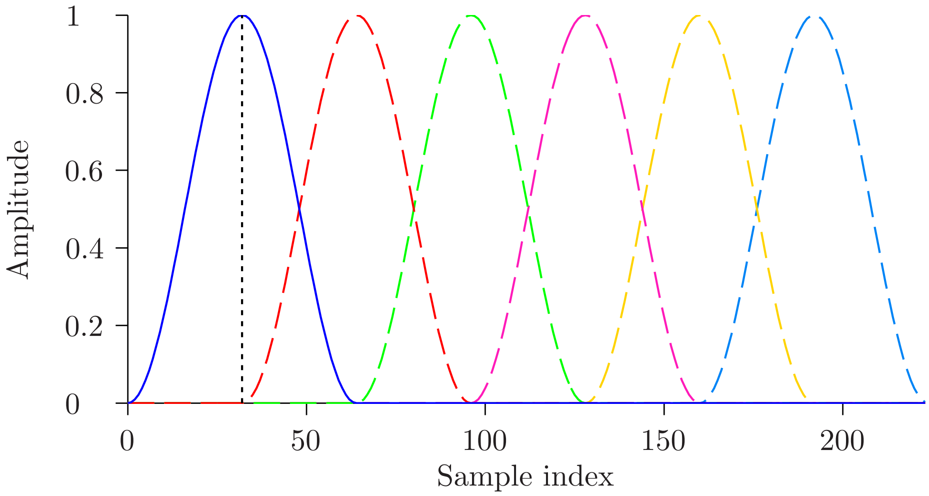

Figure A.2: A set of translated Hann windows of length 64 for short-time spectrogram analysis. A dotted line marks the centroid of the first window.

The music signal contains several harmonics. A harmonic is a component frequency of the signal that is an integer multiple of the fundamental frequency (the pitch of the sound). If the fundamental frequency is f, the harmonics have frequencies f, $2f$, $3f$, etc. While the fundamental frequency determines the current pitch of the sound, the relative strengths of the harmonics determine most other sound qualities, for instance which instrument that is playing and generally what the signal "sounds like."

If you zoom in on the frequencies below 5 000 Hz (which is the most interesting part of the figure here) the music sample should show up as a collection of horizontal line segments, moving up and down in unison over time. The harmonics in the signal are represented by bands of high intensity—low frequency harmonics are typically the most intense. The harmonics are separated by bands of lower intensity, forming striped patterns. Look in your plot to identify and point out the existence of harmonics in the music signal.

Speech consists of both voiced and unvoiced sounds. Voiced sounds such as "aaaaa" are similar to musical instruments, in that they have a certain fundamental frequency and harmonic structure, here determined by the vibration of the vocal folds. In contrast, unvoiced sounds, e.g., "fffff," are mostly noise, moderated by the human vocal tract. You can feel the difference if you touch the laryngeal prominence on your neck while speaking.

Both voiced and unvoiced sounds should be visible in the spectrogram of the speech signal. Notice that the noisy, unvoiced sounds have no single determining frequency or pattern to them, but instead contain a significant amount of energy spread over almost all frequencies.

Voiced and unvoiced sounds should also be visible in the time-domain representation of the signal. To show this, take the female speech signal

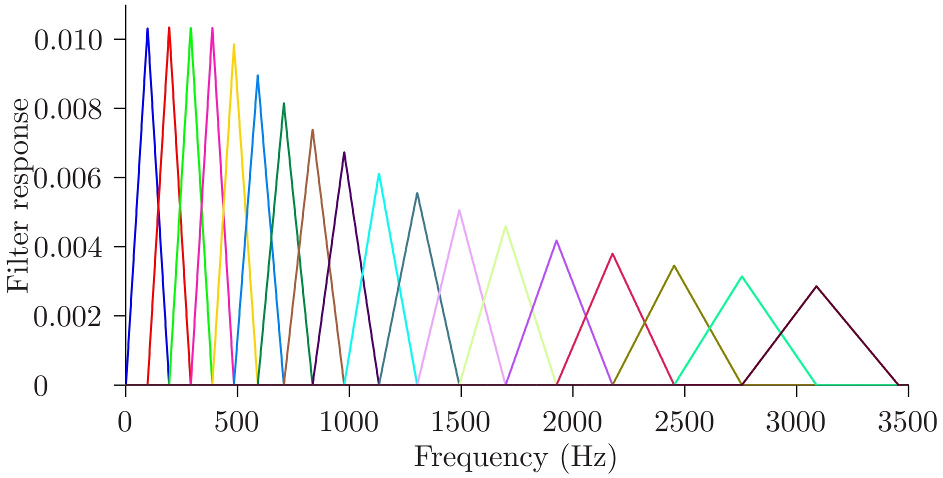

Figure A.3: Frequency responses of a mel-cepstrum-style filter bank.

from Sounds.zip, plot it, and zoom in on a range of about 20 ms in many different regions. Identify one region that corresponds to an unvoiced sound, and another that represents a voiced sound.

MFCCs

While the logarithmic short-time spectrogram representation gives a much better impression of how a sound is perceived than just a plot of the signal over time, it is still not particularly well suited for speech recognition—for one thing, the signal still has a much too high rate to be useful. Therefore further processing is applied to calculate the MFCCs. The idea of this processing is to transform each frame spectrum $I(f)$, which measures intensity as a function of frequency, to a frequency and intensity scale that better match human frequency and intensity perception (specifically, our discrimination properties), and then only keep the most general features of the spectrum in this space.

To begin with, human hearing has much coarser frequency resolution in high than in low registers. To imitate this, the frame DFT is smoothed roughly as if the signal had been analyzed through a filter bank with progressively broader sub-bands as shown in Figure A.3.[2] To approximately mimic the human auditory system, and to reduce the amount of data, the filter bank typically contains only around 30 bands in total. The specific locations of the bands are taken from psychoacoustic models of hearing such

[2] There are many MFCC variants, but most implementations actually apply the "mel" smoothing to the absolute *magnitude* of the DFT coefficients (ETSI-ES-201108, 2003), instead of the *power* as originally proposed (Davis and Mermelstein, 1980), although the summation of magnitude values violates Parseval's relation for the filter bank analogy.

as the *bark* or the *mel* scale (the M in MFCCs). Typically a setup with triangular bands like in the figure is used.

The output power values from each filter band are then logarithmically transformed. Finally, MFCCs are obtained by Fourier transforming[3] the log-intensity values within each windowed frame. The result of this process is known as the *cepstrum* for each frame. Typically, the first 13 or so of these coefficients are used as feature vectors in speech recognition systems. The zeroth cepstrum coefficient is related to the overall signal level, while the remainder relate to the general spectral envelope (spectral shape) of the sound. Sometimes the first and second time derivatives of the cepstral coefficients are also included in the feature vector.

