

CS 188 Project #3: HMMs for ASR

CS188 - Spring 2006

Due: April 18

In this project, we will harness the power of Hidden Markov Models (HMMs) for Automatic Speech Recognition (ASR). This document follows a tutorial format, in which the presentation of theory is interleaved with short exercises intended to demonstrate the applications.

Contents

1	Python Code Preliminaries	2
2	Fundamental Graphical Models	3
2.1	Markov Chains	3
2.1.1	Representations	4
2.1.2	Viterbi algorithm	5
2.1.3	Forward algorithm	7
2.1.4	Parameter estimation	8
2.1.5	Exercises	8
2.2	Mixture Models	10
2.2.1	Representations	10
2.2.2	Parameter estimation	11
2.3	Hidden Markov Models	12
2.3.1	Representation	12
2.3.2	Viterbi algorithm	13
2.3.3	Forward algorithm	13
2.3.4	Parameter estimation	14
2.3.5	Exercises	14
3	Automatic Speech Recognition	16
3.1	Background	16
3.1.1	Phonetics	16
3.1.2	Digital Signal Processing	18
3.1.3	Optional exercises	19
3.2	Acoustic models	20
3.2.1	Concatenative HMM units	20
3.2.2	Speech modeling units	21
3.2.3	Recognition networks	23
3.2.4	Exercises	25

1 Python Code Preliminaries

The utilities in `util.py` may be helpful in implementing this project, although you may prefer to modify them to suit your own coding style. The primary objects are based on Python's native `dict` data structure:

- **Counter**: a standard counter, which can be normalized to produce a `ProbDist`
- **ProbDist**: a probability distribution over finitely many values
- **CondCounter**: a counter for conditional event `Counters`, used to produce a `CondProbDist`
- **CondProbDist**: accesses a collection of conditional probabilities, each a `ProbDist`

As subclasses of Python's `dict`, these objects can be instantiated in many ways:

```
>>> from util import *
>>> c = Counter({True:5, False:5})
>>> p = ProbDist([('val1', 0.2), ('val2',0.8)])
>>> cc = CondCounter(one=c)
>>> cp = CondProbDist()
>>> c,p,cc,cp
({False: 5, True: 5}, {'val2': 0.8000000000000004, 'val1': 0.2000000000000001},
{'one': {False: 5, True: 5}}, {})
```

To retrieve elements, use the bracket notation, or any `dict` accessor function. Unlike `dict`, these objects will not raise a `KeyError` exception, and will instead return a default value for unhashed keys.

```
>>> c[False]
5
>>> p.keys()
['val2', 'val1']
>>> p['val3']
0.0
>>> cp['missingkey']['missingkey'] = 0.1; cp
{'missingkey': {'missingkey': 0.1000000000000001}}
>>> cp.clear(); cp
{}
```

You can convert a counter into a probability distribution:

```
>>> c.makeProbDist()
{False: 0.5, True: 0.5}
>>> cp = cc.makeCondProbDist(); cp
{'one': {False: 0.5, True: 0.5}}
```

A probability distribution can generate a random sample:

```
>>> cp['one'].sample()
False
>>> [p.sample() for i in range(10)]
['val2', 'val2', 'val2', 'val1', 'val2', 'val2', 'val2', 'val2', 'val2', 'val1']
```

Probability distributions can also be *logified* – that is, probabilities are converted to the logarithmic domain. To prevent floating-point underflow in a product of probabilities, you may wish to sum their logarithms.

```
>>> p.logify()
{'val2': -0.22314355131420971, 'val1': -1.6094379124341003}
>>> p.delogify()
{'val2': 0.8000000000000004, 'val1': 0.2000000000000001}
```

