Guidelines for working with Markov models in Matlab with a discussion of their applications in information processing

Illustrated by an identical example used in the book "Speech and Language Processing", 3rd edition, by Dan Jurafsky and James H. Martin, Stanford University, USA

prof.dr.sc. Davor Petrinović
University of Zagreb
Faculty of Electrical Engineering and Computing
Zagreb

Version 0.9

November 2020, Zagreb

Content

Guidelines for working with Markov models in Matlab with a discussion of their applications in information processing

Introduction

To make it easier to understand how to work with HMM models and to use the appropriate functions from the corresponding program libraries, it is still necessary to explain the differences between the types of HMM models. So, in addition to the HMM models with discrete observations that we covered in the lectures on Information Processing, there are also HMM models with continuous and semi-continuous observations. Software packages will typically support all of these HMM models, so in the next few chapters we will describe the differences between each type of HMM model. Also, in order to recognize the importance of these models in many engineering applications, as a motivation for their adoption, in this introduction we will show how these models are used in systems for automatic (or machine) recognition of spoken words from acoustic notation, which in English language is typically called Automatic Speech Recognition, ASR. Examples of such commercial systems that most of you use nowadays on a number of platforms are Siri on Apple phones, OK Google and Google Assistant on Android platforms, Alexa from Amazon on a number of "Alexa-enabled" embedded computer systems, and Cortana for the Microsoft software environment, as well as a number of other lesser known solutions. It should be noted that these systems did not appear "overnight", because in their development a huge effort has been invested of hundreds of world scientists over a period of at least 30 years. Hundreds of scientists have received their doctorates in this field, and numerous scientific papers have been published in journals dedicated to this narrow topic of the application of machine learning in the processing of natural speech. It can even be argued that one of the most significant drivers of the development of machine learning theory and data science was precisely the field of natural speech and natural language processing. These theoretical approaches were later applied in other scientific disciplines, as is common in the dissemination and takeover of scientific knowledge.

Types of observed data and their statistical representation in HMM models

As part of the introductory presentation on the subject of Information Processing, we dealt exclusively with models with discrete observations, e.g. candy colors extracted from bowls, or the whole number of ice creams eaten in a day and the like. In both cases, the probabilities of symbol observation in individual states of the model can be described by histograms, i.e. by counting how many times each possible symbol from the dictionary in a particular state of the model was observed for given series of observations. In actual applications, and thus in automatic speech recognition procedures, the observed quantities are still discrete in time, but not of discrete values. It is also not just one quantity that is observed (e.g. color or number), but a whole range of values that are characteristic of the current properties of a speech signal at that short time window of a typical duration of 10 to 20 milliseconds. Although we will not really learn how to use HMM models to describe such processes, whose observations are from the set of real numbers R^N, in these introductory chapters we will still give some basic information on how HMM models can be directly generalized to such processes with a "continuous dictionary" of symbols.

Feature vectors

A set of quantities that describes the short-term properties of speech is called a feature vector. Numerous authors have proposed a variety of representations with the goal of transforming the properties of speech as we experience it into a compact numerical representation with as few parameters as possible for description. The feature vector is well chosen if it is consistent with the psycho-acoustic model of sound perception. This means that the two feature vectors should be close if the corresponding voices sound similar to us and vice versa, the two vectors should be distant if the voices to us are significantly different. Today, one of the most commonly used feature vectors for applications in speech processing is the so-called *MFCC vector*, which stands for *Mel-Frequency Cepstral Coefficients*. Without going into the details of its calculation, we say that it describes the short-term spectral properties of speech (logarithm of the magnitude of the spectrum), but calculated above the nonlinear frequency scale, also called the *Mel scale*, as opposed to the ordinary DFT transformation defined above the linear frequency scale. Ultimately, such a log-spectrum is transformed back to the time domain by a *discrete cosine transformation*, and as such it represents a kind of spectrum of the log-amplitude spectrum calculated above the nonlinear frequency scale. The fact that it is actually a “spectrum of the spectrum” is also the reason for the genesis of its original name "*Ceps-trum*", which is created by reversing the first four letters of the English word for the spectrum "*Spec-trum*", and hence the corresponding adjective "*Ceps-tral*" instead of "*Spec-tral*" . Ultimately, it turns out that only about 12 cepstral coefficients are sufficient to faithfully describe the properties of a short speech segment with duration of approximately 20 to 30 ms. This represents a significant reduction in dimensionality, because with a typical sampling frequency of 44.1 KHz, this waveform segment had nearly a thousand time samples (882 to be exact), which are summarized by this representation in these 12 cepstral coefficients, which faithfully describe our experience of that short voice section. Also, this representation is suitable for the additional reason, that our perceptual experience of the distance between two different voices is very well predicted by the ordinary Euclidean distance between the two corresponding MFCC vectors.

The rate of MFCC analysis

The feature analysis procedure must be repeated more often than the reciprocal value of the width of the analysis window used, which means that feature vectors are typically calculated about 100 times per second, i.e. with an analysis frame shift of 10 ms. This means that there is always a deliberate overlap of adjacent analysis frames, so that all the details of the temporal evolution of these features are faithfully preserved. In particular, at a typical pronunciation rate, it takes one second to say the number twenty-one in Croatian: “*dvadeset i jedan*”. Thus, these two words connected by a conjunction ‘*i*’ are composed of 14 phonemes and two pauses which will ultimately be described through MFCC analysis with approximately 100 feature vectors of dimension 12, i.e. with a total of 1200 real numbers. If we divide the number of vectors by the number of phonemes (and pauses), then we see that on average each phoneme lasts about 6 consecutive vectors. However, some voices, such as vowels, have longer duration, so the number of vectors describing an individual voice will actually depend on the sound: from 3-4 vectors for short explosive consonants, to as many as ten or more vectors for extended pronunciation of vowels.

Modeling the temporal evolution of speech feature vectors

So now that we have a vector of speech features, we need to somehow describe the probability of observing a specific combination of these 12 real values in each state of the model. The hidden states of the HMM model will model individual sounds that are uttered when pronouncing a single word or when pronouncing only one part of a word such as a syllable. The required number of model states depends on the number of phonemes in the sequence, so modeling only shorter parts of words will reduce the required number of model states, which will give simpler models. In the sequel, we will use the term syllable to describe such a short sequence of sounds, although in real applications only pairs of adjacent sounds are used, the so-called *diphones*, or in a more complex embodiment, segments of three consecutive phonemes that are called *triphones*. Words are naturally formed by concatenating sounds, so if each state models one characteristic sound, or the so-called *phoneme* in sequence, the whole HMM model will then describe the pronunciation of the syllable (diphone or triphone) or perhaps in more complex models with a larger number of states even the whole word. Now that we have described the model itself, we must further explain how to describe the probability of real valued vector observations in individual states of this model. One approach to solving the problem of how to use discrete HMM models to represent the temporal evolution of speech feature vectors is based on converting this real vector into an index, thus turning continuous representation back into the desired discrete one that we know how to model, so we will describe such a procedure in the next chapter.

Vector quantization for the purpose of discretization of a real vector process

As a simpler example, let us imagine that by means of our HMM model we want to describe the probability of a male person belonging to a certain anthropometric group, where hidden states will model the time course of his physical development from childhood to adulthood. If we choose three standard variables of anthropometric characteristics defined as measures: skeletal longitudinality, body weight and body volume, which we measure as body height in cm, body weight in kg, and as forearm circumference in cm, then for each person we can determine its position in the R^3 feature space in each state of this HMM model. It is to be expected that these vectors are grouped in certain regions of this space, because the selected variables are interdependent. It can also be expected that the described groupings are different depending on the state of the model (person's age) and the anthropometric type of person. Thus, we could divide all the observed vectors into a finite number of groups. As a suitable metric for classification, we could use Euclidean metrics in that R^3 feature space, or alternatively weighted Euclidean metrics, if certain components of this vector contribute differently to the classification. In both cases, the decision whether a particular vector belongs to a particular class is determined with the minimum (weighted) distance of that vector to the representative of each individual class.

There are two basic problems that need to be solved here: (1) how to classify an unknown vector for given set of representatives and (2) how to choose the number of classes and how to find the best representatives of each of these classes. The answer to these questions is given by the theory of *Vector Quantization (VQ)*, which describes how to solve both problems. Vector quantization is actually a generalization of the "*K-means*" algorithm, which solves the same problem for the scalar case, i.e. for the case that the classification is performed in the R^1 space. Vector quantization theory was developed in the early 1980s, primarily in the context of *digital signal processing*, and specifically in the context of *signal coding for lossy compression*. All today's encoders of media signals (speech, audio signals, images, video signals, etc.) are based on the procedures of vector quantization, due to its high compression efficiency, but also due to the numerical simplicity of its implementation. With the recent development of data science field, today we recognize that vector quantization is based on the paradigm of *competitive learning*, and is therefore closely related to models of *self-organizing maps* and models of *sparse coding* that are extensively used in "*Deep learning*" field, and specifically in so called *autoencoders*.

In conclusion, with a constructed vector codebook, i.e. with an optimally determined set of representative vectors, each unknown vector can be labeled by the index of the class to which it belongs, based on the criterion of the smallest distance, thus obtaining discrete symbols from the finite dictionary. The selected number of representatives is exactly that dimension of the dictionary (number of possible output symbols). The bounded regions of feature space that belong to a particular representative also have a special name and are called Voroni cells, after the Russian mathematician Georgy Feodos'evich Voronoi; (Гео́ргий Феодо́сьевич Вороно́й) 1868 –1908 who studied this area in the nineteenth century.