Similar to the previously described spectrogram, a *cepstrogram* can be obtained by plotting the frame cepstra over time. Because low-order cepstral coefficients typically have significantly greater average magnitude than higher-order coefficients, it is advisable to always normalize each cepstral coefficient series extracted from each utterance to have zero mean and unit variance, both when plotting and as part of speech feature extraction. This connects with cepstral mean normalization for speaker adaptation discussed in example 4.4.

While the additional Fourier transform in the MFCCs may seem surprising, it brings a number of mathematically favourable properties for speech recognition. Importantly, low-order cepstral coefficients are mostly independent of the fundamental frequency, i.e., the pitch of the speech, while retaining other relevant speech information such as spectral envelope characteristics.[4] This makes it easier for pattern recognition algorithms to perform well regardless of individual pitch variations, for instance when both male and female speakers are considered. In fact, pitch is not important for word recognition in English at all, although it matters in so called *tonal languages*, such as Mandarin Chinese or (to a limited extent) Swedish.

Furthermore, unlike neighbouring frequencies in the short-time spectrum, which can be highly correlated, MFCCs generally have small correlations between coefficients. This is useful partially because some of these correlations are an artifact of the windowed Fourier analysis, and partially because the MFCC distributions can then better be approximated with techniques that do not take correlations into account. Such methods have significantly fewer parameters to estimate, which increases the precision of each individual estimate and decreases the risk of overfitting. GMMs are one example—these are typically constructed with diagonal covariance matrices for the MFCC component distributions.

To get a feel for how MFCCs work you should now plot and compare

[3] Technically, the *type-II discrete cosine transform* is often used.

[4] Since the harmonic structure in voiced segments causes rapid variations in intensity $I(f)$ as a function of frequency f—the stripes you saw earlier—pitch information sits in high-order cepstral coefficients, which we discard.

spectra and cepstra of the female speech and the music signal. You can use the `subplot` function to show both spectrograms and cepstrograms in parallel. Which representation do you think is the easiest for you, as a human, to interpret, and why?

It may be instructive to plot and compare spectrograms and cepstrograms of both a male and a female speaker uttering the same phrase, using the speech samples provided in `Sounds.zip`. First plot the two spectrograms. Can you see that they represent the same phrase? Could a computer discover this? Why/why not? Then plot the two cepstrograms. Can you see that they represent the same phrase now? What about a computer?

Also take one of the speech signals and use the MatLab command `corr` to calculate correlation matrices for the spectral and cepstral coefficient series. Specifically, the matrices should contain correlations coefficients between the different spectral or cepstral coefficient time series (*not* correlations between different time frames in the signal). For the spectrogram, use the log spectral intensities in the calculation. Plot the absolute values of the correlation matrices with `imagesc`. Which matrix, spectral or cepstral, looks the most diagonal to you? Use `colormap gray` to get an easier-to-interpret picture of these matrices.

Dynamic Features

One problem with using raw MFCCs as speech recognizer features is that pitch is not the only thing that makes our voices sound different, and there is a large variability in what MFCCs from the same speech sound can look like. It has been discovered that relative differences in acoustic properties between sounds often are more informative than the absolute MFCC values themselves. In particular, one can estimate the time derivative (velocity) and second derivative (acceleration) of each MFCC at each point in time, and use the resulting series as additional features. These quantities can be estimated by finite differences between frames, so called deltas, and differences of these differences, called delta-deltas, all of which can be computed with the helpful MatLab command `diff`. The resulting derivative-type features are known as *dynamic features.*

To improve accuracy, virtually all successful speech recognizers use feature vectors that include both static and dynamic features (deltas and delta-deltas) for every cepstral coefficient. It is recommended, but not required, that you also use such features in your recognizer. If you wish to do so, provide example MatLab code showing how you process the cepstral coefficients from `GetSpeechFeatures` into a vector of suitable normalized static and dynamic features. Note that these feature vectors can be quite high-dimensional, e.g., $13 \times 3 = 39$ elements per frame if 13 MFCCs are considered.

Your assignment report should include

- A copy of your MatLab code that plots the female speech and the music signal over time and also zooms in on representative signal patches illustrating oscillatory behaviour, as well as voiced and unvoiced speech segments. All code should be attached either in one or more separate m-files, or as a zip archive.

- A copy of your MatLab code that plots spectrograms for the same two signals. Put markers in the graph using `annotation` to demonstrate that you have identified the occurrence of harmonics in the music sample, as well as voiced and unvoiced segments in the speech.

- A copy of your MatLab code that compares the spectrogram and (normalized) cepstrogram representations of the two signals above, and the same for a female and a male speaker uttering the same phrase.

- A copy of your MatLab code that plots and compares correlation matrices for the spectral and cepstral coefficient series.

- Answers to the questions in the text associated with the plots.

- A working MatLab function which computes feature vector series that combine normalized static and dynamic features, if you wish to use this technique in your recognizer.

- Some thoughts on the possibility of confusing the MFCC representation in a speech recognizer. Can you think of a case where two utterances have noticeable differences to a human listener, and may come with different interpretations or connotations, but still have very similar MFCCs? (*Hint*: Think about what information the MFCCs remove.) What about the opposite situation—are there two signals that sound very similar to humans, but have substantially different MFCCs?

Always be sure to clearly state what each figure in your report depicts and which variable that is plotted on which axis.

A.2.3 Song Recognition

The goal of this project task is to design a pattern recognition system capable of distinguishing between a few brief snippets of whistled or hummed melodies. This is known as *query by humming.* While it may sound like an esoteric problem, there are numerous online and mobile song recognition services which support query by humming, including SoundHound and Midomi.com. Query by humming is also a useful feature for karaoke machines, and is even considered in the MPEG-7 standard.

In this particular assignment you will design a MatLab feature extractor suitable for this melody recognition task. For your assistance, you will be given a partial implementation of a feature extractor that computes some useful starting quantities for further processing. To complete the implementation it is useful to know a little about music signals, Fourier analysis, music theory, and human pitch and loudness perception, as described in the following sections. This information will help you design a good feature extractor.

Another classification project that would be possible using very similar features would be to have people sing a certain test melody, and then recognize which persons that are professional singers. Notify the course project assistant if you would rather work on that problem instead.

Music Signal Representation and Analysis

On CDs and in the computer, sound is usually represented by discrete samples of fluctuations in air pressure, forming "sound waves." These sampled signals typically have a very high rate such as 1 411 200 bits per second for CD-quality stereo sound.

While this representation is good for high-fidelity sound playback, it is not very similar to how humans interpret sound. (It is also likely to overload a simple melody recognizer with information.) When we listen to music or speech we do not notice each of these fluctuations, rather we perceive the rates of the fluctuations as different *frequencies*, which may form musical chords or relate to the pitch of a singing or talking voice.

To get a better impression of this, download the sound signal package `Songs.zip` from the course project homepage. Each separate sound file in the package can be loaded into MatLab using the function `wavread`. You can use the function `sound` to listen to the songs; use the sampling frequency returned by `wavread` to get the correct playback rate.