Lastly, we provide the handy function `fullrange`, which is similar to Python's `range`, but includes the endpoint. This will be useful in `for` loops, so that indices that match the formulas in the following sections.

```
>>> fullrange(1,10)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

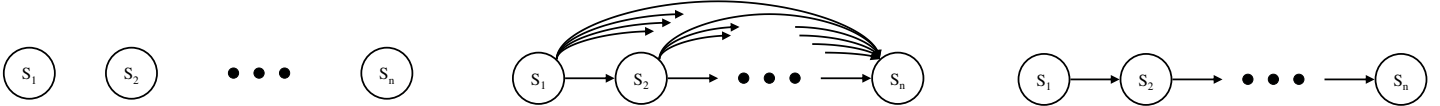


Figure 1: Left: independent events. Center: general Bayes' network. Right: first-order Markov chain.

2 Fundamental Graphical Models

Probabilistic graphical models, of which Bayesian networks are a subtype,¹ constitute a paradigm for representing large joint probability distributions as graphs. Random variables correspond to nodes in the graph, and arcs connecting nodes assert independence relations among the variables. The local relationships define a *factorization* of the joint probability distribution as the product of conditional probability distributions which can be more compactly parameterized.

In this section we will explore three kinds of graphical models, along with efficient algorithms for performing inference queries on them. We begin with Markov chains for describing sequences of random variables. Next, mixture models are briefly introduced for situations in which some variables are unobservable. These are then combined in the Hidden Markov Model, for sequences of random variables which are not completely observable. It may be more intuitive to call these Markov Models, Mixture Models, and Markov Mixture Models; alas, this is not the standard nomenclature.

2.1 Markov Chains

Consider a sequence of random variables $\mathbf{S} = (S_1, \dots, S_n)$ where each $S_i \in \mathcal{S}$. For reasons that may be clear later, we will refer to these as *states*. In some scenarios, these are assumed to be independent events (Figure 1, left), such that their joint probability is a product of the marginals:

$$P(\mathbf{S}) = \prod_{i=1}^n P(S_i) \quad (1)$$

A further simplification would treat these as independent identically distributed (*i.i.d.*) random variables, so that only one distribution would need to be parameterized. Unfortunately, the probabilistic world does not consist solely of sequences of die rolls or coin flips, and we often do not wish to make such independence assumptions. In the general case, the sequence can be represented as a Bayesian network with a causal structure determined by the variable ordering S_1, \dots, S_n (Figure 1, center). By the chain rule:

$$P(\mathbf{S}) = P(S_1) \prod_{i=2}^n P(S_i | S_1, \dots, S_{i-1}) \quad (2)$$

Each arc in the network corresponds to a conditional probability distribution. For m discrete variables, each taking on d values, the distribution can be parameterized with a conditional probability table (CPT) having d^m entries.² For the general Bayesian network, we would parameterize $n(n-1)/2$ CPTs of size $O(d^n)$.

Rather than having each S_i depend on all variables that precede it in the ordering, we can make a *Markov assumption*, limiting the dependence to a small number of directly preceding states. In particular, the first-order Markov assumption asserts that S_i depends only on S_{i-1} .³ This drastically reduces the complexity of the graphical model (Figure 1, right), called a *Markov chain*, which can be parameterized by n CPTs of size $O(d)$. A problem remains: how can we apply this model to sequences that are longer than n ? There are too many model parameters, and yet they are too specific to generalize to some sequences.

¹Bayes' nets are sometimes called *directed* graphical models

²In the AIMA book, CPTs are compactly (and perhaps confusingly) shown as $(d-1)^m$ entries, since the d -th value can be retrieved by subtracting probabilities for the $d-1$ other values.

³Any higher-order assumption can be re-formulated as first-order, with new variables to represent tuples of values.

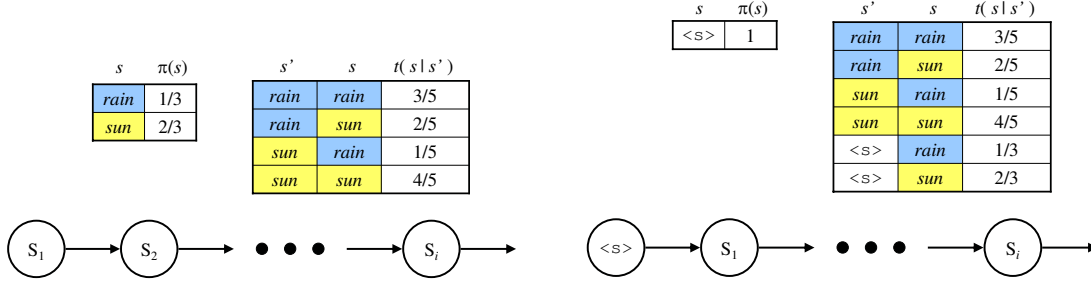


Figure 2: Left: the weather Markov chain with π and t parameters. Right: inserting a fixed start symbol.

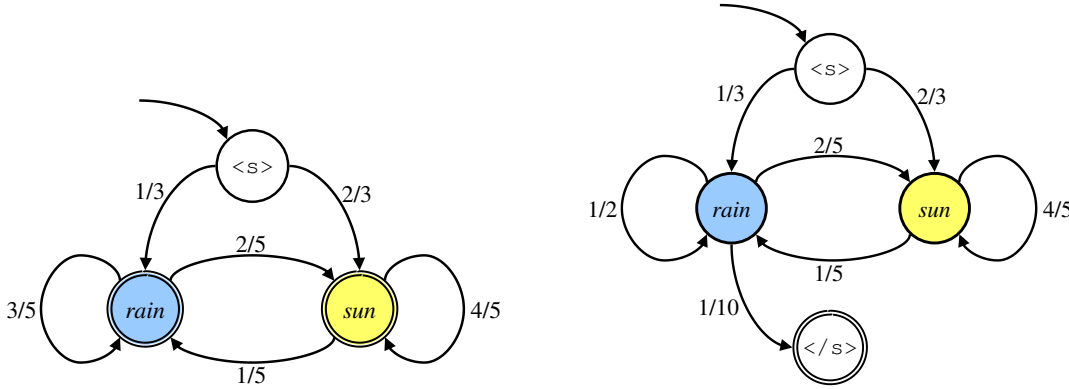


Figure 3: Left: the weather Markov chain as a FSA. Right: inserting a fixed stop state.

We address this with the *stationarity* assumption, such that all conditional probability distributions are identical. We now parameterize a single CPT that is shared among all arcs of the chain. Calling this the *transition model*, denoted $t(s|s')$:

$$t(s|s') = P(S_i = s | S_{i-1} = s') \quad \forall i, s, s' \quad (3)$$

We will only consider Markov chains with this property, also called *time-invariant* or *homogenous*. To complete the parameterization, we specify the marginal distribution over the initial state, $\pi(s) = P(S_1 = s)$. Then the likelihood of a sequence $\mathbf{s} = (s_1, \dots, s_n)$ can be expressed as:

$$P(\mathbf{s}) = \pi(s_1) \prod_{i=2}^n t(s_i | s_{i-1}) \quad (4)$$

2.1.1 Representations

Figure 2 (left) depicts a Markov chain: the probabilities can be interpreted in terms of the weather domain, simulating a sequence rainy and sunny days. The tables explicitly parameterize the distributions π and t .

It is often convenient to define $\langle s \rangle$, the *start symbol*, which must always precede a sequence. Treating $\langle s \rangle$ as the initial state, then the initial distribution is trivially $\pi(\langle s \rangle) = 1$, and the transition table is extended with entries $t(S_1 | \langle s \rangle) = P(S_1)$, as in Figure 2 (right).

By the stationarity assumption, the distributions at step i are no different from those at any other step. So rather than representations like in Figure 2, which “roll out” the Markov chain across time, we can instead draw the model as in Figure 3 (left). States are connected with arcs corresponding to probabilities from the transition model. The starting state $\langle s \rangle$ is denoted by a single incoming arc with no source, and we draw a

double-outline around all *accepting states*, which are the all values that can terminate a sequence. If you are familiar with the theory of *regular languages*, you may have observed a resemblance between Markov chains and (non-deterministic) finite-state automata.

Thus far, if we queried the Markov chain for the probability of a sequence, $\mathbf{s} = (s_1, \dots, s_n)$, it would actually give the probability of all sequences of length n or greater that begin with the subsequence \mathbf{s} . We might be more interested in the probability that the model generates exactly the sequence \mathbf{s} and then halts. For models that answer this query, we introduce $\langle /s \rangle$, the *stop symbol*, which is at the end of any sequence. Once reached, no more states can be generated and so $P(s|\langle /s \rangle) = 0$.⁴ The Markov chain in Figure 3 (right) can now be interpreted as a simulation that runs for some number of steps before terminating. In this example, the world might come to an end after a rainy day. Or, for a non-apocalyptic interpretation: suppose this is the weather report from an outdoor festival, which could be canceled due to rain.

We will choose to represent Markov chains as in Figure 3 (right), for which we only need to specify a set of states \mathcal{S} and the transition model t . The likelihood of a sequence $\mathbf{s} = (s_1, \dots, s_n)$ is then:

$$P(\mathbf{s}) = P(\langle s \rangle, s_1, \dots, s_n, \langle /s \rangle) = t(s_1|\langle s \rangle) \left(\prod_{i=2}^n t(s_i|s_{i-1}) \right) t(\langle /s \rangle|s_n) \quad (5)$$

Note that when we refer to a sequence $\mathbf{s} = (s_1, \dots, s_n)$, we implicitly insert the start and stop symbols as if they were s_0 and s_{n+1} . In the next section, we will also refer to a *partial sequence* of length $i < n$ that is not terminated by $\langle /s \rangle$, but by some state s_i :

$$P(\langle s \rangle, \dots, s_i) = t(s_1|\langle s \rangle) \prod_{j=2}^i t(s_j|s_{j-1}) \quad (6)$$

2.1.2 Viterbi algorithm

Suppose we wish to determine \mathbf{s}^* , the most likely sequence of n states that is generated by a Markov chain:

$$\mathbf{s}^* = \arg \max_{\mathbf{s}} P(\mathbf{s}) = \arg \max_{s_1, \dots, s_n} P(\langle s \rangle, s_1, \dots, s_n, \langle /s \rangle) \quad (7)$$

A rather slow algorithm for this would enumerate the $O(|\mathcal{S}|^n)$ exponentially many sequences and choose the one of highest likelihood, from Eq. 5. Fortunately, there are more efficient techniques.

We can “roll out” a Markov chain to form a *trellis* (Figure 4), showing the possible states at each step connected by arcs specifying transition probabilities. Note the similarity between the trellis and a search graph. In fact, we can provide a search problem formulation:

States: $(s, i) \in \{\langle s \rangle, \langle /s \rangle\} \cup \mathcal{S} \times \mathcal{N}$

Initial state: $(\langle s \rangle, 0)$

Successor function: from a state $(s', i - 1)$, generate new states $\{(s, i) : t(s|s') > 0\}$

Goal state: $(\langle /s \rangle, n + 1)$.

Cost function: The step cost from $(s', i - 1)$ to (s, i) is $t(s|s')$.

The cost of a path to (s, i) is computed as the product of all step costs.

The path cost to reach (s, i) equals the likelihood of a partial sequence of length i that is terminated by state s . Then to find \mathbf{s}^* , we would search for a maximum-cost path to $(\langle /s \rangle, n + 1)$ in the trellis.

