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/-/blob/main/2023/labs/lab2/lab2.py

[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 iPython

As you may already know, there are different ways to run Python code. In this class, we will use the Spyder IDE, based on the iPython interpreter. To start the IDE, just type the following into a terminal:

```
spyder3 &
```

You'll get a new full-screen window, with a text editor on the left and two smaller sub-windows on the right: A **Usage** note on top and an iPython console below.

Using the tabs at the bottom edge of the top one, change it to the **Variable explorer** view.

In the iPython console you should see 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
```

Now open `lab2.py` in the editor (File -> Open...) and run the code by clicking the green arrow button on the main menu bar. If a dialogue window pops up the first time you click the arrow, please leave all options as default and click `Run` in that 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:

```
%run lab2.py
```

in the iPython console window. If you get a `file not found` error, you may be in the wrong directory: try typing `cd ~/anlp/labs/lab2` in the interpreter and then try again.

(Note: commands that start with '%' are specific to iPython, and may not work in other Python interpreters. Also, UNIX commands like `cd` *and* `ls` *work in iPython, but not all interpreters.)*

Now uncomment the line `plot_distributions(distribution, str_probs)` near the end of the file. If you run the code using `%run lab2.py` you'll see no difference, because your change was not saved. But if you use the green arrow button, the file will be automatically saved and the new version will be used, so you will now see a plot in the output. Remember to save your changes when using `%run`.

(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. 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)
```

After running a code file in Spyder, you have access to all the variables and functions defined in it. In fact we just used one of them: the `plot_histogram()` function.

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

Variables are also displayed in the variable explorer window. Try double clicking on the `distribution` variable there to see what is displayed.

In the remainder of the lab we provide commands which you can run in the interpreter window of Spyder.

Getting help in iPython (and in general)

Before moving on, let's make sure you know how to get help. There are several ways to do this:

1. Simply type `help()` at the iPython interpreter prompt; you can then type a module name (e.g., `string`) and you will get some documentation. Actually a *lot* of documentation, but maybe not the most helpful kind. So, it may be better to try options 2 or 3. (Hit `q` to exit help.)

2. To get a list of all the methods available for a particular variable, you can use tab completion in iPython. This can also work to tell you all the methods available for particular datatype, like a string. For example, type the following two lines into the interpreter, but don't hit <enter> after the second line:

   ```
   ss = "a string"
   ss.
   ```

   Now hit the <tab> key on your keyboard. You should see a list of all of the methods you can call on `ss`, which is to say, all the methods for strings.

3. You can use the `help` tab in the upper right window of spyder. Notice that you can even get help for functions defined in your own code, if you have written documentation for them. For example, click on the call to `plot_histogram`

further back up in the console window, then type `Ctrl+i` and see what appears in the Help window. Or just type `help(plot_histogram)` in the interpreter. Notice where the documentation comes from in `lab2.py`.

4. Often it's most appropriate to just search the Internet, if you want more general information, don't know the name of the function you're looking for, want examples of how to use particular functions, or whatever.

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, type `%run lab2.py` 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 earler 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].

Week 3 Lab: Answers and explanations

Author: Edoardo M. Ponti, Sharon Goldwater

Date: 2014-09-01, updated 2017-09-30

Examining the initial output

- `str_list` is a list containing a sequence of random outcomes from the distribution we defined.
- `str_counts` is a dictionary containing the counts of each outcome in that sequence.
- `str_probs` is a dictionary which is supposed to contain the estimated probability of each outcome, but currently also just contains the counts.

Normalizing a distribution

`normalize_counts` is supposed to normalize a set of counts or weights to create a probability distribution that sums to one, by dividing each count by the sum of all counts. At the moment, it does nothing, just returning the same counts that are passed in.

See our answer code[2] for two possible ways to write the correct function, one with a list comprehension and one with a for loop.

You could implement an error check on this function to make sure the sum of the values in the dictionary is 1, in the same way we did on the true distribution (lines 169-171 in our answer code). You could go further and make sure each value is between zero and one, although this is probably better implemented as a check on the *input* of the function, to make sure someone doesn't pass in crazy values.

