Markov Models, Hidden and Otherwise.*

Greg Kochanski
http://kochanski.org/gpk

2005/03/08 10:47:58 UTC

1 Introduction

Markov Models (also known as Markov Processes and Markov Chains)[1] are a powerful and commonly used technique. In essence, a Markov Model is a way of generating a random sequence that contains patterns. Used in reverse, a Markov model is a way of finding certain patterns in a sequence.

To have a Markov chain, you first need a set of allowed states. The Markov process then hops from one state to another, generating a chain of states. Mathematically, Markov chains are generated by a probabilistic rule that looks at the last few values of the sequence to select the next state. Specifically, when calculating a first-order Markov chain, one needs to look at the current state to decide what states are possible for the next step and what is the probability of moving into each of the possible next states.

Second- or higher-order Markov Chains: Higher-order Markov chains depend on more historical information. For instance, a third-order chain depends on the last three states it visited: At a given time t, the probability of hopping to state s at time $t+1$ is $P(s|q,u,v)$, where q, u, and v are the current and two preceding states. We focus on first-order Markov chains because any higher-order Markov chain can be converted into a first-order chain.
For instance, if, in a third-order Markov chain, you visit states ..., **a, c, d, a, b, e, a,** ..., when you are actually in state **b**, it computes the probability of its next hop based on its recent history: **d, a, b**. To construct the equivalent first-order representation, we name the states so that the history can be reconstructed from the state name. You collect three states worth of history and encode the information into the name of the current state. When the third-order Markov chain is in state **b**, the equivalent first order one would be in state **dab**, and it would visit states ..., **??a, ?ac, acd, cda, dab, abe, bea, ea?, a??,** .... Each of those state names gives you the same information that the third-order Markov chain has at the same moment. That means that any information you might get from the history in the higher-order representation can be deduced from the (longer) name in the first-order representation.
Obtaining a first-order representation is crucial, because that's the only kind of Markov chain that can be efficiently solved by the Viterbi algorithm. There is a cost to this conversion, however. The first-order representation of a higher-order Markov process usually has far more states than the higher-order representation, because you need one state name for each possible history.

Figure 1 shows the states and allowed transitions in an example Markov process. If you start in state **a**, at the next clock tick you will jump to state **b**, then either **c** or **d**, and so on. In a Markov chain[2], the probability

[1] Strictly speaking, the Markov Process is a random process that generates a sequence of symbols. The sequence is known as the Markov chain. Finally, if you use a Markov process to simulate or predict some linguistic phenomenon, you have a Markov model.

[2] Specifically, a first-order time-independent Markov chain. Higher-order chains are possible, which depend on more history. You can have time-dependent Markov chains, too, where the probability of making a hop may change with time. They will be mentioned in the section on Hidden Markov Models.

Uses of higher order Markov chains.: Higher order Markov chains are used in speech recognition systems to represent co-articulation effects between phonemes. A third-order Markov process is often used, so it can represent variants of a phoneme, as modified by its immediate neighbour on either side. In that case, with about 40 different phonemes in the third-order representation, converting it to the equivalent first-order representation will yield $40^3 = 64000$ possible states.
This points out a serious limitation of Markov chains for linguistic purposes: they cannot capture the long-range structure of language. Real human language includes references to words many sentences away. Conversations can revolve around the same topic for an hour or more, or can return to their beginnings after long digressions. Books will often place characters in a situation, and then come back to look at the outcome tens of thousands of words later. Markov chains simply cannot represent this important fact about language; their value is that they can simply and efficiently capture the short-range structure in language. While this is only a fraction of what we'd like to do, it is better than capturing no structure at all.

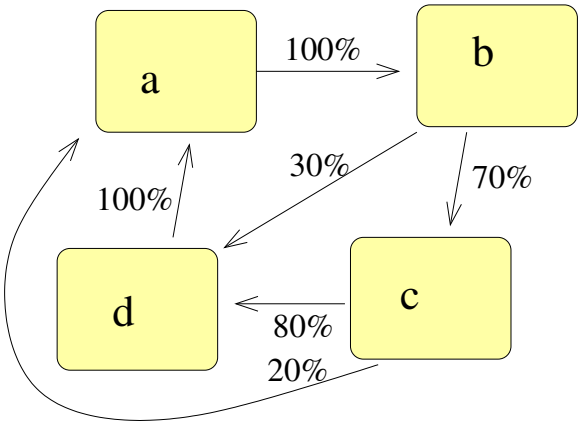

Figure 1: A description of a Markov process that generates **abcabda**.... Boxes are states, and arrows show transitions between states.

of hopping from state q at time t to state s at time $t+1$ is called the transition probability between states q and s, and is written as $P(s|q)$.