How do we find the maximum-cost path? We could adjust the search problem formulation to make use of optimal (i.e. minimum-cost) search strategies. For example: take the logarithm of all probabilities, and multiply them by -1 . Now the path cost is the sum of the step costs, which are non-negative, and the most likely sequence \mathbf{s}^* corresponds to the optimal solution path. We might try uniform-cost search, but this could be very inefficient if it explores a very long path consisting of low-cost steps, corresponding to a long

⁴The self-transition is also zero, $P(\langle /s \rangle|\langle /s \rangle) = 0$, so there is not a proper distribution of transitions conditioned on $\langle /s \rangle$.

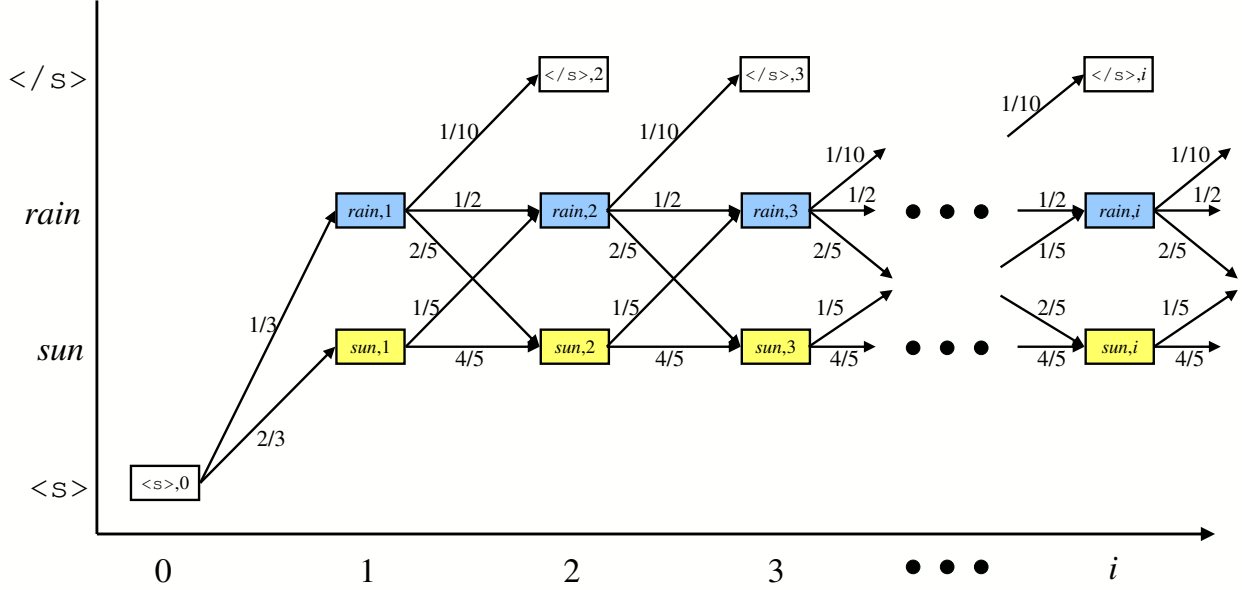


Figure 4: A trellis for the weather Markov chain. Unreachable states are not displayed.

sequence of high-probability transitions.⁵ If the goal is $(\langle s \rangle, n + 1)$, there is no reason to explore the search tree at depths greater than $n + 1$. Knowing the solution depth suggests that we should try a depth-limited search; but the first solution found is not guaranteed to be optimal because step-costs are not uniform.

There is an alternative in the Viterbi algorithm, a *dynamic programming* technique that exploits the structure of the trellis search space. This is one of a family of related algorithms that make use of a *forward recurrence* by noting that the cost of a path to (s, i) can be considered as the cost of a path to $(s', i - 1)$ incremented by the step cost from s' to s . Thus, instead of considering the $O(|\mathcal{S}|^i)$ possible paths into (s, i) , we can simply consider the $O(|\mathcal{S}|)$ steps from the preceding states of column $i - 1$ in the trellis.

A maximum-cost path to (s_i, i) corresponds to a most likely partial sequence of length i that terminates with the state s_i . Let us denote this quantity of interest as a *message*, $m[i][s_i]$:

$$m[i][s_i] = \max_{s_1, \dots, s_{i-1}} [P(\langle s \rangle, \dots, s_i)] \quad (8)$$

By separating out the last term in Eq. 6:

$$m[i][s_i] = \max_{s_1, \dots, s_{i-1}} [P(\langle s \rangle, \dots, s_{i-1}) \cdot t(s_i | s_{i-1})] \quad (9)$$

Re-arranging the indices of this maximization:

$$m[i][s_i] = \max_{s_{i-1}} \left[\max_{s_1, \dots, s_{i-2}} [P(\langle s \rangle, \dots, s_{i-1}) \cdot t(s_i | s_{i-1})] \right] \quad (10)$$

The messages can therefore be defined recursively:

$$m[i][s_i] = \max_{s_{i-1}} [m[i-1][s_{i-1}] \cdot t(s_i | s_{i-1})] \quad (11)$$

⁵Also, UCS might not even be complete if there are zero-cost steps, due to transitions of probability 1.

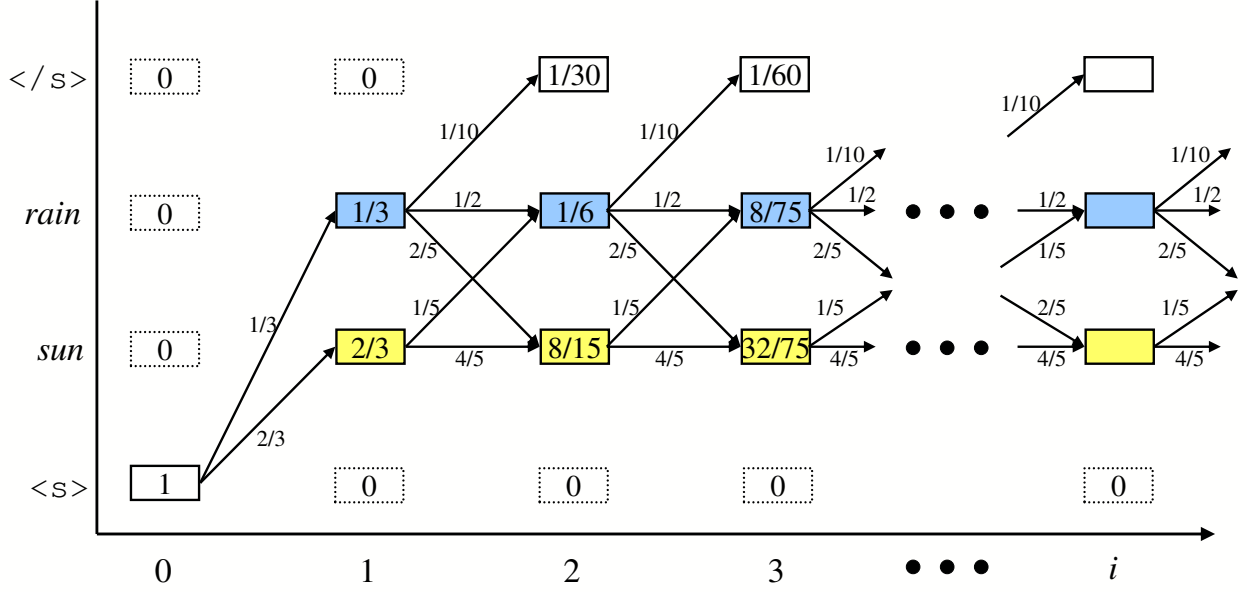


Figure 5: A trellis for the weather example, in which nodes are labeled with the Viterbi messages $m[i][s]$.

To initialize the recursion:

$$m[0][s] = \begin{cases} 1 & \text{if } s = \langle s \rangle \\ 0 & \text{if } s \neq \langle s \rangle \end{cases} \quad (12)$$

The likelihood of the most likely sequence of n states is thus:

$$P(\mathbf{s}^*) = \max_{s_1, \dots, s_n} P(\langle s \rangle, s_1, \dots, s_n, \langle /s \rangle) = m[n+1][\langle /s \rangle] \quad (13)$$

To recover the most probable sequence \mathbf{s}^* , we can store back-pointer messages, $p[i][s_i]$:

$$p[i][s_i] = \arg \max_{s_{i-1}} [m[i-1][s_{i-1}] \cdot t(s_i | s_{i-1})] \quad (14)$$

Then \mathbf{s}^* is read back in reverse order by following the back-pointers, starting at $p[n+1][\langle /s \rangle]$.

Figure 5 demonstrates the calculation of the Viterbi algorithm on the weather example. To implement this, we would need to store arrays for m and p , each of dimension $(|\mathcal{S}| + 2) \times (n + 2)$. This algorithm for finding the most likely sequence of length n therefore has linear time and space complexity, $O(|\mathcal{S}|n)$. Dynamic programming techniques such as this Viterbi algorithm, and also the Fast Fourier Transform algorithm, are easily implemented in hardware and are prevalent in many modern communication devices.

2.1.3 Forward algorithm

Another algorithm that is very closely related to the Viterbi algorithm is the forward algorithm. This can be used to answer a query about the *total likelihood* of all sequences \mathbf{s} of length n :

$$\ell = \sum_{\mathbf{s}} P(\mathbf{s}) = \sum_{s_1, \dots, s_n} P(\langle s \rangle, s_1, \dots, s_n, \langle /s \rangle) \quad (15)$$

In terms of the trellis, ℓ is the sum of the costs of all paths to $(\langle /s \rangle, n + 1)$. As with the Viterbi algorithm, we can exploit the forward recurrence by caching messages. Now $m[i][s_i]$ represents the total likelihood of a

partial sequence of length $i < n$ that is terminated by s_i :

$$m[i][s_i] = \sum_{s_1, \dots, s_{i-1}} P(\langle \mathbf{s} \rangle, \dots, s_i) \quad (16)$$

The recursive definition simply replaces the maximization of the Viterbi algorithm with a summation:

$$m[i][s_i] = \sum_{s_{i-1}} [m[i-1][s_{i-1}] \cdot t(s_i | s_{i-1})] \quad (17)$$

The total likelihood is then a message in the last column of the trellis: $\ell = m[n+1][\langle \mathbf{s} \rangle]$

2.1.4 Parameter estimation

Given a training set of example sequences, it is fairly easy to estimate parameters of a Markov chain's transition model with relative frequencies that will maximize the likelihood of the training data. We gather empirical counts, $c(s|s')$, that count the number of times that s followed s' in the training data. The estimated parameters are calculated by normalization:

$$\hat{t}(s|s') = \frac{c(s|s')}{\sum_{s''} c(s''|s')} \quad (18)$$

2.1.5 Exercises

Instructions: Supply solution code in the file `mc_exercises.py` and provide any written responses in `mc_exercises.txt`. For all exercises, use probabilities in the standard non-logarithmic domain.

Exercise 0. Carefully read the documentation in `MarkovChain.py`. Look at the first bit of code in `mc_exercises.py`, and be sure to understand it. This shows how to construct the weather Markov chain example depicted in Figure 3 (right). In addition, a training set of weather sequences is then randomly generated, and used to estimate the parameters of a new Markov chain.

Exercise 1. Write the function `calcLikelihood` to compute the likelihood of a sequence. For the weather Markov chain, what is the likelihood of $(rain, sun, rain)$? What is the likelihood of $(sun, rain, sun)$?

Exercise 2. The `sequenceGenerator` function enumerates all sequences of length n . Using the approach of Eq. 7, write the function `mostLikelySequence_Slow` to find the most likely sequence of length n . Write the function `totalLikelihood_Slow`, which sums the likelihoods of all sequences of length n . For the weather example, what values of n cause the computation to run for an unbearably long time?

Exercise 3. Fill in the missing recursive step of the Viterbi algorithm in the function `viterbi`, and find the most likely sequence of length n with `mostLikelySequence_Viterbi`. Fill in `forward` for the forward algorithm, and use the function `totalLikelihood_Forward` to compute the total likelihood of all sequences of length n . For the weather example, what values of n cause the computation to exhibit underflow errors?

Exercise 4. For the weather example, what pattern do you notice among the most likely sequences of various lengths? How does this compare to your intuition of what a “likely” weather pattern should be, and how might you augment the model to better reflect reality?

Exercise 5. The *Viterbi approximation* estimates the total likelihood as the probability of a single most likely sequence. Under what circumstances is this a good approximation?

Exercise 6. What is the expected length of a sequence generated by the weather Markov chain?

Hint: There are at least three ways to approach this question: inspect the model structure and work it out analytically (give exact answer); approximate the expectation with a partial summation (give a lower bound); or empirically estimate the quantity from randomly sampled data (give confidence bounds).

Extra Non-Credit. Try out some of these ideas:

- **Implementation issues:** To avoid numerical underflow, the Viterbi algorithm should almost always be implemented with probabilities in the logarithmic domain. For the forward algorithm, there is a clever way to re-scale the messages at each step so that they stay within a safe numerical interval.
- **Beam search:** This faster Viterbi search explores a limited number of paths through each column of the trellis. This is not guaranteed to find the optimal solution, however.
- **N-best:** Modify the Viterbi algorithm to return the N most likely sequences.

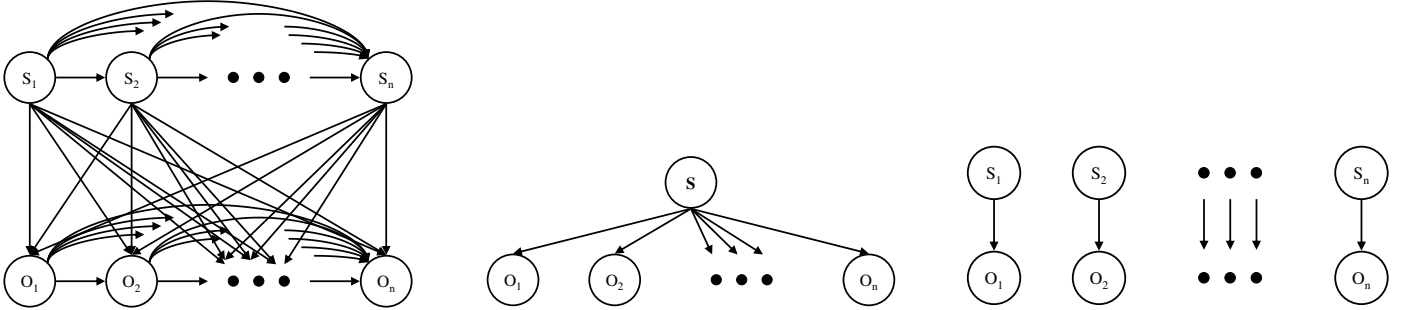


Figure 6: Left: general Bayes' network. Center: Naive Bayes' Model. Right: mixture model.

2.2 Mixture Models

In a large class of scenarios, we are concerned with some sequence of events in a world that is not completely observable. However, there is often some other evidence available from which we may attempt to model the situation. In general, the evidence can be denoted as a sequence of *observations*, $\mathbf{O} = (O_1, \dots, O_n)$, from which we hope to use inference algorithms to find the related sequence of *states*, $\mathbf{S} = (S_1, \dots, S_n)$. We will consider generative models in which observations are generated by these *hidden states*.⁶