Comparing estimated and true probabilities

The type of estimate produced here is called a relative frequency estimate or maximum-likelihood estimate. In general, items that have low probability will be estimated incor-

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

[2] https://git.ecdf.ed.ac.uk/anlp/course_materials/-/blob/main/2023/labs/lab2/lab2_sol.py

rectly because if they happen to appear in the observed data, their probability is usually overestimated; whereas if they happen not to appear their probability is underestimated.

Effects of sample size

The estimated probabilities should be more accurate as the amount of data increases. For the particular distribution we used here, you will be able to set the sample size large enough so that no zero probabilities are estimated (with, say, 500 or certainly 1000 samples). We could likely do the same with natural language if our distribution was over *characters*, as here, because there is only a finite number of characters. However, it isn't possible to do that with a distribution over *words* in natural language because there is an infinitely long tail of words with arbitrarily low probabilities, so no matter how much data we have, there will be some words that are theoretically possible but so improbable that they don't appear in the data. (This is even more true if we consider higher-order n-grams over words.)

Computing the likelihood

To see two different implementations of the likelihood function, see our answer code[2].

The exact likelihoods you get will depend on the particular random sequence you generated, but you should find that (for a sequence of 50 characters) the likelihood of the true distribution ranges roughly from 1e-30 to 1e-42, and the likelihood of the estimated distribution ranges roughly from 1e-28 to 1e-39. However, although you can get sequences that vary in probability by 10-12 orders of magnitude, you will *always* find that the probability under the true distribution is *lower* than the probability under the estimated distribution, usually by about three orders of magnitude. This is precisely because the estimated distribution is the maximum-likelihood estimate, i.e., it is the estimate with the highest possible likelihood.

Log likelihood

Note The likelihood printed out for a sequence of 1000 characters is 0 because the actual probability is so small that the computer is unable to represent its value. (This problem is called *numerical underflow* and is something we need to watch out for when multiplying together many very small probabilities.)

We can't fix the problem by taking the log of the output of the likelihood function, because the output is already zero and taking the log of zero is undefined.

To see how the log likelihood should actually be computed, see our answer code[2].

The log likelihood of a random sequence of 1000 characters will again vary, but you should be seeing numbers roughly in the range of -700 to -600.

Introduction to NumPy (optional)

Most answers should be self-explanatory.

The following will give errors:

```
l+1
a+c
l[m]
```

These two treat the `b` (or `c`) array as a set of indices, and return the items in `a` corresponding to those indices:

```
a[b]
a[c]
```

This does the same thing, by implicitly turning `m` into an array:

```
a[m]
```

This one creates a new array, where the item at index `i` is `sum(a[:i+1])`:

```
np.cumsum(a)
```

And `digitize(x,y)` assumes `y` is a set of "bins", and tells you which bin (by index `j`) each `x[i]` falls into, i.e., where `y[j-1] <= ``x[i]` < `y[j]`.

Lab for week 7: Probabilistic Parsing

Author: Henry Thompson
Author: Bharat Ram Ambati
Author: Sharon Goldwater
Author: Shay Cohen
Date: 2016-10-30 (modified 2023-09-19)

Goals and motivation of this lab

In this lab we explore probabilistic phrase structure grammars and chart parsers which use such grammars. We are using data from nltk that inclues 3914 treebanked sentences from the Wall Street Journal.

As always, we have written most of the code for the lab 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.

Preliminaries

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

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

Download the files lab3.py[2], lab3_helper.py[3] BetterICP.py[4] into your `lab3` directory: From the Lab 3 web page[5], right-click on the link and select *Save link as...*, then navigate to your lab3 directory to save.

Open the file with gedit and start up ipython:

```
gedit lab3.py &
ipython
```

Running the code

If you haven't done so yet, you will first need to download the relevant nltk data:

[1] http://creativecommons.org/licenses/by-nc/4.0/.
[2] https://git.ecdf.ed.ac.uk/anlp/course_materials/-/blob/main/2023/labs/lab3/lab3.py
[3] https://git.ecdf.ed.ac.uk/anlp/course_materials/-/blob/main/2023/labs/lab3/lab3_helper.py
[4] https://git.ecdf.ed.ac.uk/anlp/course_materials/-/blob/main/2023/labs/lab3/BetterICP.py
[5] https://git.ecdf.ed.ac.uk/anlp/course_materials/-/blob/main/2023/labs/lab3/lab3.pdf
[6] https://gist.github.com/nlothian/9240750

```
%import nltk
%nltk.download('treebank')
```

You can then run this week's lab as follows:

```
%run lab3.py
```

It should print the first parsed sentence and the productions in it.

