# Automata for Language Processing

Language is inherently a sequential phenomena. Words occur in sequence over time, and the words that appeared so far constrain the interpretation of words that follow. In the last lecture we explored probabilistic models and saw some simple models of stochastic processes used to model simple linguistic phenomena. In this chapter we review basic concepts of finite state automata (FSA), and develop probabilistic FSA's that are useful in natural language applications.

Sequential information is critical in a natural language like English. Only certain sequences of words are considered to be "grammatical" in a language. For instance, we all can agree that

I can can the peaches in a can. (1)

is a sentence of English, while

Can I peach a can the in can. (2)

is not, even though it consists of the same words. How do we know this? It isn't because we have heard all the possible sentences before and remember them. Its unlikely any of you have heard or read sentence (1) before. We know that it is English because of general rules and constraints that we know about language. One large part of this knowledge concerns how words can be ordered in meaningful sentences. This chapter introduces a range of mechanisms that can be used to capture this information so it can be used for various tasks.
□

# 1. Finite State Machines

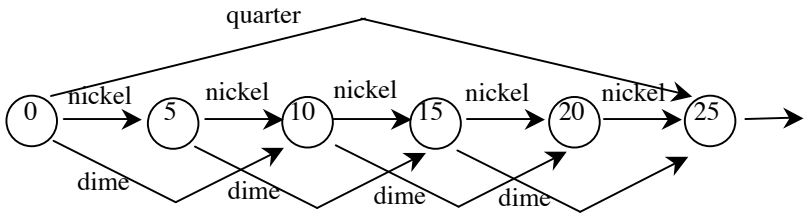



Figure 1: A Finite State Machine for a gum machine

One of the simplest models of sequential processes is the **finite state machine** (FSM). FSM consists of a set of **states**, of which there is a special state called the starting state, and at least one state called an end state, and a set of connections called **transitions** that allow movement between states. Depending on the particular style of FSM used, either the transitions or states may have an associated **output** value. An FSM define a set of sequences that it is said to **accept** by the following methods: a sequence is accepted by an FSM if we can start at the start state, and find a path through the FSM where the outputs on the transitions generate the sequence and ends in an end state. For example, figure 1 shows an FSM that models the behavior of a gumball machine, where you have to put in exactly 25 cents to get a gumball (if you put in too much, you lose your money!). It accepts any sequence of coins that adds up to 25 cents, such as the sequence consisting of five nickels, or a dime, a nickel, then a dime, or even the simple sequence of one quarter. The start state is the state that has an incoming arrow that comes from nowhere (in this case the state labeled 0, and the end states are indicated with an arrow that leads nowhere (in this case its only the state labeled 25). The set of sequences that an FSM (or any other formal automata) accepts is called its

**Language.**

Thus this FSM can output “nickel dime dime”, by moving from the start state 0, through 5 and 15, to the end state 25. This same FSM could not output “dime dime dime” as we would traverse states 0, 10, 20 and then not be able to move forward over an arc labeled “dime” from state 20. A real gumball machine, of course, would either accept the coin and not give any change, or move to a state that gives 5 cents change.

Another way to write FSMs is to put the outputs on the states. In this case, we need to find a path through the states that have the right outputs. Figure 2 shows an FSM for the same gumball machine.

In this case, the input “nickel dime dime” could be generated by the path 5.1, 15.2 and 25.3, which is an end state. The input “dime dime dime”, on the other hand, could not be generated. We could move through start state 10.2, to state 20.2, but then there is not succeeding state reachable from state 20.2 that outputs “dime”.

These two notations are formally equivalent in the sense for every FSM defined in one form, the language it accepts is accepted by another FSM in the other form. In other words, they can express the same class of languages. This does not mean, however, that the equivalent FSMs are equivalent in other ways. The FSM

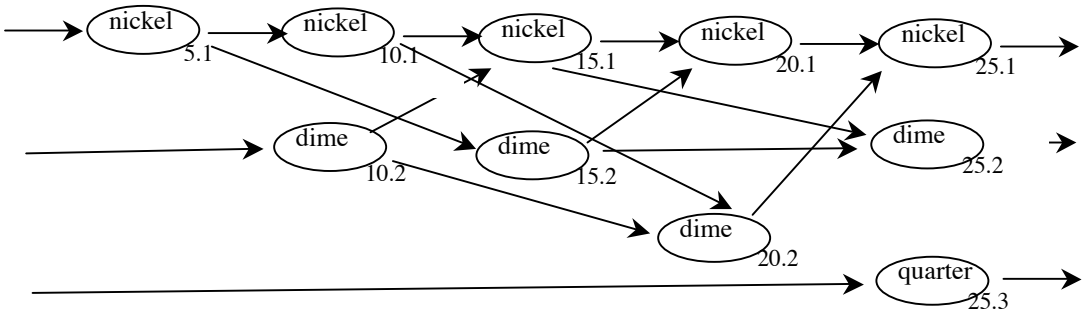



Figure 2: Another Finite State Machine for the gum machine

in figure 2, for instance, requires 4 more states to express the same language. In more complex languages, the differences can be dramatic. In addition, the first formulation seems more intuitive in this case, as the states correspond to how money has been put in so far.