Disadvantages of discretized models for describing observations

The disadvantage of the previously described procedure of discretization of the real valued observation vector is primarily in the fact that part of the information in that procedure is irretrievably lost, because vector quantization, as well as ordinary scalar quantization are not invertible operations. If the input vector is located exactly on the edge between two classes, it will eventually belong to only one of these two classes, although it is actually equally distant from both representatives of those two classes, i.e. it does not actually belong to either. If we chose a larger number of classes, the quantization error might be smaller, because the representatives would be more densely distributed in the feature space. However, the representation error will always exist, i.e. it is not possible to reconstruct the original real value of the input vector from its class index, since the actual value is replaced with the value of the representative of the class to which the vector is assigned.
Such classification can be considered a "rigid" decision, because we have actually defined that it is 100% probable that the vector is in that class and 0% in all other classes, provided that the vector belongs to the Voroni cell of that nearest class. However, we have no information in which part of the Voronoi region the vector is actually located, whether it is in the middle, or perhaps at the edges of that cell. For the case of an input vector located near a generalized bisector between two classes, it would be more logical to assume that the probability of belonging to these classes is 50% and 50%, not 100% in only one, because the vector was only infinitesimally closer to the center of the one chosen.

HMM models with continuous observations

The described problem, which is inevitable in discretized HMM models, is solved by using HMM models that have continuous observations (also called models with infinite dictionary), whose probabilities of observing the real valued vector are described by selected parametric models of probability densities. The most well-known such parametric model is the Gaussian distribution defined in N-dimensional space, because it is compact in description and at the same time well adapted to numerous natural processes. For the previously described problem of modeling anthropometric classes, we could build a model of Gaussian distribution in 3-dimensional space from the whole set of measured data. Such a model is described with a relatively small number of parameters, namely, the three-dimensional expectation vector (point in R^3 feature space where the origin of this distribution is placed) and a covariance matrix of dimension 3x3 describing scatterings in all directions, i.e. the shape of data distribution around its expected value. Since this covariance matrix is always a symmetric matrix, it contains only 6 unique values in total, which together with the three expected center values makes a total of 9 free parameters of this density.
Such a model is called the *Gaussian probability density model*, and by evaluating this density function of given parameters, it is easy to determine for each unknown vector the likelihood

of its belonging to that density, which will represent the emission probability of the HMM model in a particular state. Thus, in the HMM model, we could associate each state with one such parametrically described density, the parameters of which are adjusted to the statistics of that vector observation in that particular state.
Analogous to the procedure of training discrete HMM models, where we iteratively adjusted the matrix of discrete observations B for each state of the model, in the process of training continuous HMM models training is reduced to iteratively finding optimal Gaussian density parameters in each state of the model, i.e. the mean vector and the covariance matrix. Such parametric description solves the problem of "rigid" decision of discretized observations, because the distance of the input vector from the center of density is directly reflected in the likelihood of this vector given the Gaussian model. The closer to the center, the argument of the exponential function will be smaller (closer to zero) and its probability higher, and therefore continuous HMM models can much more accurately describe the statistics of observations in individual states.

HMM models with observation probabilities based on Gaussian mixture models

Finally, instead of associating independent separately trained Gaussian densities with each state of the HMM model, which ensures the highest modeling accuracy, there is another alternative, called *semi-continuous HMMs*. It is based on a similar assumption on which the procedure of discretization of observations by vector quantization is based. Thus, we could divide the input vector process that we model with HMM into M classes, but which are not described solely by the representation vector of each class as in vector quantization, but are rather described by complete Gaussian density. The dimensionality of the feature space in which this Gaussian density is defined is determined by the length of the observation vector. So, again for the same anthropometric example, one might wonder if we can statistically describe the whole set of all measured data (each data value is a vector of three numbers) using a finite number of Gaussian distributions in R^3 space, where at this point we will not try to relate these classes to states of the model (i.e. time course of a person's physical development). We will build this model solely on the basis of the criteria of the best possible statistical representation of the entire data set by the model.
Such a statistical model where the data set is not described by a single Gaussian density but by a combination of several such densities is called a *Gaussian Mixture Model (GMM).* With the chosen number of M mixture components, it is possible to determine the set of M Gaussian densities that will best represent the entire data set of vectors. If the quality criterion of GMM model is the log-likelihood of observation of all data set vectors with a given model consisting of a mixture of Gaussian distributions, then the process of learning the optimal parameters of the components of this mixture can be achieved by *Expectation Maximization (EM)* algorithm. It is based on the same theoretical principle as the iterative procedure we use to build the optimal parameters of the HMM model. For Gaussian mixture model, it is necessary to define additional scalar weighting factor for each component of the mixture, because the GMM model by its definition is the simple weighted sum of the Gaussian components that make it up. For GMM to continue to be a probability density function, its integral over the entire domain must be equal to unity, and this will be achieved if these weights of the individual components add up to one, because each component of the mixture already meets the unit integral condition.
By tuning these scalar weights, we give appropriate importance to each individual component of the mixture. For example, if the first component has the highest weight, significantly higher

than all the others, then the characteristics of that GMM model will be very close to the properties of the first density that makes it up. If the first two weights are equal and the others are close to zero then by such a mixture we are actually describing an equal combination of the first two components and analogously for the other weight combinations. The described property allows us, in semi-continuous HMM models, instead of associating states with Gaussian density parameters that are individually optimized for each state, to actually associate states of HMM models only with weights of individual component mixtures, while components themselves were trained in advance based on the whole data set regardless of the state in which they were observed.
The described simplification significantly reduces the number of parameters of the HMM model, because now instead of describing the entire Gaussian density in each state of the model we only have a set of weights (defining significance or importance) with which individual densities are observed in that state, while all states share the same set Gaussian components which were trained on the entire data set. Hence the name of this model, "*semi-continuous model*" because it is based on a discrete number of possible Gaussian components, which are only weight combined in individual states of the HMM model. These weights can be any set of positive real numbers that satisfies the condition that they add up to a unit. By selecting appropriate weights, we can compactly describe the emission probabilities of individual states. In the process of the HMM model training, only these weights are updated for each state of the HMM model such that the total likelihood of observation sequence of real valued feature vectors used for training is maximized (as high as possible). The parameters of the Gaussian components themselves are not touched during HMM training, because they are already predetermined when building the GMM model over the entire data set of observation vectors.

Training of HMM models for real applications

In reality, HMM models are trained not only with one such sequence of observed feature vectors, but with multiple series of observations of potentially different lengths, all belonging to the same class of models, based on “manually labeled” input data. This process of manual labeling is also called *annotation* and often requires extremely great human effort in the preparation of such reference training materials. Often the same material is annotated in parallel by a number of independent annotators in order to be able to unambiguously confirm the final valid label (*ground truth*). In all large IT corporations that offer automatic speech recognition systems, this manual annotation task is performed by entire "armies" of educated human listeners, who must listen to and label (transcribe) hours and hours of such speech material in order to expand the data set to be used in model training.
It is precisely in these events that the problem of insufficient *protection of personal information* has recently been highlighted, as often the persons whose speech materials (recordings) are used were not aware that they were being recorded, or did not give their direct consent to use their voice recordings, i.e. they didn’t expect that recording will be listened by human operators rather than just a computer (robot).
In the context of syllable recognition for the purpose of automatic recognition of *natural connected speech* (spontaneous as opposed to dictated), where one HMM model describes one syllable (in reality it is typically a diphone or triphone), the model training data set consists of multiple sequences of MFCC vectors collected when pronouncing the same syllable by one or preferably more different persons, in all possible words where that same syllable occurs.
In the case of multiple observation sequences, the criterion used to train the models is that

the product of the observation likelihoods of all these series is maximal for such optimally determined model coefficients. When forming a training data set, it is necessary to ensure sufficient representativeness, because the built model will be optimized for this used set. If, during the actual use of the model for the purpose of determining the class of an unknown sequence of feature vectors, a completely new observation appears that still belongs to the same class according to the "manual label" but which differs significantly from the sequences used for model training, then it can be expected that the likelihood of this observation will be significantly less than the likelihood of the training sequences, which may cause a wrong decision.

As an example, if the pronunciations of only adult male speakers were used to train the model, and in actual use the pronunciations of female or child speakers are also used for model evaluation, then significantly worse recognition results can be expected because the model is not built on a representative sample. In the formal training procedures of these HMM models, similar *cross-validation procedures* are applied as in other machine learning algorithms (by dividing the total database into a training set and an exclusive evaluation set, and repeating this partitioning by randomly selecting part for the training and part for evaluation with a predefined ratio, typically 90% for learning and 10% for evaluation). Cross-validation procedures can be used to verify that a learning set is sufficiently representative, because if any random partition gives comparable accuracy on the evaluation set, this indicates that the training set is sufficiently representative. However, such a conclusion is valid only if the whole data set is truly representative of the expected application, because if new observation sequences appear in the application that were not contained in this initial set in any form, the behavior of the system will be completely unpredictable.

Syllable recognition and concatenation of syllable models into word and sentence models

Finally, we must also explain how the computer distinguishes between different syllables, i.e., how the final decision is made about the unknown observed sequence of feature vectors. The principle is based on building a series of parallel HMM models, one for each syllable that need to be recognized for a given set of words that the system will be trained to recognize. The unknown sequence is evaluated simultaneously with all these pre-trained HMM models, and the model that gives the highest likelihood of observation of that unknown sequence is selected. These models are evaluated continuously and in parallel and their "activations" are monitored, i.e. which model gives the greatest likelihood at given time instance. The sequence of activations of these models indicates a possible sequence of uttered syllables, which then form words, and then the words form sentences.