Following lab 5, we use the Penn Phrase Structure Treebank for this lab as well. This data consists of around 3900 sentences, where each sentence is annotated with its phrase structure tree. We use NLTK libraries to load this data.

(Don't forget to do:

```
%run lab3.py
```

after you've done some edits, in order to get your new function definitions.)

Probabilistic Phrase Structure Grammar (PCFG)

Probabilistic Phrase Structure Grammars (PCFGs) are Phrase Structure Grammars, where each production has a probability assigned to it. Consider the toy PCFG grammar. In NLTK, the data type of this grammar is `ProbabilisticGrammar`:

```
# Grammatical productions.
S -> NP VP [1.0]
NP -> Pro [0.1] | Det N [0.3] | N [0.5] | NP PP [0.1]
VP -> Vi [0.05] | Vt NP [0.9] | VP PP [0.05]
Det -> Art [1.0]
PP -> Prep NP [1.0]
# Lexical productions.
Pro -> "i" [0.3] | "we" [0.1] | "you" [0.1] | "he" [0.3] | "she" [0.2]
Art -> "a" [0.4] | "an" [0.1] | "the" [0.5]
Prep -> "with" [0.7] | "in" [0.3]
N -> "salad" [0.4] | "fork" [0.3] | "mushrooms" [0.3]
Vi -> "sneezed" [0.6] | "ran" [0.4]
Vt -> "eat" [0.2] | "eats" [0.2] | "ate" [0.2] | "see" [0.2] | "saw" [0.2]
```

Each production has a probability assigned to it. Change the probability of the production `NP -> NP PP` from 0.1 to 0.01. Re-run the code. What is the error given by the code?

Change the value back before continuing!

Comment out the three lines that are printing out the first sentence and its productions.

PCFG Parser:

We made a simple class called `BetterICP`. It uses NLTK's `InsideChartParser` module which is a probabilistic chart parser (`help(nltk.InsideChartParser)` for more details).

BetterICP class has a method `parse` which parses a sentence and prints all possible parses. This method takes three arguments. First argument is `tokens`, which is a list of words in the sentence. Second argument is `notify`. `notify=True` will print each parse as it is found without waiting until the end of the process. Third argument `max` defines the number of possible parses to be printed. The parsing process will stop once `max` number of parses have been found.

Un-comment the lines in `lab3.py` which initialise the `BetterICP` class with our simple grammar and call the method `parse`, then reload.

The figures to the right of the parse gives the probability, the total cost of the tree and the cost of the spanning `S` edge.

Run the parser on sentences "he sneezed" and "he sneezed the fork". What is the output for each of these sentences ?

Ranking parses

Uncomment the next two parses, so the PCFG parser runs on the sentences "he ate salad with mushrooms", and "he ate salad with a fork". Note that the parses are ranked based on their probabilities (high to low)/costs(low to high).

Observe the PP-attachment (preposition phrase attachment) and identify which VP production is preferred. In the best parse, is PP attached to NP (noun phrase) or VP (verb phrase) ?

Edit the grammar and switch the probabilities of the two rules involved. Re-load, with appropriate commenting so that only sentence2 and 3 are parsed. What changes ?

Turn tracing on, by adding the following *before* the `sppc.parse` lines in `lab3.py`:

```
    sppc.trace(1)
```

and rerun the three parses you've done so far.

Can you see how the order in which edges are added is determining what analysis gets found first?

Can you see now the parse is effectively breadth-first? Try to spot where a shorter edge that will eventually not be part of the best overtakes a longer one that will.

Try changing the rule probabilities back to where they started, and watch again.

Working with the real grammar

In the above sections, we worked with a toy grammar created by hand. For example, consider `Prep -> "with" [0.7] | "in" [0.3]` production in our toy grammar. This means that out of all the possibilities there are only two prepositions in the language, and that e.g. when generating they should be chosen with the associated probabilities.