In natural language applications, we often talk of an FSM accepting a sentence in the language, whereas in the discussion above we’ve talked about machines generating output. Formally, a machine accepts a sentence S if it can generate sentence S. FSMs have turned out to be very useful in many aspects of language processing. Figure 3 shows a simple FSM that accepts a set of noun phrases including

The one very old cow
The one old cow
The one old cow

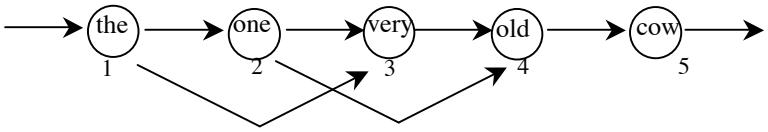



Figure 3: A Finite State Machine for simple cow phrases

## Deterministic and Non deterministic FSMs

The FSMs we developed so far have a special property given a particular sequence, to check whether it is accepted, we can tell which transition to take at every stage. For instance, from state 2 in the FSM in figure three, given the next word in the sequence is *very*, we know to follow the transition to state 4. When an FSM has this property from any state and any input sequence, it is called a **deterministic finite state machine (DFSM)**. DFSM are computationally very attractive because the algorithm to check whether an input is accepted or not is very efficient. For instance, we know that the sequence *The cow* is not accepted,

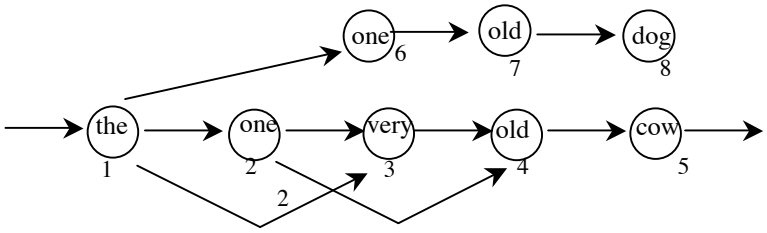



Figure 4: A Non Deterministic Finite State Machine

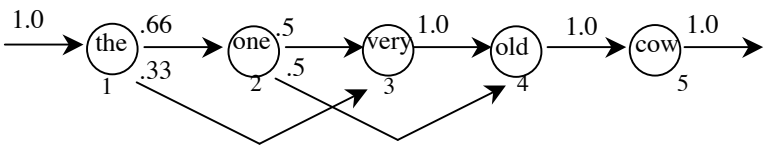
for we can pass from states 1 to state 2 and output *the,* but then state 2 has no transition leaving it with output *cow*. So the sequence is not valid. Not all FSMs are deterministic. For instance, say want to modify the noun phrase FSM so that is also accepts *the one old dog*, but not *the one very old dog*. We might do this as shown in Figure 4. Now, when trying to decide whether to accept a sequence beginning with the word *the one*, there are two choices of transitions from state 1. We cannot know which is the right choice until we see the last word of the sequence. As a result, algorithms to check whether a sequence is accepted by an NFSM are mode complex and computationally expensive. We will consider such algorithms later.

Note that it is possible to convert any NFSM into a deterministic FSM that accepts the same language. But the DFSM may be considerably larger (potentially exponentially larger–an NFSM with N states may require a DFSM with $2^N$ states).
□

# 2. Weighted Finite State Machines

In many applications, we are not so interested in whether a sequence is a valid sentence in a language as much as computing the probability of a sequence. For instance, speech recognition engines typically use a model of word transition probabilities (e.g., how often a word w follows a word w') to drive the recognition process. Adding such a model on top of straight acoustic recognition can double the word accuracy rate. We can model this data with a variant of an FSM called a **weighted finite state machine (WFSM)**. A WFSM is just like an FSM expect that it has numbers on the transitions.

For example, say we interested in language about old cows, and have the FSM shown in figure 5, which is



simply the figure 3 FSM with weights assigned to the transitions. If the sum of the weights from each arc add up to 1, and are never negative, then we have a **probabilistic finite state machine (PFSM).**

A deterministic probabilistic finite state machine, as this one is, is often called a **Markov Chain**. It has the nice properties of DFSM in that we can very quickly determine whether a sequence is accepted by the machine. In addition, we can compute the probability of a sequence by simply multiplying the probabilities on each of the transitions together. For instance,

P(The one old cow) = P(transition 1 to 2) * P(transition 2 to 3) * P(transition 3 to 5)
= .66 * .5 * 1.0 = .33

or one third.