2.2.1 Representations

As a starting point, we might order the variables $(S_1, \dots, S_n, O_1, \dots, O_n)$ and draw a general Bayes' network (Figure 6, left) that makes no independence assertions. Clearly, such a model is overly specified and will be too difficult use. We are often dealing with situations in which the observations are drawn from a very large domain \mathcal{O} which could have infinitely many values, possibly continuous; the states, on the other hand, are generally considered from a small domain \mathcal{S} of finitely many discrete values. Thus, we will prefer models that parameterize distributions conditioned on the states, rather than conditioning on observations.

Under one simplifying assumption we have a Naive Bayes' model (Figure 6, center) which posits that the sequence of observations \mathbf{O} are effects generated by a single underlying cause, \mathbf{S} . If \mathbf{S} takes on values from the domain \mathcal{S}^n , there will be too many parameters to be reasonably trained from data.

Alternatively, we might suppose that each individual observation O_i was generated by the state at the same instant, S_i , and that each of these relationships is independent of the processes at other time steps (Figure 6, right). The joint distribution of states and observations is factorized as:

$$P(\mathbf{S}, \mathbf{O}) = P(\mathbf{S})P(\mathbf{O}|\mathbf{S}) = \prod_{i=1}^n P(S_i)P(O_i|S_i) \quad (19)$$

To generalize to sequences of arbitrary length, we can make an assumption of *i.i.d.* sampling. This becomes a *mixture model* for which we only have to learn the parameters of two distributions:

$$\begin{aligned} \pi(s) &= P(S_i = s) & \forall i, s \\ e(o|s) &= P(O_i = o|S_i = s) & \forall i, o, s \end{aligned} \quad (20)$$

We call the latter the *emission* model and denote it by $e(o|s)$.⁷

Mixture models exhibit a strong independence assertion: events at step i are independent of events at all other steps. In a model with no sequential dependencies, the order of the states and observations is irrelevant. Nonetheless, this is a common simplifying assumption, and mixture models of this sort can be built into very effective *classifiers*, even for time-varying phenomena such as speech.

⁶These are sometimes called *latent* variables.

⁷More commonly, the parameters $e(o|s)$ are *mixture components* and $\pi(s)$ are the *mixing proportions*.

Conditioned on a given a state sequence \mathbf{s} , the likelihood of an observation sequence \mathbf{o} is factorized as a product of independent emission probabilities:

$$P(\mathbf{o}|\mathbf{s}) = \frac{P(\mathbf{s}, \mathbf{o})}{P(\mathbf{s})} = \prod_{i=1}^n e(o_i|s_i) \tag{21}$$

We will use this result in the next section, to factorize the Hidden Markov Model.

2.2.2 Parameter estimation

Given observation sequences along with the state sequences that generated them, the parameters of the emission model are easily learned as relative frequencies based on empirical counts $c(o|s)$, the number of times that observation o was generated by state s in the training data:

$$\hat{e}(o|s) = \frac{c(o|s)}{\sum_{o'} c(o'|s)} \tag{22}$$

Unfortunately, this maximum-likelihood estimate only works *if we are given the state sequence*, which is quite often not available in real-world situations. To learn the parameters of a distribution in which some variables are unobserved in the training data, we must generally resort to the Expectation-Maximization (EM) algorithm. We defer consideration of this subject until the following section.

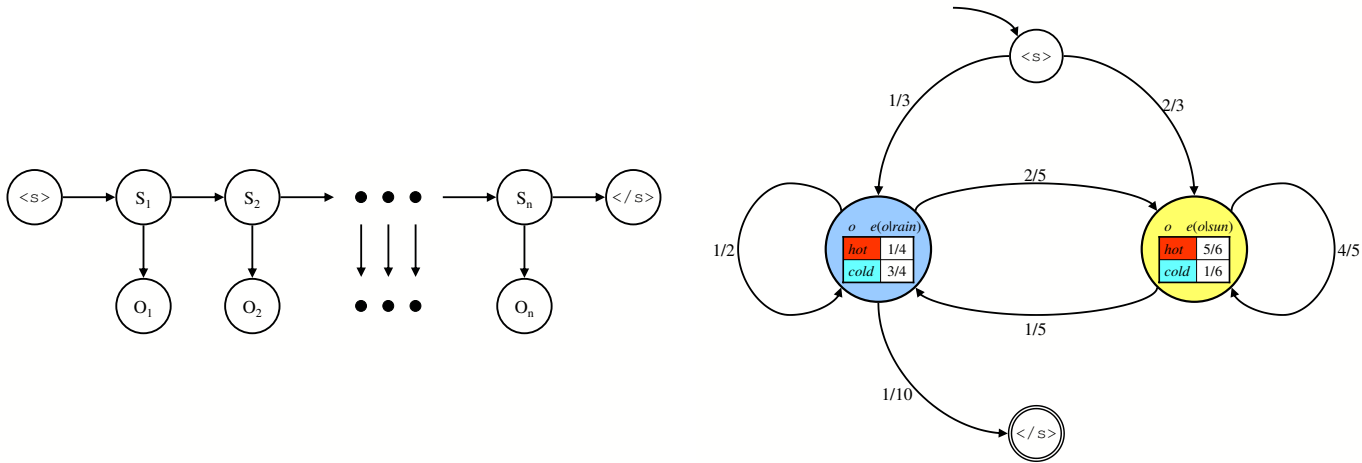


Figure 7: Left: a Hidden Markov Model, “rolled out”. Right: HMM description for the weather example.

2.3 Hidden Markov Models

To continue the weather example from the Markov chain section, suppose that by the cruelty of fate you find yourself locked inside a metal cargo container aboard a freighter somewhere on the Pacific Ocean. Unable to directly observe *rain* or *sun*, your daily observations consist of two temperature sensations: *hot* or *cold*.

Given a sequence of temperature observations, how can you infer the most likely sequence of weather conditions? A Hidden Markov Model allows you to perform this type of query, as well as others: if you have different competing models of the world, how can you compare them in terms of your observations? Also, if you did not know the parameters of a model, how could you learn them from incomplete training data in which some variables are never observed?

In this section, we present HMMs. With a solid understanding of Markov chains and mixture models, it is straightforward to interpret HMMs as a combination of the two previous models.

2.3.1 Representation

With the HMM framework, we assume that there is structure to the state sequence – it is a Markov process. To derive the HMM, we can simply augment the Markov chain with an emission model specifying the probability of observations. Or we can view this as a mixture model in which the states are no longer generated independently, but as a Markov process.

Figure 7 (left) shows the structure of an HMM “rolled out” across time. Figure 7 (right) depicts an HMM for the weather example in terms of the Markov chain’s state transition diagram. In this representation, we associate the parameters of the emission model $e(o|s)$ with the conditioning state.⁸ Note in particular that there are no emission models associated with the states $\langle s \rangle$ and $\langle /s \rangle$. These are called *null* states, for which the emission probability distribution is not clearly defined. To simplify an intricate issue, we could assume that empty symbols ε are attached at the ends of the observation sequence: $o_0 = o_{n+1} = \varepsilon$. Then the emission models for the null states are specially defined as $e(\varepsilon|\langle s \rangle) = e(\varepsilon|\langle /s \rangle) = 1$.

There are several interesting independence relations that can be read off the HMM structure. Without knowing the state sequence, the observations could have some dependence. However, conditioning on any state S_i effectively separates the network into three sets of random variables: the “present” observation O_i is conditionally independent of everything else; furthermore, all “past” variables with indices less than i become independent of any “future” variables with indices greater than i . Note that conditioning on observations results in no independence assertions: given observations of O_1, \dots, O_{n-1} , then the value of O_n at one end of the network could still depend on state S_0 all the way at the front of the chain.

⁸In an alternative representation that more closely resembles FSA, observations are generated by the arcs rather than states.

2.3.2 Viterbi algorithm

Recall that an HMM is a Markov chain combined with a mixture model. Thus the joint likelihood of a state sequence \mathbf{s} and an observation sequence \mathbf{o} can be factorized from the HMM structure as the product of the likelihood of the state sequence in the Markov chain (Eq. 5) and the likelihood of an observation sequence given the state sequence for a mixture model (Eq. 21):

$$P(\mathbf{s}, \mathbf{o}) = P(\mathbf{s})P(\mathbf{o}|\mathbf{s}) = t(s_1|\langle \mathbf{s} \rangle) \left(\prod_{i=2}^n t(s_i|s_{i-1}) \right) t(\langle / \mathbf{s} \rangle | s_n) \prod_{i=1}^n e(o_i|s_i) \quad (23)$$

A common query is to determine the most likely state sequence given a sequence of observations \mathbf{o} :

$$\mathbf{s}^* = \arg \max_{\mathbf{s}} P(\mathbf{s}|\mathbf{o}) = \arg \max_{\mathbf{s}} \frac{P(\mathbf{s}, \mathbf{o})}{P(\mathbf{o})} \quad (24)$$

The quantity $P(\mathbf{o})$ can be quite difficult to compute, particularly for large state spaces. Yet, noting above that $P(\mathbf{o})$ is constant because \mathbf{o} is a fixed observation, the term is irrelevant to the maximization:

$$\mathbf{s}^* = \arg \max_{\mathbf{s}} P(\mathbf{s}, \mathbf{o}) \quad (25)$$

With a formula for the joint likelihood, we could determine \mathbf{s}^* by enumerating $O(|\mathcal{S}|^n)$ sequences.⁹ As with Markov chains, however, the Viterbi algorithm can be summoned for the same purpose.¹⁰

For HMMs, the cached Viterbi messages $m[i][s_i]$ are maximizations over the joint likelihoods of partial state sequences terminated by s_i and a partial observation sequence (o_1, \dots, o_i) :

$$m[i][s_i] = \max_{s_1, \dots, s_{i-1}} [P(\langle \mathbf{s} \rangle, s_1, \dots, s_i, o_1, \dots, o_i)] \quad (26)$$

These messages can be defined recursively, where each new message incorporates an additional transition from the previous column in the trellis and an emission from the most recently generated state:

$$m[i][s_i] = \max_{s_{i-1}} [m[i-1][s_{i-1}] \cdot t(s_i|s_{i-1}) \cdot e(o_i|s_i)] \quad (27)$$

The maximizing joint likelihood $P(\mathbf{s}^*, \mathbf{o})$ is given by the message $m[n+1][\langle / \mathbf{s} \rangle]$, and the desired most likely state sequence \mathbf{s}^* can be retrieved by following the back-pointers, starting at $p[n+1][\langle / \mathbf{s} \rangle]$.

2.3.3 Forward algorithm

