Computational Probability and Inference

Colin Leach, from original by MIT course staff September 2016

Abstract

This is a set of notes for the online course '6.008.1x Computational Probability and Inference' given by the staff of MIT.

Contents

1 Probability and Inference 2
1.1 Introduction to Probability 2
1.2 Probability Spaces and Events 4
1.3 Random Variables 7
1.4 Jointly Distributed Random Variables 7
1.5 Conditioning on Events 7

1 Probability and Inference

1.1 Introduction to Probability

Introduction Probabilities appear in everyday life and feed into how we make decisions. For example:

The weather forecast might say that "tomorrow there is a 70% chance of rain". This 70% chance of rain is a probability, and if it is sufficiently high, then we may want to bring an umbrella when we go outdoors.

We could predict that the probability of car traffic is higher during rush hour than otherwise, so if we don't want to be stuck in traffic while driving, we should avoid driving during rush hour.

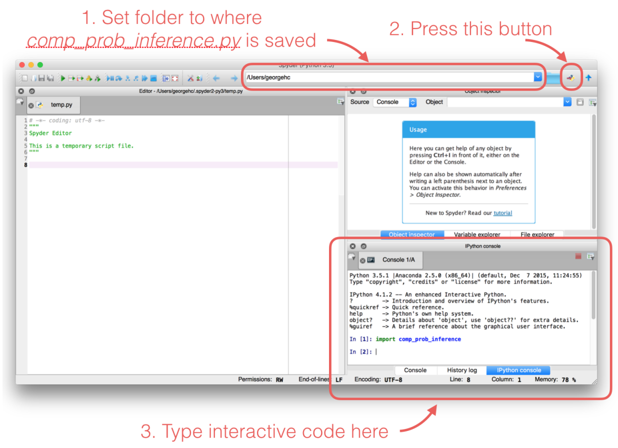
We aim to build computer programs that can reason with probabilities.

A First Look at Probability Perhaps the simplest example of probability is flipping a fair coin for which we say that the probability of heads is 1/2 and, similarly, the probability of tails is also 1/2. (Don't worry, we'll see much more exciting problems soon!) What do we mean when we say that the probability of heads is 1/2?

The basic idea is that if we repeat this experiment of flipping a coin a huge number of times, say n, then the number of heads we should see should be close to $n/2$ as we increase n. While you could certainly try this out in real life by flipping a coin, say, 100,000 times, doing this would be disastrously tedious. Let's simulate these flips in Python instead.

Simulating Coin Flips Follow along in an IPython prompt within Spyder.

We have provided a package `comp_prob_inference.py`, which you should save to your computer. Within Spyder, do the following:



```
# -*- coding: utf-8 -*-
"""
Spyder Editor

This is a temporary script file.
"""

```

Here you can get help of any object by pressing **Ctrl+I** in front of it, either on the Editor or the Console.

Help can also be shown automatically after writing a left parenthesis next to an object. You can activate this behavior in *Preferences > Object Inspector*.

New to Spyder? Read our tutorial

```
Python 3.5.1 |Anaconda 2.5.0 (x86_64)| (default, Dec  7 2015, 11:24:55)
Type "copyright", "credits" or "license" for more information.

IPython 4.1.2 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.
%guiref   -> A brief reference about the graphical user interface.

In [1]: import comp_prob_inference

In [2]: |
```

Permissions: RW End-of-lines: LF Encoding: UTF-8 Line: 8 Column: 1 Memory: 78 %

Let's start by importing the package comp_prob_inference:

```
> import comp_prob_inference
```

To simulate flipping a fair coin, enter:

```
> comp_prob_inference.flip_fair_coin()
```

You should get either 'heads' or 'tails'. Try re-running the above line a few times. You should see that the coin flip results are random.