To model these higher hierarchical structures, hidden Markov models can again be used successfully, only that they then model the conditional probabilities of syllable strings within words or the conditional probabilities of word strings within a sentence. This is especially important in the case where the accuracy of this acoustic level recognition of voice sequences is relatively low. Unfortunately, this is most often the case, precisely because of the very high acoustic variability of pronunciation, so it is extremely difficult to determine with high certainty which model is the right one, especially when the number of such parallel HMM models is extremely large. Specifically, if we take the Croatian language with approximately 30 phonemes as an example, then the total number of different triphones would be 27000 = 30*30*30. However, many of these triphones do not exist in actual language, mostly because they are not pronounceable, so in reality the number of such parallel HMM models of Croatian triphones is about 9000. Thus, the computer should evaluate the likelihood of observing this unknown observation sequence given all 9000 pre-built models and choose the model that gives the greatest likelihood. Since the probability of wrong selection with so many models is rather high, instead of only one best candidate, i.e. instead of only one best HMM model of the recognized syllable currently activated, a list of several possible candidates is forwarded to higher levels along with their evaluated acoustic observation likelihoods. The final decision on the "proper" recognized syllable is left to the higher hierarchical level of the model, which describes the conditional probabilities of syllables concatenation, and ultimately selects the combination from the acoustic recognition layers that forms a valid word from the dictionary with the highest likelihood of all possible words in the dictionary.

The same principle is then used for the highest hierarchical level, which chooses from such series of multiple candidates of valid words those candidates whose likelihood of observation is the highest, while respecting the model of valid lexical sentence structure, which is also known as the *language model*. By combining these multiple levels of recognition, the overall accuracy of the speech recognition can be improved significantly. Commercial speech recognition systems today have accuracies of at least 95% or more, meaning that out of 20 spoken words in a row, only one may be mistaken, making them usable in a number of everyday applications. The recognition accuracy at the lowest acoustic level does not necessarily have to be equally high, because it is the addition of these higher hierarchical layers that ensures the final accuracy of the system operation. Typically, the recognition accuracy at the acoustic level is around 60-70%, which means that only in two of the three cases is the syllable model that had the highest likelihood of acoustic level observation the one that is ultimately selected, while in the remaining third of cases alternative candidates are selected which are lower on a sorted list of models according to the likelihood of acoustic observation.

Algorithmic latency of natural speech recognition systems

The described hierarchical decision-making structure of the ASR system for recognizing natural connected speech also has one undesirable consequence, and that is the inevitable latency of the system. The final decision on the recognized first syllable in the first word in the spoken utterance may be made only when the computer "listens", collects and processes the whole sentence, because all hierarchical layers must evaluate the conditional probabilities of left and right context and from all multiple candidates select those combinations that will produce the greatest likelihood of observation at the level of the entire spoken utterance.

It is very much similar to the way that people in the brain process the speech message they have heard. Although they may not have understood every word spoken due to the adverse surrounding noise, they will still be able to reconstruct the original spoken message when they listen to the whole sentence. Often people in such conditions of impaired communication will need to briefly "think" what the interlocutor could say, similarly combining the conditional probabilities of left and right context for individual words they could not understand due to the surrounding noise, but also respecting the context of the conversation. Therefore, modern ASR systems are closely mimicking the natural speech recognition process in the human brain.

When using the automatic speech recognition system, the user should not be surprised by such an inevitable algorithmic delay of the system, and in doubt why the recognized spoken words are not displayed immediately, immediately impatiently repeat the query or parts of the query. Such repetition will only complicate or even disable the ASR system in successful sentence recognition.

Contextually adapted speech recognition systems

In some applications, it is possible to significantly reduce the dictionary size of all possible valid words. Specifically, if it is a narrower context of the application of the ASR system (e.g. automatic transcription of the financial report on the stock market), then in fact the total number of different words used is relatively modest (from a few hundred to a maximum of several thousand). This automatically reduces the number of required syllable models (triphones), because in this subset of valid words, half of the model may be unnecessary, because they never appear as part of any valid word in the dictionary used. This reduces the so-called *perplexity* of the problem, which will automatically ensure greater accuracy of recognition, because the number of possible outcomes from which to choose the right one is significantly smaller. Also, depending on the context of application, the language model, i.e. the model of sentence structure can be very specific. Only certain constructions of common sentences are used for meteorological report or weather forecast in relation to the sentences that the political commentator will use in his newspaper article. Therefore, state-of-the-art automatic speech recognition systems try to independently determine the context of current use, and then without the need for user intervention choose models that are most appropriate and specially built for that automatically identified context (e.g. hairdressing appointment service, or e.g. table reservation in the restaurant, etc.).

From all of the above discussions, it is clear that for the real-world ASR application is actually a huge number of HMM models that all must be relevantly trained, in order to generalize their decision for completely new and unknown observations that may occur when using the system. For this reason, the advantage of semi-continuous HMM models is understandable, because they have a significantly smaller number of "free parameters" of the model, which automatically improves training processes and robustness of the model. For relevant model training, the number of observations used must always be several orders of magnitude greater than the total number of model parameters, so this is the main reason why so much human and financial effort is invested in continuously upgrading the datasets used to re-train these models. With everything said, we can conclude that the system for automatic speech recognition is indeed completely built and defined by data (i.e. *data driven*), because the algorithm itself is like an empty book with "white pages", which only based on the actual content learned from input data becomes a "bestseller".

Toolbox used for laboratory exercises with HMM models

Since the hidden Markov models belong to the field of machine learning in the class of algorithms for unsupervised learning, and in Matlab machine learning is supported in a special package "Statistics Toolbox", it includes five basic functions specifically for working with HMM models:

hmmdecode	Hidden Markov model posterior state probabilities
hmmestimate	Hidden Markov model parameter estimates from emissions and states
hmmgenerate	Hidden Markov model states and emissions
hmmtrain	Hidden Markov model parameter estimates from emissions

hmmviterbi	Hidden Markov model most probable state path

However, instead of this “native” Matlab implementation of HMM functions, another more comprehensive Toolbox will be used for the Information Processing lab exercise, which is specifically written to support the HMM models within Matlab, even before Matlab developed its own support. This is a package called:

Hidden Markov Model (HMM) Toolbox for Matlab
Written by Kevin Murphy, 1998.
Last updated: 8 June 2005.
Distributed under the MIT License

The University of British Columbia, UBC, Vancouver Campus

https://www.cs.ubc.ca/~murphyk/Software/HMM/hmm.html

A brief description of this package lists its key functionalities, which should be clear to you based on the introductory chapter in which we described the various types of HMM models:

„This toolbox supports inference and learning for HMMs with discrete outputs (dhmm's), Gaussian outputs (ghmm's), or mixtures of Gaussians output (mhmm's). The Gaussians can be full, diagonal, or spherical (isotropic). It also supports discrete inputs, as in a POMDP. The inference routines support filtering, smoothing, and fixed-lag smoothing.“

This package supports a much wider set of functionalities for working with HMM models compared to the introductory description presented in the lectures, so as part of this exercise we will explore working with only a few basic functions of this package for modeling HMM models with discrete observations.

Instructions for download and installation of the "HMM Toolbox" software package

This Toolbox can be downloaded as a single ZIP file from the following link:

https://www.cs.ubc.ca/~murphyk/Software/HMM/hmm_download.html

Number of hits since 23 October 2002:

Download
Click here. Unziping creates a directory called HMMall, which contains 4 subdirectories.

The full link (behind the text 'here') for ZIP file download is:
https://www.cs.ubc.ca/~murphyk/Software/HMM.zip

Installing the HMM Toolbox in Matlab

The same ZIP file was also uploaded to the materials for the 6th and 7th lectures on the IP (OI) course website. For installation purposes, you need to unzip this ZIP archive to a folder called

HMMall that can be located anywhere in your computer's file system, but it's best to put it in a Matlab working folder that will include your remaining source files. Matlab must have access to the folder where all the programs that are an integral part of this Toolbox are located. You can add this new folder to the search path of the files used by Matlab with this command below (in the example it is assumed that the HMMall folder is located in the root of the C drive (C:/), but it can be anywhere in the tree structure, only then you must correctly specify the path to that folder as an argument to the genpath function).

Assuming you unzip it to C:/HMMall...
>> addpath(genpath('C:/HMMall'))

To quickly verify that the Toolbox is installed correctly, run this script:

>> testHMM

We can now demonstrate the most important functions from this Toolbox on an identical example of weather modeling based on records of the number of ice creams eaten in a given day that we used as a basis for lectures. All three basic tasks related to HMM models (likelihood calculation, decoding and model learning) will be demonstrated by calling the appropriate functions from this Toolbox.

Examples of working with the software package "HMM Toolbox" in Matlab

With these following examples, we will try to follow exactly the presentation in the document that we used for the lectures from the IP (OI) course, exactly for the identical example that was used to illustrate the operation of discrete HMM models. The achieved results of performing these key functions will then be compared with the expected results presented in the lecture materials.

