# Chapter 7
# Probabilistic State Estimation

Consider trying to drive through an unfamiliar city. Even if we are able to plan a route from our starting location to our destination, navigation can fail on two counts: sometimes we don't know where we are on a map, and sometimes, due to traffic or road work or bad driving, we fail to execute a turn we had intended to take.

In such situations, we have some information about where we are: we can make observations of our local surroundings, which give us useful information; and we know what actions we have taken and the consequences those are likely to have on our location. So, the question is: how can we take information from a sequence of actions and local observations and integrate it into some sort of estimate of where we are? What form should that estimate take?

We'll consider a probabilistic approach to answering this question. We'll assume that, as you navigate, you maintain a *belief state* which contains your best information about what state you're in, which is represented as a probability distribution over all possible states. So, it might say that you're sure you're somewhere in Boston, and you're pretty sure it's Storrow drive, but you don't know whether you're past the Mass Ave bridge or not (of course, it will specify this all much more precisely).

We'll start by defining probability distributions on simple and complex spaces, and develop a set of methods for defining and manipulating them. Then, we'll formulate problems like the navigation problem discussed above as *state estimation* problems in *stochastic state machines*. Finally, we'll develop an algorithm for estimating the unobservable state of system, based on observations of its outputs.

## 7.1 State spaces

We have been using state machines to model systems and how they change over time. The *state* of a system is a description of the aspects of the system that allow us to predict its behavior over time. We have seen systems with finite and infinite state spaces:

- A basic enumeration of states, such as `('closed', 'closing', 'open', 'opening')` for an elevator controller, we will call an *atomic* finite state space. The states are atomic because they don't have any further detailed structure.
- A state space that is described using more than one variable, such as a counter for seconds that goes from 0 to 59 and a counter for minutes that goes from 0 to 59 can be described as having two *state variables*: seconds and minutes. We would say that this state space is *factored*: a state is actually described by a value of each of the two variables.

- A state space described by a single integer is a countably infinite atomic state space.
- A state space described by a real number is uncountably infinite, but still atomic.
- A state space described by more than one integer or real number (or a combination of continuous and discrete state variables) is a factored state space.

In this chapter, we will concentrate on factored, finite state spaces, and see how to represent and manipulate probability distributions on them.

## 7.2 Probability distributions on atomic state spaces

Probability theory is a calculus that allows us to assign numerical assessments of uncertainty to possible events, and then do calculations with them in a way that preserves their meaning. (A similar system that you might be more familiar with is algebra: you start with some facts that you know, and the axioms of algebra allow you to make certain manipulations of your equations that you know will preserve their truth).

The typical informal interpretation of probability statements is that they are long-term frequencies: to say "the probability that this coin will come up heads when flipped is 0.5" is to say that, in the long run, the proportion of flips that come up heads will be 0.5. This is known as the *frequentist interpretation* of probability. But then, what does it mean to say "there is a 0.7 probability that it will rain somewhere in Boston sometime on April 29, 2017"? How can we repeat that process a lot of times, when there will only be one April 29, 2017? Another way to interpret probabilities is that they are measures of a person's (or robot's) *degree of belief* in the statement. This is sometimes referred to as the *Bayesian interpretation*. In either interpretation, the formal calculus is the same.

So, studying and applying the axioms of probability will help us make true statements about long-run frequencies and make consistent statements about our beliefs by deriving sensible consequences from initial assumptions.

We will restrict our attention to discrete sample spaces[45], so we'll let $\mathcal{U}$ be the *universe* or sample space, which is a set of *atomic events*. An atomic event is just a state: an outcome or a way the world could be. It might be a die roll, or whether the robot is in a particular room, for example. Exactly one (no more, no less) event in the sample space is guaranteed to occur; we'll say that the atomic events are "mutually exclusive" (no two can happen at once) and "collectively exhaustive" (one of them is guaranteed to happen).

[45] In probability, it is typical to talk about a 'sample space' rather than a 'state space', but they both come to the same thing: a space of possible situations.

*Example 13.*

- The sample space for a coin flip might be {`H`, `T`}, standing for heads and tails.
- The sample space for a sequence of three coin flips might be {`HHH`, `HHT`, `HTH`, `HTT`, `THH`, `THT`, `TTH`, `TTT`}: all possible sequences of three heads or tails.
- The sample space for a robot navigating in a city might be the set of intersections in the city.
- The sample space for a randomly generated document might be all strings of fewer than 1000 words drawn from a particular dictionary.

An *event* is a subset of $U$; it will contain zero or more atomic events.

*Example 14.*

- An event in the three-coin-flip space might be that there are at least two heads: {HHH, HHT, HTH, THH}.
- An event in the robot navigation problem might be that the robot is within one mile of MIT.
- An event in the document domain might be that the document contains, in sequence, the words '6.01' and 'rules'.

A probability measure $Pr$ is a mapping from events to numbers that satisfy the following axioms:

$$Pr(U) = 1$$

$$Pr(\{\}) = 0$$

$$Pr(E_1 \cup E_2) = Pr(E_1) + Pr(E_2) - Pr(E_1 \cap E_2)$$

Or, in English:

- The probability that something will happen is 1.
- The probability that nothing will happen is 0.
- The probability that an atomic event in the set $E_1$ or an atomic event in the set $E_2$ will happen is the probability that an atomic event of $E_1$ will happen plus the probability that an atomic event of $E_2$ will happen, minus the probability that an atomic event that is in both $E_1$ and $E_2$ will happen (because those events effectively got counted twice in the sum of $Pr(E_1)$ and $Pr(E_2)$).

Armed with these axioms, we are prepared to do anything that can be done with discrete probability!

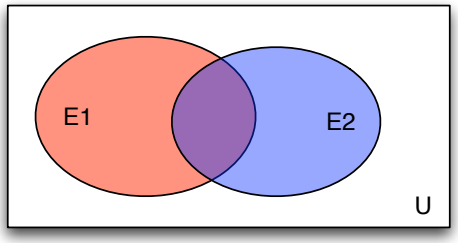
## Conditional probability

One of the most important operations in probabilistic reasoning is incorporating evidence into our models. So, we might wonder what the probability of an event is *given* or *conditioned on* some other relevant information we have received. For example, we might want to know the probability of getting a die roll greater than 3, if we already know that the die roll will be odd.

We will write conditional probabilities as $\Pr(E_1 \mid E_2)$, pronounced "the probability of $E_1$ given $E_2$", where $E_1$ and $E_2$ are events (subset of the atomic sample space). The formal definition of conditional probability is this:

$$\Pr(E_1 \mid E_2) = \frac{\Pr(E_1 \cap E_2)}{\Pr(E_2)} \quad .$$

In the figure below, $E_1$ is the red ellipse, $E_2$ is the blue ellipse, and $E_1 \cap E_2$ is the purple intersection. When we condition on $E_2$, we restrict our attention entirely to the blue ellipse, and ask what percentage of the blue ellipse is also in the red ellipse. This is the same as asking for the ratio of the purple intersection to the whole blue ellipse.




*Example 15.* What is the conditional probability of getting a die roll greater than 3, given that it will be odd? The probability of a die roll greater than 3 (in a fair six-sided die) is 1/2. But if we know the roll will be odd, then we have to consider ratio of the probabilities of two events: that the die roll will be odd and greater than three, and that the die roll will be odd. This is 1/6 divided by 1/2, which is 1/3. So, learning that the die roll will be odd decreases our belief that it will be greater than three.