This particular Markov process is *ergodic* because if you run it long enough, all the states will be visited[3], and visited in an unpredictable manner. Most Markov chains that you encounter will be ergodic, and it is an important condition for many mathematical results in this area.

States vs. Links: You can transform any Markov process into an equivalent process by writing a state for each link and a link for each state. Think of the Markov process as driving a car: it is possible to specify a route either by giving a list of cities (from which your listener can deduce the roads that you must have used) or giving a list of roads (from which the listener can deduce cities).
The Markov chain equivalence is stronger than the transportation analogy, because there is only one link (road) between any two Markov states (cities). Also, each link (road) touches only two states (cities). In such a world, as might be maintained by obsessive and overenthusiastic transportation planners, once you get on the London–Oxford motorway, there would be nowhere to stop except Oxford.

One can also represent the same Markov chain in a way that explicitly shows the way the state hops around from time to time. Figure 2 does this. There are two things one normally does with a Markov model: run it in

[3] Mathematically, ergodicity means that there is some finite integer k, such that after exactly k steps, any state has a nonzero probability of changing to any other state. The smallest such value of k for this example is 8. Ergodicity excludes Markov chains that are perfectly periodic, and also chains that are broken into several pieces.

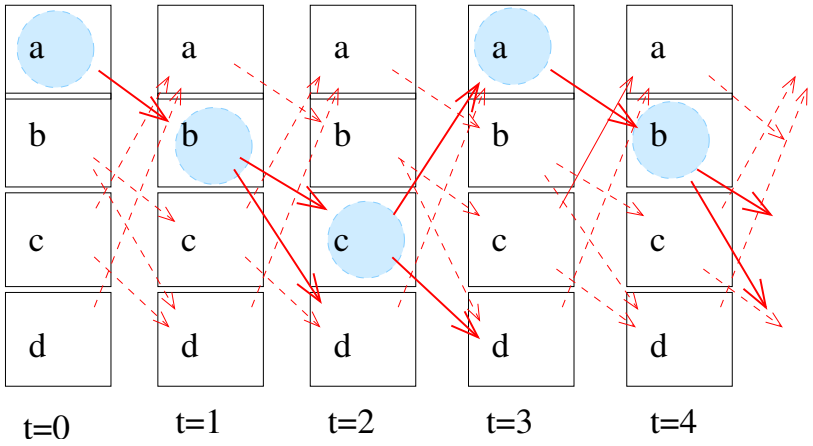

Figure 2: A description of a Markov process that generates **abcabda**.... Each column represents the possible states of the process at a given time, with the actual state at each moment coloured. The arrows show possible transitions. This figure shows the same process as Figure 1, but shows the state changing over time.

the forward direction to produce a random sequence or to deduce statistical properties about the sequence, or run it in reverse, and deduce the transition probabilities from an observed sequence of states.