Calculation of observation likelihood for a given model

Let's start first by defining the model parameters. Although in the ‘weather & ice-cream’ model the hidden states are marked with the letters C for C(old) and H for H(ot), in Matlab we have to index the states with numbers, so we select:

```
% State indexing
% C(old) state will be the state 1
% H(ot) state will be the state 2
%
% Colum vector with initial probability distribution
%
prior0=[
    0.2  % C(old) state
    0.8  % H(ot) state
];
```

In a similar way, we define a transition probability matrix A that defines the probabilities of state change, where the first row corresponds to state C and its probabilities of transition to state C and H, while the second row corresponds to state H and its transitions to state C and H:

```
% Probability matrix A of state transitions
%
% a11 a12
% a21 a22
%
transmat0=[
0.5 0.5       % P(C|C) P(H|C)
0.4 0.6       % P(C|H) P(H|H)
];
```

The sum of each row must be equal to one, because it shall be a probability distribution. We also introduce the variable Q which will denote the number of model states, because it will be used as an argument in some functions:

```
Q=size(prior0,1);
```

Finally, we also define the emission probability matrix B, i.e. for each state we define the probability of observing all output symbols from the dictionary. Our output dictionary has only three symbols "1", "2" and "3" which indicate the number of ice creams eaten per day. If the output symbols were the colors of the candies drawn from a white or green bowl, as we illustrated in a live experiment at the beginning of the first lecture, then we would have to associate each color with a numerical index just as we indexed the hidden states H and C, since Matlab is only supporting indexed (numeric) dictionary symbols.

```
% Emission probability matrix B
% each row corresponds to one state, and
% each column corresponds to one output symbol
obsmat0=[
0.5 0.4 0.1    % P(1|C) P(2|C) P(3|C)
0.2 0.4 0.4    % P(1|H) P(2|H) P(3|H)
];
O=size(obsmat0,2);
```

The variable O will represent the total number of dictionary symbols, which in our example is 3 (one, two or three eaten ice creams).
We now define a specific series of observations, for which we will try to determine its likelihood of observation with a given model using the "Forward" algorithm. We will choose an identical series of three observed symbols "1 3 1" in time instances t = 1, t = 2 and t = 3, which was also used in lecture materials.

```
data=[3 1 3];
```

The length of this observation sequence T can be arbitrary, and as described in the introductory chapters, we can generally have multiple series of observations of different lengths all belonging to the same HMM model, so using multidimensional matrices in Matlab for their storage is not suitable. Such a data structure expects a fixed (identical) dimension of all observation series along all "axes". Instead of multidimensional matrices, a „cell“ can be used in Matlab, because this data structure in each cell can contain a numerical element of arbitrary dimensionality (and type). E.g. in the first cell there can be a row with 10 elements, in the second cell a column of 17 elements and finally in the third cell a 3x7 matrix, etc. This "cell" data structure is therefore much more flexible and allows each element to have a row with a series of observations of arbitrary length T. The above defined observation row 'data' is therefore converted into such a cell structure by this sequence of commands:

```
if ~iscell(data)
  data = num2cell(data, 2);
end
ncases = length(data);
```

The variable ‘ncases’ will contain the number of observation sequences in the new cell structure ‘data’, and for our simple example this will be 1 because we specified only one observation string. We will now execute the following Matlab excerpt for these default observation symbols and for the given model, which processes the observation sequences one by one in the loop and calculates their observation likelihood for the given model. This likelihood P $(O|\lambda)$ is usually represented in logarithmic domain with the natural logarithm (base e), and in this form it is called the *log-likelihood* of the observation O. The use of the logarithm is significant, because by increasing the length of the observed sequence T the likelihood P $(O|\lambda)$ as a rule assumes very small values. In the Forward algorithm, probabilities are multiplied that are all numbers less than one, and such small products are summed and forwarded to the next time step in accordance with the induction of the Forward algorithm, which again results in their further reduction of values. If instead of one observation series O we have two series O1 and O2 which we assume belong to the same HMM model λ then we can define their likelihood of simultaneous observation as the product P $(O1|\lambda)$*P $(O2|\lambda)$, which in the case of logarithmic representation corresponds to the sum of two log-likelihoods:

$$\log(P(O1|\lambda) * P(O2|\lambda)) = \log(P(O1|\lambda)) + \log(P(O2|\lambda))$$

The program section below will just calculate (accumulate) such log-likelihood to determine the total log-likelihood of the whole set of observational sequences of arbitrary lengths stored in the 'cell' structure 'data'.

```
loglik = 0;
errors = [];

for m=1:ncases
  obslik0 = multinomial_prob(data{m}, obsmat0);
  [alpha, beta, gamma, ll] = ...
      fwdback(prior0, transmat0, obslik0, 'scaled', 0);
  if ll==-inf
    errors = [errors m];
  end
  loglik = loglik + ll;
end
```

To better understand the method of calculating log-likelihood, consider the first (and only in this case) iteration of this loop for the first series of observations of length three symbols. If we fix the loop index to m = 1:

```
>> m=1

m =

   1
```

This is the contents of that first (and only) cell of the 'data' structure:

```
>> data{m}

ans =

   3    1    3
```

Now we run only the first command from the loop:

```
>> obslik0 = multinomial_prob(data{m}, obsmat0)

obslik0 =

  0.1000    0.5000    0.1000
  0.4000    0.2000    0.4000
```

We see that the matrix 'obslik0' has a T (= 3) column and a Q (= 2) row and that it contains the probabilities of observing the selected sequence "3 1 3" in the first state of the model (in the first row of the matrix for state C) and in the second state of the model (in the second matrix row for the state H). These are the following conditional probabilities:

```
% P(3|C) P(1|C) P(3|C)
% P(3|H) P(1|H) P(3|H)
```

As this is an HMM model with discrete output probabilities (because we have a finite number of output symbols), this function ('*multinomial_prob*') simply recalls the values of the output probability matrix B (which we named 'obsmat0') based on the index of observed symbols for a given set of observations in the 'data {1}' field. Thus, we converted the numerical indices of the output symbols into the corresponding observation probabilities for each state of the model and for all time instances t=1 to t=T=3. This auxiliary matrix 'obslik0' will then be used as an input argument to the '*fwdback*' function which will implement the "Forward" algorithm.

“Forward” algorithm

```
  [alpha, beta, gamma, ll] = ...
      fwdback(prior0, transmat0, obslik0, 'scaled', 0);
```

This function returns in the output matrix 'alpha' the calculated forward probabilities for each hidden state of the model α_t(state) for each time instance t=1 to T, where the rows again correspond to states, while the columns correspond to the time moments of observation. Note that the same function can, based on the same set of input arguments, calculate the backward probabilities of the HMM model β_t(state) for t=T, T-1,… down to 1, and return them in the output matrix 'beta'. Finally, it also returns the 'gamma' matrix calculated from the forward and backward probabilities of each trellis node, which is used in the algorithm for training the parameters of the HMM model in the so-called 'the third HMM task'.
Let's look at what we got in the output matrix 'alpha':

```
>> format long
>> alpha

alpha =

   0.020000000000000   0.069000000000000   0.005066000000000
   0.320000000000000   0.040400000000000   0.023496000000000
```

Let us compare these values with the probabilities shown in Figure A.5 in the lecture materials that illustrate the application of the "Forward" algorithm for an identical example. We recognize that the first row of the matrix corresponds to the hidden state C(old) which is indexed as state 1, while the second row corresponds to the state H(ot) which is indexed as the second state. Each column of this matrix corresponds to a particular time instant t=1, t=2 and t=T=3. We see that in the second column of this matrix we obtain the forward probabilities $\alpha_2(1)=0.069$ and $\alpha_2(2)=0.0404$ which are manually calculated and marked in Figure A.5. If the observational sequence were limited to only the first two symbols, then the total likelihood of observation of this abbreviated sequence would be exactly equal to the sum of these two probabilities P(„3 1"|λ) = $\alpha_2(1)+\alpha_2(2)$ = 0.069 + 0.0404 = 0.1094

By summing the forward probabilities over all the hidden states (i.e. across all the columns of the matrix) we get the likelihood of observing all the shortened series of observations: "3", "3 1" and finally "3 1 3" for the whole sequence.

```
>> sum(alpha)

ans =

   0.340000000000000   0.109400000000000   0.028562000000000
```

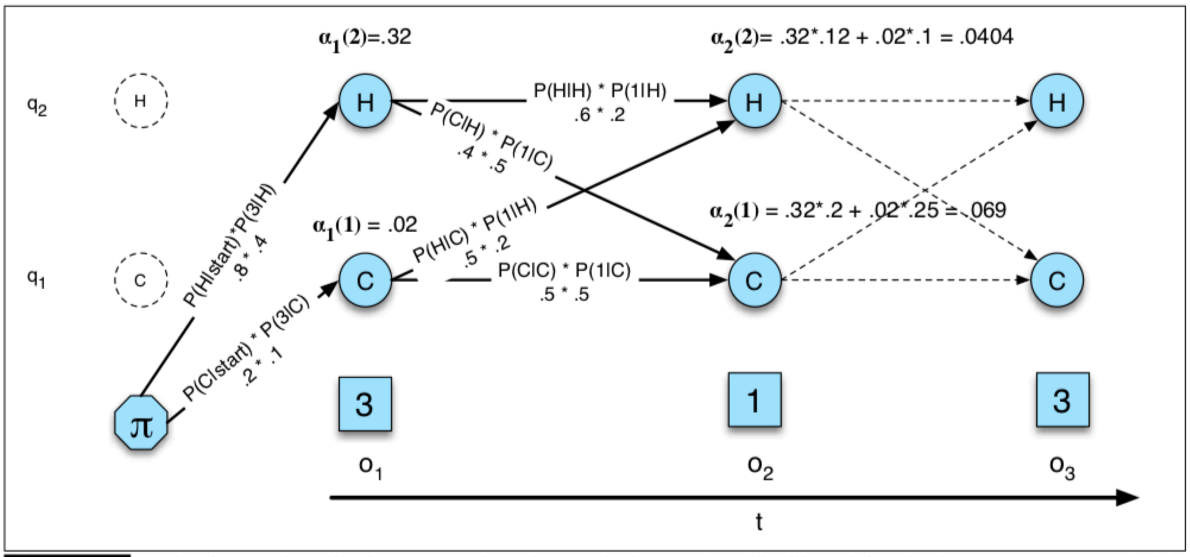


Figure A.5 The forward trellis for computing the total observation likelihood for the ice-cream events *3 1 3*. Hidden states are in circles, observations in squares. The figure shows the computation of $\alpha_t(j)$ for two states at two time steps. The computation in each cell follows Eq. A.12: $\alpha_t(j) = \sum_{i=1}^{N} \alpha_{t-1}(i)a_{ij}b_j(o_t)$. The resulting probability expressed in each cell is Eq. A.11: $\alpha_t(j) = P(o_1, o_2 \ldots o_t, q_t = j|\lambda)$.