In this section, we shall work with the grammar from the treebank. `psents` contains all the parsed sentences in the treebank. `prods = get_costed_productions(psents)` gives all the productions in the treebank with their costs.

And with the treebank-based grammar, as we see from `get_costed_productions`, the cost is not some made-up number, but the -base-2-log of the maximum-likelihood estimate of the probability of each production, based on its frequency in the treebank.

Create a PCFG by uncommenting the `prods=...` and `ppg=PCFG(Nonterminal('NP'), prods)` lines in `lab3.py` and re-loading.

The first argument to `PCFG` is the start symbol and the second argument is the productions with their costs. The first argument *just* determines the start symbol for parsing purposes, it doesn't restrict the productions that are included. That is, *all* the productions in `prods` will be in `ppg`.

Now try `sprods = ppg.productions(Nonterminal('S'))`, which gives a list of all the productions that start with 'S'. How many productions are there in the treebank that start with 'S'?

There is a useful complete listing of the tagset used in the Penn Treebank[6] online.

Print the first 10 productions in this list to get a feel of the kind of productions used in the treebank, and why we're going to start with NP for our little experiment.

Note also that punctuation symbols do appear both in the trees and the grammar. Many common punctuation marks occur as pre-terminals for themselves (and sometimes some close friends), that is, for example, `, -> ','` and `. -> '.' | '?' | '!'`, but in order to avoid confusion with the tree displays, this does *not* apply to brackets, which use three-letter acronyms, as follows:

```
-LRB- -RRB- -RSB- -RSB- -LCB- -RCB-
  (     )     [     ]     {     }
```

Finally, irritatingly, the `-digits` which regularly appear at then end of some tags are coreference markers which pair up e.g. a relative pronoun and a subsequent gap, as in:

```
(NP
  (NP (DT the) (NN executive) (NNS functions) )
  (SBAR
    (WHNP-2 (WDT that) )
            (S
              (NP-SBJ (DT the) (NNP Confederation) (NNP Congress) )
              (VP (VBD had)
                (VP (VBN performed)
                  (NP (-NONE- *T*-2) ))))))
the executive functions that the Confederation Congress had performed
```

Un-comment the following lines and run the code:

```
ppc=BetterICP(ppg,1000)
print "beam = 1000"
ppc.parse("the men".split(),True,3)
```

What happened? Why do you think that is?

Now comment out the last two of the above three lines, and remove the comments from the *next* three lines, to widen the beam to 1075, and run again.

What's surprising about the results, given our discussions in class about the relative frequency of 'the' as DT and 'the' as JJ??

Uncomment three more lines to try width 1200 (but *do not* uncomment the `trace` lines).

You should now be able to see three different parses for this simple noun phrase. These parses are ranked based on increasing costs. The number printed with the parse tree is the cost. So, the lower the number, the higher the probability of that parse. We're *still* not seeing the simple `DT NNS` structure we might expect.

Finally uncomment on the last three lines to widen the beam rather a lot more, and run again.

What has finally happened?

How could *increasing* the beam width have improved the results the way it did?

Finally turn tracing on by uncommenting the:

```
ppc.trace(1)
```

line, and arranging the other comments so only the beam-width 1900 parse will be attempted. *Do not* run this from inside ipython (or use Control-C to interrupt it if you do :-).

From the terminal command line, do:

```
> python lab3.py > ptrace.txt
```

Use:

```
> less ptrace.txt
```

to look at the results, in the form of a line for each edge. The ASCII art in the left-hand column is a bit hard to make sense of at first, but the remaining columns are easy:

- the interval covered (e.g. "[1:2]" for an edge between vertex 1 and vertex 2)
- the word or dotted rule on the edge -- if the `*` (dot) is at the end, the edge is inactive, otherwise active
- the cost

You can also use 'grep' to find what's happening to the two rules we care about, as follows:

```
> egrep -n 'the|men|NP -> ([*]\s)?(DT|JJ)\s([*]\s)?(NNS|NP)(\s[*])?\s*\[' ptrace.tx
```

Note that tracing at this level shows edges as the come *off* the agenda and into the chart, so we can't see exactly why the result we're looking for never appeared.