It is sometimes interesting to determine the total likelihood an observation sequence, $\ell(\mathbf{o})$, the sum of joint likelihoods over all possible state sequences that could have generated the given observation sequence:

$$\ell(\mathbf{o}) = \sum_{\mathbf{s}} P(\mathbf{s}, \mathbf{o}) \quad (28)$$

For example, total likelihood could be used to compare different HMM model structures that might have generated a sequence of observations. Or it could compute the normalizing denominator from Eq. 24.

As with Markov chains, the forward algorithm for HMMs is similar to the Viterbi algorithm, replacing maximizations with summations. The forward messages $m[i][s_i]$ are thus recursively defined:

$$m[i][s_i] = \sum_{s_{i-1}} [m[i-1][s_{i-1}] \cdot t(s_i|s_{i-1}) \cdot e(o_i|s_i)] \quad (29)$$

The total likelihood of \mathbf{o} is then given by the message $m[n+1][\langle / \mathbf{s} \rangle]$.

⁹This method will give us the most likely state sequence, given an observation, but not its conditional likelihood.

¹⁰This is by far the most common application of the Viterbi algorithm. Typically, it is presented in the context of HMM inference, since the most likely sequence in a Markov chain is not as meaningful or interesting.

2.3.4 Parameter estimation

If we are given the state sequences that generated certain observation sequences in some training data set, then it is possible to learn the maximum-likelihood estimates of the HMM parameters. The state transitions are simply counted as in Eq. 18, and the emissions are counted as in Eq. 22.

More commonly, we have a partially observed training set in which the state sequences are unknown, so we need to learn the parameters in some other manner. One approach is Viterbi training, a process of iterative re-estimation which works as follows:

1. Initialize parameters t and e , perhaps randomly.
2. Using the current parameters, construct an HMM and use the Viterbi algorithm to find the most likely state sequence \mathbf{s}^* that generated each observation \mathbf{o} in the partially observed training set.
3. Create a new training set comprising “fully observed” pairs $(\mathbf{o}, \mathbf{s}^*)$.
4. Update t and e with maximum-likelihood estimates (Eqs. 18 and 22) from this “fully observed” data.
5. Repeat from Step 2

This process could also be called “hard EM”. It is related to the more general Expectation-Maximization algorithm, which uses posterior likelihoods to weight the posited training examples. The training procedure is usually run for just a few iterations, or until the data likelihood reaches convergence.

2.3.5 Exercises

Instructions: Supply solution code in the file `hmm_exercises.py` and provide any written responses in `hmm_exercises.txt`. For all exercises, use probabilities in the standard non-logarithmic domain.

Exercise 0. Carefully read the documentation in `HiddenMarkovModel.py`. Look at the first bit of code in `hmm_exercises.py`, and be sure to understand it. This shows how to construct the weather HMM depicted in Figure 7 (right). In addition, a fully observed training set of weather examples (pairs of state sequences and observations sequences) is then randomly generated, and used to learn maximum-likelihood estimates of the parameters for a new HMM.

Exercise 1. Fill in the missing recursive step of the Viterbi algorithm in the function `viterbi`, and use the function `mostLikelyStateSequence` to find the most likely state sequence given an observation sequence. Fill in `forward` for the forward algorithm, and use the function `totalLikelihood` to compute the total likelihood of an observation sequence.

Exercise 2. The example HMM in Figure 7 (right) models typical weather in the Bay Area. Suppose you become trapped in a cargo container at the Port of Oakland, then escape four days later.¹¹ Over that time, your temperature observations were *hot, cold, hot, cold*. What was the most likely weather pattern?

Oops! You were a bit confused about the observations, which were actually *cold, hot, cold, hot*. What was the most likely weather pattern for these revised observations?

Exercise 3. You have the unfortunate habit of falling into cargo containers and then being shipped around the world. Being well-traveled, you know the weather HMMs for places such as Alaska (Figure 8, left). Now let’s say you find yourself inside a cargo container somewhere in the Pacific Northwest, but you don’t know if you’re in the Bay Area or Alaska. You observe the temperature sequence *(cold, hot, hot)*. Which weather model was most likely to have caused these observations? What if you observed *(hot, hot, cold)*?

Suppose that your prior probability of being in the Bay Area is 99%, and your observations consist only of cold days. How many observations do you need before concluding that you are probably in Alaska?

¹¹To interpret this increasingly contrived example: observations end after a rainy day because lightning cracks open the container. Or perhaps the dampness causes the walls of the cargo container to rust through. I don’t know...

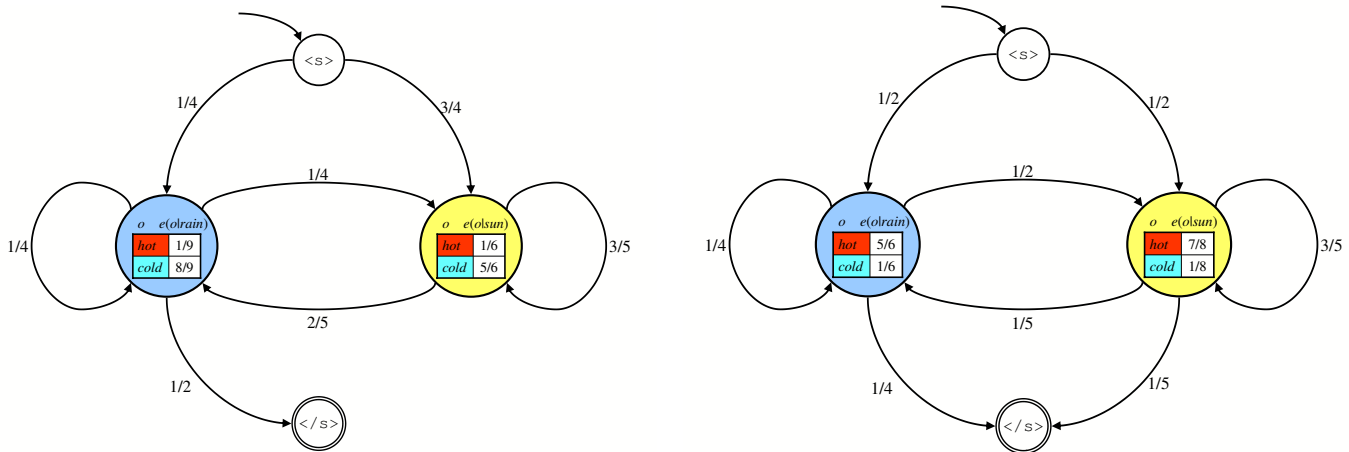


Figure 8: Left: weather HMM for Alaska. Right: weather HMM for Hawaii.

Exercise 4. The weather HMM for Hawaii is shown in Figure 8 (right). Suppose you did not know the parameters of this model, but somehow obtained a set of temperature observations from a hapless fellow who frequently fell into cargo containers at the Port of Honolulu.

Explore the code for this exercise in `hmm_exercises.py`, in which you try to learn the parameters of the Hawaiian weather HMM from observation sequences. Run the `viterbiTrainer` with variously initialized HMM models. Note how many iterations it usually takes for the Viterbi training procedure to reach convergence. What happens to the data likelihood after each iteration?

Are the learned parameters reliable? Try a few training sets of different sizes, or use `averagedTrainer` to combine the results of a number of training runs. Discuss any interesting conclusions that you are able to draw from these experiments.

Extra Non-Credit. Try out some of these ideas:

- **EM algorithm:** The “soft” EM algorithm¹² has several properties which make it much better than the “hard” approximation we have presented here. To learn model parameters effectively, you will need to calculate marginal posteriors over individual states in a sequence. This may require you to implement the *backward* algorithm, which is unsurprisingly similar to the forward algorithm.
- **HMM concatenation:** Suppose you had a collection of HMMs, such as weather models for the Bay Area, Alaska, and Hawaii. Now consider a scenario in which you are inside a cargo container aboard a freighter that is cruising around the Pacific Ocean. The ship’s route visits the three locations according to some Markov random process. How can you link together the weather HMMs into a larger model that describes this scenario? This is the concept of *concatenative* modeling in which small HMM units are joined together to form larger networks.¹³ In speech recognition, for example, small phonetic HMMs are “strung” together to model words and phrases.
- **Optimization:** There are many ways to optimize the code we have developed in this tutorial, to make it suitable for more wide-scale applications. Numerically, probabilities should generally be represented in the logarithmic domain. Structurally, efficiency can be gained by representing HMMs as matrices (AIMA, Ch. 15.3). Algorithmically, there are some instances of concatenative HMMs for which the Viterbi search is outperformed by different decoding strategies (which find the most likely sequence of larger units, such as words, rather than states).

¹²For the specific case of HMMs, and particularly in the ASR community, this is sometimes called Baum-Welch re-estimation.

¹³This was the motivation for representing HMMs with special start and stop null states.

3 Automatic Speech Recognition

The speech recognition problem has a well-known noisy channel formulation in which a speaker’s intended words \mathbf{W} are encoded into an acoustic signal \mathbf{O} . The task of the recognizer is to adequately model this process and recover the hypothesized word string $\hat{\mathbf{W}}$ that best matches some acoustic evidence \mathbf{O} . In a probabilistic model, this is the maximization of the quantity:

$$\hat{\mathbf{W}} = \arg \max_{\mathbf{W}} P(\mathbf{W}|\mathbf{O}) \tag{30}$$

We can frame this problem differently by applying Bayes’ formula:

$$\hat{\mathbf{W}} = \arg \max_{\mathbf{W}} \frac{P(\mathbf{O}|\mathbf{W})P(\mathbf{W})}{P(\mathbf{O})} \tag{31}$$

When the observation \mathbf{O} is fixed, we seek the maximum *a posteriori* estimate:

$$\hat{\mathbf{W}} = \arg \max_{\mathbf{W}} P(\mathbf{O}|\mathbf{W})P(\mathbf{W}) \tag{32}$$

The observation likelihood $P(\mathbf{O}|\mathbf{W})$ can be computed with an *acoustic model*, while the prior $P(\mathbf{W})$ is generally provided by a *language model*. The desired $\hat{\mathbf{W}}$ maximizes the combination of these components (Eq. 32); finding this recognition hypothesis is the *decoding* problem, which is intractable under the naïve approach of enumerating all possible word sequences.