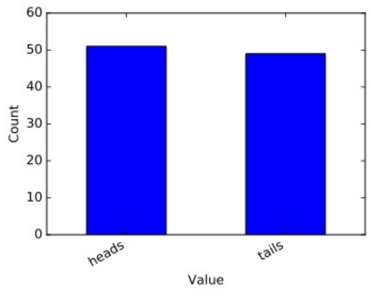
To flip the fair coin 100 times, enter:

```
> flips = comp_prob_inference.flip_fair_coins(100)
```

Let’s plot how many times we see the two possible outcomes in the same bar graph, called a histogram:

```
> comp_prob_inference.plot_discrete_histogram(flips)
```

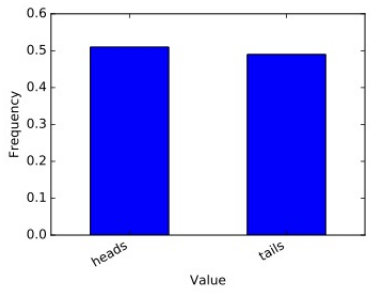
For example, we get the following plot:



Often what we will care about in this course is the fraction (also called the frequency of times an outcome happens. To plot the fraction of times heads or tails occurred, we again use the plot_discrete_histogram function but now add the keyword argument frequency=True:

```
> comp_prob_inference.plot_discrete_histogram(flips, frequency=True)
```

Doing so, we get the following plot:



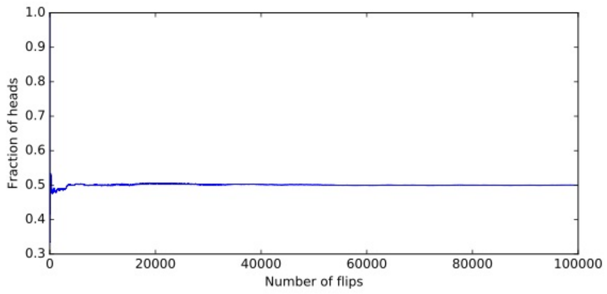
Next, let’s plot the fraction of heads as a function of the number of flips (going up to 100,000 flips).

```
n = 100000
heads_so_far = 0
fraction_of_heads = []
for i in range(n):
    if comp_prob_inference.flip_fair_coin() == 'heads':
        heads_so_far += 1
    fraction_of_heads.append(heads_so_far / (i+1))
```

Note that `fraction_of_heads[i]` tells us what the fraction of heads is after the first i tosses. Then to actually plot the fraction of heads vs the number of tosses, enter the following:

```
import matplotlib.pyplot as plt
plt.figure(figsize=(8, 4))
plt.plot(range(1, n+1), fraction_of_heads)
plt.xlabel('Number of flips')
plt.ylabel('Fraction of heads')
```

For example, when we run this we get the following plot:



The fraction of heads initially can be far from 1/2 but as the number of flips increases, the fraction stabilizes and gets closer to 1/2, the probability of heads.

Computer note: Many times in this course, it will be helpful to run simulations to test code and plot histograms for different outcomes to get a sense of how likely the outcomes are. Simulations and visualizations can be powerful not only in making sure your code is working correctly but also to present results to people!

Probability and the Art of Modeling Uncertainty Since probability effectively corresponds to a fraction, it is a value between 0 and 1. Of course, we can have impossible events that have probability 0, or events that deterministically happen and thus have probability 1. Each time we model uncertainty in the world, there will be some underlying experiment (such as flipping a coin in our running example). An event happens with probability $q \in [0, 1]$ if in a massive number of repeats of the experiment, the event happens roughly a fraction q of the time; more repeats of the experiment make it so that the fraction gets closer to q.

Some times, an underlying experiment cannot possibly be repeated. Take for instance weather forecasting. Whereas we could actually physically flip a coin many times to repeat the same experiment, we cannot physically repeat a real-life experiment for what different realizations of tomorrow's weather will be. We could wait until tomorrow to see the weather, but then we would need a time machine to go back in time by one day to repeat and see what the weather is like tomorrow (and this assumes that there's some inherent randomness in tomorrow's weather)! In such a case, our only hope is to somehow model or simulate tomorrow's weather given measurements up to present time.

Different people could model the same real world problem differently! Throughout the course, a recurring challenge in building computer programs that reason probabilistically is figuring out how to model real-world problems. A good model — even if not actually accurate in describing, for instance, the science behind weather — enables us to make good predictions.

Once a weather forecaster has anchored some way of modeling or simulating weather, then if it claims that there's a 30% chance of rain tomorrow, we could interpret this as saying that using their way of simulating tomorrow's weather, in roughly 30% of simulated results for tomorrow's weather, there is rain.

