`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 $(\boldsymbol{x}_{t+1} \dots \boldsymbol{x}_T)$, given an HMM λ and the state $S_t = i$ at time t. The result is known as the *backward variables*, defined through

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

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 $(c_1 \dots c_T)$ 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 $\hat{\beta}_{j,t}$, $t = 1 \dots 3$ 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.