**ICP-4721: Natural Language Processing**

**Laboratory 1: Introduction to Text Processing in Python for NLP**

*Bill Teahan*

# 

# Exercises

*For this module’s laboratory sessions, you will need to produce a report that includes the answers to each of the lab exercises and then submit the report to Blackboard. You will have two weeks to complete the report. Marks (out of 100) are shown at the top of each exercise. Try to complete the exercises within the two lab sessions otherwise you will need to complete the exercises in your own time. There will be four labs for this module, so as the lab weighting is 50%, this means that each lab report is worth 12.5%. Feedback on the previous week’s lab will be provided the following week so have your lab report and programs ready for checking.*

***Please do not include the questions in the lab report – just the answers – as this will affect the similarity index for your report for the TurnItIn plagiarism software.***

# Submission Deadline

All lab work will be due at midnight on Tuesday of the week when it is due (i.e. the night before the next lab session).

# Documentation

Official Python website – Python.org:

<http://www.python.org/>

For further information and documentation, you can also try the Python Wiki:

<http://wiki.python.org/moin/>

O’Reilly Media – “Learning Python” :

<http://oreilly.com/catalog/9780596513986/>

## Strings in Python

Strings are a built-in data type in Python which you all should now be familiar with from previous modules on programming you will have taken. Strings are also very important for NLP as most NLP programs will represent and manipulate natural language text as strings.

You can learn more about strings by reading the Python online documentation:

<https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str>

There are many useful Python string methods which you can use to manipulate strings. A full list of methods can be also found in the Python online documentation:

<https://docs.python.org/3/library/stdtypes.html#string-methods>

Familiarise yourself with each of these methods to learn about what you can do to easily process text in Python in various ways.

A few programs have been provided in Blackboard in the Lab01 folder as examples:

* convertupper.py

This contains a simple Python function called capitalise() which has the following docstring that documents what the function does:

""" Creates a new file with the prefix 'upper\_'

added to the name of the original file.

All the alphabetic characters in the new

are capitalized. This function does not

disturb the contents of the original file. """

* runconvert.py

This imports the convertupper module and uses the capitalise() function to capitalise all the words in a file called declaration.txt. This text file contains a slightly noisy (i.e. messy) version of the American Declaration of Independence.

|  |
| --- |
| Exercise 1.1 – Basic Encryption Using Strings in Python (10 Marks) If you run the runconvert.py program, you will see that it produces an output file called upper\_declaration.txt. For this exercise, modify the program so that it produces a different output file instead: encrypted\_declaration.txt. Use the following string methods to encrypt the text in the file declaration.txt so that the text is completely changed from the original and becomes much more difficult to decipher:   * reverse() * translate()   **Post your code, and encrypted output into the report.** |

# Dictionaries in Python

One of the most useful built-in data types in Python for NLP are dictionaries. This is illustrated by the groupwords.py program which can also be found in the Lab01 folder. This program uses a dictionary called groups to create lists of words which all have the same length.

|  |
| --- |
| Exercise 1.2 – Coping with Noisy Text (10 Marks) Run the groupwords.py program, using the declaration.txt filename as the first command line argument. Include the output in your report. **Discuss in your report what the problems are** **with the output the program produces.** *Be very careful* – there are many more problems than you will spot at first glance. (Note: This is a very common issue that you will encounter when processing texts and writing programs for NLP. You always need to be very careful when checking that your output is fully correct).  **Edit the program to try to remove these problems and run again.**  **Post your corrected code, and new output into your report.** |

# Zipf’s Law

In 1949, George Zipf showed that a remarkable number of observations in the social sciences, including word frequencies, follow hyper-geometric laws. He noticed that if you rank word types in order of decreasing frequency, with the most frequent type ranked 1, the next most frequent ranked 2, and so on, then the product of rank and frequency remains constant.

Restated, if *t* is the number of types in the text, the probability of a type of rank *r* is given by:

p(*r*) = μ / *r*, *r* = 1, 2, ..., *t*.

Zipf estimated the constant μ for word frequency distributions to be roughly 0.1 using data from James Joyce’s *Ulysses*.

We can check this out using the create\_Zipfs\_Law\_plot.py program in the Lab01 folder. This produces a Zipf’s Law plot for three very different English texts (which can also be found in the Lab01 folder):

* the Brown corpus which contains approximately 1 million words of American English text (in the file Brown.txt);
* the complete works of Shakespeare (in the file Shake.txt);
* the King James version of the Bible (in the file Bible.txt).