1.2 Probability Spaces and Events

Two Ingredients to Modeling Uncertainty When we think of an uncertain world, we will always think of there being some underlying experiment of interest. To model this uncertain world, it suffices to keep track of two things:

The set of all possible outcomes for the experiment: this set is called the sample space and is usually denoted by the Greek letter Omega Ω. (For the fair coin flip, there are exactly two possible outcomes: heads, tails. Thus, $\Omega = \{\text{heads}, \text{tails}\}$.)

The probability of each outcome: for each possible outcome, assign a probability that is at least 0 and at most 1. (For the fair coin flip, $\mathbb{P}(\text{heads}) = \frac{1}{2}$ and $\mathbb{P}(\text{tails}) = \frac{1}{2}$.)

Notation: Throughout this course, for any statement $\mathscr{S}$, "$\mathbb{P}(\mathscr{S})$" denotes the probability of $\mathscr{S}$ happening.

In Python:

```
> model = {'heads': 1/2, 'tails': 1/2}
```

In particular, we see that we can model uncertainty in code using a Python dictionary. The sample space is precisely the keys in the dictionary:

```
> sample_space = set(model.keys())
{'tails', 'heads'}
```

Of course, the dictionary gives us the assignment of probabilities, meaning that for each outcome in the sample space (i.e., for each key in the dictionary), we have an assigned probability:

```
> model['heads']
0.5
> model['tails']
0.5
```

A few important remarks:

- The sample space is always specified to be *collectively exhaustive*, meaning that every possible outcome is in it, and *mutually exclusive*, meaning that once the experiment is run (e.g., flipping the fair coin), exactly one possible outcome in the sample space happens. It's impossible for multiple outcomes in the sample space to simultaneously happen! It's also impossible for none of the outcomes to happen!
- Probabilities can be thought of as fractions of times outcomes occur; thus, probabilities are nonnegative and at least 0 and at most 1.
- If we add up the probabilities of all the possible outcomes in the sample space, we get 1. (For the fair coin flip, $\mathbb{P}(\text{heads}) + \mathbb{P}(\text{tails}) = \frac{1}{2} + \frac{1}{2} = 1$.)

Some intuition for this: Consider the coin flipping experiment. What does the fraction of times heads occur and the fraction of times tails occur add up to? Since these are the only two possible outcomes (and again, recall that these outcomes are exclusive in that they can't simultaneously occur, and exhaustive since they are the only possible outcomes), these two fractions will always sum to 1. For a massive number of repeats of the experiment, these two fractions correspond to $\mathbb{P}(\text{heads})$ and $\mathbb{P}(\text{tails})$; the fractions sum to 1 and so these probabilities also sum to 1.

Probability Spaces At this point, we've actually already seen the most basic data structure used throughout this course for modeling uncertainty, called a *finite probability space* (in this course, we'll often also just call this either a *probability space* or a *probability model*):

A *finite probability space* consists of two ingredients:

- a sample space Ω consisting of a *finite* (i.e., not infinite) number of collectively exhaustive and mutually exclusive possible outcomes
- an assignment of probabilities: for each possible outcome $\omega \in \Omega$, we assign a probability $\mathbb{P}(\text{outcome } \omega)$ at least 0 and at most 1, where we require that the probabilities across all the possible outcomes in the sample space add up to 1:
  $\sum_{\omega \in \Omega} \mathbb{P}(\text{outcome } \omega) = 1$

Notation: As shorthand we occasionally use the tuple "$(\Omega, \mathbb{P})$" to refer to a finite probability space to remind ourselves of the two ingredients needed, sample space Ω and an assignment of probabilities $\mathbb{P}$. As we already saw, in code these two pieces can be represented together in a single Python dictionary. However, when we want to reason about probability spaces in terms of the mathematics, it's helpful to have names for the two pieces.

Why finite? Of the two pieces making up a finite probability space $(\Omega, \mathbb{P})$, the sample space Ω being finite is a fairly natural constraint, corresponding to how we typically work with Python dictionaries where there is only a finite number of keys. As we'll see, finite probability spaces are already extremely useful in practice. Pedagogically, finite probability spaces also provide a great intro to probability theory as they already carry a wealth of intuition, much of which carries over to a more complete story of general probability spaces!

Table Representation A probability space is a data structure in that we can always visualize as a table of nonnegative entries that sum to 1. Let's see a concrete example of this, first writing the table out on paper and then coding it up.