## 7.3 Random variables

Just as some state spaces are naturally described in their factored form (it's easier to say that there are 10 coins, each of which can be heads or tails, than to enumerate all $2^{10}$ possible sequences), we often want to describe probability distributions over one or more variables. We will call a probability distribution that is described over one dimension of a state space a *random variable*.

A discrete random variable is a discrete set of values, $v_1 \ldots v_n$, and a mapping of those values to probabilities $p_1 \ldots p_n$ such that $p_i \in [0, 1]$ and $\sum_i p_i = 1$. So, for instance, the random variable associated with flipping a somewhat biased coin might be $\{heads : 0.6, tails : 0.4\}$. We will speak of random variables having distributions: so, for example, two flips of the same biased coin are actually two different random variables, but they have the same distribution.

In a world that is appropriately described with multiple random variables, the atomic event space is the *Cartesian product* of the value spaces of the variables. So, for example, consider two random variables, C for *cavity* and A for *toothache*. If they can each take on the values T or F (for *true* and *false*), then the universe is pairs of values, one for each variable:

$$C \times A = \{(T, T), (T, F), (F, T), (F, F)\} \quad .$$

We will systematically use the following notation when working with random variables:

- $A$: capital letters stand for random variables;
- $a$: small letters stand for possible values of random variables; so $a$ is an element of the domain (possible set of values) of random variable $A$;
- $A = a$: an equality statement with a random variable in it is an event: in this case, the event that random variable $A$ has value $a$

Some non-atomic events in a universe described by random variables $A$ and $C$ might be $C = c$ (which includes atomic events with all possible values of $A$), or $C = A$ (which includes all atomic events consisting of pairs of the same value).

## Joint distribution

The *joint distribution* of a set of random variables is a function from elements of the product space to probability values that sum to 1 over the whole space. So, we would say that $(C = c, A = a)$ (with a comma connecting the equalities), which is short for $(C = c \textit{ and } A = a)$ is an atomic event in the joint distribution of $C, A$.

In most of our examples, we'll consider joint distributions of two random variables, but all of the ideas extend to joint distributions over any finite number of variables: if there are $n$ random variables, then the domain of the joint distribution is all $n$-tuples of values, one drawn from the domain of each of the component random variables. We will write $\Pr(A, B, \ldots, N)$ to stand for an entire joint distribution over two or more variables.

## Conditional distribution

In **section 7.2** we gave the basic definition of conditional probabilities, in terms of events on atomic subspaces. Sometimes, it will be useful to define conditional probabilities directly. A *conditional probability distribution*, written $\Pr(A \mid B)$, where $A$ and $B$ are random variables (we can generalize this so that they are groups of random variables), is a function from values, $b$, of $B$, to probability distributions on $A$. We can think of it this way:

$$\Pr(A \mid B) = \lambda b.\, \Pr(A \mid B = b)$$

Here, we are using $\lambda$ in the same way that it is used in Python: to say that this is a function that takes a value $b$ as an argument, and returns a distribution over $A$.

*Example 16.* Conditional distributions are often used to model the efficacy of medical tests. Consider two random variables: $D$, which has value *disease* if someone has a disease and value *nodisease* otherwise; and $T$, which has value *positive* if the test comes out positive and value *negative* otherwise. We can characterize the efficacy of the test by specifying $\Pr(T \mid D)$, that is, a conditional distribution on the test results given whether a person has the disease. We might specify it as:

$$\Pr(T \mid D) = \begin{cases} \{\textit{positive} : 0.99, \textit{negative} : 0.01\} & \text{if } D = \textit{disease} \\ \{\textit{positive} : 0.001, \textit{negative} : 0.999\} & \text{if } D = \textit{nodisease} \end{cases}$$

### 7.3.1 Python representations of distributions

We can represent distributions in Python in a number of ways. We'll use a simple discrete distribution class, called `DDist`, which stores its entries in a dictionary, where the elements of the sample space are the keys and their probabilities are the values.

```
class DDist:
    def __init__(self, dictionary):
        self.d = dictionary
```

The primary method of the `DDist` class is `prob`, which takes as an argument an element of the domain of this distribution and returns the probability associated with it. If the element is not present in the dictionary, we return 0. This feature allows us to represent distributions over large sets efficiently, as long as they are sparse, in the sense of not having too many non-zero entries.

```
    def prob(self, elt):
        if elt in self.d:
            return self.d[elt]
        else:
            return 0
```

(The expression `elt in self.d` is a nicer way to say `self.d.has_key(elt)`. That is a call to a built-in method of the Python dictionary class, which returns `True` if the dictionary contains the key `elt` and `False` otherwise.)

It is useful to know the `support` of the distribution, which is a list of elements that have non-zero probability. Just in case there are some zero-probability elements stored explicitly in the dictionary, we filter to be sure they do not get returned.

```
    def support(self):
        return [k for k in self.d.keys() if self.prob(k) > 0]
```

If we want to use the probability distribution to make a simulation, or to do something like shuffle or deal cards, it's useful to be able to `draw` from it. This method returns an element from the sample space of the distribution, selected at random according to the specified distribution.

```
    def draw(self):
        r = random.random()
        sum = 0.0
        for val in self.support():
            sum += self.prob(val)
            if r < sum:
                return val
```

We can represent a joint distribution on two random variables simply as a `DDist` on pairs of their values. So, for example, this distribution

```
dist.DDist({(0, 0) : 0.5, (0, 1): 0.2, (1, 0): 0.1, (1, 1): 0.2})
```

can be seen as the joint distribution on two random variables, each of which can take on values 0 or 1. (Remember that expressions like `key1 : v1, key2 : v2` create a new dictionary).

Finally, we will represent conditional distributions as Python procedures, from values of the conditioning variable to distributions on the conditioned variable. This distribution

$$\Pr(\mathsf{T} \mid \mathsf{D}) = \begin{cases} \{\mathit{positive} : 0.99, \mathit{negative} : 0.01\} & \text{if } \mathsf{D} = \mathit{disease} \\ \{\mathit{positive} : 0.001, \mathit{negative} : 0.999\} & \text{if } \mathsf{D} = \mathit{nodisease} \end{cases}$$

would be represented in Python as

```
def TgivenD(D):
    if D == 'disease':
        return dist.DDist({'positive' : 0.99, 'negative' : 0.01})
    elif D == 'nodisease':
        return dist.DDist({'positive' : 0.001, 'negative' : 0.999})
    else:
        raise Exception, 'invalid value for D'
```

To find a value for $\Pr(\mathsf{T} = \mathit{negative} \mid \mathsf{D} = \mathit{disease})$, we would evaluate this Python expression:

```
>>> TgivenD('disease').prob('negative')
```

## 7.4 Operations on random variables

Now that we can talk about random variables, that is, distributions over their sets of values, we can follow the PCAP principle to define a systematic way of combining them to make new distributions. In this section, we will define important basic operations.