2 Simulation and Statistics

Simulation via brute-force techniques is straightforward and the following tiny program will do the trick:

```
#!/usr/bin/env python

import random                   # Grab a random number generator
import sys                      # Grab a print routine.

# Internally, we use natural numbers (0, 1, 2, 3, ...) to
# label different states.    The StateNames array is just
# there to make the printout easier for humans to read.
StateNames = ['.\n', 'I went', ' slowly', ' home']
Nstates = len(StateNames)                       # How many states are there?

# This is a two-dimensional array, where P[i][j] contains
# the probability of making a transition from state i to state j.
TransitionProbs = [
                                [0.0,  1.0,  0.0,  0.0],        # P(*|a)
                                [0.0,  0.0,  0.7,  0.3],        # P(*|b)
                                [0.2,  0.0,  0.0,  0.8],        # P(*|c)
                                [1.0,  0.0,  0.0,  0.0]         # P(*|d)
                                ]

StartState = 0                  # Which state should the simulation start in?

state = StartState                      # Initialize.
for time in range(40):                  # Run for 40 steps.
        P = TransitionProbs[state]              # Get the probabilities for
                                                # transitions out of the current state.
        r = random.random()                     # Pick a random number.
        for PossibleState in range(Nstates):            # This loop goes over all possibliities
                                                        # for the next state.
                if r < P[PossibleState]:                # Found the random choice for next state.
                        state = PossibleState                   # Prepare for next step.
                        sys.stdout.write( StateNames[PossibleState] )   # Print.
                        break                                   # Leave the inner loop.
                r -= P[PossibleState]                   # Prepare to check next PossibleState.
```

... and produce the following result:

> I went slowly home. I went slowly. I went home. I went home. I went slowly. I went home. I went slowly home. I went slowly home. I went slowly. I went slowly home. I went slowly home. I went home

From such a Monte-Carlo simulation, one can, in principle, compute anything you want, such as the probabilities of of seeing particular words or of seeing particular sequences or words.

Analytic techniques are also available for solving many problems. If one works through the algebra to compute the probability of each state at time $t+1$, given a probability distribution across states at time t, one finds:

$$P(s|t+1) = \sum_q P(s|q) \cdot P(q), \tag{1}$$

where $P(s|q)$ are the various transition probabilities. $P(s|q)$ is zero if there is no link from state q to state s. Perhaps by coincidence, perhaps not, this equation is simply the multiplication of a matrix by a vector:

$$p_s^{[t+1]} = \sum_q T_{s,q} \cdot p_q^{[t]}, \tag{2}$$

where $T_{s,q}$ are all the state-to-state transition probabilities, arranged as a matrix, $p^{[t+1]}$ is the probability of being in each state at time $t+1$, arranged as a vector and likewise, $p_q^{[t]}$ is the probability of being in state q at time t.

Once that connection to matrix multiplication is realized, all the machinery of linear algebra can come into operation. For instance, all ergodic Markov processes eventually settle down to a stable probability distribution, no matter what state they are stated in[4]. After a long enough time, the Markov chain gradually forgets what states it has started from[5], and the probability of being in state **a**, for instance, becomes independent of time and independent of the starting point.

These analytic results explain why, if one builds Monte-Carlo grammars, and runs them to produce text, the results are generally disappointing. The problem is that the Markov chain forgets what it is talking about every few words. Markov chains for composing music were tried in the 1980s and had similar problems.

Markov Chain Text: For fun, and to show the short-range order of text produced by a Markov process, I built a second-order Markov process from these lecture notes, then generated text from it. (The program is in Appendix A.) Words, defined by white space, were used as symbols. You can see that the generated text is fairly coherent on a word-to-word basis, in the sense that one doesn't encounter two adjacent words that clash, but the topic hops from one lecture to another, and there can often be more than one shift inside a sentence, rendering the result semantically incoherent and often ungrammatical:

> This work is available almost by inspection. All this is often useful to help convince yourself that the median almost violates point 5. Second, we will do is to try to focus on first-order Markov chains depend on the street has spoken before a group of people, and the third flip, *et cetera*.
>
> Imagine that Alice wants to meet Bob, but Bob doesn't want to know the probability distribution close to one. The arithmetic is different if you're on B *vs.* a more general correlation I had grand plans of helping the classifier to perform worse on the measurement by e-mailing to a set union operation, $\neg$ for "not", $\cap$ for logical "and" or a Monte Carlo simulation. It can be though of as a feature either: long documents will have some relation to recent English names, and too many names with seven or more dimensions.

In a future work, I will explore the societal implications of seven-dimensional names.

Still, despite their limitations, Markov chains (and their cousins, probabilistic finite state machines) are intensively used in language for two critically important reasons:

- there are reliable known techniques for deriving the Markov model from data, which isn't true for a lot of other models, and
- Markov models provide a complete representation of local structure: they can represent any linguistic behaviour that does not require a long memory.