We begin this section by very briefly discussing the acoustic phonetics and signal processing background, principally introducing the concept of formant frequencies and their role in defining vowel quality. We then present a variety of HMM acoustic modeling techniques for representing speech sounds. Lastly, we consider ways in which trained speech units can be joined together to form recognition networks for various tasks.

3.1 Background

ASR is a specialization at the intersection of three larger fields of study: computer science, linguistics, and electrical engineering. In this computer science course we explore the models and algorithms which have made ASR a classic and reasonably successful application of statistical AI. However, it is also very important to understand the phonetic and phonological foundations which enable an analysis of speech sounds, and the digital signal processing that forms the crucial first stage of any ASR system.

3.1.1 Phonetics

Speech is produced in humans by intricate manipulations of a physiological apparatus that result in a variety of distinctive sounds. Beginning with a controlled exhalation, most speech sounds are initially generated by the vibration of the vocal folds in the larynx as air from the lungs rushes up the trachea and through the constricted glottis. This air then continues through the vocal tract, from the pharynx and then into the oral cavity. Within the mouth, the tongue and lips can be moved into different configurations that critically determine the quality of the resulting sound. The sound can also be characterized by other factors, such as whether the vocal cords vibrate and whether the velum is lowered to divert airflow to the nasal cavity. These mechanisms are studied as *articulatory phonetics*, which provides a system in which sounds of the world’s languages can be classified according to physiological factors such as place and manner of articulation.

Although generative models attempt to simulate the process by which a phenomenon is produced, most ASR systems do not directly manage speech at the articulatory level, in part because data of this kind is very complicated (and uncomfortable) to collect.¹⁴ However, the taxonomy of speech sounds with respect to articulation is quite useful as a system for representing modeling units. Rather than creating a large number

¹⁴A very notable exception: <http://speech.llnl.gov>

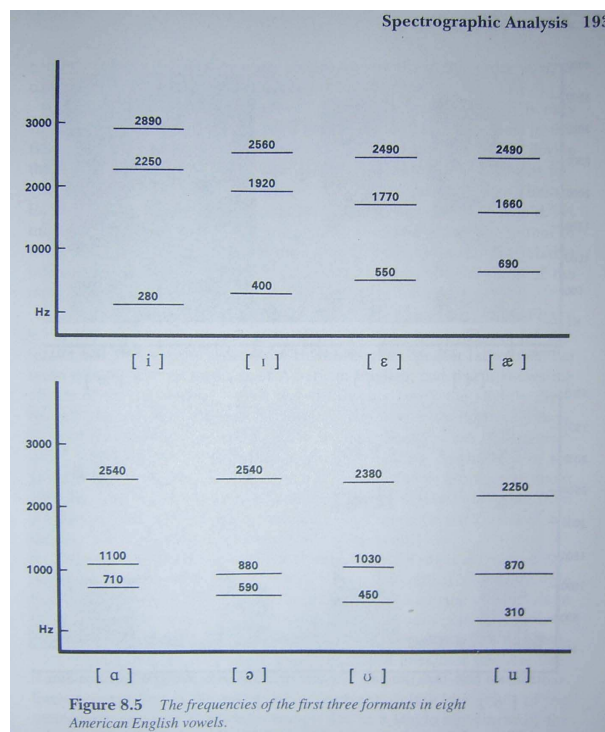
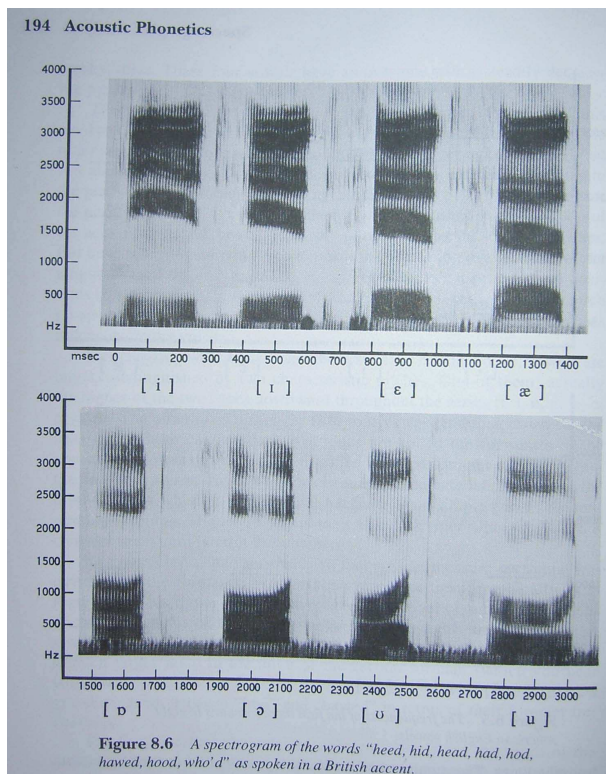


Figure 9: A spectrogram (left), and formant locations (right). (From Ladefoged, *A Course in Phonetics*)

of models for individual words, a phonetic inventory can be used to assemble words as sequences of smaller models drawn from a set of *phones*.¹⁵

After being generated during the articulation process, a speech sound becomes part of a time-varying signal that is physically transmitted as the propagation of waves of air pressure. These signals are received by an ear or a microphone – both electro-mechanical transducers, in a way – before ultimately being decoded as the intended phones and words. Of particular interest to the study of *acoustic phonetics* is this relationship between a linguistic speech unit and its realization as a sound signal. Inverting this relationship is the key to ASR, since these acoustic observations are the only evidence available.

Although some of the earliest attempts at ASR worked with sound objects in the natural time-amplitude domain, more comprehensive analysis is possible by examining the signal in the time-frequency domain. That is, consider the frequency components present within a short amount of time, a moving *window* centered at each time instant. A *spectrum* of the relative amplitudes of each frequency component can be characteristic of a particular speech sound. It is even more informative to view a dynamic series of spectra, a *spectrogram* representation (Figure 9, left) in which spectral peaks show up as dark horizontal bars.

The *source-filter* theory of speech production is a model which explains sounds as being generated by a periodic source signal (vibration of the vocal folds) that is modified by a filter (the vocal tract) which selectively attenuates certain frequency components. The filter function can be derived from the shape of an *open-tube* approximation of the vocal tract, which is principally determined by the placement of the tongue and opening of the jaw. The frequency bands which are amplified by the filter correspond to the resonant frequencies of the vocal tract; these local peaks in the frequency spectra are known as *formants*. In particular, the lowest such formant is denoted F_1 , and the higher formants are denoted F_2 , F_3 , and F_4 .¹⁶

¹⁵These are really *phonemes* determined by a language's *phonology*, but the distinction is usually not made.

¹⁶To be confusing, phoneticians also refer to F_0 , which is not a formant but the fundamental frequency, or first harmonic.

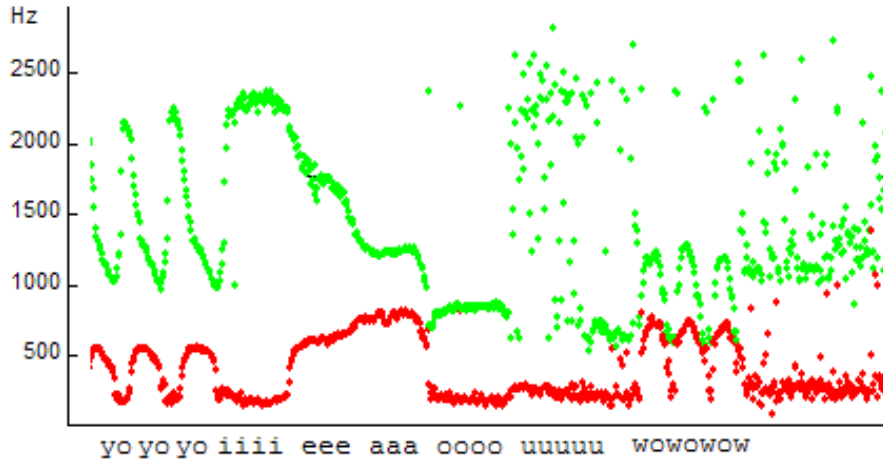


Figure 10: First and second formant estimates (red and green, respectively) produced with the Snack Sound Toolkit. Note that when F_2 is low the tracker has difficulty locating it, and finds F_3 instead.

There is an interesting relationship between the auditory perception of vowel sounds and the formant properties of the acoustic signal. Unlike consonants, which can be classified according to a discrete set of articulatory features, the vowel space does not quite partition in this way.¹⁷ An acoustic analysis, however, shows that vowels are mainly determined by the location of their first two formants (Figure 9, right). In terms of the source-filter model of speech production, these formants correspond to the front and back cavity resonances of the vocal tract configuration for a given vowel.

3.1.2 Digital Signal Processing

Although sounds are continuous-time analog signals, it is more convenient to use discrete-time digital representations. Also, we wish to work with a compact speech feature representation.

Digital microphones *sample* measurements of air pressure (or rather, the electrical voltages induced by a vibrating diaphragm) at a rate between 8000 and 44,100 samples per second (Hz). To avoid issues of misrepresenting continuous signals in discrete time, an *anti-aliasing* filter is used to block all frequency components above the *Nyquist frequency*, or half the sampling rate. In effect, an 8 kHz sample rate approximates telephone-quality sound whereas a 44.1 kHz sample rate is considered high-fidelity sound; because most speech information is conveyed in the lower frequency range, a typical sample rate for ASR is 8-20 kHz. In addition, the analog-to-digital amplitude conversion results in values that are typically *quantized* as 8, 12, or 16-bit integers; to avoid problems such as quantization noise or clipping when digitizing speech, care must be taken to correctly calibrate the microphone volume sensitivity.

For ASR, we prefer to use more manageable feature representations rather than directly accessing a digitized speech signal. In the time-domain, features are represented as over-lapping *frames*, which are usually derived from 25 millisecond windows of the speech signal centered at 10 millisecond intervals. The basic computational operation is the Fast Fourier Transform, which is used to find the frequency spectrum for each frame. Some examples of these short-term spectral-based features:

- **Formant features:** In this project, a formant-tracking tool (Figure 10) estimates formant frequencies from spectral peaks. These two-dimensional features comprise an estimate of F_1 and F_2 for each frame.
- **MFCC features:** *Mel-Frequency Cepstral Coefficients*: these are standard ASR features. To account for the non-linearities of the human auditory system's frequency response, a filter-bank of 24 *mel*-spaced

¹⁷By tradition, vowels are qualitatively described in terms of placement on front-back, low-high axes. These labels were intended to correlate with tongue and jaw articulator positions, although in fact they better describe relative auditory qualities.

filters is used to derive *cepstral* coefficients. These are then reduced to 12 components by a process of orthogonalization, and a thirteenth feature is often appended for the signal's log-energy.

- **Delta features** Speech feature vectors are often appended with a set of Δ (first derivative) and $\Delta\Delta$ (second derivative) features which are useful for capturing some local temporal dynamics.

3.1.3 Optional exercises

Instructions: These are optional exercises, to help understand the feature representation used in this project.

Exercise 0. Download and install a speech analysis tool, such as Praat¹⁸ or Snack¹⁹. Snack is installed on some of the newer machines in 275 Soda Hall, but you'll need headphones and a microphone.

Exercise 1. Generate spectrograms of speech you record, track formant estimates, and play with a formant-based vowel synthesizer. In Snack, these are the demos in `spectrogram.py`, `vowelspace.tcl`, and `formant.tcl`. In Praat, these features are a bit tricky to find, but they're there...

Update: Improved demos are available from the project webpage, along with Windows stand-alone executables. See `spectrogram.(tcl|exe)`, `formant-track.(tcl|exe)`, and `formant-synth.(tcl|exe)`.