### 7.4.1 Constructing a joint distribution

A convenient way to construct a joint distribution is as a product of factors. We can specify a joint distribution on $C$ and $A$ by the product of a distribution $\Pr(A)$ and a conditional distribution $\Pr(C \mid A)$ by computing the individual elements of the joint, for every pair of values $a$ in the domain of $A$ and $c$ in the domain of $C$:

$$\Pr(C = c, A = a) = \Pr(C = c) \Pr(A = a \mid C = c)$$

It is also true that

$$\Pr(C = c, A = a) = \Pr(A = a) \Pr(C = c \mid A = a)$$

*Exercise 7.1.* Use the definition of conditional probability to verify that the above formulas are correct.

*Example 17.* In the domain where the random variable C stands for whether a person has a cavity and A for whether they have a toothache, we might know:

- The probability of a randomly chosen patient having a cavity:

$$\Pr(C) = \{T : 0.15, F : 0.85\}$$

- The conditional probability of someone having a toothache given that they have a cavity:

$$\Pr(A \mid C) = \begin{cases} \{T : 0.333, F : 0.667\} & \text{if } C = T \\ \{T : 0.0588, F : 0.9412\} & \text{if } C = F \end{cases}$$

Then we could construct the following table representing the joint distribution:

| | | C | |
|---|---|---|---|
| | | T | F |
| A | T | 0.05 | 0.05 |
| | F | 0.1 | 0.8 |

The numbers in the table make up the joint probability distribution. They are assignments of probability values to atomic events, which are complete specifications of the values of all of the random variables. For example, $\Pr(C = T, A = F) = 0.1$; that is, the probability of the atomic event that random variable C has value T and random variable A has value F is 0.1. Other events can be made up of the union of these primitive events, and specified by the assignments of values to only some of the variables. So, for instance, the event $A = T$ is really a set of primitive events: $\{(A = T, C = F), (A = T, C = T)\}$, which means that

$$\Pr(A = T) = \Pr(A = T, C = T) + \Pr(A = T, C = F) \ ,$$

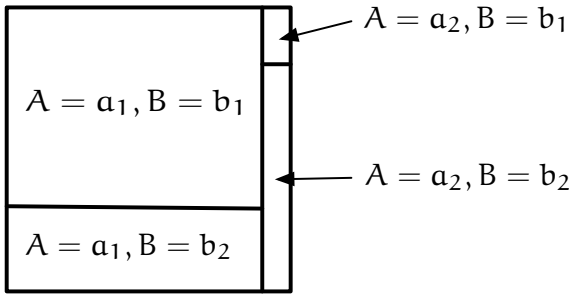
which is just the sum of the row in the table.

Here is another example of forming a joint distribution. Imagine that we are given the following distributions:

$$\Pr(A) = \{a_1 : 0.9, a_2 : 0.1\}$$

$$\Pr(B \mid A) = \begin{cases} \{b_1 : 0.7, b_2 : 0.3\} & \text{if } A = a_1 \\ \{b_1 : 0.2, b_2 : 0.8\} & \text{if } A = a_2 \end{cases}$$

You can visualize the joint distribution spatially, like this:




The sample space is divided vertically according to the distribution $\Pr(A)$, and then, for each value of $A$, it is divided horizontally according to the distribution $\Pr(B \mid A = a)$. This joint distribution is represented numerically by the table:

| | | $A$ | |
|---|---|---|---|
| | | $a_1$ | $a_2$ |
| $B$ | $b_1$ | 0.63 | 0.02 |
| | $b_2$ | 0.27 | 0.08 |

We can also think of this joint distribution as just another regular distribution on a larger state space:

$$\Pr(A, B) = \{(a_1, b_1) : 0.63, (a_1, b_2) : 0.27, (a_2, b1) : 0.02, (a_2, b_2) : 0.08\}$$

More generally, we can construct a joint distribution on an arbitrary number of random variables, $\Pr(V_1, \ldots, V_n)$, as follows:

$$\begin{aligned}\Pr(V_1 = v_1, \ldots, V_n = v_n) = \; & \Pr(V_1 = v_1 \mid V_2 = v_2, \ldots, V_n = v_n) \\ & \cdot \Pr(V_2 = v_2 \mid V_3 = v_3, \ldots, V_n = v_n) \\ & \ldots \\ & \cdot \Pr(V_{n-1} = v_{n-1} \mid V_n = v_n) \\ & \cdot \Pr(V_n = v_n)\end{aligned}$$

This can be done with the variables in any order.

### 7.4.2 Marginalization

A *marginal distribution* over any individual random variable can be obtained from the joint distribution by summing over all assignments to the other random variables in the joint distribution. In two dimensional tables, this means summing the rows or the columns:

$$\Pr(A = a) = \sum_b \Pr(A = a, B = b)$$

*Example 18.* In our example with toothaches and cavities, we can compute the marginal distributions:

$$\Pr(A) = \{T : 0.1, F : 0.9\}$$
$$\Pr(C) = \{T : 0.15, F : 0.85\}$$

Although you can compute the marginal distributions from the joint distribution, **you cannot in general compute the joint distribution from the marginal distributions!!** In the **very special case** when two random variables $A$ and $B$ do not influence one another, we say that they are *independent*, which is mathematically defined as

$$\Pr(A = a, B = b) = \Pr(A = a) \Pr(B = b) \quad .$$

If we only knew the marginals of toothaches and cavities, and assumed they were independent, we would find that $\Pr(C = T, A = T) = 0.015$, which is much less than the value in our joint distribution. This is because, although cavity and toothache are relatively rare events, they are highly dependent.

### 7.4.3 Conditioning

One more important operation on a joint distribution is *conditioning*. It is fundamentally the same operation as computing a conditional probability, but when the conditioning event (on the right hand side of the bar) is that a random variable has a particular value, then we get a nice simplification. So, for example, if we wanted to condition our joint toothache/cavity distribution on the event $A = T$, we could write $\Pr(C, A \mid A = T)$. But since the value for $A$ is already made explicit in the conditioning event, we typically write this as $\Pr(C \mid A = T)$. It is obtained by:

- Finding the row (or column) of the joint distribution corresponding to the conditioning event. In our example, it would be the row for $A = T$, which consists of $\Pr(A = T, C = T)$ and $\Pr(A = T, C = F)$.
- Dividing each of the numbers in that row (or column) by the probability of the conditioning event, which is the marginal probability of that row (or column). In our example, we would divide 0.05 by 0.1 to get

$$\Pr(C \mid A = T) = \{T : 0.5, F : 0.5\}$$

So, in this example, although a cavity is relatively unlikely, it becomes much more likely conditioned on knowing that the person has a toothache.

We have described conditioning in the case that our distribution is only over two variables, but it applies to joint distributions over any number of variables. You just have to think of selecting the entries whose value for the specified random variable equals the specified value, and then renormalizing their probabilities so they sum to 1.

### 7.4.4 Bayesian reasoning

Frequently, for medical diagnosis or characterizing the quality of a sensor, it's easiest to measure conditional probabilities of the form $\Pr(\textit{Symptom} \mid \textit{Disease})$, indicating what proportion of diseased patients have a particular symptom. (These numbers are often more useful, because they

tend to be the same everywhere, even though the proportion of the population that has disease may differ.) But in these cases, when a patient comes in and demonstrates some actual symptom $s$, we really want to know $\Pr(\mathit{Disease} \mid \mathit{Symptom} = s)$. We can compute that if we also know a *prior* or *base rate* distribution, $\Pr(\mathit{Disease})$. The computation can be done in two steps, using operations we already know:

1. Form the joint distribution: $\Pr(\mathit{Disease}, \mathit{Symptom})$
2. Condition on the event $\mathit{Symptom} = s$, which means selecting the row (or column) of the joint distribution corresponding to $\mathit{Symptom} = s$ and dividing through by $\Pr(\mathit{Symptom} = s)$.