3 Solving for Transition Probabilities

Finding the Markov process that best represents an observed sequence of states is not very hard[6]. Since the behaviour of the Markov process is defined by the transition probabilities, one just needs to estimate the various values of $P(s|q)$. To do that, just count the number of times the sequence $\ldots$**qs**$\ldots$ is seen, and divide by the total number of times that that $\ldots$**q**$\ldots$ is seen. As you might recognize, this is just the maximum likelyhood estimate for that probability, and is all you need if you have lots of observations of $\ldots$**qs**$\ldots$. In the more normal case where data is sparse, you would want to use Good-Turing or some other estimation method, as appropriate.

[4] You compute this steady-state probability by requiring that $p^{[t+1]} = p^{[t]}$, which leads to an eigenvalue equation. The stable probability after many steps is given by the eigenvector associated with the largest eigenvalues. One can prove that (for an ergodic chain) there is always exactly one eigenvalue that is exactly equal to one, and it is the largest eigenvalue. Equation 2 is a discrete difference equation, and the other eigenvalues tell you how fast the Markov chain forgets the initial conditions.

[5] Assuming it is ergodic. If it is non-ergodic, it will remember some things, like which of the two disconnected sub-graphs it started upon, but will forget other details.

[6] If the sequence is ergodic.

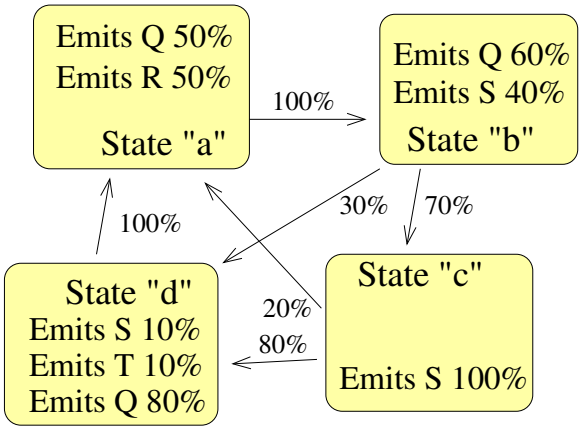

Figure 3: A description of a Hidden Markov process that generates **QQSQSSR**... while going through states **abcabda**.... Boxes are states, and arrows show transitions between states. The basic system is the same as in Figure 1, but the states are decorated with information about which output symbols can be produced from a given state.

Another thing to think about – which few people do – is that Maximum Likelyhood or Good-Turing estimates of $P(s|q)$ are simply estimates of the underlying probability. The real value of $P(s|q)$ is almost certainly somewhat different.

$P(s|q)$ may be a transition probability, but it is also just a parameter in a model that describes the data. Consequently, the data doesn't force you into a unique value of $P(s|q)$: a range of different values for $P(s|q)$ will do about equally well at describing the data, and one can speak of a probability distribution of the values of $P(s|q)$. In fact, using Bayes Theorem, one can compute a probability distribution for $P(s|q)$. Generally, $P(s|q)$ will be part of a joint distribution of all the transition probabilities, all correlated, so that changes in one are compensated by changes in the others.

4 Hidden Markov Models

Hidden Markov Models (HMMs) are exactly what their name implies. They are a Markov model which is hidden, in that you cannot directly observe the sequence of states. Obviously, a completely hidden Markov model would be pretty useless; it would be on par with a write-only memory or the psilent "p" in psychiatrist. So, a HMM gives you some clues about the current state of the Markov process by having each state randomly emit one of several symbols[7].

Figure 3 shows the state diagram of a sample HMM, and Figure 4 shows its behaviour as a function of time.

The Markov process ticks along under the covers, and occasionally emits symbols. As with a open Markov process, you can use this in either the forward direction, where you simulate the state transitions and the output, or in the reverse direction. In reverse, you observe the symbols and attempt to deduce the transition probabilities, the sequence of states, and/or the emission probabilities for various symbols from each state.

5 HMM simulation

Simulating an HMM is little more complex than simulating a Markov process. The only difference is that instead of simply printing the state information, the code to simulate a HMM has an extra call to a random number generator to allow it to choose which symbol to emit.

