Lab 1: Working with probability distributions

Use the html version[1] of this lab if you want to easily access the links or copy and paste commands.
Or use the pdf version[2] if you want nicer formatting or a printable sheet.

Goals and motivation of this lab, and what to submit

This week's lab is intended to help build your intuitions about estimating probability distributions from data, and to give you some example code for generating bar plots and for sampling from a discrete probability distribution.

We have written most of the code here already and included a lot of explanatory comments, but we will ask you to add a few things here and there. For students with more programming background, the 'Going Further' section will give you a chance to explore some more advanced topics and efficiency issues.

We encourage you to *work with a partner* on the lab and discuss the questions and solutions with each other. Try to help with each other's weaknesses: if one student is stronger with programming, let the other student take the keyboard and give them a chance to work things out. And maybe they can help you with the math!

At the end of today's lab session, please email me a short message including the following information:

- your name and your partner's name;
- a few sentences answering the questions in the 'Effects of sample size' section;
- a note saying how far you got in the lab, if you didn't finish;
- an attachment with your solution code (as far as you've gotten).

I will post solutions to each lab but will not provide individual feedback, I just want to make sure that you are actually making an effort.

Preliminaries

Before starting, you'll need to download the lab materials. We suggest creating a folder to hold all the labs (e.g., `introCL-labs`), with a subfolder for each one (e.g., `lab1` for this lab).

Download the file lab1.py[3] into your new folder. (In Windows, this can be done by right-clicking on the link and selecting *Save link as...*, then choosing the folder to save in.)

Using a Python editor or development environment, open the `lab1.py` file so you can see the code, and start a Python interpreter. (If you've installed Anaconda, run its development environment, called Spyder, and open the file there. The interpreter console is in the bottom right.)

Examining and running the code

Run the code in the `lab1.py` file. (If using Spyder, click on the green triangle.)

You should see a plot appear with two sets of bars, "True" and "Est". "True" plots the probabilities of the different outcomes defined by the `distribution` dictionary at the top of the main body of code. "Est" will eventually plot the probabilities as estimated from some data, although at the moment the values of the bars are incorrect.

[1] http://homepages.inf.ed.ac.uk/sgwater/teaching/lsa2015/labs/lab1.html
[2] http://homepages.inf.ed.ac.uk/sgwater/teaching/lsa2015/labs/lab1.pdf
[3] http://homepages.inf.ed.ac.uk/sgwater/teaching/lsa2015/labs/lab1.py
[4] http://stackoverflow.com/questions/11373192/generating-discrete-random-variables-with-specified-weights-using-scipy-or-numpy
[5] https://wiki.python.org/moin/PythonSpeed/PerformanceTips

Now look at the text that was printed out in the interpreter window. (In some versions of Python, you might need to close the plot window first.) By looking at the printout, the code, and/or debugging information in the top right window of Spyder, can you see what `str_list`, `str_counts`, and `str_probs` are intended to be, and what datatypes they are? Which of these three variables has an incorrect value?

Normalizing a distribution

Take a look at the function `normalize_counts`. What is this function supposed to do? What is it actually doing? Fix the function so that it does what it is supposed to do. (You may want to use the `sum` function to help you.)

What is a simple error check you could perform on the return value of `normalize_counts` to make sure it is a probability distribution?

Note: you may find as you are working through this lab that some of the numerical results you get are not quite what you expect, for example you might get 0.20000000000000001 as the answer to 1/5. (If you want to test this in the interpreter, you will need to type `from __future__ import division` first.) This is because the answer to 1/5 is easy to represent in base 10 (which we use), but not in base 2 (which the computer uses internally). There is a tiny amount of error involved when the computer converts from base 2 back to base 10. Most programming languages by default only show numbers to five or six decimal places, so you won't see the error, but Python shows more, so you do see it.

Comparing estimated and true probabilities

If you correctly fixed the `normalize_counts` function, you should be able to rerun the whole file and see a comparison between the true distribution over characters and an estimate of that distribution which is based on the observed counts of each character in the randomly generated sequence (which in turn was generated from the true distribution). What is the name for the kind of estimate produced here?

Now look at the plot comparing the true and estimated distributions. Notice that there are several letters with very low probability under the true distribution. You will probably see that some of them occur in the randomly generated sequence and some do not, just due to random chance. For the low-probability letters that *do* occur in the sequence, are their estimated probabilities generally higher or lower than the true probabilities? What about for the low-probability letters that *do not* occur in the sequence?

Effects of sample size

Change the code so that the length of the generated sequence is much smaller or much larger than 50 and rerun the code. Are the estimated probabilities more or less accurate with larger amounts of data? Are you able to get estimates that no longer include zero probabilities? Would it be possible to adjust the sample size in a natural language corpus to avoid zero probabilities while still using the same kind of probability estimation you have here? Why or why not?

Computing the likelihood

In class, we discussed the *likelihood*, defined as the probability of the observed data given a particular model. Here, our true distribution and estimated distribution are different possible models. Let's find the likelihood of each model.

First, change the code back so that you are generating sequences of length 50 again. You may also want to comment out the line that produces the plot, since we won't be using it again.

Next, fill in the correct code in the `compute_likelihood` function so that it returns the probability of a sequence of data (first argument) given the model provided as the second argument.

The function you just defined is being used to compute two different likelihoods using the generated sequence of 50 characters: first, the likelihood of the true distribution, and then the likelihood of the estimated distribution. Look at the values being printed out. Which model assigns higher probability to the data? By how much? (You may want to run the code a few times to see how these values change depending on the particular random sequence that is generated eaach time.)