Pick a sound signal in the file and plot it. Label your plot to show which variable is shown along which axis, and make sure the time axis has the right scale and units. Then zoom in on a range of 200 samples or so in many different regions of your plot. You will find that the signal in most sections has a "wavy" appearance. It is the frequency of these oscillations that humans perceive.

The frequencies that make up a sound can be extracted from the signal using the important mathematical tool known as *Fourier series analysis*, which you probably are familiar with from other courses.[5] However, music is a dynamic process, where the sound and its component frequencies change over time. While standard Fourier analysis identifies the frequencies

[5] If you have not heard of Fourier analysis before, the topic is covered in many signal processing textbooks and on Wikipedia. A nice interactive demonstration of Fourier series approximations in Java is available at `http://www.jhu.edu/~signals/fourier2/`.

present in a given signal section, along with their average amplitudes, it does not say at what times in the signal these frequencies appear or disappear. To accommodate frequency content changing over time, sound signals are typically divided into many short pieces, called *windows* or *frames*, each of which is Fourier analyzed separately. This gives an impression of what the signal sounds like at each particular moment. Typically, the windows overlap a bit (often 50%) and are on the order of 20 ms long. Figure A.2 in Section A.2.2 provides an illustration of how overlapping windows may be arranged in practice.

Each frame of the sound contains a wealth of information about the signal at that particular point. Much of this information, extracted by the Fourier analysis, is not even perceived by the listener. Furthermore, many other aspects of the signal may be audible, but not important for our purposes: in addition to the melodies and notes played, the sequence of analysis frames also tells us about the instruments that are playing and their distinct timbres, the vocals (both lyrics and what the different voices sound like), and various special effects and noises present. The latter also includes the rhythm section, with drums and percussion, which may be very loud.

The notes that are playing constitute a very small part of the overall information content in a music signal. Yet this small piece of information is the most important part in defining the melody, and by extension the song, that is being played.

Filtering out relevant information in music is often a challenging problem. To make things easier, we shall not consider full-blown music recognition here. Instead, we concentrate on recognizing short snippets of melodies, hummed, played, whistled, or sung in an otherwise quiet environment. This avoids complexities such as recognizing and separating different instruments or speech sounds, removing percussion and effects, and handling chords and polyphony (several notes played simultaneously).

In a melody recognition task, music is boiled down to its bare essentials: the succession of notes played, their durations, and the durations of pauses between them. Decent features for recognizing melody snippets can then be computed just by knowing the dominant pitch and the intensity of the signal in each frame. (The intensity, or sound volume, is relevant for pause detection.) The MatLab package `GetMusicFeatures`, which can be downloaded from the course web page, extracts exactly this information from a given signal. It returns a matrix

$$\texttt{frIsequence} = \begin{pmatrix} f_1 & f_2 & \cdots & f_T \\ r_1 & r_2 & \cdots & r_T \\ I_1 & I_2 & \cdots & I_T \end{pmatrix},$$

where T is the number of analysis frames, which depends on the signal duration. For each frame t the matrix contains an estimated pitch f_t in Hz, an estimated correlation coefficient r_t between adjacent pitch periods, and the frame root-mean-square intensity I_t. You can run `help GetMusicFeatures` to get to know more about how the function works.

However, to perform efficient melody recognition, it is necessary to apply some additional tweaks and processing to the data from `GetMusicFeatures`. This post-processing, forming the final steps in designing a good feature extractor for melody recognition, will be up to you to propose and implement. To do so, you need to know a little about music theory and human hearing.

Music Theory and Human Hearing

The musical qualities of a sequence of notes is not determined by the absolute frequencies involved, but the relative difference between them. More specifically, it is the quotient between different frequencies that decides what is harmonic and pleasant, and what is dissonant or out of tune.

The most fundamental relationship between two pitches is that one is double the frequency of the other. This interval is known as an *octave*, and forms the basis of all forms of tonal music. There are very fundamental reasons why this interval is so important, relating to harmonic analysis and the physics of vibrating strings and objects—if notes offset by an octave did not harmonize well together, many simple instruments (including the human voice) would, for instance, appear out of tune with themselves!

An octave is a big leap on any frequency scale. In Western music, the octave is further subdivided into twelve smaller steps, known as *semitones*. In the commonly used tempered tuning, the quotient between the pitches of two adjacent semitones is the same everywhere. This places all semitones at equally spaced distances on a logarithmic scale.

Combinations of semitones form the basis of all Western music. A melody is formed by concatenating a number of stretches of various semitones with different durations, potentially with silent segments in between, to form a sequence of notes. Any interval between notes that is not close to an integer number of semitones may be perceived as being out of tune. The musical notation used in sheet music is a way to write down these note sequences and represent them visually. (As each note in a melody often is shorter than a second, and musical pieces typically last several minutes, entire songs constitute very long, specific sequences of notes picked from the scale; sheet music often requires several sheets. For simplicity, the melody recognition task concentrates on short song snippets only.)

Another reason why musical scales and pitch perception is logarithmic is the wide frequency range of human hearing, which approximately fits in the interval from 20 to 20 000 Hz. This covers three orders of magnitude, or approximately ten octaves. In a linear representation, the lowest octave (20 to 40 Hz) covers a mere 20 Hz, a very small part of the entire 20 kHz range. By operating on a roughly logarithmic scale, humans are able to distinguish between notes in all octaves with about equal accuracy.

In order to perform melody recognition that is robust to variations, it is important that your features account for the logarithmic nature of musical pitch and human frequency perception, typically by basing your features on the logarithm of the pitch track. Furthermore, the presence of distinct semitones allows devising a discrete representation of the sound, if you like.

Another vital aspect of music perception is that most people, possibly excluding those with perfect pitch, still perceive the same melody if all notes in it are *transposed* (moved) the same number of semitones up or down on the scale. This need not be an integer number of semitones, either, just as long as the frequency ratios always remain unchanged. It is therefore crucial to devise a feature extractor where the output remains largely unchanged if the entire input is transposed up or down by an arbitrary amount. Another way of saying this is that the offset of the pitch track should not matter. In practice, singers and other musicians may often use tuning forks and similar tools to ensure they all use the same offset when they play together.

A final note concerns signal intensity and loudness perception. The intensity component returned by `GetMusicFeatures` is proportional to the sound power in each window. Just like pitch perception, humans can perceive a wide range of sound energies: the scale from the faintest audible sound power to the pain threshold covers about 15 orders of magnitude. Loudness perception, like our pitch perception, therefore follows an approximately logarithmic scale. This improves discriminative accuracy between sounds on the fainter end of the intensity spectrum.

Because of the great differences in intensity that may be expected in your recordings, it is probably a good idea to base any intensity features on the logarithm of the signal intensity as well. Such features may be useful for handling pauses in the input melodies. This quantity may then be processed further, discretized, etc., depending on what kind of features you create. (Note that, in recordings, the output level and other characteristics of the recorded signal also depend on the microphone and its properties.)