Likewise, the analytic calculations one can do with a HMM are identical to those for a Markov chain, except that a final multiplication by another matrix is added. This extra matrix converts from the probability of being in a certain state to the probability of emitting a certain symbol.

[7] You can, equivalently, make the emission of symbols occur as the Markov process transverses a link.

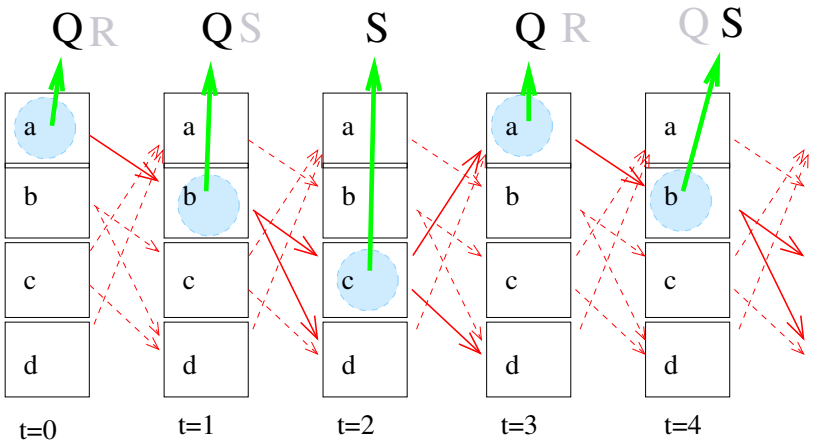

Figure 4: A view of a Hidden Markov process as it generates QQSQSSR... while going through states **abcabda**.... Boxes are states, and arrows show transitions between states. The letters on top are the emitted symbols and are the only information an observer can get about the process. The basic process is the same as in Figure 3, but the behaviour is shown across several different times.

Note that the emitted symbols do not have to be discrete: they can be chosen from a continuous probability distribution, such as a Gaussian. These continuously-valued HMMs are seen in speech recognition systems. In most speech recognition (ASR) systems, a HMM is used that (conceptually, at any rate) generates sets of acoustic parameters every 10 milliseconds or so, based on the changing state. In these models, the HMM state identifies the current phone (or a section of a phone). The state ticks through a complex graph that represents all the utterances that the ASR system can recognise, built hierarchically out of words and phones[8].

One then matches this model to acoustic data by means of Bayes' Theorem, or an approximation thereof. The HMM generates $P(\text{acoustic data}|\text{utterance})$ for the various utterances that the ASR system can recognise, and Bayes' Theorem (as usual) reverses that to obtain the probability for each utterance, given the observed acoustic data.

A Markov model is used in an ASR system for two main purposes:

- allowing the model to stretch or compress to match to the duration of the actual speech, and
- allowing for alternate pronunciations.

Figure 5 shows some fragments of the state transition diagram for a speech recogniser that allows flexibility with respect to duration and Figure 6 shows how two alternate pronunciations would be implemented.

A typical state diagram for a speech recogniser that is designed to decide among a dozen semantic classes might have hundreds of states per word (several alternative pronunciations times several phones times several states per phone) and tens to hundreds of alternative wordings[9] for each semantic class.

6 Deducing HMM parameters

While simulating a HMM is straightforward, deducing the parameters from an observed sequence of symbols is not. Unlike an open Markov chain where there is a simple and direct connection between the matrix of

[8] Without meaning to beat up on Markov models too much, it must be noted that if you use a speech recogniser's Markov model to actually generate speech by having it emit acoustic parameters and then synthesising signals from the acoustic parameters, the result is atrocious. Oddly enough, the reason is not because the ASR models have only short-range correlations: their graphs are so large that they do include (by sheer brute force) correlations on the scale of an utterance. The problem actually comes from the uncorrelated random emission of symbols from the Markov process: in one frame, the chain may emit the acoustic parameters for a fully articulated /a/, while 10 milliseconds later, the very same state may emit a rather schwa-like /a/.

[9] "Yes.", "Yah", "Uh, yes.", "Yeah.", "OK, Yeah.", "Yeah, OK.", "Uh, OK. Fine.", "Sure.", "Uh huh.", "Well, OK.", "Very good!", *et cetera*.