Example: Suppose we have a model of tomorrow's weather given as follows: sunny with probability 1/2, rainy with probability 1/6, and snowy with probability 1/3. Here's the probability space, shown as a table:

		Probability
	sunny	1/2
Outcome	rainy	1/6
	snowy	1/3

Note: This a table of 3 nonnegative entries that sum to 1. The rows correspond to the sample space $\Omega = \{\text{sunny}, \text{rainy}, \text{snowy}\}$.

We will often use this table representation of a probability space to tell you how we're modeling uncertainty for a particular problem. It provides the simplest of visualizations of a probability space.

Of course, in Python code, the above probability space is given by:

```
prob_space = {'sunny': 1/2, 'rainy': 1/6, 'snowy': 1/3}
```

A different way to code up the same probability space is to separately specify the outcomes (i.e., the sample space) and the probabilities:

```
outcomes = ['sunny', 'rainy', 'snowy']
probabilities = np.array([1/2, 1/6, 1/3])
```

The i-th entry of `outcomes` has probability given by the i-th entry of `probabilities`. Note that `probabilities` is a vector of numbers that we represent as a Numpy array. Numpy has various built-in methods that enable us to easily work with vectors (and more generally arrays) of numbers.

More on Sample Spaces In the video, we saw that a sample space encoding the outcomes of 2 coin flips encodes all the information for 1 coin flip as well. Thus, we could use the same sample space to model a single coin flip. However, if we really only cared about a single coin flip, then a sample space encoding 2 coin flips is richer than we actually need it to be!

When we model some uncertain situation, how we specify a sample space is not unique. We saw an example of this already in an earlier exercise where for rolling a single six-sided die, we can choose to name the outcomes differently, saying for instance "roll 1" instead of "1". We could even add a bunch of extraneous outcomes that all have probability 0. We could add extraneous information that doesn't matter such as "Alice rolls 1", "Bob rolls 1", etc where we enumerate out all the people who could roll the die in which the outcome is a 1. Sure, depending on the problem we are trying to solve, maybe knowing who rolled the die is important, but if we don't care about who rolled the die, then the information isn't helpful but it's still possible to include this information in the sample space.

Generally speaking it's best to choose a sample space that is as simple as possible for modeling what we care about solving. For example, if we were rolling a six-sided die, and we actually only care about whether the face shows up at least 4 or not, then it's sufficient to just keep track of two outcomes, "at least 4" and "less than 4".

Probabilities with Events

Events as Sets

Code for Dealing with Sets in Python In the video, the set operations can actually be implemented in Python as follows:

```
sample_space = {'HH', 'HT', 'TH', 'TT'}
A = {'HT', 'TT'}
B = {'HH', 'HT', 'TH'}
C = {'HH'}
A_intersect_B = A.intersection(B) # equivalent to "B.intersection(A)" or "A & B"
A_union_C = A.union(C) # equivalent to "C.union(A)" and also "A | C"
B_complement = sample_space.difference(B) # equivalent also to "sample_space - B"
```

Probabilities with Events and Code From the videos, we see that an event is a subset of the sample space Ω. If you remember our table representation for a probability space, then an event could be thought of as a subset of the rows, and the probability of the event is just the sum of the probability values in those rows!

The probability of an event $\mathscr{A} \subseteq \Omega$ is the sum of the probabilities of the possible outcomes in $\mathscr{A}$:

$\mathbb{P}(\mathscr{A}) \triangleq \sum_{\omega \in \mathscr{A}} \mathbb{P}(\text{outcome } \omega)$,

where "$\triangleq$" means "defined as".

We can translate the above equation into Python code. In particular, we can compute the probability of an event encoded as a Python set `event`, where the probability space is encoded as a Python dictionary `prob_space`:

```
def prob_of_event(event, prob_space):
    total = 0
    for outcome in event:
        total += prob_space[outcome]
    return total
```

Here's an example of how to use the above function:

```
prob_space = {'sunny': 1/2, 'rainy': 1/6, 'snowy': 1/3}
rainy_or_snowy_event = {'rainy', 'snowy'}
print(prob_of_event(rainy_or_snowy_event, prob_space))
```

1.3 Random Variables

1.4 Jointly Distributed Random Variables

1.5 Conditioning on Events