Exercise 2. Formants characterize steady-state vowels (monophthongs), while formant transitions characterize diphthongs and give evidence for some classes of consonant sounds. For some aperiodic signals – sounds such as fricatives or silence – there are no formants, *per se*. Nonetheless, the formant tracking utility will still provide estimates. Can you see any regularity in the spurious “formants” that are estimated for such sounds, and notice how these are distinguishable distributions that can be exploited for ASR?

¹⁸<http://praat.org>

¹⁹<http://speech.kth.se/snack>

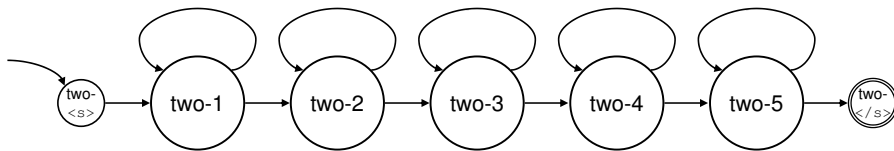


Figure 11: A typical concatenative HMM unit (5-state whole-word model)

3.2 Acoustic models

Hidden Markov Models can be used to model a speech process, using *concatenative* modeling units. There are two distinct phases in ASR system development: in training, parameters for the model units are learned from possibly unlabeled data; in testing, the model units are composed into a recognition network from which the most likely word sequence can be retrieved.

3.2.1 Concatenative HMM units

The HMM formulations that we used in the preceding sections are unfortunately not always easy to use for some sorts of real-world applications in which the model can become very large and complex. For sequential modeling tasks, such as ASR, we will find it useful to work with *concatenative* models. In this framework, elementary *units* can be assembled into very large *networks*, and all across all levels of combination and composition there is an underlying abstraction: everything is an HMM.

A concatenative HMM has two significant properties: it has a specified name used to designate its start and stop states, which are sometimes referred to as *entry* and *exit* states; also, there can be additional non-emitting null states other than the entry and exit states. Indeed, this is not very different from the previous formulation, and in code our concatenative HMM will be implemented as a subclass of regular HMMs. It is convenient to view the structure of a concatenative unit as in Figure 11. Note that non-emitting null states are drawn smaller than observation-emitting states, and that the entry and exit states are uniquely identified by the model’s name.

The basic operation on concatenative HMMs is – of course – concatenation. Given a set of models and a sequence of units, concatenation simply strings together the units in order by adding a transition from the exit state of one model to the entry state of the next, resulting in a serially concatenated HMM. It is also possible to join a set of model units in parallel, such that all the exit states at one step will transition to all the entry states at the following step. To reduce the number of transitions that need to be created, it is often helpful to insert null states as “glue” between concatenated units. In Figure 12 for example, the 20 transitions through `glue1` provide an economical alternative to the 100 transitions from 10 exit states to 10 entry states. Note also that the concatenation must rename states at each step with unique subscripts to indicate which copy of an HMM unit is being referenced.

The introduction of null states unfortunately complicates the Viterbi algorithm on concatenative HMMs. Because the null states do not emit observations, we must re-interpret the Viterbi messages to be maximizations over sequences of $i + k$ states, of which exactly k are non-emitting null states:

$$m[i][s_{i+k}] = \max_{s_1, \dots, s_{i+k}} P(o_1, \dots, o_i, s_1, \dots, s_{i+k}) \quad (33)$$

A recursive relation is defined:

$$m[i][s_{i+k}] = \max_{s'} m[i-1][s'] \cdot \tau(s', s_{i+k}) \cdot e(o_i | s_{i+j}) \quad (34)$$

where $\tau(s', s_{i+k})$ is the maximal probability of a path from s' to s_{i+k} that emits a single observation at state s_{i+j} for $j \leq k$.²⁰ The back-pointer messages store the paths corresponding to $\tau(s', s_{i+k})$.

²⁰This is extremely tricky. See `ConcatenativeHMM.py` for an implementation based on topological state ordering.

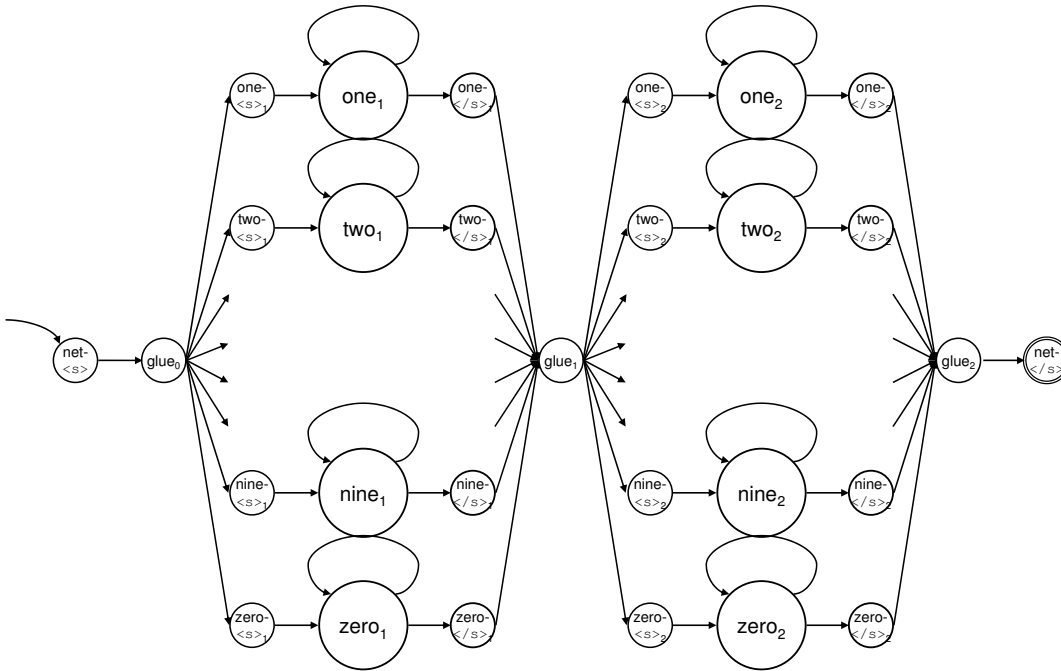


Figure 12: A concatenated HMM network (for 2-digit sequences, with single-state whole-word models)

3.2.2 Speech modeling units

The concatenative HMM units used to model speech are typically of the kind in Figure 11, a left-to-right structure in which each non-null state either transitions to itself or to one successive state – this is generally called a *Bakis* topology. Two common kinds of these speech modeling units are *whole-word* and *monophone* models, which we will explore in this section.²¹

The most simple unit would be a single-state HMM for each word in the task vocabulary, as in Figure 12. Note that there are only two non-trivial distributions to be learned for each of these word models: a transition probability conditioned on the single state, either a self-loop to itself or a transition to the stop state; and an emission probability of observations conditioned on the single state. These are essentially a mixture model for each word and capture no sequential information other than some limited notion of word duration. A better solution might be to model each word with more than a single state; Figure 11 depicts a word model with five states. In addition to capturing some elements of the temporal structure within a word, a multi-state HMM unit also more accurately describes the word duration.²²

To train these models, we can begin by initializing the HMM parameters uniformly or randomly. For each training utterance, we have acoustic observations and their word-level transcriptions – but no state sequences. If we concatenate the HMMs for the model units that correspond to the word sequence, we have a model for the entire utterance and can use the Viterbi algorithm to perform *forced-alignment*, finding the most likely state sequence that might have generated the observations. This state sequence can then be used to train the maximum-likelihood estimates of the model parameters with the iterative Viterbi training procedure (“hard EM”).²³ But in practice, most ASR systems use Baum-Welch re-estimation (“soft EM”).

²¹For large-vocabulary ASR, typical units are *tied context-dependent triphones*, which are derived from monophones.

²²A single-state HMM can only describe durations that have a geometric distribution, whereas the multi-state HMM can model durations that have a more realistic negative binomial distribution.

²³This is not the standard use of Viterbi forced-alignment. Rather, the forced-alignment is usually used to determine the

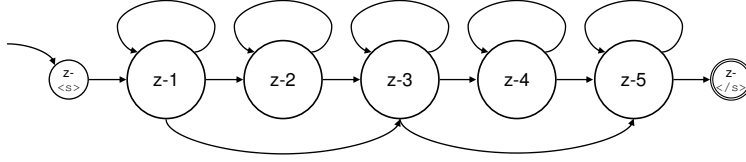


Figure 13: A 5-state left-to-right phone model with skip transitions

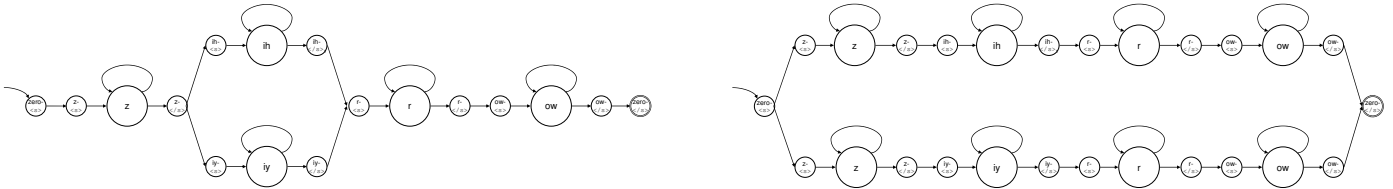


Figure 14: Equivalent word models formed by concatenating pronunciations. (Single-state phone models)

To initialize the multi-state whole-word HMM units, we could copy parameters from previously trained single-state whole-word models; in particular, we might copy the emission models rather than starting from uniformly or randomly initialized parameters. This common technique of initializing new models from previously trained ones is called *bootstrapping*. These new bootstrapped multi-state HMM units can then be refined with a few iterations of the Viterbi training procedure. A caveat when training a new set of models from initial random, uniform, or bootstrapped models: the initial Viterbi alignment can sometimes produce very unintuitive alignments when breaking ties, so it sometimes makes sense to manually align the utterances in the first iteration by evenly spacing out the state transitions among all emitting states.

Whole-word model units work particularly well for small-vocabulary tasks, such as digit recognition. For larger tasks, however, there are too many words to reliably train a model for each. In this situation, sub-word phonetic units can provide a system for modeling an arbitrary vocabulary of words. A small number of units evidenced across many words, these are intended to serve as a phonemic representation of the sounds of a language. One of the major drawbacks to using phonetic units, however, is the requirement of a handmade pronunciation dictionary which specifies possible model sequences for every word in the vocabulary.

A standard monophone unit has a three-state Bakis topology, a design choice that was initially inspired by the sub-phonetic structure of some speech sounds, like the distinct articulatory phases involved in the production of a plosive. A five-state topology is also common and will often model duration more effectively, possibly capturing more sub-phonetic structure; but it also enforces a greater minimum duration, since every phone must be at least 50 milliseconds long. This is sometimes problematic for fast speakers, so a system architect might wish to add *skip transitions* to a five-state unit, as in Figure 13.

