Computing Transitional Probabilities

Contents

1 Background **2**

2 Before coding **2**

3 Writing the code **3**
- 3.1 Loading the data . . . 3
- 3.2 Counting frequencies . . . 3
- 3.3 Computing TPs . . . 4
- 3.4 Computing the probability of any sequence . . . 4
- 3.5 Computing probability for test items . . . 5
- 3.6 Computing average probability . . . 5
- 3.7 Plotting . . . 6
- 3.8 Generating sequences . . . 6

References **6**

Goals. The goal of this computer lab is that you become familiar with Python. This document will guide you through the steps of writing a very simple program that computes Transitional Probabilities. We also provide you with a Python tutorial[1], which includes slides and Jupyter notebooks. You can choose to either start with the notebook tutorial and continue with the assignment afterwards, or to directly start with the assignment and consult the slides or the related notebook when you want to find out more about some particular aspect of the language. Whichever strategy you use, you can always ask your lecturer and TA whenever you need some clarification. Of course you can also check the official Python documentation and the recommended tutorials.

Requirements. For this lab, you will need an installation of Python3, the package `matplotlib`, and `jupyter notebooks`. You can install Python and `matplotlib` at the same time by installing Anaconda, a distribution for scientific computing that contains many useful packages. Alternatively, you can install Python and

Last updated on July 10, 2019. Written by Raquel G. Alhama and Samira Abnar (2018) Updated by Bas Cornelissen (2019).

[1]The tutorial was developed by the Computational Cognitive Neuroscience Summer School (2016)

matplotlib without Anaconda. To do so, follow these instructions. Once Python is installed, you can install `matplotlib` as indicated here. Finally, follow these instructions to install Jupyter notebooks.

Files. In order to complete the assignment, you also need to use the jupyter notebook `materials/transitional-probabilities.ipynb` and the files containing the familiarization stimuli: `materials/SaffranAslinNewport1996_1A` and `materials/SaffranAslinNewport1996_2A`.

1 Background

Saffran, Aslin, and Newport (1996) showed that 8 month old infants are sensitive to transitional probabilities in a speech stream, and can use them so identify word-like units. The transitional probability between two syllables X and Y is defined as the conditional probability that Y occurs after X:

$$P(Y|X) = \frac{\text{counts}(XY)}{\text{counts}(X)}$$

In this computer lab, we will develop a very simple Python program that takes as input the transcribed speech stream used in the experiment by Saffran and colleagues, and computes the transitional probabilities in it.

2 Before coding

First of all, it is useful to identify the subtasks that our program will need to perform:

1. Load the input data.
2. Compute the counts for each syllable (*unigram*) and for each consecutive pair of syllables (*bigram*).
3. Compute the TPs of any given pair of syllables.

When programming, it is good practice to create a different *function* for each specific task. A function is an encapsulated block of code that (optionally) receives some input parameters, does something, and (optionally) returns an output; you can think about it as a small program inside the bigger program. For instance, the following function receives two parameters as input, performs some operation (in this case, it sums the parameters) and returns the result of the sum.

```
def function_name(parameter_1, parameter_2):
    result = parameter_1 + parameter_2
    return result
```

Part of this work has already been done for you: open the file `transitional-probabilities.ipynb` with jupyter. In the last cell you can find this:

```
if __name__ == '__main__':
    print("This is the lab on transitional probabilities")
    stimuli = readStimuli("SaffranAslinNewport1996_2A.txt")
    unigram_dict, bigram_dict = count(stimuli)
    print("These are the counts for unigrams: ", unigram_dict)
```

The first line is the entry point from where the program starts running, and it looks the same in every program. This program starts printing a friendly message (*This is Lab 2 of CMLM :)*), and then *calls* one of the functions defined above, `readStimuli`, and provides it with one parameter (the name of the file that contains the stimuli). At this point, the program would jump to the function, execute it, and store the returned value in the variable called `stimuli`. The next line performs another function call, and stores the two returned values in two variables. Finally, the value of one of the returned values is printed. You can run the program and see the output.

3 Writing the code

3.1 Loading the data

The first task we identified is loading the data. This entails opening the file that contains the stimuli, loading the stimuli into memory (so that it is available for latter processing), and closing the file.

In the code you have been provided, this function is called `readStimuli`. The lines starting with `#` , and the text enclosed in `''' (some text) '''` are ignored when the program is executed. They are very useful to keep the code readable: the former, called *comment*, can be added anywhere in the program, while the latter is mostly used to generate automatic documentation of the code.

Read the code in this function and try to understand it. If it is difficult, it may be useful to solve the tutorial notebooks 1 and 2.

3.2 Counting frequencies

According to the formulas for computing TPs, we need the count of single syllables (unigrams) and pairs of syllables (bigrams). Python offers us a very useful structure that we can use to store these counts: the dictionaries. In the tutorial notebook 5 you can find a short tutorial about using dictionaries.

