Chapter 2

Computational Statistics

2.1 Distributions

In statistics a **distribution** is a set of values and their corresponding probabilities.

For example, if you roll a six-sided die, the set of possible values is the numbers 1 to 6, and the probability associated with each value is 1/6.

As another example, you might be interested in how many times each word appears in common English usage. You could build a distribution that includes each word and how many times it appears.

To represent a distribution in Python, you could use a dictionary that maps from each value to its probability. I have written a class called `Pmf` that uses a Python dictionary in exactly that way, and provides a number of useful methods. I called the class Pmf in reference to a **probability mass function**, which is a way to represent a distribution mathematically.

`Pmf` is defined in a Python module I wrote to accompany this book, `thinkbayes.py`. You can download it from `http://thinkbayes.com/thinkbayes.py`. For more information see Section 0.3.

To use `Pmf` you can import it like this:

```
from thinkbayes import Pmf
```

The following code builds a Pmf to represent the distribution of outcomes for a six-sided die:

```
pmf = Pmf()
for x in [1,2,3,4,5,6]:
    pmf.Set(x, 1/6.0)
```

`Pmf` creates an empty Pmf with no values. The `Set` method sets the probability associated with each value to 1/6.

Here's another example that counts the number of times each word appears in a sequence:

```
pmf = Pmf()
for word in word_list:
    pmf.Incr(word, 1)
```

`Incr` increases the "probability" associated with each word by 1. If a word is not already in the Pmf, it is added.

I put "probability" in quotes because in this example, the probabilities are not normalized; that is, they do not add up to 1. So they are not true probabilities.

But in this example the word counts are proportional to the probabilities. So after we count all the words, we can compute probabilities by dividing through by the total number of words. `Pmf` provides a method, `Normalize`, that does exactly that:

```
pmf.Normalize()
```

Once you have a Pmf object, you can ask for the probability associated with any value:

```
print pmf.Prob('the')
```

And that would print the frequency of the word "the" as a fraction of the words in the list.

Pmf uses a Python dictionary to store the values and their probabilities, so the values in the Pmf can be any hashable type. The probabilities can be any numerical type, but they are usually floating-point numbers (type `float`).

2.2 The cookie problem

In the context of Bayes's theorem, it is natural to use a Pmf to map from each hypothesis to its probability. In the cookie problem, the hypotheses are B_1 and B_2. In Python, I represent them with strings:

```
pmf = Pmf()
pmf.Set('Bowl 1', 0.5)
pmf.Set('Bowl 2', 0.5)
```

This distribution, which contains the priors for each hypothesis, is called (wait for it) the **prior distribution**.

To update the distribution based on new data (the vanilla cookie), we multiply each prior by the corresponding likelihood. The likelihood of drawing a vanilla cookie from Bowl 1 is 3/4. The likelihood for Bowl 2 is 1/2.

```
pmf.Mult('Bowl 1', 0.75)
pmf.Mult('Bowl 2', 0.5)
```

`Mult` does what you would expect. It gets the probability for the given hypothesis and multiplies by the given likelihood.

After this update, the distribution is no longer normalized, but because these hypotheses are mutually exclusive and collectively exhaustive, we can **renormalize**:

```
pmf.Normalize()
```

The result is a distribution that contains the posterior probability for each hypothesis, which is called (wait now) the **posterior distribution**.

Finally, we can get the posterior probability for Bowl 1:

```
print pmf.Prob('Bowl 1')
```

And the answer is 0.6. You can download this example from `http://thinkbayes.com/cookie.py`. For more information see Section 0.3.

2.3 The Bayesian framework

Before we go on to other problems, I want to rewrite the code from the previous section to make it more general. First I'll define a class to encapsulate the code related to this problem:

```
class Cookie(Pmf):

    def __init__(self, hypos):
        Pmf.__init__(self)
        for hypo in hypos:
            self.Set(hypo, 1)
        self.Normalize()
```

A Cookie object is a Pmf that maps from hypotheses to their probabilities. The `__init__` method gives each hypothesis the same prior probability. As in the previous section, there are two hypotheses:

```
hypos = ['Bowl 1', 'Bowl 2']
pmf = Cookie(hypos)
```

`Cookie` provides an `Update` method that takes data as a parameter and updates the probabilities:

```
def Update(self, data):
    for hypo in self.Values():
        like = self.Likelihood(data, hypo)
        self.Mult(hypo, like)
    self.Normalize()
```

`Update` loops through each hypothesis in the suite and multiplies its probability by the likelihood of the data under the hypothesis, which is computed by `Likelihood`:

```
mixes = {
    'Bowl 1':dict(vanilla=0.75, chocolate=0.25),
    'Bowl 2':dict(vanilla=0.5, chocolate=0.5),
    }

def Likelihood(self, data, hypo):
    mix = self.mixes[hypo]
    like = mix[data]
    return like
```

`Likelihood` uses `mixes`, which is a dictionary that maps from the name of a bowl to the mix of cookies in the bowl.

Here's what the update looks like:

```
pmf.Update('vanilla')
```

And then we can print the posterior probability of each hypothesis:

```
for hypo, prob in pmf.Items():
    print hypo, prob
```

The result is

```
Bowl 1 0.6
Bowl 2 0.4
```

which is the same as what we got before. This code is more complicated than what we saw in the previous section. One advantage is that it generalizes to the case where we draw more than one cookie from the same bowl (with replacement):

```
dataset = ['vanilla', 'chocolate', 'vanilla']
for data in dataset:
    pmf.Update(data)
```

The other advantage is that it provides a framework for solving many similar problems. In the next section we'll solve the Monty Hall problem computationally and then see what parts of the framework are the same.