To see more detail edit the file again to switch to beam-width 1200, and up the level of detail in the trace to show pruning:

```
ppc.trace(3)
```

Again, run from *outside* ipython:

```
> python lab3.py > ptrace2.txt
```

Use the `less` and the `egrep` command on `ptrace2.txt` this time to see what went wrong. Can you see why we're 'losing' the correct answer? Where are the edges coming from that are filling up the beam?

You may want to try exploring `ptrace2.txt` with `less -N` and its `/regexp` command which skips forward to the next occurrence of that regexp. Also note that you don't have to quit and type `less -N` to back up and look for something else, you can just use `b` to *b* ack up a page, and `g` to *g* o back to the beginning.

If you're more comfortable searching inside an editor, you can just load `ptrace2.txt` into an editor and do that.

Aside from the speed of our old DICE machines, what's wrong with the grammar that's causing so much wasted effort? (Hint: A simpler answer than the ones we've looked at in class is all that's needed.)

Going Further (for enrichment):

1. Further to the last question, try sorting `prods` in increasing order of cost -- what stands out about many of the lowest cost productions? (Hint: what's the easiest way for some production to have cost 0?)

You can check your guess by referring to the counts you get from:

```
pd=production_distribution(psents)
```

using the supplied `production_distribution` function, similar to the one we used last time, but which just counts productions without separating lexical from non-lexical.

2. Draw the 'correct' phrase structure tree for the sentence "he ate salad with a fork" manually according to the toy grammar. Compare your tree with the output trees when parsed with `sppc`.

To draw your correctly bracketed structure, represented as a string, use NLTK commands as shown below, using your own string:

```
treeStr = "(S (NP (Pro he)) (VP (Vt ate) (NP (N salad) ) ) )"
tree = nltk.tree.Tree(treeStr)
tree.draw()
```

3. Update the grammar to handle imperative sentences such as "eat the salad".

4. Take some sample sentences, draw their correct phrase structures, get the first best output from the PCFG parser (set the beam width as low as you can!) for these sentences and evaluate them using parseval.

5. Create a text file called "input.txt" and write three sentences "he ate salad", "he ate salad with mushrooms", and "he ate salad with a fork" into this file, where each sentence is in a separate line. Write a function which reads sentences from this file, parses them and print the output into "output.txt" file.

Extra credit: Recompute the productions while ignoring all the `-digit*` suffixes on Nonterminal symbols which occur throughout the treebank. How many productions now?. Rebuild the grammar. Re-run some tests -- any changes?

Even more extra credit: Edit the code to a) actually store costs, not probabilities and b) use a figure of merit as discussed in class rather than the simple sum of costs. This *should* fix the problem observed at the end of the lab above.

Probabilistic Phrase Structure Grammar (PCFG)

- What is the error given by the code ?

  Productions for NP do not sum to 1. Sum of the probabilities of all the productions for a particular non-terminal should be 1.

PCFG Parser:

- Run the parser on setennces “he sneezed” and “he sneezed the fork”. What is the output for both these sentences ?

  For the first sentence we get its parse tree “(S (NP (Pro he)) (VP (Vi sneezed)))” as the output. For the second sentence, we get none as there is no parse for this sentence.

Ranking parsers

- In the first best parse, is PP attached to NP (noun phrase) or VP (verb phrase) ?

  PP is attached to NP in the best-first parse for both the sentences.

- Edit the grammar and switch the probabilities of the two rules involved. What changes ?

  Change VP -> Vi [0.05] | Vt NP [0.9] | VP PP [0.05] to VP -> Vi [0.05] | Vt NP [0.05] | VP PP [0.9].

  This makes the parser to prefer VP -> VP PP instead of VP -> Vt NP and hence the PP is attached to VP this time rather than NP.

Working with the real grammar

- How many productions are there in the treebank that start with ‘S’ ?

  The number of productions that start with ‘S’ can be identified using the command `len(ppg.productions(Nonterminal('S')))` which is 772.