The pronunciation dictionary specifies one or more sequences of phones for each word in the vocabulary, which are concatenated to form a word model. Figure 14 displays two possible concatenated word models for the word *zero*, given a pronunciation dictionary which specifies two variants for the first vowel. Note that the glue states and many auxiliary subscripts formed during the concatenation process have been omitted from the figures, although in the implementation the structure is far more complicated.

appropriate model sequence for a given utterance, choosing among various word pronunciations or detecting untranscribed silences and noises. Forced-alignment is also used to reject training utterances of low likelihood, indicating a possible mis-transcription, inappropriate segmentation, or otherwise poor data quality.

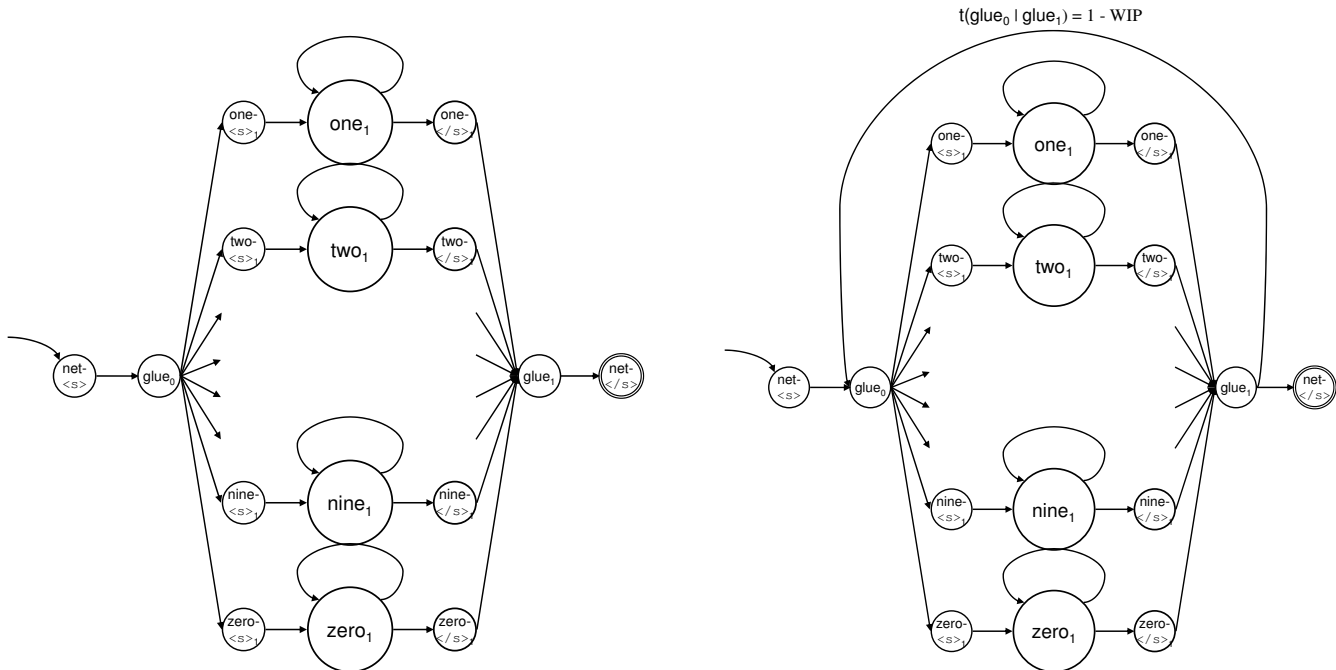


Figure 15: Isolated-digit (left) and digit-loop (right) recognition networks. (Single-state whole-word models)

3.2.3 Recognition networks

After training a set of HMM units, either whole-word models or phone models that can be concatenated into word sequences using a pronunciation dictionary, the ultimate task of an ASR system is to perform recognition of test utterances. In essence, this is the computation of Eq. 32. In order to be able to represent the acoustic likelihood $P(\mathbf{O}|\mathbf{W})$, we must first construct a recognition network by assembling the trained speech units into a larger HMM.

The simplest network to build is an *isolated*-word recognizer. For the case of digit recognition with a vocabulary of ten words, the recognizable word sequences are the set of exactly ten one-word sentences. Figure 15 displays such a recognition network, in which the only allowable paths will pass through only one word model. If we assume that all digits are equally likely – that is, $P(\mathbf{W})$ is irrelevant – then the decoding problem is to find the word sequence which is most likely given the observations. In this instance, we could run the forward algorithm and compare the messages $m[n][\text{word-}</s>]$ for each word in the fairly small vocabulary. In practice, however, we typically use the *Viterbi approximation* and assume that the most likely state sequence passes through the word sequence which has the greatest total likelihood.

For the more challenging task of *connected*-digit recognition, the recognition network is altered by adding a transition from the exit states of each word model back to the entry states; more precisely, this is done by adding a single transition from glue_1 back to glue_0 . This kind of network is called a *word-loop* recognizer, for which we again decode using the Viterbi approximation. A problem with these connected-word networks is that the recognized word sequences are generally too long,²⁴ due mostly to the poor duration modeling of Bakis HMM units. One very common hack to address this problem is to introduce a *word insertion penalty*, lowering the probability of the transition from glue_1 back to glue_0 . Another solution, if we knew the exact length of word sequences, would be to set up a fixed-length network as in Figure 12.

The concerned reader may wonder why we have not yet specified the probabilities of transitions introduced in model concatenations. These transitions always have probability one by default – but this results in a

²⁴In the standard ASR scoring terminology, these are insertion errors.

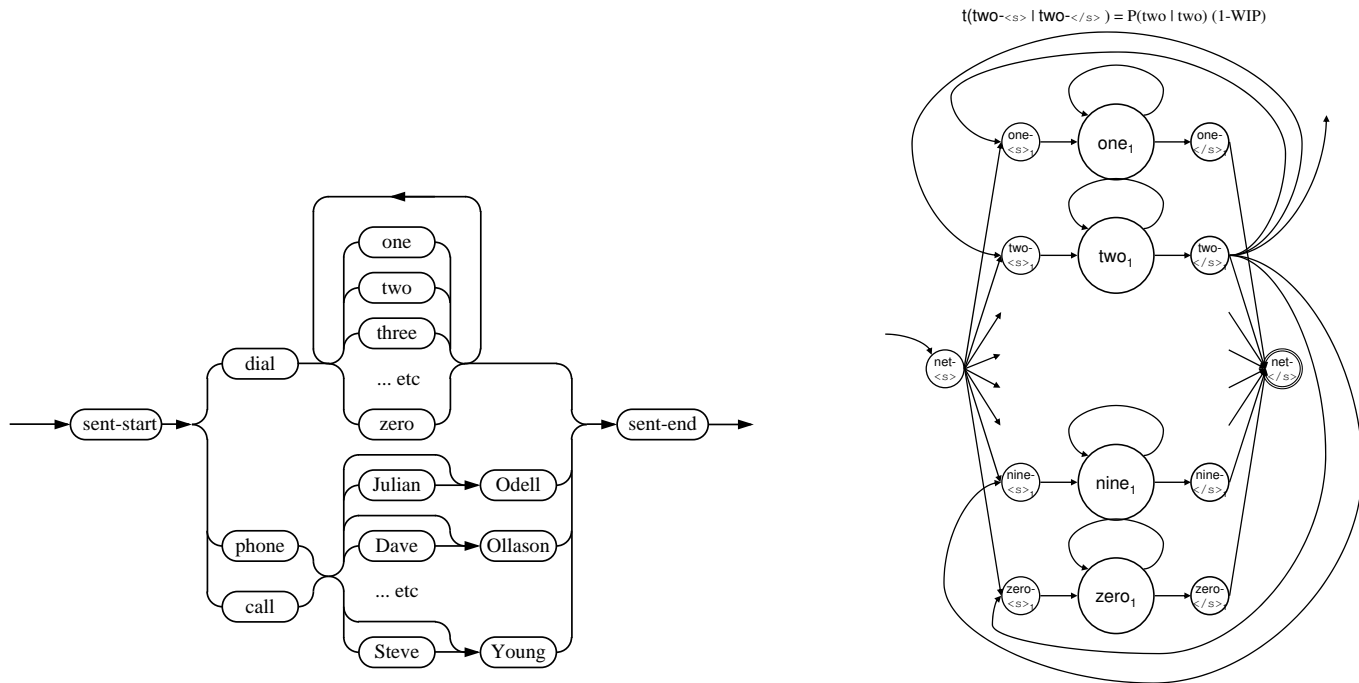


Figure 16: Left: A word network for a voice-dialing grammar. (From The HTK Book.) Right: Incorporating a bigram language model into a connected-digit recognizer. (Not all transitions are shown.)

concatenated HMM whose transition model is not properly normalized! Why not set the new transitions according to a probability distribution? For instance, perhaps the transitions splitting the two paths for pronunciation variants of “zero” in Figure 14 should each have probability 0.5 rather than 1.0. The crucial difference here is to consider whether we are going to do Viterbi decoding to find the most likely state sequence or a more thorough decoding technique which finds the word sequence of highest total likelihood. In the case of Viterbi decoding, pronunciation variants “compete” against each other, despite belonging to the same word sequence; we do not want to “handicap” paths through a word sequence which is modeled by multiple pronunciations, and so we set all concatenated transitions to 1.0. In the case of full-likelihood decoding, however, the pronunciation variants each contribute to the likelihood of the same word sequence; to avoid “favoring” words that are modeled by multiple pronunciations, we must therefore set the concatenated transitions according to some distribution – this could even be learned during acoustic model training.

Thus far, we have assumed that the language model $P(\mathbf{W})$ is uniform; while this is often a fair assumption for digit-recognition tasks, most other ASR tasks involve some more interesting word structure. Sometimes this is encoded by the ASR developer as a *task grammar*, a language specifying the allowable word sequences, sometimes probabilistically weighted. These grammars can be converted into word networks, as in Figure 16 (left), and replacing the words with HMM acoustic models results in a recognition network. Another common alternative for language modeling is to use n -gram models over word sequences, essentially Markov Chains of order $n - 1$. A bigram model provides the probability of each word conditioned on a preceding word. The word-loop recognition network can be modified to incorporate the word bigram language model by adding transitions from the exit states back to the entry states, as in Figure 16 (right). Note that glue states are bypassed in this network, and there are very many transitions which are omitted from the figure.

Much of the research effort in ASR goes into developing decoding strategies which are fast, making appropriate approximations, and incorporating as much of the language model as possible. The Viterbi decoder has long been a favorite because of its simplicity, effectiveness, and in particular the fact that the forward direction of its computation enables *online* recognition – it can start decoding the beginning of

an utterance before it has finished recording the end. Because decoders are generally unable to tractably incorporate more than trigram language models, a common technique is to use a fast decoder to generate N -best hypothesis lattices which can later be rescored by a higher-order language model.

3.2.4 Exercises

Instructions: These exercises might involve very minimal coding, but will require you to use some provided tools to build and test ASR systems. If you write any code that you wish to submit, please place it in `asr_exercises.py` or at least make it accessible from there. These exercises are fairly open-ended, and you should submit a brief writeup of your experiments either electronically (submit `asr_exercises.[txt|pdf]`) or in hardcopy (homework drop-box in 283 Soda Hall).

Exercise 0. Look through the code in `ConcatenativeHMM.py`. This includes the class for creating concatenative HMMs, as well as a version of the Viterbi algorithm.²⁵ The code in `asr.py` will probably make more sense if you read it in the context of `asr_exercises.py`. If you want to get a feel for the data you'll be working with, explore the `audio-data` directory and play the `.wav` sound files. If you are curious about how the features in `feature-data` were generated, you may want to examine `features.py`.

Exercise 1. The functions `wholeWordExpt` and `phoneticExpt` in `asr_exercises.py` provide a framework in which you can train and test various kinds of acoustic models. The arguments you can supply are:

- `numStates`: number of states per whole-word or monophone unit.
- `mode`: specify the task. Either `'connected'`-digit or `'isolated'`-digit recognition.
- `trainIters`: how many Viterbi training iterations to perform.
- `trainExamples`: how many examples to train on.
- `testExamples`: how many examples to test on.
- `resolution`: affects the discretization of features.
- `wip`: set the word insertion penalty.

Run a variety of experiments and note any interesting observations in your writeup.

Exercise 2. Train monophone acoustic models and examine the emission probabilities of various states. For a state in a vowel model, locate the mode of the joint distribution over formant values – or perhaps a mean – and compare the F_1 and F_2 values against the linguistic expectation (see Figure 9). From the Snack Sound Toolkit (available in 275 Soda), use the `formant.tcl` demo to synthesize a vowel sound with the formant values from your trained models. How does it sound?

Exercise 3. What kind of network would you construct to recognize U.S. telephone area codes (spoken 3-digit sequences)? Justify any design choices you make, and provide a graphical depiction of the network structure; for clarity, you may omit all null states, subscripts, and numerical parameters. If instead of recognizing the digit sequence that was spoken you were supposed to recognize the U.S. state to which that area code belonged, how might your system differ? For real data, see www.thedirectory.org/pref.

²⁵The Viterbi algorithm for HMMs with null states is tricky. It probably won't help you with the earlier exercises.