The code in this section is available from `http://thinkbayes.com/cookie2.py`. For more information see Section 0.3.

2.4 The Monty Hall problem

To solve the Monty Hall problem, I'll define a new class:

```
class Monty(Pmf):

    def __init__(self, hypos):
        Pmf.__init__(self)
        for hypo in hypos:
            self.Set(hypo, 1)
        self.Normalize()
```

So far `Monty` and `Cookie` are exactly the same. And the code that creates the Pmf is the same, too, except for the names of the hypotheses:

```
hypos = 'ABC'
pmf = Monty(hypos)
```

Calling `Update` is pretty much the same:

```
data = 'B'
pmf.Update(data)
```

And the implementation of `Update` is exactly the same:

```
def Update(self, data):
    for hypo in self.Values():
        like = self.Likelihood(data, hypo)
        self.Mult(hypo, like)
    self.Normalize()
```

The only part that requires some work is `Likelihood`:

```
def Likelihood(self, data, hypo):
    if hypo == data:
        return 0
    elif hypo == 'A':
        return 0.5
    else:
        return 1
```

Finally, printing the results is the same:

```
for hypo, prob in pmf.Items():
    print hypo, prob
```

And the answer is

```
A 0.333333333333
B 0.0
C 0.666666666667
```

In this example, writing `Likelihood` is a little complicated, but the framework of the Bayesian update is simple. The code in this section is available from `http://thinkbayes.com/monty.py`. For more information see Section 0.3.

2.5 Encapsulating the framework

Now that we see what elements of the framework are the same, we can encapsulate them in an object—a `Suite` is a `Pmf` that provides `__init__`, `Update`, and `Print`:

```
class Suite(Pmf):
    """Represents a suite of hypotheses and their probabilities."""

    def __init__(self, hypo=tuple()):
        """Initializes the distribution."""

    def Update(self, data):
        """Updates each hypothesis based on the data."""

    def Print(self):
        """Prints the hypotheses and their probabilities."""
```

The implementation of `Suite` is in `thinkbayes.py`. To use `Suite`, you should write a class that inherits from it and provides `Likelihood`. For example, here is the solution to the Monty Hall problem rewritten to use `Suite`:

```
from thinkbayes import Suite

class Monty(Suite):

    def Likelihood(self, data, hypo):
        if hypo == data:
            return 0
```

```
        elif hypo == 'A':
            return 0.5
        else:
            return 1
```

And here's the code that uses this class:

```
suite = Monty('ABC')
suite.Update('B')
suite.Print()
```

You can download this example from `http://thinkbayes.com/monty2.py`. For more information see Section 0.3.

2.6 The M&M problem

We can use the `Suite` framework to solve the M&M problem. Writing the `Likelihood` function is tricky, but everything else is straightforward.

First I need to encode the color mixes from before and after 1995:

```
mix94 = dict(brown=30,
             yellow=20,
             red=20,
             green=10,
             orange=10,
             tan=10)

mix96 = dict(blue=24,
             green=20,
             orange=16,
             yellow=14,
             red=13,
             brown=13)
```

Then I have to encode the hypotheses:

```
hypoA = dict(bag1=mix94, bag2=mix96)
hypoB = dict(bag1=mix96, bag2=mix94)
```

`hypoA` represents the hypothesis that Bag 1 is from 1994 and Bag 2 from 1996. `hypoB` is the other way around.

Next I map from the name of the hypothesis to the representation:

```
hypotheses = dict(A=hypoA, B=hypoB)
```

And finally I can write `Likelihood`. In this case the hypothesis, `hypo`, is a string, either `A` or `B`. The data is a tuple that specifies a bag and a color.

```
def Likelihood(self, data, hypo):
    bag, color = data
    mix = self.hypotheses[hypo][bag]
    like = mix[color]
    return like
```

Here's the code that creates the suite and updates it:

```
suite = M_and_M('AB')

suite.Update(('bag1', 'yellow'))
suite.Update(('bag2', 'green'))

suite.Print()
```

And here's the result:

```
A 0.740740740741
B 0.259259259259
```

The posterior probability of A is approximately 20/27, which is what we got before.

The code in this section is available from `http://thinkbayes.com/m_and_m.py`. For more information see Section 0.3.

2.7 Discussion

This chapter presents the Suite class, which encapsulates the Bayesian update framework.

`Suite` is an **abstract type**, which means that it defines the interface a Suite is supposed to have, but does not provide a complete implementation. The `Suite` interface includes `Update` and `Likelihood`, but the `Suite` class only provides an implementation of `Update`, not `Likelihood`.

A **concrete type** is a class that extends an abstract parent class and provides an implementation of the missing methods. For example, `Monty` extends `Suite`, so it inherits `Update` and provides `Likelihood`.

If you are familiar with design patterns, you might recognize this as an example of the template method pattern. You can read about this pattern at `http://en.wikipedia.org/wiki/Template_method_pattern`.

Most of the examples in the following chapters follow the same pattern; for each problem we define a new class that extends `Suite`, inherits `Update`, and provides `Likelihood`. In a few cases we override `Update`, usually to improve performance.

2.8 Exercises

Exercise 2.1. *In Section 2.3 I said that the solution to the cookie problem generalizes to the case where we draw multiple cookies with replacement.*

But in the more likely scenario where we eat the cookies we draw, the likelihood of each draw depends on the previous draws.

Modify the solution in this chapter to handle selection without replacement. Hint: add instance variables to `Cookie` *to represent the hypothetical state of the bowls, and modify* `Likelihood` *accordingly. You might want to define a* `Bowl` *object.*