*Note: these three text files have been preprocessed and converted to 27-character English. This is where all alphabetic characters in the text have been converted to lower case, and contiguous sequences of non-alphabetic characters in the text have been replaced by a single newline character. This results in “words” appearing on separate lines in the text file. (Examine the text files to see what this looks like.)*

*Defining what “words” are can be very problematical for NLP – even for English which is a predominantly word-based language. One expedient for processing English texts is simply to convert the text to 27-character English as in this example.*

|  |
| --- |
| Exercise 1.3 – Understanding Zipf’s Law (20 Marks) Run the create\_Zipfs\_Law\_plot.py program. Post a screenshot of the output that is produced in your report. **Discuss what this output shows.**  **Also in your report explain how the program works.**  **Try out your program using several more English texts to see what happens. Discuss what you find out. [Make sure your texts aren’t too short!]**  **Include screenshots of the output(s) in your report.**  *[Hint: This may also be something useful for your assignment.]* |

# The Sparse Data Problem

Stanley F. Chen in his 1996 Harvard Ph.D. thesis “*Building probabilistic models for natural language*” stated the following:

“*The sparse data problem refers to a situation when there is insufficient data to train one’s model accurately. This problem is ubiquitous in statistical modeling; the models that perform well tend to be very large and thus require a great deal of data to train*.”

|  |
| --- |
| Exercise 1.4 – Dictionaries Again – and the Sparse Data Problem (20 Marks) The create\_Zipfs\_Law\_plot.py program used a dictionary called frequency to store the frequencies of words in each of the three texts. Modify the program so that it outputs the percentage of the words in each of the three texts that have frequency 1 [these are called word singletons, or *hapax legomena*], the percentage that have frequency 2, and the percentage that have frequency > 2.  **Post the modified program and output it produces into your report.**  **Discuss what the results mean and their significance for NLP.** |

# The Zero Frequency Problem

Another major problem for NLP is what is called the Zero Frequency Problem. This is where words do not show up in our training data because they are rare or very infrequent. Both the Sparse Data Problem and the Zero Frequency Problem has significant implications for NLP because it means we must use incredibly large training texts to train models for NLP. This is why these models are called Large Language Models (or LLMs for short).

|  |
| --- |
| Exercise 1.5 – The Zero Frequency Problem (20 Marks) The purpose of this exercise is to solve the following problem – how much of the text is missing? i.e. has zero frequency.  We can again use the three Python dictionaries we created above to help us answer this question. Write a program to find out how many words are missing between the three dictionaries. We’d expect a substantial number missing of course, but it would be interesting to see whether the English bible has more words in common with the complete works of Shakespeare than more recent American English.  **Post the modified program and output it produces into your report.**  **Discuss what the results mean and their significance for NLP.** |

# Types versus Tokens and Heap’s Law

There are further ways we can visualize relationships amongst words in natural language texts. For example, we can characterize words in the following two ways:

* word types: These are the set of unique words in the text. i.e. the vocabulary.
* Word tokens: These are the list of running words in the text.

For example, in the well-known palindromic text “*a man a plan a canal panama*” [this text string has the same alphabetic sequence both forwards and backwards], we have:

* 5 word types: *a, man, plan, panama*
* 7 word tokens: *a, man, a, plan, a, canal, panama*

As a text gets larger and larger, both the number of word types and word tokens will keep on growing. Obviously, the number of word types will grow much more slowly compared to the number of word tokens. Although the number of word types will eventually plateau, it will never stop increasing because of the sparse data and zero frequency problems.

An interesting question is whether there exists a relationship between word types and word tokens (similar to the relationship between word frequency and word rank for Zipf’s Law).

There is in fact a law for this relationship – it is called Heap’s Law.

Heap's law states that the number of unique words *V* (i.e. the number of word types) in a collection with *N* words (i.e. the number of word tokens) is approximately *sqrt(N)*.

Two programs can be found in the Lab01 folder that can be used to visualize the relationship between word types and word tokens:

* create\_Heaps\_Law\_plot.py
* create\_Types\_vs\_Tokens\_plot.py

|  |
| --- |
| Exercise 1.6 – Understanding Heap’s Law (20 Marks) Run the following two programs: create\_Heaps\_Law\_plot.py and create\_Types\_vs\_Tokens\_plot.py. Post screenshots of the output that is produced by each in your report. **Discuss what these outputs show.**  **Also in your report explain how the programs work.**  **Try out your programs using several more English texts to see what happens. Discuss what you find out. [Make sure your texts aren’t too short!]**  **Include screenshots of the output(s) in your report.**  **Discuss what the results mean and their significance for NLP.**  *[Hint: This may also be something useful for your assignment.]* |