Thus, the final sum 0.028562 in the third step represents the likelihood of observing the whole sequence P ("3 1 3"|λ) where this final step is not shown in Figure A.5 but is only indicated by dashed lines for the transition from moment t=2 to moment t=3 in the state trellis (forward probabilities were calculated only up to step t = 2). Everything described is consistent with the definition of the recursive algorithm "Forward":

1. Initialization:

$$\alpha_1(j) \;=\; \pi_j b_j(o_1) \;\; 1 \le j \le N$$

2. Recursion:

$$\alpha_t(j) = \sum_{i=1}^{N} \alpha_{t-1}(i)a_{ij}b_j(o_t); \quad 1 \le j \le N, 1 < t \le T$$

3. Termination:

$$P(O|\lambda) = \sum_{i=1}^{N} \alpha_T(i)$$

“Backward” algorithm

Let’s look at what the second output argument ‘beta’ of the function contains. As we announced, this matrix contains backward probabilities:

```
>> beta

beta =

   0.090500000000000   0.250000000000000   1.000000000000000
   0.083600000000000   0.280000000000000   1.000000000000000
```

It is calculated in the same function from the last column t=T=3 backwards to the first column t=1 in accordance with the definition of the algorithm "Backwards":

1. **Initialization:**

$$\beta_T(i) = 1, \quad 1 \le i \le N$$

2. **Recursion**

$$\beta_t(i) = \sum_{j=1}^{N} a_{ij}\, b_j(o_{t+1})\, \beta_{t+1}(j), \quad 1 \le i \le N, 1 \le t < T$$

3. **Termination:**

$$P(O|\lambda) = \sum_{j=1}^{N} \pi_j\, b_j(o_1)\, \beta_1(j)$$

The third (last) column of the matrix 'beta' corresponds to the initialization step, while the second and then the first column are calculated in the recursion step of the algorithm, specifically for t=2, and then for t=1. The likelihood of the observation of the given sequence "3 1 3" can also be determined on the basis of these backward probabilities β_t(state) in accordance with the expression for the final step (Termination). According to this expression, the initial probability distribution vector, the output probability vector of the particular observed symbol in the first time step (symbol "3") and the calculated backward probabilities in the first column of the beta matrix containing $\beta_1(1)$ and $\beta_1(2)$ should be multiplied.

```
>> beta(:,1).*prior0.*obslik0(:,1)

ans =

   0.001810000000000
   0.026752000000000
```

In Matlab, the sum of these two products of all three factors can be found as:

```
>> sum(beta(:,1).*prior0.*obslik0(:,1))

ans =

   0.028562000000000
```

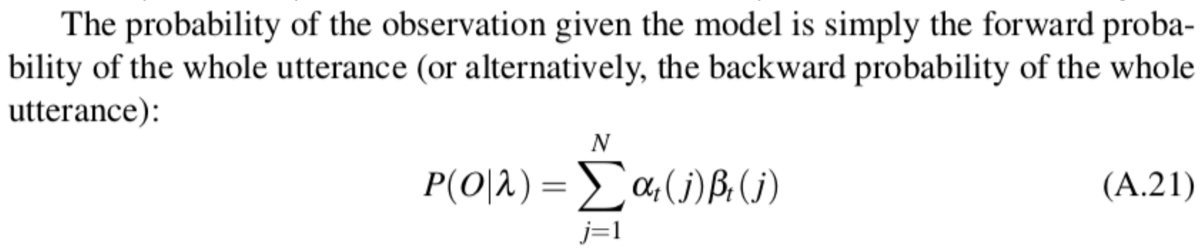
As expected, we see that we obtain an identical solution for the likelihood of the whole sequence P ("3 1 3"$|\lambda$) as with the application of the "Forward" algorithm.

The identical value of likelihood P (O$|\lambda$) must be obtained if we multiply the forward and backward probabilities 'alpha' and 'beta' and add them over all hidden states (interestingly, we get the same value at each time instance for t=1 to T):

```
>> sum(alpha.*beta)

ans =
   0.028562000000000   0.028562000000000   0.028562000000000
```

This corresponds to expression A.21 we derived in the chapter on HMM model training:

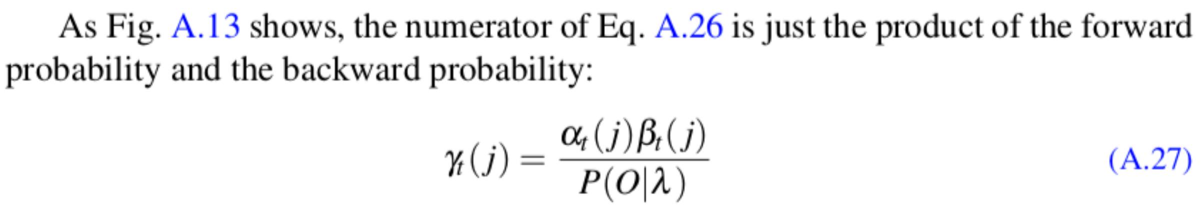


The probability of the observation given the model is simply the forward probability of the whole utterance (or alternatively, the backward probability of the whole utterance):

$$P(O|\lambda) = \sum_{j=1}^{N} \alpha_t(j)\beta_t(j) \qquad (A.21)$$

Calculating the auxiliary probability matrix gamma, for re-estimation of HMM model parameters

Finally, the third output argument also returned by the 'fwdback' function is the 'gamma' matrix defined by expression A.27 in Chapter A.5, which is used in training HMM models from given observations when re-estimating the output probabilities of HMM models in matrix B.



As Fig. A.13 shows, the numerator of Eq. A.26 is just the product of the forward probability and the backward probability:

$$\gamma_t(j) = \frac{\alpha_t(j)\beta_t(j)}{P(O|\lambda)} \qquad (A.27)$$

Let's check whether the calculated output matrix 'gamma' corresponds to the above analytical expression:

```
>> (alpha.*beta)/sum(alpha(:,end)) - gamma

ans =

   1.0e-16 *

                   0                   0                   0
                   0   0.555111512312578                   0
```

We see that these are identical values (up to within the numerical error of the representation in the double float format).

Calculation of log-likelihood of observation for a given model

Finally, the last argument returned by the 'fwdback' function is 'll', log-likelihood of the observation O, which is simply ln(P(O|λ)), i.e. the natural logarithm of the likelihood, which we can easily confirm :

```
>> ll

ll =

  -3.555678115951395

>> log(sum(alpha(:,end)))

ans =
  -3.555678115951395
```

Function for calculating the log-likelihood of the observed sequences for a given HMM model

The described in detail analyzed section of Matlab code which illustrated the way of calculating the log-likelihood of a specific observation O for a given model λ, in several different ways, is actually an integral part of the function 'dhmm_logprob.m' of this Toolbox, that computes the log-likelihood of observations in 'data' for a given model. This observation matrix can be initialized manually as in our example, but it can also be generated automatically based on the given model. Below is the complete Matlab code for this short function:

```
function [loglik, errors] = dhmm_logprob(data, prior, transmat, 
obsmat)
% LOG_LIK_DHMM Compute the log-likelihood of a dataset using a 
discrete HMM
% [loglik, errors] = log_lik_dhmm(data, prior, transmat, obsmat)
%
% data{m} or data(m,:) is the m'th sequence
% errors  is a list of the cases which received a loglik of -
infinity

if ~iscell(data)
  data = num2cell(data, 2);
end
ncases = length(data);

loglik = 0;
errors = [];
for m=1:ncases
  obslik = multinomial_prob(data{m}, obsmat);
  [alpha, beta, gamma, ll] = fwdback(prior, transmat, obslik, 
'fwd_only', 1);
  if ll==-inf
    errors = [errors m];
  end
  loglik = loglik + ll;
end
```

We could call this function immediately with the given input arguments, so we would immediately get the desired log-likelihood of observation, without analyzing all the elements of the calculation in detail:

```
data=[3 1 3];
>> dhmm_logprob(data, prior0, transmat0, obsmat0)

ans =

  -3.555678115951395
```

It should be noted that in the actual implementation of the function 'dhmm_logprob.m', that the function 'fwdback' is called with one additional option „'fwd_only',1" which specifies that instead of calculating all output arguments only the forward probability 'alpha' and log-likelihood ‘ll’ are returned, while empty vectors [] are returned instead of ‘beta’ and ‘gamma’. Also, compared to the previously analyzed section, we see that when calling the 'fwdback' function, we do not use another option that we used earlier, specifically: „`'scaled',0`“. This second option actually requires that the calculation of the forward and backward probabilities be performed according to the original analytical expressions described in the lecture materials, without scaling, while omitting this option is actually identical to setting the default value to „`'scaled',1`“. Such a default option actually activates the automatic scaling of the alpha and beta probabilities, thus achieving greater numerical stability of their calculation. This scaling has no effect on the final log-likelihood (because it is canceled) except that it makes it numerically more stable with long observational sequences.