So, given an actual observation of symptoms $s$, we can determine a conditional distribution over *Disease*, by computing for every value of $d$,

$$\Pr(\mathit{Disease} = d \mid \mathit{Symptom} = s) = \frac{\Pr(\mathit{Symptom} = s \mid \mathit{Disease} = d)\Pr(\mathit{Disease} = d)}{\Pr(\mathit{Symptom} = s)} \quad .$$

The formula, when written this way, is called *Bayes' Rule*, after Rev. Thomas Bayes who first formulated this solution to the 'inverse probability problem,' in the 18th century.

### 7.4.5 Total probability

Another common pattern of reasoning is sometimes known as *the law of total probability*. What if we have our basic distributions specified in an inconvenient form: we know, for example, $\Pr(A)$ and $\Pr(B \mid A)$, but what we really care about is $\Pr(B)$? We can form the joint distribution over $A$ and $B$, and then marginalize it by summing over values of $A$. To compute one entry of the resulting distribution on $B$, we would do:

$$\Pr(B = b) = \sum_a \Pr(B = b \mid A = a)\Pr(A = a)$$

but it's easier to think about as an operation on the whole distributions.

### 7.4.6 Python operations on distributions

We can implement the operations described in this section as operations on `DDist` instances.

#### Constructing a joint distribution

We start by defining a procedure that takes a distribution $\Pr(A)$, named `PA`, and a conditional distribution $\Pr(B \mid A)$, named `PBgA`, and returns a joint distribution $\Pr(A, B)$, represented as a Python `dist.DDist` instance. It must be the case that the domain of $A$ in `PA` is the same as in `PBgA`. It creates a new instance of `dist.DDist` with entries `(a, b)` for all `a` with support in `PA` and `b` with support in `PB`. The Python expression `PA.prob(a)` corresponds to $\Pr(A = a)$; `PBgA` is a conditional probability distribution, so `PBgA(a)` is the distribution $\Pr(B \mid A = a)$ on $B$, and `PBgA(a).prob(b)` is $\Pr(B = b \mid A = a)$.

So, for example, we can re-do our joint distribution on $A$ and $B$ as:

```
PA = dist.DDist({'a1' : 0.9, 'a2' : 0.1})
def PBgA(a):
    if a == 'a1':
        return dist.DDist({'b1' : 0.7, 'b2' : 0.3})
    else:
        return dist.DDist({'b1' : 0.2, 'b2' : 0.8})

>>> PAB = JDist(PA, PBgA)
>>> PAB
DDist((a1, b2): 0.270000, (a1, b1): 0.630000, (a2, b2): 0.080000, (a2, b1): 0.020000)
```

We have constructed a new joint distribution. We leave the implementation as an exercise.

## Marginalization

Now, we can add a method to the `DDist` class to marginalize out a variable. It is only appropriately applied to instances of `DDist` whose domain is pairs or tuples of values (corresponding to a joint distribution). It takes, as input, the index of the variable that we want to *marginalize out*.

It can be implemented using two utility procedures: `removeElt` takes a list and an index and returns a new list that is a *copy* of the first list with the element at the specified index removed; `incrDictEntry` takes a dictionary, a key, and a numeric increment and adds the increment to the value of the key, adding the key if it was not previously in the dictionary.

```
def removeElt(items, i):
    result = items[:i] + items[i+1:]
    if len(result) == 1:
        return result[0]
    else:
        return result

def incrDictEntry(d, k, v):
    if d.has_key(k):
        d[k] += v
    else:
        d[k] = v
```

Now, we can understand `marginalizeOut` as making a new dictionary, with entries that have the variable at the specified index removed; the probability associated with each of these entries is the sum of the old entries that agree on the remaining indices. So, for example, we could take the joint distribution, `PAB`, that we defined above, and marginalize out the variable A (by specifying index 0) or B (by specifying index 1):

```
>>> PAB.marginalizeOut(0)
DDist(b1: 0.650000, b2: 0.350000)
>>> PAB.marginalizeOut(1)
DDist(a1: 0.900000, a2: 0.100000)
```

## Conditioning

We can also add a `conditionOnVar` method to `DDist` which, like `marginalizeOut`, should only be applied to joint distributions. It takes as input an index of the value to be conditioned on, and a value for that variable, and returns a `DDist` on the remaining variables. It operates in three steps:

- Collect all of the value-tuples in the joint distribution that have the specified value at the specified index. This is the new universe of values, over which we will construct a distribution.
- Compute the sum of the probabilities of those elements.
- Create a new distribution by removing the elements at the specified index (they are redundant at this point, since they are all equal) and dividing the probability values by the sum of the probability mass in this set. The result is guaranteed to be a distribution in the sense that the probability values properly sum to 1.

Now, we can compute, for example, the distribution on $A$, given that $B = b_1$:

```
>>> PAB.conditionOnVar(1, 'b1')
DDist(a1: 0.969231, a2: 0.030769)
```

Note that this is *not* a conditional distribution, because it is not a function from values of $B$ to distributions on $A$. At this point, it is simply a distribution on $A$.

*Exercise 7.2.* Define a method `condDist(self, index)` of the `DDist` class that makes a new conditional probability distribution. Remember that a conditional distribution is *not a distribution*. It is a function that takes as input a value of the random variable we are conditioning on, and returns, as a result a probability distribution over the other variable(s).
So, this method takes an index (of the variable we are conditioning on) and returns a conditional probability distribution of the other variables in the joint distribution, given the variable at the specified index.
Answer:

```
def condDist(self, index):
    return lambda val: self.conditionOnVar(index, val)
```

## Bayesian Evidence

The operation of updating a distribution on a random variable $A$, given evidence in the form of the value $b$ of a random variable $B$, can be implemented as a procedure that takes as arguments the prior distribution $\Pr(A)$, named `PA`, a conditional distribution $\Pr(B \mid A)$, named `PBgA`, and the actual evidence $b$, named `b`. It starts by constructing the joint distribution $\Pr(A, B)$ with `JDist`. Then, remembering that the order of the variables in the joint distribution is `(A, B)`, it conditions on the variable with index 1 (that is, $B$) having value `b`, and returns the resulting distribution over $A$.

So, for example, given a prior distribution on the prevalence of disease in the population

```
pDis = dist.DDist({True: 0.001, False: 0.999})
```

and the conditional distribution of test results given disease:

```
def pTestGivenDis(disease):
    if disease:
        return dist.DDist({True: 0.99, False: 0.01})
    else:
        return dist.DDist({True: 0.001, False: 0.999})
```

we can determine the probability that someone has the disease if the test is positive:

```
>>> dist.bayesEvidence(pDis, pTestGivenDis, True)
DDist(False: 0.502262, True: 0.497738)
```

*Exercise 7.3.* Does the result above surprise you? What happens if the prevalence of disease in the population is one in a million? One in ten?