- What happened (first, beam = 1000, run)?

  ```
  Nothing.  No parses.  Beam too narrow.
  ```

- What’s surprising about the results (second, beam = 1075, run)?

  ```
  Parses now, but with ''(JJ the)'' instead of ''(DT the)'' and
  also spurious ''(NP (NNS men))'' instead of just ''(NNS men)''
  ```

- What has finally happened at beam width 1950?

  ```
  Lower-cost, better-looking results have finally made it to the
  top
  ```

- How could *increasing* the beam width have improved the results the way it did?

  When beam=1075, best-first parse has JJ as tag for ‘the’. At 1200, we get DT for ‘the’ but the wrong parse for ‘men’. When beam=1950, best-first parse we finally get the ‘right’ analysis. My guess is that the `NP -> Det NNS *`, with *three* costs summed together, was too expensive to make the beam early on, when there were 100s and 100s of cheap single-word (one- and two cost-sums) to push it out, but once most of those had died, then the relatively expensive, but not *quite* expensive enough to get kicked out, `NP -> DT * NNS`, could come off the agenda and get through.

```
Note that as we increased the beam width the parser got slower

Looking carefully at the trace output, particularly the
''trace(3)'' version, with the egrep pattern, seems to confirm
this analysis.
```

- Aside from the speed of our old DICE machines, what's wrong with the grammar that's causing so much wasted effort?

  Each non-terminal has between a few hundreds and thousands of productions. There are around 1500 productions that start with NP, compared to 4 productions in our toy grammar. The parser is much slower as it has to check thousands and thousands of rules. Further more, as many of these productions appear only once, their cost is 0 (probability 1) so they fill the agenda and push reasonable, but non-0-cost, possiblities off the agenda because of the beam width.

Going Further:

\1. Further to the last question, try sorting prods in increasing order of cost – what's stands out about many of the lowest cost productions?

```
Most of the low-cost productions are the cases where the
production occured only once in the entire data.
```

2. Draw the phrase structure tree for the sentence "he ate salad with a fork" manually using the toy grammar. Compare your tree with the output trees::

   treeStr = "(S (NP (Pro he)) (VP (VP (Vt ate) (NP (N salad))) (PP (Prep with) (NP (Det (Art a)) (N fork)))))" tree = nltk.tree.Tree.fromstring(treeStr) tree.draw()

\3. Update the grammar to handle imperative sentences like "eat the salad".

```
Adding a production "S -> VP" can handle imperative sentences.
```

4. Take some sample sentences, draw their correct phrase structures, get the best-first output from the PCFG parser (set the beam width as low as you can!) for these sentences and evaluate them using parseval.

   See lecture notes on how evaluate using parseval.

5. Create a text file called "input.txt" and write three sentences "he ate salad", "he ate salad with mushrooms", and "he ate salad with a fork" into this file, where each sentence is in a separate line. Write a function which reads sentences from this file, parses them and print the output into "output.txt" file.

   See `lab3-sol.py`.

Lab for week 9: Text Classification

Author: Ida Szubert
Author: Sharon Goldwater
Author: Henry S. Thompson

Goals and motivation of this lab

The idea of text classification is to assign labels to pieces of text, based on their content. Examples include:

- Classifying news articles by topic: sport, politics, fashion etc.
- Classifying student essays according to grade categories.
- Classifying tweets by language.
- Classifying books as potentially interesting or not for a particular customer.

In this lab we will demonstrate simple generative (Naive Bayes) and discriminative (logistic regression) classifiers, first on a toy data set and then on a more realistic text collection. Along the way you will be introduced to some powerful and widely used Python libraries for numerical computation (numpy) and machine learning (scikit-learn).

We want you to see some of the practical differences between the two methods and the types of features used in text classification. You'll also learn to experiment with feature design and practice evaluating results of your experiments.

Preliminaries

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

```
cd ~/anlp/labs
mkdir lab4
cd lab4
```

Download the file lab4.zip[2] and unzip it in your `lab4` directory. You should find `lab4_toy.py`, `lab4_toy-sol.py`, `lab4_big.py`, and `lab4_helper.py`.