Sometimes, there may not be a clear-cut threshold intensity separating notes and non-notes in the input. Moreover, the sound intensity can be high even if there is no note playing, for instance if the signal has a lot of noise. For more robust note detection, one may look at the estimated correlation coefficient between pitch periods, also computed by `GetMusicFeatures`. This correlation will be high (near one) in signal segments with a clear pitch, but lower in non-harmonic (noisy or silent) regions of the sound. A third strategy for coping with pauses can be to look at the pitch estimate itself, and see how it behaves in silent or unvoiced signal segments. Certain pitch extraction techniques may consistently indicate very high or very low pitch frequencies in such regions. Other pitch extractors just return noisy, seemingly random values when no tones are present.

Feature Extractor Design

You now know all the theory necessary to design and implement a set of features for melody recognition, based on signal pitch, correlation, and intensity series from `GetMusicFeatures`. Your features may be either continuous and vector-valued or discrete, but need to have the following properties:

1. They should allow distinguishing between different melodies, i.e., sequences of notes where the ratios between note frequencies may differ.
2. They should also allow distinguishing between note sequences with the same pitch track, but where note or pause durations differ.
3. They should be insensitive to transposition (all notes in a melody pitched up or down by the same number of semitones).
4. In quiet segments, the pitch track is unreliable and may be influenced by background noises in your recordings. This should not affect the features too much, or how they perceive the relative pitches of two notes separated by a pause.

It is also desirable, but not necessary, if your features satisfy:

5. They should not be particularly sensitive if the same melody is played at a different volume.
6. They should not be overly sensitive to brief episodes where the estimated pitch jumps an octave (the frequency suddenly doubles or halves). This is a common error in some pitch estimators, though it does not seem to be particularly prevalent in this melody recognition task.

To help you develop your features you should use the example recordings in `Songs.zip`: two of these are from the same melody, while one is from a different melody. Make a plot of the three pitch profiles of these recordings together, and another plot with the three corresponding intensity series. Place time along the horizontal axis in your plots. Also try changing the axes in the plots to use logarithmic scales (e.g., using `set(gca,'YScale','log')`), and look at the plots again. To see the melodies more clearly in the pitch profile plots here, you may want to look at the frequency range 100–300 Hz especially.

The graphs should give you an understanding of the data series you will be working with. Make sure you understand the relationship between the plots and what the example melodies sound like.

To assist you in your understanding, the code you downloaded also includes a command, `MusicFromFeatures`, capable of creating sound signals that closely match a given `frIsequence`. You can use this command to convert the output from `GetMusicFeatures` back into sounds. Because feature extraction is lossy, the restored audio often sounds very different from the original. You should be able to hear that information about the melody is retained in the `frIsequence`, while most other information is gone.[6]

When you design your feature extractor, think about all the requirements above and what you can do to address them. Then code a MatLab function that implements your ideas for feature extraction and fits with the `PattRecClasses` framework. Use your knowledge, and your mathematical and engineering creativity to come up with a good method! Keep in mind that there are many ways to solve this problem. You can test your work and ideas on the example files in `Songs.zip`.

Your features need to match the output distributions you plan to use. Specify if you plan to use either `DiscreteD`, `GaussD`, or `GaussMixD` output distributions for your HMMs. Then make sure that the feature values you generate fit with your choice of output distribution, for instance that they have the same range.

When working on your features, it makes sense to think about the kind of processes described and generated by HMMs. To be described well by HMMs, your feature sequences should look like something an HMM can produce. As seen in the previous assignment, HMM output tends to consist of stationary segments with similar behaviour. Transitions between segments are instantaneous. If your feature sequences similarly have stationary or slowly changing regions with relatively well-defined boundaries, they will probably be modelled well by HMMs.

Conversely, if your features do not look like something that the `HMM` class from `PattRecClasses` can easily produce, chances are your classifier will work poorly. In particular, here are some things you should try to avoid:

- Do not mix and match discrete and continuous values in your feature sequences. Decide on either continuous-valued or discrete features, and stick with your choice.

- If your features are continuous, do not let the exact same numerical value appear more than once. (Do not have segments where the output value shows no variation at all, for example.) If the same numerical value appears more than once in the training data, this can lead to variance estimates being exactly zero, causing divisions by zero and errors in your classifier later on.

- If your features are continuous-valued, do not put in isolated points in the feature sequence with radically different behaviour from the

[6] Sampling from a trained model to generate new signals similar to the training data is known as *synthesis*. Sampling from an HMM trained on output from `GetMusicFeatures`, for example, yields new `frIsequence`s which can be played back using `MusicFromFeatures`. This shows what the model has learned. You are welcome to try this later in the project if you like. The results may surprise you.

rest. Even if it might be theoretically possible to create an HMM that describes such data, it will be difficult to learn such a model with the tools given in this course. If there are highly different, isolated points in the data sequence, these will look a lot like outliers, and the information in them is likely to be lost on the HMM.

- If your features are discrete-valued, only use scalar, positive integer values with a finite, fixed upper bound, as these are the only numbers the `DiscreteD` class can handle. You can always convert negative, non-integer, and vector-valued discrete variables to positive integers by enumerating all possible output values or output vectors: the set $\{0, 0.5, 1.5\}$ can be mapped to $\{1, 2, 3\}$, for instance.

- Don't necessarily try to remove "noise" from the output. As seen in A.1.2 point 5, noise can carry information about the state or class, and the HMM is designed to be able to model uncertainty and variability. Unless you are confident some piece of information is unhelpful, leave it in.

- Don't overthink things. Most of the complicated and intricate feature extractors that have been submitted have worked worse than the clean and simple schemes. In general, it is the HMM, rather than the feature extractor, that should supply any intelligence here.

Checking Your Feature Extractor

Once you have designed and started coding a feature extractor proposal, it is a good idea to graphically inspect the feature series produced by your extractor for the given example files. Figure out an informative way to plot or graphically illustrate your features, so that different feature series can be compared. If the regular `plot` command isn't good enough for you (say if you are using high-dimensional feature vectors), you can look into the command `imagesc` to see if it better fits your needs.

Plotting the features series from the example files should let you confirm that the series are reasonably similar between two examples of the same melody, but differ more between examples from different melodies. Keep in mind, though, that there is a difference between curves that are visually similar to the human eye, and mathematical similarities that a pattern recognition system can exploit.

To verify that your features are transposition independent, multiply the pitch track returned by `GetMusicFeatures` by 1.5, and use this in your feature extractor. The output should be virtually the same as with the original pitch. If it isn't, you likely have to rethink parts of your feature extractor. Make a plot to show that your features are equivalent before and after pitch track multiplication. Since two of the examples in `Songs.zip` are from the same melody, these should have similar feature sequences as well. Verify this in another plot. If your feature function passes these tests, it is likely you have something that will give decent results in practice.

