Week 3 Lab: Working with probability distributions

Author: Edoardo M. Ponti, Sharon Goldwater, Ida Szubert, Henry S. Thompson

Date: 2014-09-01, updated 2015-10-01, 2016-09-29, 2017-09-25, 2018-08-20, 2019-09-20, 2022-09-30, 2023-09-17

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 most of the code has been written already and we 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 by itself. However you may also want to do the 2019's Lab 2[2] for some additional practice with Python and further heavily-commented example code. There are also instructions on how to access the Python help documentation. 2019's Lab 2 solutions[3] are also available, but you should work through as much as you can before looking at them.

If you are experienced with Python

We have kept 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.

[1] http://creativecommons.org/licenses/by-nc/4.0/.

[2] http://www.inf.ed.ac.uk/teaching/courses/anlp/labs/l02_2019/anlp_l02_2019.html

[3] http://www.inf.ed.ac.uk/teaching/courses/anlp/labs/l02_2019/anlp_l02_2019_sol.html

[4] https://git.ecdf.ed.ac.uk/anlp/course_materials/-/raw/main/current/labs/lab2/lab2.py?inline=false

[5] http://stackoverflow.com/questions/11373192/generating-discrete-random-variables-with-specified-weights-using-scipy-or-numpy

[6] https://wiki.python.org/moin/PythonSpeed/PerformanceTips

Please 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 lab2
cd lab2
```

Download the file lab2.py[4] into your `lab2` directory: click on the link, then navigate to your `lab2` directory to save.

Running code in Visual Studio Code

As you may already know, there are different ways to run Python code. In this class, we will use the Visual Studio Code IDE (VSCode for short). To start the IDE, just click on its icon. You'll get a new full-screen window. To open a terminal inside VSCode click `Terminal > New Terminal` or use the shortcut `Ctrl+Shift+``. In the terminal, run the command `python3`. In the terminal you should now see a >>> prompt, and can now do things like this (where the second line is the interpreter's response):

```
>>> 3+4
7
```

Now open `lab2.py` in the editor (`File > Open...`) and run the code in interactive mode by right-clicking anywhere on the file, then selecting `Run Current File in Interactive Window`.

What happened? Can you see which lines of code in `lab2.py` generated the text output you got?

Alternatively, you can run the code by typing:

```
python3 lab2.py
```

in the terminal window. If you get a `file not found` error, you may be in the wrong directory: try typing `cd ~/anlp/labs/lab2` in the terminal and then try again. However, this will not give you access to the value of the variables in the file.

Now uncomment the line `plot_distributions(distribution, str_probs)` near the end of the file. If you run the code using `python3 lab2.py` you'll see no difference, because your change was not saved. But if you run in interactive mode, the displayed version of the file will be run, so you will now see a plot in the output. Remember to save your changes before running the file if you use `python3 lab2.py`.

(Note: the plot generated by `plot_distributions` will be incorrect at the start. You will have to fill in the code for the functions it relies upon during the course of this laboratory to make it work properly!)

You can also run individual commands in the interpreter of the interactive window. For example, to produce a histogram of some counts we generated from a uniform distribution, you could type in:

```
values = list(range(1, 10))
counts = [16, 8, 12, 12, 11, 10, 9, 12, 10]
plot_histogram(values, counts)
```

To see that you can also access the variables, type `distribution` in the interpreter of the interactive window. What is the value of this variable?

In the remainder of the lab, we provide commands which you can run in the interpreter of the interactive window of Visual Studio Code.

Examining the initial output

When you type `distribution` into the interpreter, the output is 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, run the Python file again and look at the plot. 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.

Look now at what else was printed out in the interpreter window. By looking at the printout, the variable explorer window, 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

After you fix the `normalize_counts` function, rerun the whole file to 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 of the kind of estimate produced here?

Now look at the plot comparing the true and estimated distributions. Notice that there are several letters that have very low probability under the true distribution. Due to random chance, some of them will occur in the randomly generated sequence and some will not.

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 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? Can you get estimates that no longer include zero probabilities?

Suppose we're estimating unigram probabilities of words (not characters) from a natural language corpus. Would it be possible to adjust the sample size to avoid zero probabilities while still using the same kind of probability estimation you have here? (That is, could you make the corpus big enough to avoid zeros?) 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 back the code 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.

Hint: the model just tells you the unigram probability of each character, and the likelihood is P(data | model): that is, the probability of the full sequence of characters using this unigram model.

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 each 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 1000 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*.

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, 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 1000 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), so it is good to understand how they work. Our code uses functions and datatypes from the NumPy (`numpy`) library. NumPy contains many things that are useful for doing numerical computing. One of its most basic parts, which 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 likely to do a lot of numeric programming in Python, we recommend familiarizing 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. Do you run into any problems? Compare the GT 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. As of Python 3.7, `dict` and all classes derived from it are guaranteed to preserve insertion order. Try removing the calls to `sorted` in the `print` calls at the end of the lab and see if everything still works as expected. What about the calls to `sorted` in `plot_distributions`? Try removing *them* and see if things are still OK. (Which version of Python is default on DICE?)

   Even if you're using Python 3.7, removing these calls has a downside: it renders the code no longer backwards-compatible. How might you ensure that a version without the calls to `sorted` doesn't fail silently when run on earlier versions of Python?

3. Some algorithms in NLP and machine learning require huge numbers of random outcomes, so the efficiency of the sampling function is very important. Usually library functions are written to be efficient, but according to the StackOverflow page[5] 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 if the comments are correct: is the library function really slower, and by how much? (For a fair comparison, you may need to modify your code slightly. Make sure you are not timing anything *other* than random sampling: you will need to pre-compute the list of values and probabilities rather than recomputing them each time you call for a sample.)

   Now consider `numpy.random.multinomial()`. 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 program efficiency in Python, you may want to look at this page[6].