Note: make sure you are familiar with the *floating-point notation* used by Python (and other computer programs): A number like 1.2e4 means 1.2 x 10^4 or 12000, similarly 1.2e-4 means 1.2 x 10^{-4} or .00012

Log likelihood

Increase the length of the random sequence of data to 500 characters. You should find that your program now says both likelihoods are 0. Is that correct? Do you know why this happened?

The problem you just saw is one practical reason for using *logs* in so many of the computations we do with probabilities. So, instead of computing the likelihood, we typically compute the *log likelihood* (or negative log likelihood).

One way to try to compute the log likelihood would be to just call the likelihood function you already wrote, and then take the log of the result. However, we don't do things that way. Why not?

To correctly compute the log likelihood, you will need to fill in the body of the `compute_log_likelihood` function with code that is specific to the purpose. Please use log *base 10* (see note below). *Hint*: remember that $\log xy = \log x + \log y$.

Note: For this lab, we ask you to use log base 10 because it has an intuitive interpretation. For any x that is a power of 10, $\log_{10} x$ is the number of places to shift the decimal point from 1 to get x. For example $\log_{10}$ 100 = 2, and $\log_{10}$ .01 is -2. Numbers falling in between powers of 10 will have non-integer logs, but rounding the log value to the nearest integer will still tell you how many decimal places the number has. Consider: what is the relationship between $\log_{10} x$ and the floating-point representation of x?

Now, what is the log likelihood of your random sequence of 500 characters?

Arrays in NumPy (optional)

Some of you have probably used the NumPy (`numpy`) library before. If you have, you can probably skip this section. If you haven't, and are planning to continue with numerical computing (including computational linguistics), it's a very useful package to know about---especially if you want to reuse or modify any of our code here!

There is a huge amount of stuff in NumPy but one of the most basic concepts, and one we used here in several places, is the NumPy `array` datatype. An array stores a sequence of items, like a list, except that all the items must have the same type (e.g., all strings or all integers). If the items are numbers, then the array can be treated as a vector. To see how it works, let's create some arrays and lists we can manipulate. (Note that we imported `numpy` as `np` at the top of our file)

```
l=range(4)
m=[1,0,2,3]
a=np.arange(4)
b=np.array([2,0,1,1])
c=np.array([2,3])
```

Now, try to predict what each of the following lines will do, then check to see if you are right. Note that a few of these statements will give you errors.

```
l
a
l+[1]
a+[1]
l+1
a+1
l+m
a+b
a+c
2*l
2*a
np.array(l)
a[b]
a[c]
a[m]
l[m]
sum(l)
sum(a)
np.product(c)
```

This section is just a tiny taste of NumPy, and even of arrays (e.g., we can create multi-dimensional arrays and use them as matrices with functions available for standard matrix operations). If you are going to be doing a lot of numeric programming in Python, we recommend spending the time to familiarize yourself with more of NumPy (and SciPy, another package for scientific computing in Python, which we will refer to in the Going Further section).

Sampling from a discrete distribution (optional)

You don't need to understand how most of the functions in this lab are implemented in order to do the previous parts of the lab. However, you may want to reuse/modify some of them in the future, and for that reason it is a good idea to understand how they work. First, make sure you understand the previous section on NumPy arrays. Then, look at the following two statements, which are used in the `generate_random_sequence` function. Can you figure out what these statements do, and how the whole function works? You may need to try calling `cumsum` or `digitize` yourself on some examples and/or look up the documentation.

```
np.cumsum(a)
np.digitize([.1, .7, 2.3, 1.2, 3.1, .3], a)
```

Going Further (for the ambitious)

1. Using either the distribution we gave you or one of your own choosing, generate a sequence of outcomes that is small enough to ensure you get some zero-count outcomes. Write a function to compute the Good-Turing estimate of the probability that the next outcome would be (any) previously unseen event (see textbook for definition of Good-Turing, we'll discuss it briefly in class). Compare this estimate to the actual probability of getting one of the unseen events on the next draw from the distribution. Does the quality of the estimate vary depending on things like the proportion of unseen events, the sample size, or the actual probabilities of the unseen events?

2. In some areas of computational linguistics that rely on developing sophisticated machine learning models, it may be necessary to generate huge numbers of random outcomes. As a result, the efficiency of the sampling function becomes very important. Usually library functions are written to be efficient, but according to the StackOverflow page[4] where I got the code that I modified to make `generate_random_sequence`, SciPy's built-in function for generating discrete random variables is much slower than (the original version of) the hand-built function here. Take a look at fraxel's code (second answer on the page; similar to mine) and EOL's code (fourth answer, using `scipy.stats.rv_discrete()`) and the comments below it. Use the `timeit` function to see whether the comments are correct: is the library function really slower, and by how much? (For a fair comparison, you may need to modify code slightly. Make sure you are not timing anything *other* than random sampling, i.e., you will need to pre-compute the list of values and probabilities rather than recomputing them each time you call for a sample.)

   You can go even further with this question by looking at the `numpy.random.multinomial()` function as well. It does not compute a sequence explicitly, instead returns only the total number of outcomes of each type. But, if that is all you care about, is this function more efficient than the other two?

   If you pursue this question carefully, please send your code and results! It will be especially interesting to compare if we get results from multiple people.

   In general, if you are interested in questions of program efficiency in Python, you may want to look at this page[5].