While feature extractor design is one of the most fun and creative aspects of this project, we also recognize it could be the most challenging assignment. As always, the teaching assistants are available to answer your questions. But before you ask, make sure to read about, think about, and work with the problem in MatLab as much as possible! The more effort you have made to understand, the better your questions will be, and the more useful the answers.

Your assignment report should include

- Plots of the pitch and intensity profiles of the three recordings from `Songs.zip`, two from the same melody, and one from another song. MatLab code files for these plots should also be included. Make sure that the scales and units are correct, and show which curve corresponds to which recording.

- A clear specification of the design of your feature extractor, how it integrates with the `PattRecClasses` output distributions, and how your scheme addresses each of the requirements listed previously. It must be clear that you understand your extractor and why it works.

- Working MatLab code for your feature extractor, either attached as one or more separate m-files, or in a zip archive.

- Plots illustrating how your extracted features behave over time for the three example files. It should be clear that two of the series are quite similar, whereas the third is significantly different. Again, code for generating these plots should be included.

- A plot that compares your feature output between a melody with a transposed pitch track and the original recording, showing equivalence, along with code for generating the plot.

- A discussion of aspects not captured by your feature extraction system, and possible cases where your feature extraction scheme may work poorly. Is there any way a human can confuse the system, and perform a melody or create a sound signal that we think sounds similar to another performance, but where the features produced by your function are quite different between the two performances? What about the opposite situation—can we create two recordings that sound like two different songs altogether, but which actually generate very similar feature sequences?

Always be sure to clearly state what each figure in your report depicts and which variable that is plotted on which axis.

A.2.4 Character Recognition

The goal of this project task is to design a pattern recognition system capable of distinguishing between a few different characters, drawn on the computer screen with the help of a mouse. This kind of task, where you have access to the pen movements used when writing a character, is known as *on-line character recognition*, in contrast to *off-line character recognition*, where only an image of the final character is available. This recognition problem is of obvious practical interest, for instance to simplify text input on portable devices such as the recently-popular tablet computers. It is also related to the topic of *mouse gestures*, where mouse movement patterns are used to issue commands to a computer.

In this particular assignment you will design and implement a feature extractor MatLab function that operates on data from mouse movements.

Feature Extractor Design

For the graphical symbol recognition task you will use an interface function `DrawCharacter` that can be downloaded from the course webpage. When the `DrawCharacter` function is called a blank window appears. The user can draw a character in this window, using the mouse as a pen, by holding down the left mouse button. Releasing the mouse button lifts the pen from the paper, and the mouse can be moved without leaving traces.

When the user is finished and closes the window, the function returns a matrix containing a sequence of Cartesian coordinates (and one additional variable), sampled at points along the motion of the mouse in the window, e.g.,

$$\texttt{xybpoints} = \begin{pmatrix} x_1 & x_2 & \cdots & x_L \\ y_1 & y_2 & \cdots & y_L \\ b_1 & b_2 & \cdots & b_L \end{pmatrix}.$$

The length L of the sequence depends on how complicated your symbol is, your drawing speed, and possibly the speed of your MatLab system.

The extra variable b in the series is a binary bit which is 1 at points where the left mouse button was pressed and 0 elsewhere. It thus indicates whether the user was drawing or not at any given point in time.

Your assignment is to use the coordinate and mouse-button data series to create features for recognizing the character drawn by the user. Of course, the features should facilitate good and robust classification performance as much as possible. There are therefore a number of requirements that your feature extractor must satisfy:

1. The feature output should be similar regardless of exactly *where* on the screen the user draws the symbols.

2. The features should disregard the absolute *size* of your symbols.

3. The path of the mouse when not drawing should not matter, since it leaves no visible results. However, you should still take into account the relative position between the end of one pen stroke and the start of the next. If not, the characters "T" and "+" would not be possible to distinguish, since the only difference between these characters is the offset between the vertical and the horizontal lines.

4. Data from before the user starts drawing and after the last stroke of the character has been drawn should be ignored.

5. The feature extractor should not fail if two adjacent frames are identical.

You may use either discrete or continuous features for your feature extractor. There is no single right answer to this design problem, so use your common sense, along with your mathematical and engineering creativity to come up with a good method!

Your feature extractor should be implemented as a MatLab function that operates on data series from `DrawCharacter`, and returns another series of features. As stated earlier, you should not build any advanced intelligence into your feature extractor; that is for the HMM to supply. The main information that your feature sequence should convey to the classifier is your pen (mouse) movements while you were drawing on the screen, and the relative position between different strokes. Notice that two identical-looking lines or curves may appear different to the system if drawn forwards or backwards.

Your features need to match the output distributions you plan to use. Specify if you plan to use either `DiscreteD`, `GaussD`, or `GaussMixD` output distributions for your HMMs. Then make sure that the feature values you generate fit with your choice of output distribution, for instance that they have the same range.

When working on your features, it makes sense to think about the kind of processes described and generated by HMMs. To be described well by HMMs, your feature sequences should look like something an HMM can produce. As seen in the previous assignment, HMM output tends to consist of stationary segments with similar behaviour. Transitions between segments are instantaneous. If your feature sequences similarly have stationary or slowly changing regions with relatively well-defined boundaries, they will probably be modelled well by HMMs.

Conversely, if your features do not look like something that the `HMM` class from `PattRecClasses` can easily produce, chances are your classifier will work poorly. In particular, here are some things you should try to avoid:

- Do not mix and match discrete and continuous values in your feature sequences. Decide on either continuous-valued or discrete features, and stick with your choice.

- If your features are continuous, do not let the exact same numerical value appear more than once. (Do not have segments where the output value shows no variation at all, for example.) If the same numerical value appears more than once in the training data, this can lead to variance estimates being exactly zero, causing divisions by zero and errors in your classifier later on.

- If your features are continuous-valued, do not put in isolated points in the feature sequence with radically different behaviour from the rest. Even if it might be theoretically possible to create an HMM that describes such data, it will be difficult to learn such a model with the tools given in this course. If there are highly different, isolated points in the data sequence, these will look a lot like outliers, and the information in them is likely to be lost on the HMM.

- If your features are discrete-valued, only use scalar, positive integer values with a finite, fixed upper bound, as these are the only numbers the `DiscreteD` class can handle. You can always convert negative, non-integer, and vector-valued discrete variables to positive integers by enumerating all possible output values or output vectors: the set $\{0, 0.5, 1.5\}$ can be mapped to $\{1, 2, 3\}$, for instance.

- Don't necessarily try to remove "noise" from the output. As seen in A.1.2 point 5, noise can carry information about the state or class, and the HMM is designed to be able to model uncertainty and variability. Unless you are confident some piece of information is unhelpful, leave it in.

- Don't overthink things. Most of the complicated and intricate feature extractors that have been submitted have worked worse than the clean and simple schemes.