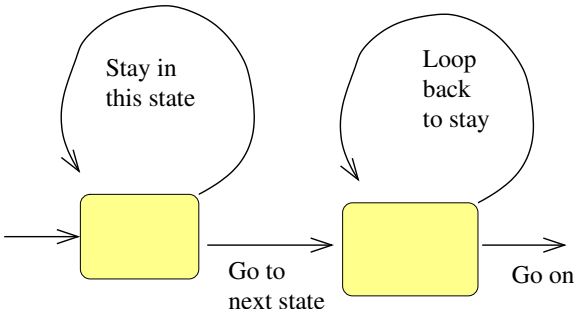

Figure 5: A section of the state transition graph that allows a phone to have a longer (by looping back) or shorter (by going straight through) duration.

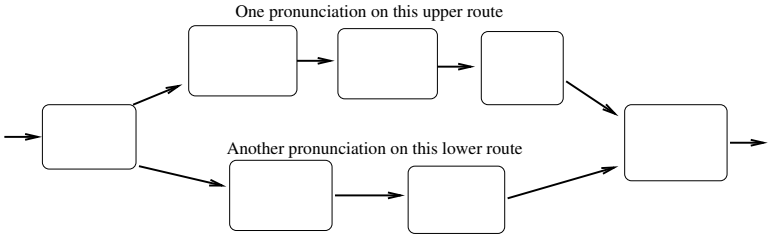

Figure 6: A section of the state transition graph that allows two alternative pronunciations by taking either the upper or lower phone sequence.

transition probabilities, and the observations, a HMM derives its emitted symbols from a combination of two random variables: the state sequence and the random choice of emitted symbol from each state. Consequently, one simply does not know whether a given **Q** comes from state **a** or state **b**, and estimating the probabilities are not straightforward.

However, good techniques exist for finding the various probabilities, if given sufficient data.

6.1 Speech Recognition - Training the Model

When training a speech recognition system, it is usually assumed that we know the phone sequence for a given training utterance. Training utterances are transcribed by hand and/or machine to yield a sequence of phones[10].

[10] There are lots of complications here. For instance, it is very expensive to hand-segment enough speech to train an ASR system, so often humans do the relatively fast and cheap transcription to a sequence of words, then a pronunciation model turns that into a sequence of phones, then a modified speech recognition system time-aligns the phone boundaries with the speech data. Each of those components requires training data of its own, so the overall process of building a speech recogniser is complex, expensive, and

Markov Chains and Finite-state Machines: There is a close connection between Markov Chains and finite-state machines. Specifically, any probabilistic finite-state machine (PFSM) that is fed a string of identical tokens internally walks through a sequence of states which is a (time-independent) Markov chain. Note, however, that the symbols that a PFSM *emits* may not be a Markov chain. The emitted symbols are the result of a Hidden Markov model (HMM), which is a slightly more complex beast.
A PFSM driven by a general sequence of tokens goes through a sequence of states that corresponds to some time-dependent Markov process, where the time-dependence is chosen to match the sequence of tokens fed into the PFSM. Likewise, the emitted tokens are equivalent to the result of some time-dependent HMM.
Mathematically, they are closely related; one thinks of something as a Markov chain or HMM if it drives itself and as a PFSM if one wishes to focus on how the output depends on the input tokens.

Given this sequence of phones, the Markov transition probabilities are obtained trivially (just as in an open Markov model, above), because the sequence of states is not hidden at this stage.

Given a sequence of phones that are time-aligned with the acoustic data, it is also conceptually simple to deduce the probabilities of a given state emitting each possible symbol: you apply appropriate signal processing to the acoustic data[11] to convert it into a symbol, and (if one had enough data and time), simply count how many of each symbol[12] one saw on each state of the Markov process. In reality, to reduce the amount of data required and to deal with the continuous nature of the acoustic parameters in a reasonably elegant manner, the probability density of obtaining each symbol is assumed to be a multivariate Gaussian[13], and the parameters that define the Gaussian are fit to the set of all acoustic data that aligns with each phone.