The task of decoding hidden states by the Viterbi algorithm

Let us now show by which function we will determine the most likely sequence of hidden model states for a given model and observations, which is also known as the decoding task. For this purpose, we use the 'Viterbi' algorithm, and the corresponding function for finding the optimal path in this software package which is called 'viterbi_path.m'. For each function in the package, you can get brief instructions for use by using the Matlab 'help' command, after which you enter the name of the function (or script) that is within the current search path. So for example for this function 'viterbi_path.m' we get:

```
>> help Viterbi_path
  VITERBI Find the most-probable (Viterbi) path through the HMM
state trellis.
  path = viterbi(prior, transmat, obslik)

  Inputs:
  prior(i) = Pr(Q(1) = i)
  transmat(i,j) = Pr(Q(t+1)=j | Q(t)=i)
  obslik(i,t) = Pr(y(t) | Q(t)=i)

  Outputs:
  path(t) = q(t), where q1 ... qT is the argmax of the above
expression.
```

Let us now call this function for the given parameters of the model (note that as the last argument we passed the matrix 'obslik0' which contains the probabilities of observing the given set of output symbols O = '3', '1', '3' in states 1 and 2):

```
% The most likely state sequence
>> vpath = viterbi_path(prior0, transmat0, obslik0)

vpath =

     2     1     2
```

Thus, the most likely sequence of hidden states of the model in time steps t=1, 2 and t=T=3 for the observation '3', '1', '3' is 2, 1, then 2, i.e. H(ot), C(old) and H(ot) using the original symbolic state notations.

Now that we have determined the most probable state path, we can determine the likelihood of observing a given sequence, but only along that most likely state path, which is also known as Viterbi path, and not for all possible paths as we calculated in the 'Forward' and 'Backward' algorithms. For this purpose, we will use the function 'dhmm_logprob_path.m' which is called with the same input arguments as the previous function, except for the row vector 'vpath' (with the desired path) which is added as the last argument:

```
% Likelihood along the most probable state path
[ll, p] = dhmm_logprob_path(prior0, transmat0, obslik0, vpath)

cp=cumprod(p)

ll =

  -4.358310108056565

p =

   0.320000000000000   0.200000000000000   0.200000000000000

cp =

   0.320000000000000   0.064000000000000   0.012800000000000

>>
>> log(cp(end))

ans =

  -4.358310108056565
```

This function returns the log-likelihood of the whole observation sequence in the output argument 'll', but it also returns the row 'p' containing the incremental probabilities along the given path 'vpath'. To obtain the required probabilities of each trellis node along the Viterbi path, it is necessary to calculate the cumulative products of these incremental probabilities from the row ‘p’ which we achieve with the Matlab ‘cumprod’ function. Thus in cp(1) we get p(1), in cp(2) we get p(1)*p(2), and in cp(3) we get the product of all three incremental probabilities p(1)*p(2)*p(3). If we calculate the natural logarithm of this last element cp(3), we obtain precisely the log-likelihood of the whole observation 'll'. Let's compare these solutions with Figure A.10 from the lecture material, which shows exactly this identical example:

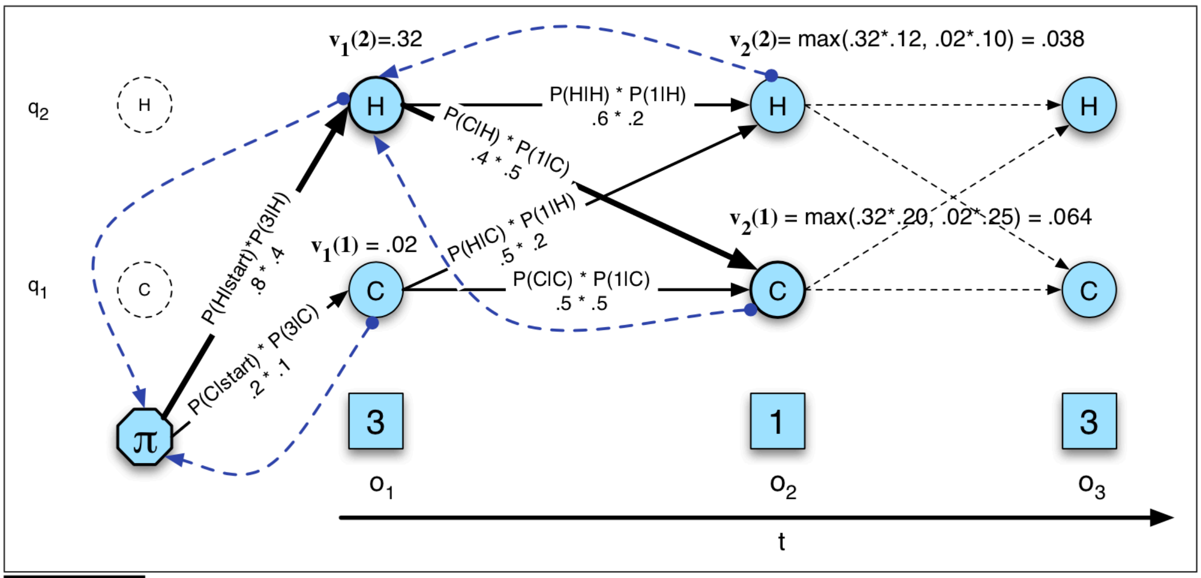


Figure A.10 The Viterbi backtrace. As we extend each path to a new state account for the next observation, we keep a backpointer (shown with broken lines) to the best path that led us to this state.

We can recognize that the first value of p(1)=cp(1) is equal to the variable $v_1(2)=0.32$, and that the second incremental probability p(2) is equal to the product shown on the diagonal path connecting state 2 in step t=1 and state 1 in step t=2, i.e. the product of P(C|H)*P(1|C)=0.4*0.5=0.2. By multiplying cp(1) with the probability along this diagonal path we get $v_2(1)$ which is contained in cp(2)=0.064. Analogously, in step t=3 we return to the final state 2, so the incremental probability p(3) along this (dashed) upward diagonal path is equal to P(H|C)*P(3|H)=0.5*0.4=0.2. Multiplying it by the previous probability $v_2(1)$ (i.e. with cp(2)) we obtain the final $v_3(2)$ located in cp(3).

Finding the likelihood of observations along all possible paths of hidden states

Finally, for the same simple example, let us try to determine the log-likelihood of observing the same observation O = '3', '1', '3', but individually along all possible paths of the state lattice, which has N^T, i.e. $2^3=8$ combinations. We can do this with the following short section of Matlab code, where we will call the same function 'dhmm_logprob_path.m' in the loop for all possible paths:

```
% Matrix with all possible state paths
mpath=[
1 1 1
1 1 2
1 2 1
1 2 2
2 1 1
2 1 2
2 2 1
2 2 2];

llm=zeros(8,1);  % Vector for log-likelihoods
for i=1:8,
  [llm(i), p] = dhmm_logprob_path(prior0, transmat0, obslik0,
mpath(i,:));
end;
```

Let's look at the obtained log-likelihood of observation for all 8 possible state paths:

```
>> llm

llm =

  -8.294049640102028
  -6.907755278982137
  -9.433483923290392
  -7.641724454062337
  -5.744604469176456
  -4.358310108056565
  -6.478573644256656
  -4.686814175028601
```

We see that indeed the greatest likelihood is obtained for the sixth case, and this is precisely the combination which corresponds to the Viterbi path 2, 1, 2, with log-likelihood of -4.35831. This ‘llm’ solution also provides us with an additional way to check the ‘Forward’ and ‘Backwards’ algorithms which calculated the total likelihood of the observation O over all possible state paths. This means that by summing the exponential values of the column 'llm' we should get P ("3 1 3"|λ):

```
>> sum(exp(llm))

ans =

   0.028562000000000
```

We see that we really got the same value of likelihood as by the previously described algorithms, with the difference that the numerical complexity of this direct approach is proportional to N^T, while the algorithms "Forward" and "Backward" are significantly less complex due to their recursive realization.

This latest ‘llm’ auxiliary solution also allows us to determine alternative paths of decoded hidden states, not just the best Viterbi path. If we would first sort the log-likelihoods, and then exponentiate them, and determine the cumulative sum of the likelihoods of the observations from the most likely to the least likely state path, then for this example we would get:

```
% Sort paths according to observation likelihood from the most
% likely to the least likely
[sllm,illm]=sort(-llm);

% Compute cumulative sums of likelihoods from the most likely
% to the least likely state path
cumsum(exp(-sllm))

ans =

   0.012800000000000
   0.022016000000000
   0.025216000000000
   0.026752000000000
   0.027752000000000
   0.028232000000000
   0.028482000000000
   0.028562000000000
```

The specific state paths from the most likely to the least likely can be read using the ‘illm’ index returned by the negative log-likelihood sort function.

```
>> mpath(illm,:)

ans =

     2     1     2   % The most likely path
     2     2     2
     2     1     1
     2     2     1
     1     1     2
     1     2     2
     1     1     1
     1     2     1   % The least likely path
```

We may also wonder how many such best paths we need to consider to achieve a certain percentage of the total likelihood of observation. We can easily check this by dividing the previous cumulative sum by its last term which is the sum of the likelihoods of all possible paths:

```
>> cumsum(exp(-sllm))/sum(exp(llm))

ans =

   0.448147888803305
   0.770814368741685
   0.882851340942511
   0.936629087598908
   0.971640641411666
   0.988446187241790
   0.997199075694979
   1.000000000000000
```

We see that the Viterbi path covers 44.8% of the total likelihood of the observation. If we add to it the next state path in terms of likelihood, which is path 2, 2, 2, then these two paths together provide 77.1% likelihood, while even with the third alternative path, 2, 1, 1, we get as much as 88.3%. The remaining 5 combinations contribute little to the overall likelihood, so we can ignore them as possible candidates for the decoded states of the model.