Try to understand the following lines. It may be helpful to try some of the code separately in the python console. For instance, what does `len([1,2,'a'])` do? And `range(10)`?

```
for syll_idx in range(len(list_of_syllables)):
    # form unigram of syllables at index
    unigram = (list_of_syllables[syll_idx])

    # see if we have already seen this unigram
```

```
if unigram in unigram_dict:
    # if so, up the count by 1
    unigram_dict[unigram] += 1
else:
    # if not, set the count to 1
    unigram_dict[unigram] = 1
```

QUESTION 1

Extend this function so that it also counts bigrams. Report your code and the counts. Hint: you can represent two syllables in a single variable using tuples. Remember that you can test your function by calling it from `__main__`. *(2 points)*

3.3 Computing TPs

Now we have a function that computes two dictionaries that contain all the counts that we may need. Next, we need to write the code for a function that, given a particular pair of syllables X and Y, returns the transitional probability $T(Y|X)$.

QUESTION 2

Complete the code for the function `TP`. *(1 point)*

The participants in Saffran, Aslin, and Newport (1996) were familiarized with an artificial language, carefully designed by the authors so that it would have certain transitional probabilities. Concretely:

> "The only cues to word boundaries were the transitional probabilities between syllable pairs, which were higher within words (1.0 in all cases, for example, *bida*) than between words (0.33 in all cases, for example, *kupa*)."

Let's check whether that is true.

QUESTION 3

What are the transitional probabilities for *bida* and *kupa*? (Use the stimuli for experiment 1A). *(0.25 points)*

3.4 Computing the probability of any sequence

Now that our program can compute transitional probabilities between syllables, we can extend it with a function that computes the probability of a sequence of

any length. In order to do so, we need to compute the product of all the bigrams in such sequence.[2]

QUESTION 4

There is a sketch of this function in the code (called *sequenceProbability*). Complete it with the missing code. *(1 point)*

3.5 Computing probability for test items

Now we can actually compute the probability for the words and part-words in experiment 2A in Saffran, Aslin, and Newport (1996). You can use nested lists to store the sequences (i.e. lists of lists of syllables):

```
words = [['pa','bi','ku'],
            ['ti', 'bu', 'do'],
            ['go', 'la', 'tu'],
            ['da', 'ro', 'pi']]
part_words = [['tu', 'da', 'ro'], ['pi', 'go', 'la']]
```

QUESTION 5

Create a function that computes the probabilities of each sequence in a list, and prints them. *(1 point)*

3.6 Computing average probability

Let's extend the previous function so that it computes and returns the mean probability for all the items in the list.

QUESTION 6

Extend the previous function so that it computes the average probability, and returns it. Report the average probability for the words and the partwords. *(0.75 points)*

[2] This estimation of the probability of a sequence is slightly simplified, since we would also have to take into account the probability for the first syllable to occur in the first position, and the probability of the last syllable to occupy the ending position. In order to do so, n-gram models compute the probability for the first and last syllable by adding a virtual initial and final symbol. This makes sense when the input is a large corpus with a big number of sentences, since those will contain a significant number of initial and ending symbols. However, since we are dealing with unsegmented input, we could only add 1 initial symbol and 1 ending symbol, so any most syllables would have a probability of zero for starting or finishing a sentence. Thus, for this experiment we take a simplified estimation of the probability of a sequence.

3.7 Plotting

We can now use the computed probabilities to visualize them in a graph. The notebook tutorial 3 contains a tutorial for basic plotting with matplotlib.

QUESTION 7

Create a function that receives a list of probabilities as input and creates a barplot with the probabilities. Use it to plot the average probability of words versus the average probability of part-words. Hint: You can find useful graph examples in the online documentation of matplotlib. *(2 points)*

3.8 Generating sequences

Some models of language and music can also be used to *generate* instances of language or music. In our case, once we have run our program to compute the TPs of the input, we can use those TPs to generate a sequence.

Choose a corpus of natural language (e.g. you can use the Tom Sawyer corpus from the last computer lab). Compute the transitional probabilities between words and use them to generate a few new sentences.

Note: In order to change the level at which the probabilities are computed (e.g. to characters, or to words), you have to adapt the function that reads the file. The function `re.findall` uses a regular expression. Adapt the expression for one character only, or for words (you can find documentation here).

QUESTION 8

Extend the program so that it reads a natural language corpus, computes the transitional probabilities and uses these probabilities to generate a new sequence. Try it at the character level and the word level. How different are the results? What kind of errors does your model make? Why? Hint: you can generate (pseudo-)random numbers in Python with the library `random`. *(2 points)*

References

Saffran, Jenny R, Richard N Aslin, and Elissa L Newport. 1996. "Statistical Learning by 8-Month-Old Infants." *Science* 274 (5294). American Association for the Advancement of Science: 1926–8.