# 3. Hidden Markov Models

Non deterministic weighted FSMs are of great interest in language processing and are used for many applications. We develop a new form of machine that allows us to express many problems more concisely than the methods we've used so far. In particular, instead of associated a single output to a state, we allow a state to define a probability distribution over outputs. In essence, each state represents a conditional probability distribution over the outputs:

P(output = o | state = s).

While this doesn't necessarily result in a non deterministic machine, in practice it almost always does. The name "Hidden Markov Model" arises from this non determinism - given a sequence, it is not clear what sequence of states produced it because we do not know what output each state had.

Let's consider a simple example. Say we are modeling a simple language in which sentences consist of two words, either a noun followed by a verb such as *eat dinner* or a verb followed by a noun such as *dogs eat*. We have a number of nouns and verbs in the language, and want to build a model that requires we pick one verb and one noun in each sentence. In other words, we'd like the probability of *dogs dinner* or *eat eat* to be

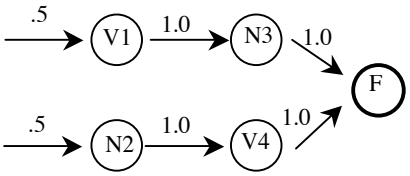



Figure 7: A Hidden Markov Model: Transitions

0. We can do this by defining the following HMM, where the nouns in the language are *dogs, dinner, flies* and *time* and the verbs are *eat, watch, time* and *flies*. We could start by defining a model with four states as shown in Figure 7 - one for an initial verb (V1), another for an initial noun (N2), and then two more for the second word as a noun (N3) and a verb (V4). The transition probabilities are set so that we an equal probability of each form of sentence. For convenience, we introduce a new special state F that is the "final state" and has no output.

So far, this looks just like our weighted FSMs except that we've moved the state label onto the state rather than placing it below. These states are now used in the output probability distributions. For instance, for V1 we have the conditional probabilities:

P(eat | V1) = .4
P(watch | V1) = .3
P(time | V1) = .3

whereas the output probability distribution for N2 is captured by

P(dogs | N2) = .25
P(flies | N2) = .35
P(time | N2) = .40

We could summarize the complete set of probability distributions in a table form as follows, where each square represents one of the conditional probabilities.
□

| P(word \| state) | eat | watch | time | dogs | flies | dinner |
|---|---|---|---|---|---|---|
| V1 | .4 | .4 | .2 | 0 | 0 | 0 |
| N2 | 0 | 0 | .4 | .25 | .35 | 0 |
| N3 | 0 | 0 | .11 | .22 | .33 | .34 |
| V4 | .25 | .25 | .25 | 0 | .25 | 0 |

## What can we do with HMMs?

HMMs are very useful for a wide range of tasks. Because of the non deterministic nature of HMMs, however, we also face some special problems.

### Determining the probability of a sequence

Because HMMs are non deterministic HMMs, computing the probability for a sequence can be difficult. This is not because it might be hard to find a path that would accept it, but because there may be many paths that could accept it. For instance, the sentence *time flies* can be generated from two different sequences

V1 N3□(i.e., a command to record the times as some flies do something)
N2 V3□(i.e., the sentence is a statement about how quickly time passes)

We can compute the probability of each path by multiplying the probabilities of the transitions together with the output probabilities from each state. For example

P(times flies generated from V1 N3) =

P(V1 is the start state) *

P(time | V1) *

P(transition V1 to N3) *

P(flies | N3) *

P(N3 is the end state, i.e., transition from N3 to F)

⇒ .5 * .2 *1.0 *.33 * 1.0 = .033

Likewise,

P(times flies generated from N2 V4) = .5 * .4 * 1.0 * .25 * 1.0 = .05

So if we want to know the probability of the output *time flies* (no matter which reading was intended), then we add up the probabilities over all the possible paths, and we would get .083 in this case.

When dealing with HMMs, we are often interested in how likely we are to be in a particular state and a certain output is observed. This is called the **forward probability.** Given an output sequence $o_1$ .. $o_t$, and a state S, the forward probability is

P(in state S at time t, output at time i was $o_i$, for i = 1,t).

Note that the informal notion of the probability that an HMM outputs a sequence O is simply the forward probability of the HMM outputting sequence O and then being in the final state F.

### Determining the maximum probability path and tagging

Alternately, we might have a sequence and want to know what single path maximizes its probability. Using the example of *time flies* again, we see that the path N2 V4 maximizes the probability of this sequence, and the answer is .05. This is a very common method for solving tagging problems. In fact, we can view this example as a part of speech tagging for a very simple language into four categories: initial verb (V1), initial noun (N2), final noun (N3) and final verb (V4). The sequence *time flies* is ambiguous, but we know the most likely interpretation corresponds to path N2 V4, i.e., *time* is an initial noun and *flies* is a final verb.