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

[2] http://www.inf.ed.ac.uk/teaching/courses/anlp/labs/l04/lab4.zip

Two paths through this lab

This lab uses the `numpy` library. If you are already familiar with `numpy`, you have the option to implement parts of the lab yourself (extracting feature counts and implementing the core of Naive Bayes). If you and your partner agree to do this, you should start from **lab4_toy.py** and **do the steps marked (Opt)**.

Otherwise, we recommend that you just start from **lab4_toy-sol.py** and **skip the steps marked (Opt)**. However, you may want to come back later to look at the code in more detail. Looking at other people's code is a great way to learn more about programming.

Toy dataset

To start, we will explore a simple binary classification problem with an artificial dataset.

Start up Spyder and open either `lab4_toy-sol.py` or `lab4_toy.py` (see "two paths" above).

You should see the data consisting of 5 training and 4 test sentences. Each sentence describes a company, and the descriptions fall into two classes: *restaurants* (1), or *furniture stores* (0).

Look at the labelled training sentences. Which words do you think might be indicative of each class?

In addition to the words in the sentences, we will also consider the POS tags and lemmas.

Feature extraction

For the purposes of classification we need to represent a document as a collection of numerical features. Transforming a document into a vector of numerical features is called *vectorization*.

The simplest features to use for text are just the frequency counts of words, i.e., the "bag-of-words" representation.

(Opt) Complete the function `encode_token_freq()`, which produces a bag-of-words representation for a collection of documents, given a vocabulary.

Now, run the file and then use the following line to extract feature vectors for each sentence, along with the vocabulary for the word features:

```
(train_word_features, word_vocab) = vectorize(train_x_data)
```

The first return value `train_word_features` is a matrix, where each row holds the vector of features for the corresponding sentence in the data set.

What should be the count for word *fresh* for the first sentence? You can check that the feature vector agrees with your answer by inspecting the entry corresponding to the first sentence and the appropriate feature:

```
i = word_vocab.index("fresh")
train_word_features[0][i]
```

In addition to word counts, we can also extract counts of POS tags and lemmas:

```
(train_pos_features, pos_vocab) = vectorize(train_x_pos)
(train_lemma_features, lemma_vocab) = vectorize(train_x_lemma)
```

From `train_pos_features`, how can you get the feature count of the POS tag 'NNS' in the third training sentence?

We'll focus on the word-based representation for the remainder of the lab, but in the Going Further section you could consider tags and lemmas again if you want.

To encode the test data as feature vectors, run:

```
test_word_features = encode_token_freq(test_x_data, word_vocab)
```

Generative modelling: Naive Bayes

First, to remind yourself about Naive Bayes classifiers, answer the following questions:

- How is $P(class|features)$ computed in a Naive Bayes classifier? (Look back at the lecture slides if you can't remember).
- In this toy data set, what is the prior probability of each class if we use Maximum Likelihood Estimation?
- What is the equation to compute the feature probabilities using add-alpha smoothing?

(Opt) Complete the function `feature_probabilities()`, which estimates the class-conditional log probabilities of the features using add-alpha smoothing, and the function `nb_posterior_log_probability()`, which computes $logP(class|observation)$.

Run your NB classifier by calling:

```
nb_model = train_naive_bayes(train_word_features, train_y_data, 0.1)
nb_classify(test_word_features, nb_model)
```

Verify that you get the same results as the Naive Bayes classifier implemented in the `sklearn` library, using the same value of alpha:

```
nb_sk = MultinomialNB(alpha=0.1).fit(train_word_features, train_y_data)
nb_sk.predict(test_word_features)
```

Find the 3 highest probability words for each class using the `most_probable_features()` function, which is defined in `lab4_helper.py`:

```
most_probable_features(nb_model, word_vocab, categories, 3)
```

Do the results seem reasonable? Do you think those features are salient and discriminative? (You might already see a problem with looking at the most probable features. If not, it may become more clear once you get to the larger dataset.)

Discriminative model: Logistic regression

Discriminative classifiers, such as logistic regression, model $P(class|observation)$ directly, without the decomposition into conditional and prior probability. Instead of modelling all of the classes, logistic regression estimates the boundaries between the classes in the feature space.