### Total Probability

Finally, we can implement the law of total probability straightforwardly in Python. Given a distribution $\Pr(A)$, called `PA`, and $\Pr(B \mid A)$, called `PBgA`, we compute $\Pr(B)$. We do this by constructing the joint distribution and then marginalizing out $A$ (which is the variable with index 0).

To compute the probability distribution of test results in the example above, we can do:

```
>>> dist.totalProbability(pDis, pTestGivenDis)
DDist(False: 0.998011, True: 0.001989)
```

## 7.5 Modeling with distributions

When we have a small number of discrete states, it is relatively easy to specify probability distributions. But, as domains become more complex, we will need to develop another PCAP system, just for constructing distributions. In this section, we'll describe a collection of relatively standard primitive distributions, and a method, called a *mixture distribution* for combining them, and show how they can be implemented in Python.

### 7.5.1 Primitives

#### Delta

Sometimes we'd like to construct a distribution with all of the probability mass on a single element. Here's a handy way to create *delta* distributions, with a probability spike on a single element:

```
def DeltaDist(v):
    return DDist({v:1.0})
```

## Uniform

Another common distribution is the *uniform* distribution. On a discrete set of size $n$, it assigns probability $1/n$ to each of the elements:

```
def UniformDist(elts):
    p = 1.0 / len(elts)
    return DDist(dict([(e, p) for e in elts]))
```

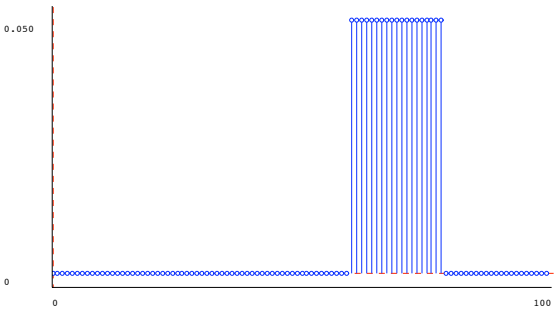
## Square

There are some distributions that are particularly valuable in numeric spaces. Since we are only dealing with discrete distributions, we will consider distributions on the integers.

One useful distribution on the integers is a *square distribution*. It is defined by parameters *lo* and *hi*, and assigns probability

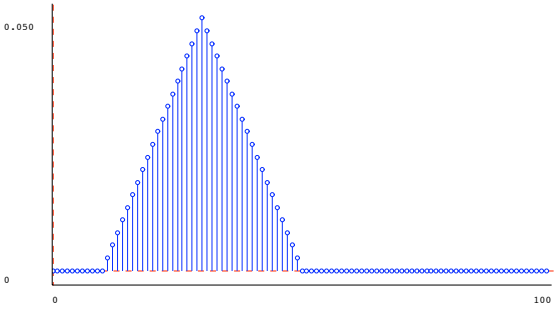
$$p = \frac{1}{hi - lo}$$

to all integers from *lo* to $hi - 1$. Here is a square distribution (from 60 to 80):




Because it is non-zero on 20 values, those values have probability 0.05.

Another useful distribution is a *triangle distribution*. It is defined by parameters *peak* and *halfWidth*. It defines a shape that has its maximum value at index *peak*, and has linearly decreasing values at each of *halfWidth* $- 1$ points on either side of the peak. The values at the indices are scaled so that they sum to 1. Here is a triangular distribution with peak 30 and half-width 20.



### 7.5.2 Mixture distribution

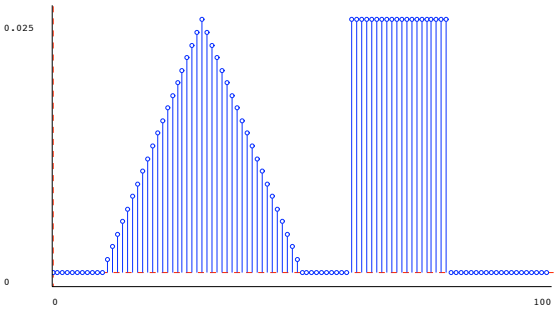
We can combine distributions by *mixing* them. To create a mixture distribution, we specify two distributions, $d_1$ and $d_2$ and a *mixing parameter* $p$, with $0 \leqslant p \leqslant 1$. The intuition is that, to draw an element from a mixture distribution, first we first flip a coin that comes up heads with probability $p$. If it is heads, then we make a random draw from $d_1$ and return that; otherwise we make a random draw from $d_2$ and return it. Another way to see it is that, if we think of a random variable $D_1$ having distribution $d_1$ and another random variable $D_2$ having distribution $d_2$, then

$$\Pr_{mix}(x) = p \Pr(D_1 = x) + (1 - p) \Pr(D_2 = x)$$

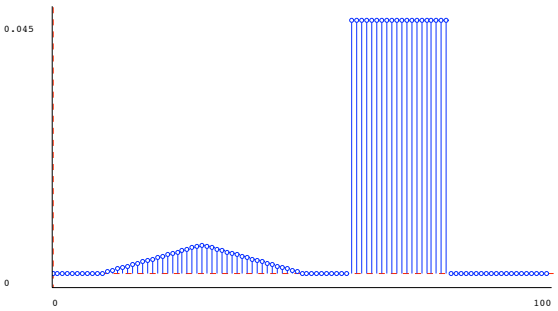
that is, the probability of an element $x$ under the mixture distribution is $p$ times its probability under distribution $d_1$ plus $1 - p$ times its probability under distribution $d_2$.

> *Exercise 7.4.* Convince yourself that if both $d_1$ and $d_2$ are proper probability distributions, in that they sum to 1 over their domains, then any mixture of them will also be a proper probability distribution.

We can make a mixture of the square and triangle distributions shown above, with mixture parameter 0.5:




Here it is, with mixture parameter 0.9, where the square is $d_1$ and the triangle is $d_2$:

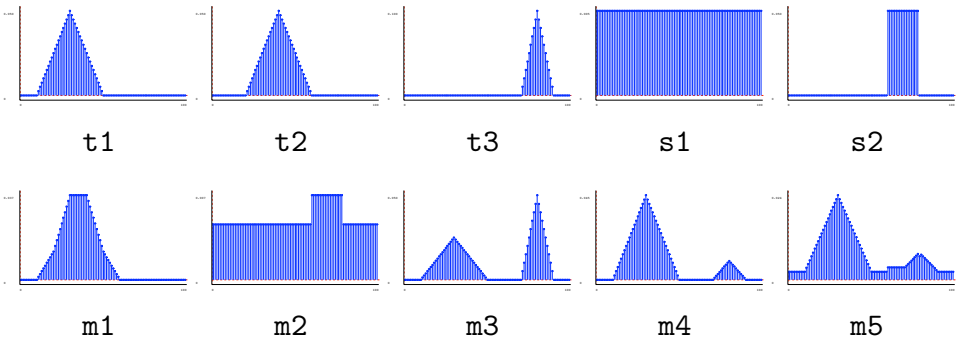



To implement a mixture distributions, we have two choices. One would be to go through the supports of both component distributions and create a new explicit `DDist`. Below, we have taken the 'lazy' approach, similar to the way we handled signal composition. We define a new class that

stores the two component distributions and the mixture parameter, and computes the appropriate probabilities as necessary.