6.2 Speech Recognition – Using the Model

Then, once the model is trained, it is presented with some acoustic data, and the task is to find the sequence of words that is most likely, given the data[14]. That task boils down to searching for the path through the Markov process that will give the largest probability of reproducing the observed data[15]. Without going into the details, the Viterbi algorithm, or the A* (also known as the stack) decoder will do the trick [Jurafsky and Martin, 2000, Chapter 7].

Although it is important, finding the best (most likely) path has its limitations. First, the Viterbi algorithm actually only solves an approximation to the actual Markov model. Viterbi makes the assumption that the probability of getting to some state is not given by the sum of the probabilities along all the alternate routes, but instead is just given by the single most probable route.

In other words, it assumes that the probability of becoming Prime Minister is just the probability of getting into Oxford times the probability of becoming a Member of Pariament (given Oxford) times the probability of becoming PM (given MP). While that may be the most probable route (25 took it), it's far from the only route (notably, Winston Churchill didn't take it). This assumption is best when the graph of links from one state to another is quite sparse, or when some transition probabilities are much larger than other ones. Many real problems fall into those categories, so the Viterbi approximation is actually quite useful.

However, it's quite possible to come up with examples where the most probable path is completely unimportant. Often, when there are thousands of possible states, there are so many paths that the sum total of all the less-probable paths swamps the most probable path. One simple example is finding the most probable path through English grammar from "Once upon a time..." to "Then they lived happily ever after. The End."

6.3 Other than ASR

Hidden Markov models are useful for things other than ASR. They are used in algorithms that deduce the parts-of-speech of words in a sentence, for instance.

In a part-of-speech (POS) tagger, the underlying model is a Markov chain of states, each labelled with a part-of-speech (noun, verb, pronoun, ...). As the state walks through the Markov chain, it emits random symbols, which are words of the appropriate variety. So, as it hops into the "noun" state, it may emit the word "pencil".

Much like a speech recogniser, you train it by giving it data where the underlying state sequence is known. For instance "Bob saw Alice" is generated by the state sequence **noun, verb, noun**. Once the tagger is trained, you use it by finding the most probable state sequence for a given text. As in the ASR case, this is found by searching the state transition diagram for the sequence of states that is most likely to yield the observed data.

time-consuming.

[11] Normally, one takes a 30 millisecond frame of the speech waveform and computes the cepstral coefficients. The emitted symbol is then a vector of the cepstral coefficients along with their time-derivatives. The necessary signal processing to obtain reasonable acoustic features is the subject of at least one complete course.

[12] To implement this process, you would have to quantise the continuous set of acoustic signals into a discrete set. In practice, the discrete set is too big for this simple approach.

[13] Or, typically a multi-humped probability distribution made of a bunch of multivariate-Gaussian humps. This is called a Gaussian mixture distribution.

[14] Often, the task is to find which semantic cluster is most probable given the data. However, once you have a string of words, it is pretty straightforward to see which cluster it came from.

[15] By application of Bayes' Theorem, the sequence of words that has the largest probability of reproducing the data is the most probable utterance, after the data is taken into account.

This most-probable state sequence is the output of the tagger: it is the best guess at the parts-of-speech that under-lie the text.

POS taggers are quite successful. Oddly enough, the inability of Markov models to represent long-range linguistic effects is a mixed blessing here. While POS taggers do make some errors because they can "forget" information from a distant part of a sentence that might disambiguate a word, they turn out to be remarkably robust for the very same reason. A typographical error or strange grammatical construction which could badly confuse the parse tree of a sentence will cause only local problems in a POS tagger: once the tagger is looking more than a few words away, the Markov process has "forgotten" the error, and is primarily operating on the basis of local information. It is thus impossible for any word in the text, no matter how strange, to disrupt more than a few tags in the document.

HMMs can also represent boundaries in text or speech. You can imagine a two-state model, where one state means "boundary" and the other state means "not boundary". Depending on the text or speech, the transition into the "boundary" state can be more or less likely in different places.