The task of generating observations for a given model

Before the final task, which will be the task of training the parameters of the HMM model from multiple observation sequences, we must first learn how to automatically generate such sequences. For this purpose, this Toolbox has a function 'dhmm_sample.m' which can be called according to this template:

```
% generate multiple observation sequences
T = 15;     % length of each sequence
nex = 10;   % number of sequences
data = dhmm_sample(prior0, transmat0, obsmat0, nex, T);
```

By executing this script, we get 10 different observation sequences of length 15 symbols (only one example of generated data is shown, because each new call of this function will generate a new random set, whose statistics are determined by the chosen parameters of the model):

```
data

data =

   2    2    2    3    1    3    2    3    1    2    2    3    1    1    2
   2    3    3    2    1    1    2    2    3    2    3    3    2    2    2
   2    2    1    3    2    2    3    1    2    1    2    2    2    2    2
   3    3    1    3    3    2    2    3    3    2    2    3    2    1    2
   2    3    1    3    2    1    3    1    1    3    3    2    2    1    2
   3    1    3    1    1    2    1    1    3    2    1    2    3    2    3
   1    2    1    1    1    3    2    2    3    1    2    2    2    3    1
   3    1    3    2    1    3    3    3    1    2    3    3    1    1    1
   1    3    1    1    1    1    2    2    1    1    1    2    3    2    2
   2    3    2    2    2    2    2    1    3    3    2    2    1    3    2
```

Note once again that these observation sequences are constructed exactly in accordance with the given parameters of the model, the same one we use in the whole presentation, i.e. based on the initial probability distribution vector 'prior0', based on the transition probability matrix 'transmat0' and finally the output symbol observation matrix in both states of the model, ‘obsmat0’. We can imagine that the function 'dhmm_sample.m' performed a series of random experiments, and in accordance with the given parameters of the model ,at each time instance: (1) based on 'obsmat0' chose one of the three possible output symbols in the current state, and (2) determined the new state based on the current state, according to the ‘transmat0’ matrix.

Long-term statistics of generated observation sequences

The question arises, what kind of solution should we have expected at all, i.e. how can we confirm that the generated data are at all consistent with the given model. A simple check can be performed by determining the frequency of occurrence of all possible output symbols for each generated observation sequence, and by comparing this empirical frequency with the expected long-term probabilities of output symbols for a given model. Specifically for the output symbol '1', the conditional observation probability of that symbol '1' in the first state (q=1) should be multiplied by the probability that the HMM model is in the q=1 state, while the conditional observation probability of the same symbol '1' in the second state q=2 should be multiplied by the probability that the model is in the state q = 2 and added to the first product, i.e. according to the expression:

$$P(1)=P(1|q=1)*P(q=1) + P(1|q=2)*P(q=2)$$

and analogously for the remaining two output symbols '2' and '3'. We already have all the conditional probabilities of observing the output symbols in the matrix 'obsmat0', but what we do not have are the probabilities of the model states P(q=1) and P(q=2). It should be noted that the state transitions of the HMM model are completely determined by the transition matrix A ('transmat0' in our example), i.e. they do not depend on observations, so it can be expected that the required state probabilities can be found directly from it. That part of the HMM model (if states were not hidden) is nothing else but a time-discrete and time-homogeneous Markov chain with a finite number of states. The condition of temporal homogeneity is met if the matrix of transition probabilities is time-constant, i.e. if it does not change with time, which is the case with us. For such a class of Markov chains, the transition probability in the k-th time step can be determined by successively multiplying the matrix of transition probabilities k times, thus as the k-th power of the matrix A: A^k. In cases where the Markov chain is irreducible and aperiodic, then there is a unique stationary distribution π_{stac}, which defines exactly the required probabilities of the state. According to Perron-Frobenius theorem, for such a case the k-th power of the matrix, A^k, converges into a matrix of unit rank (with identical rows), where each of its rows contains precisely this stationary distribution of the model states π_{stac}. Without going into proving the conditions of this convergence, (because there are many special cases that would need to be considered), let us show by our example what we would get by successively multiplying the matrix A, which in our example is called 'transmat0'.

```
>> a0=transmat0; for i=1:100, a0=a0*transmat0; end;
>> a0

a0 =

   0.444444444444445   0.555555555555556
   0.444444444444445   0.555555555555556
```

Obviously we got the desired state probabilities P(q=1)=4/9 and P(q=2)=5/9. Now we can calculate the expected frequencies of output symbols and get:

$$P(1)=P(1|q=1)*P(q=1) + P(1|q=2)*P(q=2) = (5/10)*(4/9)+(2/10)*(5/9)=(5/15)$$
$$P(2)=P(2|q=1)*P(q=1) + P(2|q=2)*P(q=2) = (4/10)*(4/9)+(4/10)*(5/9)=(6/15)$$
$$P(3)=P(3|q=1)*P(q=1) + P(3|q=2)*P(q=2) = (1/10)*(4/9)+(4/10)*(5/9)=(4/15)$$

This means that ideally a 15-symbol observation sequence should contain five '1' symbols, six '2' symbols and finally four '3' symbols. The empirical frequencies of the output symbols determined from the observational sequences of finite duration will converge to these ideal frequencies in the case when the length of these sequences T tends to infinity, because this is precisely the condition under which the stationary distribution π_{stac} was calculated. How could we get these long-term output symbol statistics directly in Matlab? We recognize that the above expressions actually correspond to the multiplication of the stationary state distribution row π_{stac} by the emission probability matrix B ('obsmat0' in our example). Will multiply the result by 15 (the common denominator of all three probabilities) to get the frequencies of the symbols in a string of length 15:

```
>> a0(1,:)*obsmat0*15

ans =
   5.000000000000001   6.000000000000000   4.000000000000001
```

Let us now check whether the long-term statistics of automatically generated data are consistent with a given model. Specifically, we count for each of the 10 generated sequences in the matrix 'data' how many times each possible output symbol appeared, for which we will use the function 'hist', to which as a second argument we list the 'bins' of data we count (Note: function 'hist' operates on columns, so we serve it the transposed data matrix).

```
>> hm=hist(data',[1 2 3])

hm =

     4     2     3     2     5     6     6     6     8     2
     7     8    10     6     5     4     6     2     5     9
     4     5     2     7     5     5     3     7     2     4
```

So, it turns out that in the first sequence we have four times '1', seven times '2' and four times '3', in the second sequence we have 2 x '1', 8 x '2' and 5 x '3', .. and so on until the tenth sequence which has 2 x '1', 9 x '2' and 4 x '3'. If we find the average number of occurrences of all output symbols over these 10 different sequences and if we divide this average by the sequence length T = 15, we should get empirical probabilities close to the expected ones (multiply them by another 15 for easier comparison with analytically derived ones):

```
>> mean(hm')/T*15

ans =

   4.400000000000000   6.200000000000000   4.400000000000000
```

According to this, out of 15 output symbols, we have on average 4.4 occurrences of symbol '1', 6.2 of symbol '2' and 4.4 of symbol '3', which has a certain similarity with the expected 5 + 6 + 4, but it is not exactly the same. Obviously, these sequences are too short for their histograms to correspond to long-term statistics. Therefore, we repeat the automatic data generation but with 100 times longer observation sequences, i.e. with T = 1500, and recalculate the empirical statistics of such long sequences:

```
% generate multiple observation sequences
T = 1500;  % length of each sequence
nex = 10;  % number of sequences
data_1500 = dhmm_sample(prior0, transmat0, obsmat0, nex, T);
% Count occurrences of each symbol
hm_1500=hist(data_1500',[1 2 3]);
% Compare with the expected probabilities
mean(hm_1500')/T*15
ans =

   4.968000000000000   6.064000000000000   3.968000000000000
```

We see that the statistics of longer observational sequences 'data_1500' follow the analytically derived expected output frequencies of the symbols much better, which confirms that the data generation function really works according to the given model parameters. What result would we get if instead of extending the sequences we increase their number, i.e. if we return the length T back to 15 but increase the number of sequences 100 times to nex = 1000?

```
% generate multiple observation sequences
T = 15;      % length of each sequence
nex = 1000; % number of sequences
data_1000 = dhmm_sample(prior0, transmat0, obsmat0, nex, T);
% Count occurrences of each symbol
hm_1000=hist(data_1000',[1 2 3]);
% Compare with the expected probabilities
mean(hm_1000')/T*15
ans =

   4.916000000000000   5.927000000000000   4.157000000000000
```