Here are some example mixture distributions, created in Python and plotted below.

```
t1 = dist.triangleDist(30, 20)
t2 = dist.triangleDist(40, 20)
t3 = dist.triangleDist(80, 10)
s1 = dist.squareDist(0, 100)
s2 = dist.squareDist(60, 80)
m1 = dist.MixtureDist(t1, t2, 0.5)
m2 = dist.MixtureDist(s1, s2, 0.95)
m3 = dist.MixtureDist(t1, t3, 0.5)
m4 = dist.MixtureDist(t1, t3, 0.9)
m5 = dist.MixtureDist(m2, m4, 0.5)
```




## 7.6 Stochastic state machines

Now we can return to the application of primary interest: there is a system moving through some sequence of states over time, but instead of getting to see the states, we only get to make a sequence of observations of the system, where the observations give partial, noisy information about the actual underlying state. The question is: what can we infer about the current state of the system give the history of observations we have made?

As a very simple example, let's consider a copy machine: we'll model it in terms of two possible internal states: *good* and *bad*. But since we don't get to see inside the machine, we can only make observations of the copies it generates; they can either be *perfect*, *smudged*, or *all black*. There is only one input we can generate to the system, which is to ask it to make a copy.

We have to start by building a model of the system of interest. We can model this problem using a probabilistic generalization of state machines, called *stochastic*[46] *state machines* (SSMs).

In an SSM, we model time as a discrete sequence of steps, and we can think about the state, input, and the observation at each step. So, we'll use random variables $S_0, S_1, S_2, \ldots$ to model the state at each time step the random variables $O_0, O_1, \ldots$ to model the observation at each time step, and the random variables $I_0, I_1, \ldots$ to model the input at each time step.

Our problem will be to compute a probability distribution over the state at some time $t+1$ given the past history of inputs and observations $o_0, i_0, \ldots, o_t, i_t$; that is, to compute

[46] 'Stochastic' is a synonym for probabilistic.

$$\Pr(S_{t+1} \mid O_0 = o_0, I_0 = i_0, \ldots, O_t = o_t, I_t = i_t) \ .$$

A stochastic state-machine model makes a strong assumption about the system: that the state at time $t$ is sufficient to determine the probability distribution over the observation at time $t$ and the state at time $t+1$. This is the essence of what it means to be a *state* of the system; that is, that it summarizes all past history. Furthermore, we assume that the system is time-invariant, so that way the state at time 3 depends on the state and input at time 2 is the same as the way that the state at time 2 depends on the state and input at time 1, and so on; similarly for the observations.

So, in order to specify our model of how this system works, we need to provide three probability distributions:

- **Initial state distribution:** We need to have some idea of the state that the machine will be in at the very first step of time that we're modeling. This is often also called the *prior* distribution on the state. We'll write it as
$$\Pr(S_0) \ .$$
It will be a distribution over possible states of the system.
- **State transition model:** Next, we need to specify how the state of the system will change over time. It is described by the conditional probability distribution
$$\Pr(S_{t+1} \mid S_t, I_t) \ ,$$
which specifies for each possible old state $r$ and input $i$, a probability distribution over the state the system will be in at time $t+1$ if it had been in state $r$ and received input $i$ at time $t$. We will assume that this same probabilistic relationship holds for all times $t$.
- **Observation model:** Finally, we need to specify how the observations we make of the system depend on its underlying state. This is often also called the *sensor model*. It is described by the conditional probability distribution
$$\Pr(O_t \mid S_t) \ ,$$
which specifies for each possible state of the system, $s$, a distribution over the observations that will be obtained when the system is in that state.

### 7.6.1 Copy machine example

We'll specify a stochastic state-machine model for our copy machine. Because there is only a single possible input, to keep notation simple, we'll omit the input throughout the example.

- **Initial state distribution:**
$$\Pr(S_0) = \{\mathit{good} : 0.9, \mathit{bad} : 0.1\}$$
- **State transition model:**
$$\Pr(S_{t+1} \mid S_t) = \begin{cases} \{\mathit{good} : 0.7, \mathit{bad} : 0.3\} & \text{if } S_t = \mathit{good} \\ \{\mathit{good} : 0.1, \mathit{bad} : 0.9\} & \text{if } S_t = \mathit{bad} \end{cases}$$
- **Observation model:**
$$\Pr(O_t \mid S_t) = \begin{cases} \{\mathit{perfect} : 0.8, \mathit{smudged} : 0.1, \mathit{black} : 0.1\} & \text{if } S_t = \mathit{good} \\ \{\mathit{perfect} : 0.1, \mathit{smudged} : 0.7, \mathit{black} : 0.2\} & \text{if } S_t = \mathit{bad} \end{cases}$$

### 7.6.2 Representation in Python

We can represent these basic distributions in Python using the `DDist` class that we have already developed. First, we have the initial state distribution, which is quite straightforward:

```
initialStateDistribution = dist.DDist({'good': 0.9, 'bad': 0.1})
```

The observation model is simply a conditional probability distribution; that is, a procedure that takes a state as input and returns the appropriate distribution over observations.

```
def observationModel(s):
    if s == 'good':
        return dist.DDist({'perfect' : 0.8, 'smudged' : 0.1, 'black' : 0.1})
    else:
        return dist.DDist({'perfect' : 0.1, 'smudged' : 0.7, 'black' : 0.2})
```

The transition model is just a little bit trickier. For reasons that we will see later, it is most convenient to organize the transition model as a procedure that takes an input `i` as input, and then returns a conditional probability distribution of $\Pr(S_{t+1} \mid S_t, I_t = i)$. Remember that the conditional probability distribution is, itself, a procedure that takes an old state and returns a distribution over new states. This seems particularly silly for the copy machine example, since we are ignoring the input, but we still have to make the definitions this way so that the types will match those of more complex models we will build later.

```
def transitionModel(i):
    def transitionGivenI(oldState):
        if oldState == 'good':
             return dist.DDist({'good' : 0.7, 'bad' : 0.3})
        else:
             return dist.DDist({'good' : 0.1, 'bad' : 0.9})
    return transitionGivenI
```

Finally, we can define a new class, called `StochasticSM`, which is a subclass of `SM`. We have had to generalize the `SM` class slightly, so that there is an option of supplying a `startState` method instead of a fixed start state, so that the actual starting state can be the result of calling a procedure. To get a starting state, it makes a random draw from the initial state distribution; to get the next values, it makes random draws from the transition and observation distributions:

```
class StochasticSM(sm.SM):
    def __init__(self, startDistribution, transitionDistribution,
                 observationDistribution):
        self.startDistribution = startDistribution
        self.transitionDistribution = transitionDistribution
        self.observationDistribution = observationDistribution

    def startState(self):
        return self.startDistribution.draw()

    def getNextValues(self, state, inp):
        return (self.transitionDistribution(inp)(state).draw(),
                self.observationDistribution(state).draw())
```

Now, we can define our copy machine:

```
copyMachine = ssm.StochasticSM(initialStateDistribution,
                               transitionModel, observationModel)
```

Because `StochasticSM` is a subclass of `SM`, we can use the standard methods of `SM`. Here, we ask the copy machine to make 20 copies, and observe the sequence of outputs it generates:

```
>>> copyMachine.transduce(['copy']* 20)
['perfect', 'smudged', 'perfect', 'perfect', 'perfect', 'perfect',
  'perfect', 'smudged', 'smudged', 'black', 'smudged', 'black',
  'perfect', 'perfect', 'black', 'perfect', 'smudged', 'smudged',
  'black', 'smudged']
```

## 7.7 State estimation

Now, given the model of the way a system changes over time, and how its outputs reflect its internal state, we can do *state estimation*. The problem of state estimation is to take a sequence of inputs and observations, and determine the sequence of hidden states of the system. Of course, we won't be able to determine that sequence exactly, but we can derive some useful probability distributions.