Bioinformatics, DNA and Proteins: Today, Markov Models are a hot topic in biology, as they can be used to efficiently search for DNA sequences that are near-matches to a template.
DNA is very much like language: the genetic information is coded as a string of chemical "symbols" drawn from an alphabet of four letters. You can think of the DNA as orthography. Most DNA codes for a protein, and three symbols in a row code for one of the 21 amino acids, which are the building blocks of proteins. Other triplets are start and stop symbols, and other things.
Amino acids are like words. They are closely related to their orthography (DNA). They are strung together in a linear sequence to make a protein.
Proteins are equivalent to the semantic or discourse level of language. Proteins (or groups of proteins) are the actors in biology: they form the structure of the cell and they control the chemistry that takes place inside each cell. Structurally, proteins may be *constructed* of a linear sequence of amino acids but as it is created, each protein folds up into a specific, complex three-dimensional structure, and that structure is essential to its function.
Chemically, the folding occurs because amino acids interact with each other even after they have been bonded together to form a chain. Certain pairs of amino acids attract and bind, other pairs don't. (This binding, known as the secondary or tertiary structure is weak compared to the basic chain, and does not disrupt the chain.) The folding of a protein into its functional shape is a complex, incompletely-understood interplay of these affinities between amino acids and geometric constraints as one part of the chain bumps into another.
Pursuing the analogy, the process of folding a protein corresponds, linguistically, to the process of assigning meaning to a string of words. The bonds that form the secondary and tertiary structure of a protein correspond to references from one word in a document to another. The folding process brings together and binds amino acids from distant places in the chain, much as the process of interpreting a document makes connections between distant words.
Don't take this analogy *too* seriously, of course. Despite some dramatic similarities, there are dramatic differences, too.

A Program for generating Markov Chain text.

```
# --  python --

"""This program constructs a Markov model for input text,
and then re-generates some text from the Markov model.

The functions here are generic, but when run as a script,
it produces a second-order Markov model.

The output is annotated with comment lines
(starting with '#') that indicate the probability of a
given choice in a given context.
"""

import random                   # Get access to a random number generator.

# The data is stored in a dictionary of lists.
# The dictionary stores (as a key) the relevant
# recent history, and a list of all possible
# outcomes as a value.
#
# So, if, the sequence "big fat fred" has been observed once,
# and "big fat george" has been observed twice,
# then the data structure would be:
# { ('big', 'fat'): ['fred', 'george', 'george'] }.
# This indicates that after seeing "big fat",
# "george" has been seen twice and "fred" once.


def add(d, hist, w):
        """Add the current n-gram to d.
        D is a dictionary-of-lists structure described above."""

        key = tuple(hist)               # No change in information.
                                        # This is needed for technical reasons,
                                        # because a tuple is immutable, therefore it
                                        # is an acceptable key for a dictionary, whereas
                                        # a sequence is not.
        if not key in d:
                d[key] = []             # If we've never seen this context before,
                                        # create a new list to store the
                                        # possibilities for the next word.

        d[key].append(w)                # Add the newly observed word to the list.



def get(d, hist):
        """Pick a random word from the dictionary-of-lists structure
        based on the specified history.    Note that this effectively
        assumes maximum-likelihood probability estimation,
        not Good-Turing."""
```

```
        key = tuple(hist)               # For technical reasons.

        # random.choice() randomly chooses one from a list:
        return random.choice( d[key] )


def build(fd, initial):
        """Read the input, break at spaces, and
        build a table (a dictionary-of-lists structure) that contains
        the number of observations of each N-gram.
        This table is proportional to the Markov transition probabilities.
        """

        history = initial
        o = {}
        while 1:
                l = fd.readline()
                if l == '':
                        break
                for word in l.split():                  # Split a line into words.
                        add(o, history, word)           # Add a word to the table.

                        history.pop(0)                  # Update the history.
                        history.append( word )

        add(o, history, None)           # This is the end-of-document mark.
        return o


def run(d, initial):
        """Generate text from the Markov process."""
        history = initial
        while 1:
                w = get(d, history)
                if w is None:       # End of document.
                        return

                print w

                history.pop(0)                  # Adjust the history.
                history.append(w)


if __name__ == '__main__':
        import sys
        bigrams = build(sys.stdin, [None, None])
        run(bigrams, [None, None])
```

References

Daniel Jurafsky and James H. Martin. *Speech and Language Processing*. Prentice-Hall, Saddle River, NJ, 2000. ISBN 0-13-095069-6.