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/`.

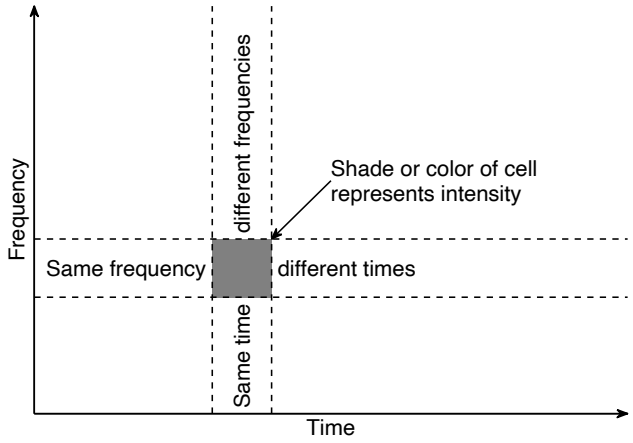

Figure A.1: Schematic illustration of the structure of a spectrogram.

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.

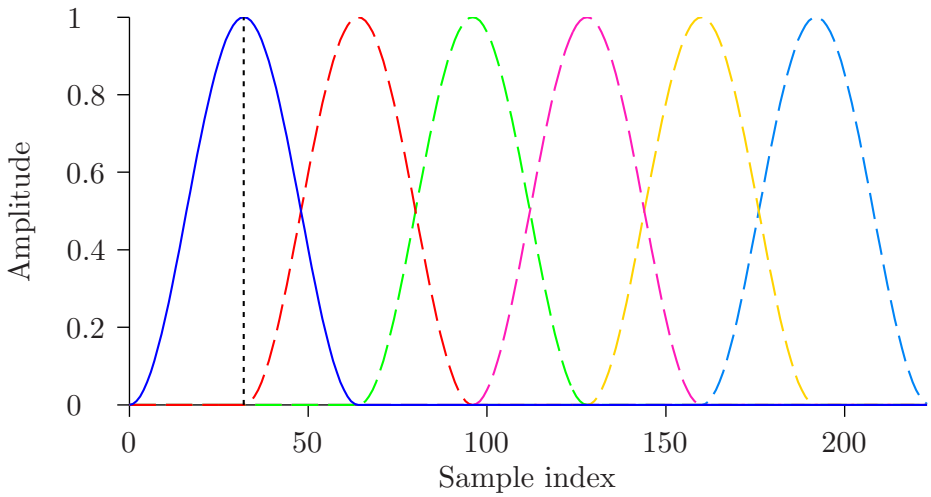

Figure A.2: A set of translated Hann windows of length 64 for short-time spectrogram analysis. A dotted line marks the centroid of the first window.

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

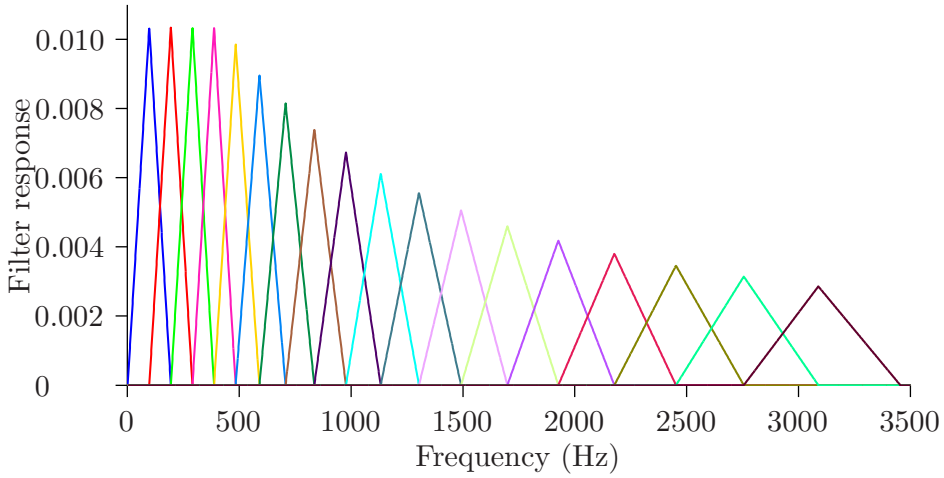

Figure A.3: Frequency responses of a mel-cepstrum-style filter bank.

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.