Verify Your Feature Extractor

When you have figured out and started coding a feature extractor proposal, you should do a number of tests to see that it works as expected and satisfies the requirements listed above. For this you may decide on a reasonably simple character such as "P" and try drawing it three times: once in the top left quadrant of the drawing area, once in the bottom right quadrant, and finally twice as big as the other two examples, filling the entire window. For each example, save the data sequence returned by `DrawCharacter` for reference. You can later feed these saved data sequences into `DrawCharacter`'s companion function `DisplayCharacter` to plot the characters you drew.

Figure out an informative way to plot or otherwise represent your feature sequences graphically, so that one can compare different sequences and see how similar they are. If the regular `plot` command isn't good enough for you (say if you are using high-dimensional vectors), you can look into the command `imagesc` to see if it better fits your needs. Use your chosen graphical representation to compare the three different examples of the same character from the previous paragraph, to verify that they are all quite similar. They should not merely be similar to the human eye, which the original examples already are, but the plots should make clear that there are mathematical similarities between the feature sequences, which a pattern recognition system may pick up on.

Also compare the feature series against the features that are produced when you draw a different character, to ensure that there are significant differences between this feature series and the three considered above.

Finally, verify that mouse movements when the pen is lifted between different strokes do not matter, but only the relative offsets between strokes. To do this, write a multi-stroke character, but move the mouse around in crazy ways between strokes. Plot the resulting feature series, and verify that it does not differ much from the series that results when you draw the same character in a normal way. Also compare the features representing the characters "T" and "+". The different offsets of the horizontal lines in the two characters should be reflected by differences in the corresponding feature series. Point out these differences in your figure (or figures), preferably using the `annotation` command.

If the feature extractor you coded passes all these tests, it is likely that your final character recognition system will be capable of good performance.

While feature extractor design is one of the most fun and creative aspects of this project, we also recognize it could be the most challenging assignment. As always, the teaching assistants are available to answer your questions. But before you ask, make sure to read about, think about, and work with the problem in MatLab as much as possible! The more effort you have made to understand, the better your questions will be, and the more useful the answers.

Your assignment report should include

- A clear specification of the design of your feature extractor, how it integrates with the `PattRecClasses` output distributions, and how it addresses each of the requirements listed previously. It must be clear that you understand your extractor procedure and why it works.

- Working MatLab code for your feature extractor, either attached as one or more separate m-files, or in a zip archive. The input to the feature extractor should be a data series from `DrawCharacter`.

- Plots of the big and small examples you drew of the same character, and informative illustrations of the corresponding feature sequences. This should confirm that the sequences are mostly similar, even though the three examples are visually different. MatLab code files for generating your plots should be included.

- Plots displaying a different character and its feature sequence. This should show substantial differences from the previous examples. Code should again be included.

- One or more figures comparing feature series resulting when writing the same character with wildly different pen movements between strokes, versus normal pen movements. This should demonstrate feature insensitivity to path while the pen is lifted. Also include code for generating the figures, and a MatLab mat-file with the original series from `DrawCharacter` for the two examples, so we can verify that the movements are different.

- One or more figures comparing the feature series of “T” and “+”, along with code that generates the figures. There should be clear differences in the features, and an `annotation` that points to these differences.

- A discussion of aspects not captured by your feature extraction system. Is there any way a human can confuse the system, and write the same character twice in different ways, so that the end result looks the same to us, but the feature series are radically different? What about the opposite situation—can we draw two characters that have different meanings to us humans, but which generate very similar feature sequences?

Always be sure to clearly state what each figure in your report depicts and which variable that is plotted on which axis.

A.3 Algorithm Implementation

In this assignment you will implement one of two very important HMM algorithms: the Forward Algorithm and the Backward Algorithm. You will be assigned to one of these two algorithms by the teaching assistants. Read the appropriate section below and implement the associated functions.

A.3.1 The Forward Algorithm

In this section you will implement and verify a MatLab function to perform the *Forward Algorithm*. This algorithm calculates conditional state probabilities, given an observed feature sequence $(\boldsymbol{x}_1 \dots \boldsymbol{x}_t \dots)$ and an HMM λ, as

$$\hat{\alpha}_{j,t} = P(S_t = j | \boldsymbol{x}_1 \dots \boldsymbol{x}_t, \lambda).$$

You can also use the *forward scale factors* $(c_1 \dots c_t \dots)$, calculated by the algorithm, to determine the total probability $P(\underline{\boldsymbol{X}} = \underline{\boldsymbol{x}} | \lambda)$ of the observed sequence given the HMM. The Forward Algorithm is also an essential step in HMM training, and you will need it for your final recognizer.

The Forward Algorithm consists of three steps, defined by equations in Sec. 5.4:

Initialization: Eqs. (5.42)–(5.44)

Forward Step: Eqs. (5.50)–(5.52)

Termination: Eq. (5.53), only needed for finite-duration HMMs.

Implement the Forward Algorithm

In the `PattRecClasses` framework the Forward Algorithm is a method of the `MarkovChain` class, because the algorithm does not need to know anything about the type of output probability distribution used by the HMM. The Forward Algorithm needs as input only a matrix `pX` with values *proportional to* the state-conditional probability mass or density values for each state and each frame in the observed feature sequence.

The proportionality scale factors, one for each frame, must be defined and stored by the calling function, if needed. This is not the responsibility of the Forward Algorithm. The purpose of the scaling is to avoid very small probability values, which can cause numerical problems in the computations.[7]

Your task is to complete the `@MarkovChain/forward` method as specified in the function interface and comments.

[7] Another approach to numerically stable HMMs is described in the technical report `http://bozeman.genome.washington.edu/compbio/mbt599_2006/hmm_scaling_revised.pdf` by Tobias P. Mann.

Note especially that your function must work for both *infinite-duration* and *finite-duration* Markov chains, and that the output results are slightly different in these two cases. Save your finished version of the function under the same file name in the same directory.

Verify the Implementation

The only way to test your implementation is to do a calculation by hand, and compare the result with the function output. However, this tedious calculation has already been done by previous students.

Create a finite-duration test HMM with a Markov chain given by

$$q = \begin{pmatrix} 1 \\ 0 \end{pmatrix}; \quad A = \begin{pmatrix} 0.9 & 0.1 & 0 \\ 0 & 0.9 & 0.1 \end{pmatrix}$$

and where the state-conditional output distribution is a scalar Gaussian with mean $\mu_1 = 0$ and standard deviation $\sigma_1 = 1$ for state 1, and another Gaussian with $\mu_2 = 3$ and $\sigma_2 = 2$ for state 2. Then use your own `forward` implementation to calculate $\hat{\alpha}_{j,t}$, $t = 1 \ldots 3$ and c_t, $t = 1 \ldots 4$, with an observed finite-duration feature sequence $\underline{x} = (-0.2, 2.6, 1.3)$. For this simple test example, the result of your `forward` function should be

```
alfaHat =
    1.0000    0.3847    0.4189
         0    0.6153    0.5811


c =
    1.0000    0.1625    0.8266    0.0581
```

assuming you applied `forward` directly to the scaled probabilities produced by `prob`. If you used the accompanying scale factors to undo the scaling before calling `forward`, the first three elements in `c` may differ. This is not recommended.