We will concentrate on the problem of *filtering* or *sequential state estimation*, in which we imagine sitting and watching the stream of inputs and observations go by, and we are required, on each step, to produce a state estimate, in the form

$$\Pr(S_{t+1} \mid O_0, \ldots, O_t, I_0, \ldots, I_t)$$

We will develop a procedure for doing this by working through a few steps of the example with the copy machine, and then present it more generally.

### 7.7.1 Our first copy

Let's assume we get a brand new copy machine in the mail, and we think it is probably (0.9) good, but we're not entirely sure. We print out a page, and it looks perfect. Yay! Now, what do we believe about the state of the machine? We'd like to compute

$$\Pr(S_1 \mid O_0 = \mathit{perfect}) \quad .$$

We'll do this in two steps. First, we'll consider what information we have gained about the machine's state at time 0 from the observation, and then we'll consider what state it might have transitioned to on step 1.

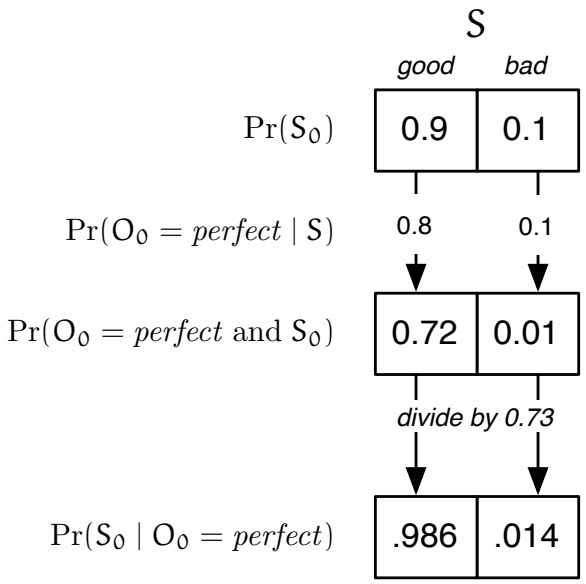
What information do we gain from knowing that the machine's first copy was perfect? We can use Bayesian reasoning to form the joint distribution between the state and observations in general, and then condition on the observation we actual received. The joint distribution has the form

| | | O | | |
|---|---|---|---|---|
| | | *perfect* | *smudged* | *black* |
| S | *good* | 0.72 | 0.09 | 0.09 |
| | *bad* | 0.01 | 0.07 | 0.02 |

Now, conditioning on the actual observation, $O_0 = \textit{perfect}$, we extract the column corresponding to the observation, $\{0.72, 0.01\}$, and divide by the sum, 0.73, to get the distribution

$$\Pr(S_0 \mid O_0 = \textit{perfect}) = \{\textit{good} : 0.986, \textit{bad} : 0.014\}$$

Here is a schematic version of this update rule, which is a good way to think about computing it by hand. Rather than creating the whole joint distribution and then conditioning by selecting out a single column, we just create the column we know we're going to need (based on the observation we got):




Because we will use it in later calculations, we will define $B'_0$ as an abbreviation:

$$B'_0 = \Pr(S_0 \mid O_0 = \textit{perfect}) \ ;$$

that is, our belief that the system is in state $s$ on the 0th step, after having taken the actual observation $o_0$ into account. This update strategy computes $B'_0(s)$ for all $s$, which we'll need in order to do further calculations.

Now, we can think about the consequences of the passage of one step of time. We'd like to compute $\Pr(S_1 \mid O_0 = \textit{perfect})$. Note that we are not yet thinking about the observation we'll get on step 1; just what we know about the machine on step 1 having observed a perfect copy at step 0.

What matters is the probability of making transitions between states at time 0 and states at time 1. We can make this relationship clear by constructing the joint probability distribution on $S_0$ and $S_1$ (it is actually conditioned on $O_0 = \textit{perfect}$). So, $\Pr(S_0, S_1 \mid O_0 = \textit{perfect})$ can be constructed from $\Pr(S_0 \mid O_0 = \textit{perfect})$ (which, it is important to remember, is a distribution on $S_0$) and $\Pr(S_1 \mid S_0)$, which is our transition model:

| | | $S_1$ | |
|---|---|---|---|
| | | *good* | *bad* |
| $S_0$ | *good* | 0.690 | 0.296 |
| | *bad* | 0.001 | 0.012 |

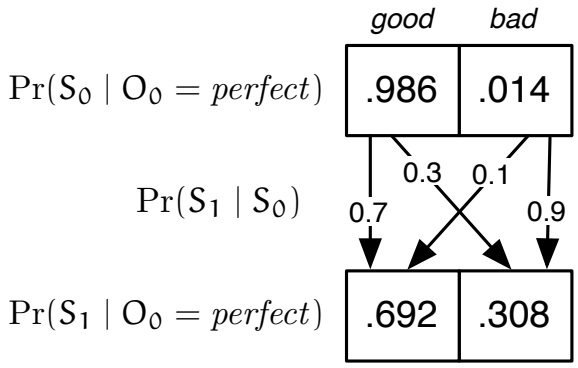
Now, we have never really known the value of $S_0$ for sure, and we can never really know it; we'd like to concentrate all of our information in a representation of our knowledge about $S_1$. We can do that by computing the marginal distribution on $S_1$ from this joint. Summing up the columns, we get

$$\Pr(S_1 \mid O_0 = \textit{perfect}) = \{\textit{good} : 0.692, \textit{bad} : 0.308\}$$

This is an application of the law of total probability.

We'll give this distribution, $\Pr(S_1 \mid O_0 = \textit{perfect})$, that is, everything we know about the machine after the first observation and transition, the abbreviation $B_1$.

