Lab for week 3: Working with probability distributions

Author: Sharon Goldwater
Date: 2014-09-01, updated 2015, 10-01

This lab is available as a web page[2] or pdf document[3].

Goals and motivation of this lab

This lab aims to build your intuitions about estimating probability distributions from data. It also provides example code, e.g., for generating bar plots and for sampling from a discrete probability distribution.

Some students are still Python beginners, so we have written most of the code already and have added a lot of explanatory comments. You'll have a chance to modify some of the code to help you understand it better.

If you are new to Python programming

This lab can be done stand-alone. However you may also want to go through last year's Lab 2[4] for some additional practice with Python and further heavily-commented example code. There are also instructions on how to access the Python help documentation. (We had a tutorial session this year instead of doing Lab 2, but the CPSLP instructor said he's happy to answer questions on the lab material. The solutions[5] are also available; we suggest working through as much as you can before looking at them.)

If you are experienced with Python

We have tried to keep the coding style here very simple, and have avoided some of the more advanced language features of Python to make the lab accessible to newcomers.

We would very much appreciate if you can help those around you who may be new to Python. Learning to explain technical concepts to others is an important transferrable skill, and solidifies your own knowledge as well.

If you finish early, the 'Going Further' section suggests some more advanced topics and efficiency issues to explore.

Preliminaries

First, create a directory for this lab inside your `labs` directory:

```
cd ~/anlp/labs
mkdir lab3
cd lab3
```

[1] http://creativecommons.org/licenses/by-nc/4.0/.
[2] http://www.inf.ed.ac.uk/teaching/courses/anlp/labs/lab3.html
[3] http://www.inf.ed.ac.uk/teaching/courses/anlp/labs/lab3.pdf
[4] http://www.inf.ed.ac.uk/teaching/courses/anlp/labs/lab2.html
[5] http://www.inf.ed.ac.uk/teaching/courses/anlp/labs/lab_solutions.html
[6] http://www.inf.ed.ac.uk/teaching/courses/anlp/labs/lab3.py
[7] http://stackoverflow.com/questions/11373192/generating-discrete-random-variables-with-specified-weights-using-scipy-or-numpy
[8] https://wiki.python.org/moin/PythonSpeed/PerformanceTips

Download the file lab3.py[6] into your `lab3` directory: From the Lab 3 web page[2], right-click on the link and select *Save link as...*, then navigate to your lab3 directory to save.

Next, open the file using your favorite text editor. If you don't have one, we suggest gedit.

```
gedit lab2.py &
```

If using gedit, you will want to set auto-indenting so that hitting <enter> will automatically insert enough spaces to line up your cursor with the line above (useful for Python coding). If you need to indent further, e.g. starting a loop, you will need to type 4 spaces.

Set auto-indent using the menu Edit->Preferences, go to the Editor tab and tick "Enable automatic indentation".

Running code in iPython

There are different ways to run Python code. The first way you probably learned was to type directly into an *interpreter* such as Idle. In this class, we will use the iPython interpreter (with Python version 2.7). To start the interpreter, just type the following into a terminal:

```
ipython
```

You should get a `In [1]:` prompt, and can now do things like this (where the `Out` line is the interpreter's response):

```
In [1]: 3+4
Out[1]: 7
```

Later, when you want to exit iPython, type `exit()`.

Our code is saved in a file and we don't want to cut and paste code into the interpreter all the time. iPython gives us with a way to do the same thing without pasting. Try typing this into the interpreter:

```
%run lab3.py
```

(Note: commands that start with '%' are specific to iPython, and may not work in other Python interpreters).

You should see a plot pop up on the screen. We will come back to the plot in a moment, so for now just close the plot window (otherwise you will not be able to type anything more into the interpreter).

You can also run the code from a terminal window, by typing:

```
python lab3.py
```

You should see the same output and plot you got before (again, close the plot window for now). But, if you use `%run` inside iPython, you have access to all the variables and functions defined in the code, which is not true if you run the code in the terminal.

To see how this works, go back to the interpreter where you typed `%run lab3.py` and type:

```
distribution
```

Note: you can use tab completion in iPython just like in a UNIX terminal. In fact you can also use lots of other UNIX commands like `ls`, `pwd`, *etc. Again, this functionality is specific to iPython and may not work in other interpreters.*

Examining the initial output

When you type `distribution`, the interpreter should output the value of the `distribution` variable. It is a dictionary that represents a probability distribution over different characters.

Look at the top of the main body of code where the probabilities of each outcome in `distribution` are defined, and compare them to the values output by the interpreter. You might see some tiny differences. For example, the probability of `a` is defined as 0.2, but might be printed as 0.20000000000000001. Other results later in the lab might also be slightly different than you expect. This is because tiny rounding errors can happen when the computer converts numbers from base 10 (which we use) to base 2 (which the computer uses internally) and back again. Most programming languages 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.

Now, type `%run lab3.py` again and look at the plot that pops up. The plot has two sets of bars, "True" and "Est". "True" plots the probabilities of the different outcomes defined by the `distribution` variable. "Est" will eventually plot the probabilities as estimated from some data. Right now the values of the bars are incorrect.

Close the plot and look now at what was printed out in the interpreter window. By looking at the printout and the code, 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

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?

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 two distributions. One is the true distribution over characters. The other is 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 there 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 can also 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?

Introduction to NumPy (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 (including on your homework assignment), and for that reason it is a good idea to understand how they work. Several of our functions use functions and datatypes from the NumPy (`numpy`) library, which is extremely useful for doing numerical computing. 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)
np.cumsum(a)
np.digitize([.1, .7, 2.3, 1.2, 3.1, .3], a)
```

We included the last two statements because they are used in the `generate_random_sequence` function. If you haven't already, see if you can now understand how that function works.

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).

Going Further

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. 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. Some algorithms in NLP and machine learning require generating 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[7] 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[8].