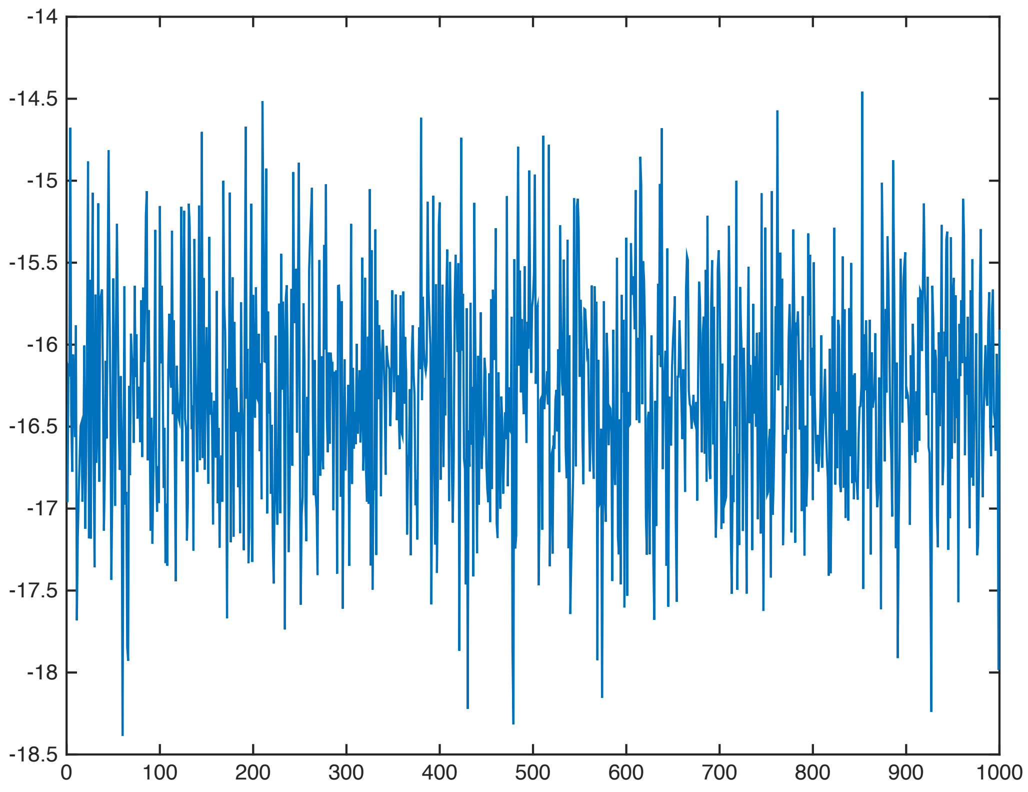
Although at first glance it seems that both approaches give a similar effect, this is not the case. For very long sequences, the initial probability distribution of the model state defined in the vector π (called 'prior0' in our example) will have no effect (due to the stationarity of the k-th power of the matrix A for the large k), while in the case of shorter sequences, e.g. T = 15, the influence of the initial probabilities will nevertheless be reflected in the experimental results. The described behavior is logical, because in long sequences the influence of the initial state of the model disappears relatively quickly, and the behavior was predominantly determined by the transition matrix A (stationary state of the model), while only at the beginning of the sequence the initial probability column π will influence the local state statistics.

Finding the likelihood of generated observation data

For these three sets of automatically generated observational sequences, we can calculate the likelihoods of their observation for a given model (the same one used to generate them). Let’s illustrate this for the set ‘data_1000’, using the following section where we compute the log-likelihood of each individual short sequence.

```
% Calculate log-likelihood of each observation
nex=size(data_1000,1);  % Number of experiments
llm=zeros(nex,1);       % log-likelihood column
for i=1:nex,
    llm(i)=dhmm_logprob(data_1000(i,:), prior0, transmat0, obsmat0);
end;
```

We can also show the result with a figure, with the Matlab command plot(llm):



We see that the range of log-likelihood is quite large, ... from -18.5 (least likely sequence) to -14.5 (most likely sequence). We can also determine the ratio of the most likely sequence to the least likely generated sequence:

```
>> exp((max(llm)-min(llm)))

ans =

  50.944825651065138
```

If we passed the whole matrix 'data_1000' to the function 'dhmm_logprob' as the first argument, the function would return the sum of log-likelihoods of all observation sequences (which corresponds to the product of their likelihood, i.e. the likelihood of simultaneous observation of all 1000 sequences given the model), but then we would not have insight into probability of individual observations.

```
>> ll=dhmm_logprob(data_1000, prior0, transmat0, obsmat0);
>> disp(sum(llm)-ll)
   0
```

In general, the dispersion of likelihood will be higher on short series, because the output statistics described by the model "did not have sufficient time" to be fully demonstrated. As a measure of this relative scatter, we can calculate the quotient of the standard deviation and the mean value of the log-likelihood column of all observations:

```
>> std(llm)/mean(llm)
```

```
ans =

  -0.040006421311045
```

from which we see that for sequences of duration T=15 this scatter is about 4% of the expected log-likelihood.

Repeat the same procedure for the long observation sequences 'data_1500' and discuss the difference from the case described. How does the final value of log-likelihood depend on the length of the sequence? What is the relative scatter of the log-likelihood for long sequences? How do you explain that?

Degenerate HMM model with identical long-term statistics

Finally, consider whether there may be alternative HMM models that have identical long-term statistics of output symbols, as the model we use as an example. What if the HMM model degenerates, and has only one state, and it is in that single state that its emission probabilities are identical to the long-term observation probabilities for our analyzed example. Is this model any different from ours and would the log-likelihood of observing the generated arrays (‘data_1500’ and ‘data_1000’) be the same given the original or degenerate model? As a starting point, consider what model these parameters describe:

```
priorL=[
    0.5  % State C
    0.5  % State H
];
% Transition probability matrix
%
transmatL=[
0.5 0.5      % P(C|C) P(H|C)
0.5 0.5      % P(C|H) P(H|H)
];
% Emission probability matrix
obsmatL=[
5 6 4   % P(1|C) P(2|C) P(3|C)
5 6 4   % P(1|H) P(2|H) P(3|H)
]/15;
```

The task of training HMM model parameters from observations

Now that we know how to generate synthetic observational data for given model parameters, we can finally show the last function of this package that is relevant for discrete HMM models, which is the 'dhmm_em' function. It is used to train HMM model parameters based on the Expectation Maximization algorithm (EM), which is described in the lecture materials in detail. The input to this function are multiple series of observations that all belong to the same class for which we want to build the HMM model. The function is called as follows:

```
% improve guess of parameters using EM
[LL, prior2, transmat2, obsmat2] = dhmm_em(data_1000, prior1,
transmat1, obsmat1, 'max_iter', 10);
```

As output, it returns the total log-likelihood 'LL' of the given observations ('data_1000' in this example) with optimally determined model coefficients, and these coefficients are: 'prior2' (initial probability row), 'transmat2' (state transition probability matrix) and 'obsmat2' (emission probability matrix). However, this function expects the initial parameters of the model in addition to the learning data, because as described in the lectures, the EM algorithm is an iterative algorithm that in each step based on the existing model calculates new model parameters that will increase the likelihood of observation. Such an iterative algorithm therefore requires for its first step a certain initial model, from which it will start. For a model with Q=2 states and a dictionary with O=3 symbols, completely random model parameters can be calculated by the following section:

```
% initial guess of parameters
prior1 = normalise(rand(Q,1));
transmat1 = mk_stochastic(rand(Q,Q));
obsmat1 = mk_stochastic(rand(Q,O))
```

Each subsequent execution of the same section will each time generate a new set of random model parameters, all of which satisfy the conditions on the sums of probabilities which must be equal to 1. Such a completely random model will be sufficient to initialize the training process, and the EM algorithm itself is guaranteed to converge, i.e. each step always necessarily increases the likelihood of the observation. The 'dhmm_em' function will display log-likelihood in each iteration of the EM algorithm, and will stop the calculation when the relative change in log-likelihood is small enough, or when the maximum number of iterations defined by the last two input arguments is reached: „, `'max_iter', 10);`"

An example of performing HMM training using this function on 1000 observation sequences of length 15 that we generated in the last chapter in the matrix 'data_1000' with described random initialization of the model gives:

```
% improve guess of parameters using EM
[LL, prior2, transmat2, obsmat2] = dhmm_em(data_1000, prior1,
transmat1, obsmat1, 'max_iter', 10);

iteration 1, loglik = -17321.546728
iteration 2, loglik = -16404.111541
iteration 3, loglik = -16374.989855
iteration 4, loglik = -16358.670188
iteration 5, loglik = -16348.640820
iteration 6, loglik = -16342.056279
iteration 7, loglik = -16337.514173
iteration 8, loglik = -16334.258950
iteration 9, loglik = -16331.854393
iteration 10, loglik = -16330.034303
```

The iterative EM algorithm shows the typical behavior of this class of algorithms, i.e. in the first few steps the log-likelihood changes significantly, while later it becomes slower and slower, because the solution converges relatively quickly to the optimal one.

Evaluation of the trained model

As described in the introductory chapters, the evaluation of the trained model must be performed on a data set that was not used in the training process, because only in this way can we confirm that the model has “generalized”. In addition to the data-set used to train the model ('data_1000'), in the last chapter we generated an additional observational set 'data', which has the same length T = 15, but only 10 such observational sequences. We can use this data as an "evaluation base", and calculate their log-likelihood of observation with all models:

- default model based on which we generated all data (prior0, transmat0, obsmat0),
- initial model with randomly selected parameters (prior1, transmat1, obsmat1),
- optimal model built by EM algorithm (prior2, transmat2, obsmat2).

```
>> ll0=dhmm_logprob(data, prior0, transmat0, obsmat0)

ll0 =

    -1.631252692507668e+02

>> ll1=dhmm_logprob(data, prior1, transmat1, obsmat1)

ll1 =

    -1.735761577055685e+02

>> ll2=dhmm_logprob(data, prior2, transmat2, obsmat2)

ll2 =

    -1.631280680263178e+02
```

We see that the likelihoods for the given and optimal model are almost identical, while the likelihood for the random model is smaller.

Compare the original and optimally determined parameters of the HMM model. Are they equal? How do you explain that?

Try rebuilding the model but with new random initializations. Investigate how much the final log-likelihoods of such models differ on the ‘data’ test set depending on the random initialization.