Here is a schematic version of the transition update:

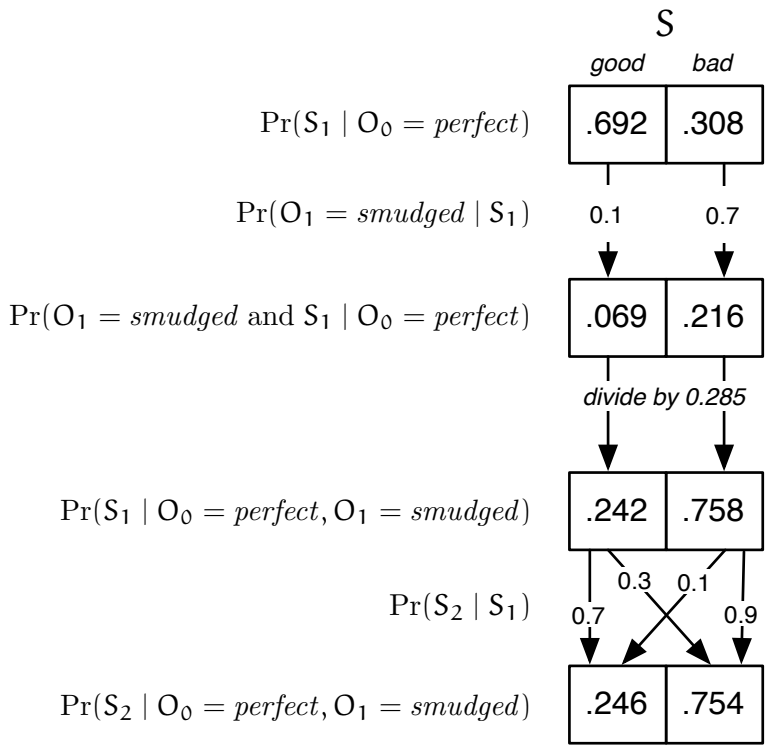



You can see that there are two ways the copier could be in a good state at time 1: because it was in a good state at time 0 (probability 0.986) and made a transition to a good state at time 1 (probability 0.7) or because it was in a bad state at time 0 (probability 0.014) and made a transition to a good state at time 1 (probability 0.1). So, the resulting probability of being in a good state is $0.986 \cdot 0.7 + 0.014 \cdot 0.1 = 0.692$. The reasoning for ending in a bad state is the same.

### 7.7.2 Our second copy

Now, let's imagine we print another page, and it's smudged. We want to compute $\Pr(S_2 \mid O_0 = \textit{perfect}, O_1 = \textit{smudged})$. This can be computed in two stages as before. We start with $B_1$, which already has the information that $O_0 = \textit{perfect}$ incorporated into it. Our job will be to fold in the information that $O_1 = \textit{smudged}$, and then the passage of another step of time.

1. Construct a joint distribution from $B_1$ and $\Pr(O_1 \mid S_1)$, and condition on $O_1 = \textit{smudged}$ to get $B_1'$. This is a Bayesian reasoning step.
2. Construct a joint distribution from $B_1'$ and $\Pr(S_2 \mid S_1)$ and marginalize out $S_1$ to get $B_2$. This is a law-of-total-probability step.




Ow. Now we're pretty sure our copy machine is no good. Planned obsolescence strikes again!

## 7.8 General state estimation

Now we'll write out the state-update procedure for SSMs in general. As a reminder, here are the random variables involved:

- State at each time $S_0, \ldots, S_T$
- Observation at each time $O_0, \ldots, O_T$
- Input at each time $I_0, \ldots, I_T$

Here are the components of the model:

- **Initial distribution** $\Pr(S_0)$
- **Observation distribution** $\Pr(O_t \mid S_t)$. Because the process is time-invariant, this is the same for all $t$.
- **Transition distribution** $\Pr(S_{t+1} \mid S_t, I_t)$. Because the process is time-invariant, this is the same for all $t$. Think of $i$ as selecting a particular conditional transition distribution to be used in the update.

Now, here is the update procedure. Assume that we have a *belief state* at time $t$, corresponding to $\Pr(S_t \mid O_{0..t-1} = o_{0..t-1})$. Then we proceed in two steps:

- **Observation update, given** $o_t$:

$$\Pr(S_t \mid O_{0..t} = o_{0..t}, I_{0..t} = i_{0..t}) = \frac{\Pr(O_t = o_t \mid S_t)\Pr(S_t \mid O_{0..t-1} = o_{0..t-1}, I_{0..t-1} = i_{0..t-1})}{\Pr(O_t = o_t \mid O_{0..t-1} = o_{0..t-1}, I_{0..t-1} = i_{0..t-1})} .$$

- **Transition update, given $i_t$:**

$$\Pr(S_{t+1} \mid O_{0..t} = o_{0..t}, I_{0..t} = i_{0..t}) \\ = \sum_r \Pr(S_{t+1} \mid S_t = r, I_t = i_t) \Pr(S_t = r \mid O_{0..t} = o_{0..t}, I_{0..t-1} = i_{0..t-1}) \ .$$

A very important thing to see about these definitions is that they enable us to build what is known as a *recursive* state estimator. (Unfortunately, this is a different use of the term "recursive" than we're used to from programming languages). If we define our belief at time $t$,

$$B_t = \Pr(S_t \mid O_{0..t-1} = o_{0..t-1}, I_{0..t-1} = i_{0..t-1})$$

and our belief after making the observation update to be

$$B'_t = \Pr(S_t \mid O_{0..t} = o_{0..t}, I_{0..t-1} = i_{0..t-1}) \ ,$$

then after each observation and transition, we can update our belief state, to get a new $B_t$. Then, we can forget the particular observation we had, and just use the $B_t$ and $o_t$ to compute $B_{t+1}$.

Algorithmically, we can run a loop of the form:

- **Condition on actual observation $O_t = o_t$.**

$$B'(s) = \frac{\Pr(O_t = o_t \mid S_t = s) B(s)}{\sum_r \Pr(O_t = o_t \mid S_t = r) B(r)}$$

- **Make transition based on input $I_t = i_t$.**

$$B(s) = \sum_r \Pr(S_{t+1} = s \mid S_t = r, I_t = i_t) B'(r)$$

where $B$ is initialized to be our initial belief about the state of the system.

### 7.8.1 Python

We can implement this state estimation process straightforwardly in Python. The nicest way to think of a state estimator is, itself, as a state machine. The state of the state estimator is $B$, the belief state, which is a probability distribution over the states of the SSM whose state we are estimating. The state estimator takes an SSM model at initialization time. At each step, the input to the state estimator is an $(o, i)$ pair, specifying the observation from and the input to the machine on the current step; the output of the state estimator is a probability distribution, represented as a `dist.DDist` over the states of the underlying system.

The initial state of the state estimator is the initial state distribution of the underlying machine:

```
class StateEstimator(sm.SM):
    def __init__(self, model):
        self.model = model
        self.startState = model.startDistribution
```

Now, to get the next values of the state estimator, we perform the observation and transition updates, as described mathematically above. We observe that the observation update is an instance

of the Bayes evidence procedure, with the current belief as the prior, the observation model as the conditional distribution, and the actual observation `o` as the value we're conditioning on. We also observe that the transition update is an instance of the law of total probability: we select the appropriate state transition distribution depending on the input `i`, then use it to make a joint distribution over $S_t$ and $S_{t+1}$, then marginalize out $S_t$, because we want to represent all of our information as a distribution on $S_{t+1}$. The output of the state estimator is the same as its state.

Now, we can do state estimation on our copy machine. We might have this sequence of observations and inputs:

```
copyData = [('perfect', 'copy'), ('smudged', 'copy')]
```

which means that we saw a perfect page, made a copy, saw a smudged page, and made another copy. Now, we can make a state estimator, using the model we defined in the previous section, and feed the data into the estimator:

```
>>> copyMachineSE = se.StateEstimator(copyMachine)
>>> copyMachineSE.transduce(copyData)
[DDist(bad: 0.308219, good: 0.691781), DDist(bad: 0.754327, good: 0.245673)]
```

Note that these are the same distributions we got when we did the state estimation by hand.

6.01SC Introduction to Electrical Engineering and Computer Science

Spring 2011