Since your implementation also must work for infinite-duration HMMs, it is a good idea to figure out a way to test this case as well.

Probability of a Feature Sequence

Later in the project you will need a function to calculate the log-probability of an observed sequence, $\log P(\underline{\boldsymbol{X}} = \underline{\boldsymbol{x}}|\lambda)$, given an HMM instance λ. The log-probability is used since the regular observation probability $P(\underline{\boldsymbol{X}} = \underline{\boldsymbol{x}}|\lambda)$ very often is too small to be represented in a computer. For this reason $P(\underline{\boldsymbol{X}} = \underline{\boldsymbol{x}}|\lambda)$ should never be used directly in any computations.

Complete the small MatLab function `@HMM/logprob` to perform the log-probability calculation by calling your `forward` method with the scaled conditional probability values supplied by the `OutputDistr` `prob`-method (try `help GaussD/prob` for more information). However, do not forget to include the scale factors in the subsequent calculation. For the HMM and observation sequence above the result should be that $\log P(\underline{X} = \underline{x}|\lambda) \approx -9.1877$, using the natural logarithm, if your implementation works for this example. Your function should also work when the input is an array of HMM objects, and compute the probability of the feature sequence under each model.

Note that if your `c`-vector looks like

```
c =
    0.3910    0.0318    0.1417    0.0581
```

it is likely that your implementation of the Forward Algorithm is correct, but that you are applying it to the true, unscaled probabilities by accounting for the scaling factors already before running `forward`. This can lead to problems later on. To get numerically robust calculations of very small log-probabilities, which is vital for later parts of the project, you will have to apply `forward` to the *scaled* probability values `pX`, as given by `prob`, first—only thereafter should the scaling factors be taken into account.

Your assignment report should include

- A copy of your finished `@MarkovChain/forward` function.
- A copy of your finished `@HMM/logprob` function.

A.3.2 The Backward Algorithm

In this project step you will implement and verify a MatLab function to perform the *Backward Algorithm*. The Backward Algorithm will be needed later for HMM training.

The Backward Algorithm is used to calculate a matrix $\boldsymbol{\beta}$ of conditional probabilities of the final part of an observed sequence $(\boldsymbol{x}_{t+1} \ldots \boldsymbol{x}_T)$, given an HMM λ and the state $S_t = i$ at time t. The result is known as the *backward variables*, defined through

$$\beta_{i,t} = P(\boldsymbol{x}_{t+1} \ldots \boldsymbol{x}_T | S_t = i, \lambda)$$

for an infinite-duration HMM. $\beta_{i,t}$ has a slightly different interpretation for finite-duration HMMs, as explained in Sec. 5.5.

In practice it is numerically preferable to calculate the *scaled backward variables* $\hat{\beta}_{i,t}$ instead, which are proportional to the regular backward variables. The Backward Algorithm calculates these variables in two steps, defined by equations in Sec. 5.5:

Initialization: Eqs. (5.64) and (5.65) define the slightly different initializations needed for infinite-duration and finite-duration HMMs.

Backward Step: Eq. (5.70) applies to any type of HMM.

Implement the Backward Algorithm

In the `PattRecClasses` framework the Backward Algorithm is a method of the `MarkovChain` class, since the algorithm does not need to know anything about the type of output probability distribution used by the HMM. The Backward algorithm takes as input a matrix with values proportional to the state-conditional probability mass or density values for each state and each element in the observed feature sequence, along with a corresponding sequence of scale factors $(c_1 \ldots c_T)$ computed by the Forward Algorithm in the previous section.

Your task is to complete the `@MarkovChain/backward` method as specified in the function interface and comments. Because of the object-oriented nature of the HMM implementation, you need not have a working Forward Algorithm in order to write the Backward Algorithm code, as seen below.

Note especially that your function must accept either an *infinite-duration* or a *finite-duration* HMM. The output has exactly the same format in both cases, although the theoretical interpretation of the output is slightly different. Save your finished version of the function under the same file name in the same directory.

Verify the Implementation

One of the most rigorous ways to test your implementation is to do a calculation by hand, and compare the result with the function output. Fortunately for you, this tedious calculation has already been carried out by previous students, and you will just use their results here.

Create a finite-duration test HMM with a Markov chain given by

$$q = \begin{pmatrix} 1 \\ 0 \end{pmatrix}; \quad A = \begin{pmatrix} 0.9 & 0.1 & 0 \\ 0 & 0.9 & 0.1 \end{pmatrix},$$

where the state-conditional output distribution is a scalar Gaussian with mean $\mu_1 = 0$ and standard deviation $\sigma_1 = 1$ for state 1, and another Gaussian with $\mu_2 = 3$ and $\sigma_2 = 2$ for state 2. Previous calculations show that, for this HMM and the observation sequence $\underline{x} = (-0.2, 2.6, 1.3)$, the Forward Algorithm gives the scale factors $\underline{c} = (1, 0.1625, 0.8266, 0.0581)$. Feeding these results into the Backward algorithm, further computations show that the final scaled backward variables $\hat{\beta}_{j,t}$, $t = 1 \ldots 3$ for this simple test example should be about

```
betaHat =
    1.0003    1.0393         0
    8.4182    9.3536    2.0822
```

These numerical results require that all calculations use the scaled state-conditional probability values `pX` as supplied by the `OutputDistr prob`-method (try `help GaussD/prob` for more information). If you used the accompanying scale factors from `prob` to undo the scaling before calling `backward`, you may get different results. This is not recommended, as the unscaled values may be very small, leading to numerical problems.

Since your implementation also must work for infinite-duration HMMs, it is recommended that you figure out a way to test this case as well.

Your assignment report should include

- A copy of your finished `@MarkovChain/backward` function.

A.4 Code Verification and Signal Database

This assignment consists of two very different parts. In one part you will verify, and possibly correct, code for your MatLab HMM implementation. This is described in Section A.4.1. In the other part you will assemble a database of example signals for training and testing your final pattern recognition system. The details of this database are task dependent: for the word recognition task see Section A.4.2, for melody recognition consult Section A.4.3, and for character recognition turn to Section A.4.4.

In case you finish the two parts of this assignment early, it might be a good idea to use any extra time to start working on your system and presentation for the final assignment. While the final assignment should not be conceptually or mathematically challenging, it is likely to require a lot of "duct-tape" code to perform all necessary training and testing steps, and to provide an interactive demo. It may therefore be a good idea to start the next assignment ahead of time, if possible.

A.4.1 Code Verification

It is easy to make mistakes in calculations and computer code, and it is therefore crucial to always check your work. For the same reason, it is frequently important to check code from other people as well. Studying someone else's code to understand how it operates, and then correcting possible errors, is a very common task for engineers and scientists today.[8]

In practice it is very important to find and eliminate all problems in whatever task you are working on, since engineers often work on projects where errors can have substantial economic consequences, and sometimes even cause bodily harm or death. Producing neat and correct code, and eliminating bugs wherever you may find them, is also a way to take pride in your work.