Train and test an LR classifier:

```
lr_sk = LogisticRegression().fit(train_word_features, train_y_data)
lr_sk.predict(test_word_features)
```

The function which describes the boundary between the classes is a linear combination of weighted features. We can look at the *weights* of the trained LR model to find the features that are the most influential:

```
most_influential_features(lr_sk, word_vocab, categories, 6)
```

Are they similar or different to the most probable features of each class in the Naive Bayes model? Would you expect them to be similar? Why or why not?

Bigger dataset

For something much more realistic in both size and variability, let's look at a database of 12000 consumer complaints. The are four topics of complaints represented in this set: *credit card*, *bank account or service*, *student loan*, and *consumer loan*.

Your task will be to run a Naive Bayes and logistic regression classifiers using a variety of features. For this part of the lab we will be using sklearn implementations of classifiers and vectorization functions.

Reset the interpreter by running:

```
%reset
```

Open `lab4_big.py` in the editor and run it. This will load the data and extract a simple bag-of-words representation for the documents in the training and test sets, and print out how many features were extracted. (See the code under "Feature extraction" near the top of the file.)

Now train and test Naive Bayes and Logistic Regression classifiers:

```
nb_model, nb_predict = nb_fit_and_predict(train_x_features, train_y,
                        test_x_features, test_y, average="weighted")
lr_model, lr_predict = lr_fit_and_predict(train_x_features, train_y,
                        test_x_features, test_y, average="weighted")
```

Notice the difference in how long each model takes to train.

Let's check the 10 most probable and most influential features for the trained models (the data is anonymised and 'xxxx' stands for a redacted word):

```
most_probable_features(nb_model, feature_names, categories, 10)
most_influential_features(lr_model, feature_names, categories, 10)
```

You should now clearly see the difference between generative and discriminative approaches by looking at those features. Are the most probable features in a generative model likely to be discriminative?

We will now explore one of the problems with simple bag-of-words features. Check the vocabulary size:

```
len(feature_names)
```

As the datasets get large, feature vectors created using our simple approach get very high-dimensional. But it's unlikely that all the features are equally important, so we may want to exclude unimportant features. Let's see if there's a better way than trying to choose features by hand.

First, let's try a simple idea: use a pre-existing stop word set from `nltk`:

```
stopset = set(stopwords.words("english"))
```

Before going further, consider the following questions:

- Will removing stopwords reduce the size of the vocabulary by much?
- Will removing stopwords change the performance of the classifiers by much?

Now, check your predictions. Uncomment and run the relevant lines (78-92) in `lab_big.py` which (a) vectorize the data without using the stopwords, (b) train and test models on the new feature representation, and (c) print out the most probable and most influential features of those models.

Did the reduction in the number of features harm performance? Are the probable/influential features more sensible? Do you see any other improvements? Have we succeeded in reducing the size of the feature vectors much?

As an alternative to removing stopwords, we can decide automatically which features to get rid of. Here, we will try removing features that have low *variance* across classes: that is, where the counts of the feature are very similar across all classes.

Re-comment the lines from the previous section and un-comment lines 97-110, which train and test the models after filtering out low variance features. How many features are there now? Did this big reduction affect performance much? Can you detect any change in how fast the models run?

Going Further

1. Try to find a variance threshold such that the performance does not suffer. You can also combine stopword filtering with variance-based reduction.
2. Take a look at the length of the documents in this dataset. Plot a histogram of the lengths:

   ```
   plot_document_lengths(train_x_words)
   ```

   Are the lengths distributed fairly uniformly? Are the documents rather short, or rather long? What sort of transformations would you apply to count based features knowing that the lengths of the documents are varied, and assuming that the length itself is not correlated with class? Try using the code we wrote for normalising each feature vector to unit length and see what happens. Does normalization affect Naive Bayes and logistic regression performance equally?

3. There are many kinds of features other than word counts. At the end of `lab4_big.py` we presented options from `sklearn`: ngrams, binarized features, and tf-idf weighted count features. Try to get a better classification performance by playing with those options, and don't forget that you can also reduce and normalize. You could also try using lemmas or POS tags if you work with nltk to get those features.

   When you are experimenting, you can inspect the most probabile / most influential features to get a feeling of how sensible your classifiers are.