### Training a model from corpora

If we had a corpus of sentences and wanted to use it to compute the probabilities for the transitions and outputs, we would have a problem. A sentence such as *time flies* is accepted via two different paths, so how would we count such a sentence when estimated the HMM probabilities? We will explore how to do this discuss later because it requires introducing some additional concepts first. □

# 4. More Powerful Formalisms

## Recursive Transition Networks

One way to extend finite-state machines to handle more general phenomena is to allow recursion in the networks. The results is a new class of machines called **recursive transition networks**. A recursive transition network looks like an FSM except we allow a transition to identify a new network instead of just a single output. To follow this transition, we must start and successfully complete the other network, at which point to return to the original transition. In essence, we have added the equivalent of procedure calls to the formalism, and with recursive calls we can encoding counting. For example, the RTN in figure 2 defines the language $a^n b^n$, i.e., all sequences consisting of n "a"'s followed by n "b"'s. It first outputs an *a*, and then either outputs a b and terminates, or recursively calls itself to generate $a^{n-1}$ $b^{n-1}$ after which it outputs the final *b*.

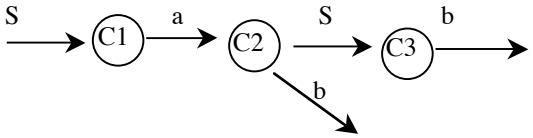


Figure 8: A Recursive Transition Network recognizing $a^n b^n$

To see how this RTN accepts the sequence a a b b, we find the following path: Node C1 -> C2 (accounts for the first "a") then from C2 we recursively traverse the network again, from C1 -> C2 (accounts for second "a") and take the arc labeled b (accounts for first "b). This network is now complete, so we return back and complete the transition from C2->C3. From C3 we follow the arc labeled "b" to account for the final "b". Note that there is no way a sequence such as a a b b b could be accepted as each level of recursion accounts for exactly one a and one b. Likewise, although it is difficult to prove, there is no finite state machine that could accept the language $a^n b^n$ It is easy to accept a superset of this language, say all strings consisting of a number of a's followed by some number of b's, but no FSM can enforce the constraint that the number of each letter is the same.

## Context-free Grammars

Context-free grammars are another formalism, expressively equivalent to recursive transition networks, yet more convenient for capturing natural language phenomena. They can be motivated by looking at the structure of a sentence such as

The birds in the field eat the corn

At the first level of analysis, this sentence has two major components, a subject phrase *The birds in the field* and a verb phrase *eat the corn*. Within the subject phrase, we can break it into an introductory specifier phrase *the*, following by the main (or "head") noun, followed by a modifier, which itself consists of a preposition and another noun phrase. Likewise, the verb phrase can be broken into a verb followed by a noun phrase. This analysis can be shown as a tree structure as shown in Figure 3. No matter how many modifiers are added to the subject phrase, it still remains "next" to the VP at the top level of the tree. Thus constraints like subject-verb agreement can be specified at this level and reach over arbitrary numbers of actual words.

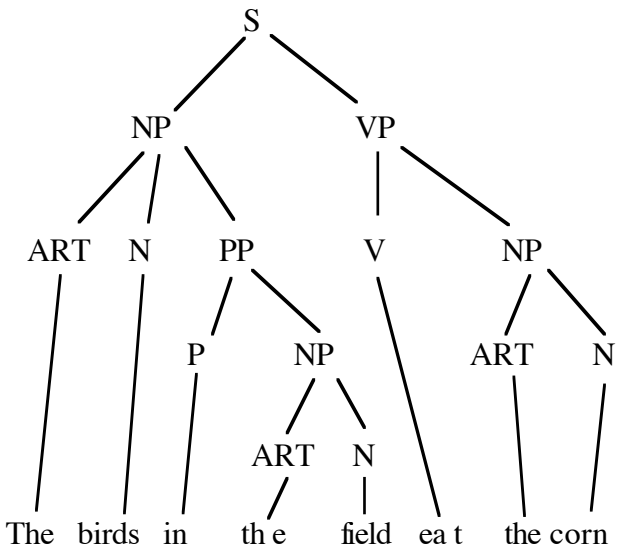



Figure 3: A tree representation of sentence structure

A context-free grammar is a set of rules that specifies what the allowable trees are. This space of trees defines a language in a similar way to how a finite-state machine or other automaton does. A context-free grammar that sanctions the tree in Figure 3 must contain the following rules:

S -> NP VP

NP -> ART N

NP -> ART N PP

PP -> P NP

VP -> V NP.

We will explore the use of context-free grammars later in the course.

□
□

□