For the code verification part of the current assignment you will work with either the Forward Algorithm or the Backward Algorithm—whichever you did *not* work with before. Instead of writing your own functions from scratch, however, you are here given code written by someone else. You will have to verify that this code is correct, fix any problems you may find, and hand in your corrected implementation. Your final code will be judged to the same standards as implementations submitted for this task on the previous assignment; the state of the code when you received it does not matter for the grading.

To get started, you should read the appropriate sections in the previous assignment, so you know what the purpose of the code is and how it should

[8] A highly recommended article about the world-leading coding and verification practices at the NASA space shuttle group can be found at `http://www.fastcompany.com/node/28121/print`.

behave. You may then inspect the code you received to make sure it does the right thing, and write tests to verify it works as expected in practice. It is important that you find and point out *all* the errors, if there are any.

Your assignment report should include

- Correct code for the Forward or Backward Algorithm task, based on the code you have been given by the course assistants.
- Everything else that is required for the corresponding original implementation task.
- A brief description of your verification procedure and code for any tests you carried out.
- A list of problems in the code you were given, how you identified them, and how you corrected them. Were they syntax errors, run-time errors, or logical flaws? What kind of consequences could be expected if any of the errors had not been fixed? How problematic would you consider these consequences to be?

A.4.2 Speech Database

To build a speech recognizer it is necessary to have some speech data to train it on. To make things more interesting you are here required to assemble this data yourself. Some of the data will also be used to estimate the performance of your recognition system.

For simplicity, you will only build a recognizer that can distinguish between a handful of different words or phrases spoken in isolation. The training data is then a database of examples of the words or phrases to be recognized. You are free to define the vocabulary yourself. Perhaps you can come up with a few words from a fun and simple application of speech recognition? Be creative!

Most fun is probably to design a word recognizer that understands your own voice. To do this reliably you will need to record some of your own speech. All you need is a PC with a microphone. Sound recording and editing software usually comes with your operating system, but alternatives are given on the project homepage. Contact the project assistant if you have no possibility of recording your own training data.

When recording, try to minimize errors and disturbances in your sound files and record at least fifteen examples of each word or phrase! For simplicity it is probably best if you use on the order of ten words and save your recordings in 22 050 Hz 16 bit mono wav files, one for each word or phrase you say. Listen to each file you generate so that there is no distortion or other problems. To get a less speaker-dependent, and thus more generally applicable recognizer, it is a good idea to use training data from several different speakers. See if you can get together with a few fellow students and do your recordings together!

Your assignment report should include

- A brief specification of the database you recorded. Which words or phrases did you use and how did you set up the recording process? How many repetitions did you record for each word? What kind of variation is there in the database between different examples of the same word?

A.4.3 Song Database

To build a melody recognizer it is first necessary to have some melody data to train it on. To make things more interesting you are here required to record this data yourself. Some of the data will also be used to estimate the performance of your recognition system.

For simplicity, you will only build a recognizer that can distinguish between a handful of different melody snippets in a quiet background. The training data is then a database with a number of example recordings for each melody to be recognized. You are free to define the song library yourself. Perhaps you can come up with a few fun and simple songs or music snippets to distinguish? Be creative! There is also the option to do "trained singer recognition," as mentioned in assignment A.2.3.

It is probably easiest if you settle for either humming or whistling melodies in your recordings. Playing the melodies on an instrument might also work, but using chords or similar might be a bad idea since the `GetMusicFeatures` function cannot handle polyphonic sounds. Singing is another possibility, but you might get lower accuracy because of the variety of speech sounds that occur in songs, some of which (e.g., "s") may be quite energetic but do not have a well-defined pitch. This is likely to confuse your simple, pitch-based feature extractor.

Most fun is probably to design a recognizer that responds well to your own voice or instrument and performance style. To do this reliably you will need to record some of your own humming, whistling, singing, or playing. All you need is a PC with a microphone. Sound recording and editing software usually comes with your operating system, but alternatives are given on the project homepage. Contact the project assistant if you have no possibility of recording your own training data.

When recording, try to minimize noise and disturbances in your sound files and record at least fifteen examples of each melody snippet! For simplicity it might be best if you use on the order of ten different melodies in your database, or maybe a little less. Save your recordings in 22 050 Hz

16 bit mono wav files, one for each example. Also listen to each file you generate so that there is no distortion or other problems!

It is probably a good idea to keep your snippets short; select a single section of each melody (often from the beginning) that takes about ten seconds or less to perform, and record at least fifteen takes of it. If you vary your style a little in each take you are likely to get a recognizer that is more robust to different performance styles. Similarly, you might get a less style dependent, and thus more generally applicable recognizer, by using training data from several different persons. See if you can get together with a few fellow students and do your recordings together!

Your assignment report should include

- A brief specification of your recording database. Which songs did you choose and why? How are the melodies performed (whistled, hummed, etc.)? Where and how did you record the examples? How many examples of each melody were recorded? What kind of variation is present in different examples of the same melody?

- An example file for each melody in your database. You may compress the audio files you send into mp3 or ogg format, or simply reduce sampling frequency and bit-depth to save space. (Note that zip file compression is notoriously poor for audio files.)

A.4.4 Character Database

To build an on-line character recognition system, it is necessary to have some pen-trace data to train it on. You will here create this database yourself. This data can also be used to estimate the performance of your recognizer.

First, try to think of a fun and interesting application of on-line character recognition. Be creative! Then pick a library of about ten symbols, characters, or glyphs relevant to your chosen application that your system will learn to distinguish between.

For each symbol in your library, you should write it at least fifteen times to build a collection of examples. Store each raw data series generated by `DrawCharacter`, so you can plot any example character later on. You can use a cell array in MatLab (delimited by the characters `{` and `}`) to store matrices of different sizes in the same multidimensional array, and save your data to disk with the command `save`. Check every example so that there are no mistakes.

If you vary your style a little between each example you draw, you are likely to get a recognizer that is more robust to stylistic variation. Similarly, you may get a less handwriting-dependent and more generally applicable recognizer by using training data from several different persons. See if you can get together with a few fellow students and assemble your databases together!

Even if there is stylistic variation, please make sure, for each character, that you always draw everything in the same basic order and direction. If not, you will generate a database that mixes very different-looking feature sequences. This variation is something simple left-right HMMs cannot describe—it will just confuse these models and make it very difficult for them to learn anything. Since the plan is for you to use standard left-right HMMs for this project, we suggest you avoid these difficulties. If you want a system that can recognize characters drawn backwards or with different stroke orders, you need to design advanced HMMs, with complicated transition matrices that essentially represent mixtures of left-right models.

Your assignment report should include

- A brief specification of your character database. Which characters did you choose and what is the application? How many examples are there of each character? What kind of variation is present between examples?

- Illustrative figures showing an example or two of each character you want to distinguish.

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.