# CS918 Exercise1 - Report

Nan Liu
Nan.Liu.1@warwick.ac.uk
4th November 2018

## 1 Introduction

This report describes the process of solving the given tasks, with a brief explanation of the methodologies used when dealing with the corresponding problem.

## 2 Programming

There are five libraries used in the whole program: **json**(read json format files), **re**(match and remove certain characters), **time**(record the run time of the code), **nltk**(create n-grams and lemmatize the word) and **math**(do operations for log transformation). The exercise consists of three parts, which will be introduced below.

### 2.1 Part A: Text Preprocessing

The first part of exercise 1 is to preprocess the text which includes removing certain characters and lemmatizing all words.

#### 2.1.1 Regular expression and character removal

To parse and clean the texts, **re.compile()** and **re.sub()** are called to define the patterns and replace them with spaces respectively.
***"[^a-z0-9 ]+"*** is used to match all non-alphanumeric characters. *"[\W]+"* was used to do the task in the beginning, but was unable to match "_" in the text.
***"\\b[a-z]\\b"*** is used to match words with 1 character.
***"\\b[0-9]+\\b"*** is used to match numbers that are fully made of digits.
***"((https?):\/\/)?([a-z0-9]+([\.\/\?\=\&*\%\-]+[a-z0-9]+)+)"*** is used to match URLs. For all URLs in the text, "http" or "https" is optional; before a special character(e.g. period, slash and so on), there should be some alphanumeric characters as domain; special characters may occur one or unlimited times; and alphanumeric characters could also appear after a special character.
The order of preprocessing should start with removing URLs, then non-alphanumeric characters. And the order of removing 1 character and digits does not matter.

#### 2.1.2 Lemmatization

When lemmatizing all words, **nltk.WordNetLemmatizer().lemmatize()** is applied to process every single word in the content. **nltk.pos_tag()** was used to attach speech tag to each word for higher accuracy, but resulted in low efficiency(much longer time to run). Therefore, lemmatize(“has”) will output “ha”, which would influence the result of Part C - 1.

## 2.2 Part B: N-Grams

The second part is to compute some calculations related to N-grams.

### 2.2.1 Tokens and types

Append all words in the corpus into a blank list *all_word*. The number of tokens should be the length of *all_word* and the vocabulary size(type) should be the length of *set(all_word)*.

### 2.2.2 Top 25 occurrences trigrams

Firstly, use **nltk.ngrams()** to generate a list of trigrams form all tokens, then use **compute_freq_dist()** to store trigrams and their corresponding frequencies in a dictionary. Finally, use **sort_freqdist()** to return the top-25-frequency trigrams on the entire corpus.

### 2.2.3 Positive and negative word counts

Read positive and negative words from the given files and store them in a dictionary where the values are initialized to 1(if positive word) and -1(if negative word) for efficiency(compared with list). Use **count_opinion_word()** to count the number of positive and negative words.

### 2.2.4 Positive and negative news counts

Use **count_opinion_word()** and iteration to count the number of positive and negative words in each single news story in the content, then compare the positive and negative numbers to check which kind of words takes the majority.

## 2.3 Part C: Language Models

The last part of exercise 1 is to set up a trigram language model based on the first 16,000 rows of the content to predict a sentence beginning with the bigram “is this”, and evaluate the remaining rows by computing the perplexity.

### 2.3.1 Sentence prediction

To begin with, use **compute_freq_dist()** to generate a dictionary of trigrams in the first 16,000 rows and use **sort_frequdist()** to sort the frequency of trigrams in descending order. Then, search for the most common trigrams beginning with the given bigram and append the third word into the predicted list; repeat this step until 10 words are found.

### 2.3.2 Perplexity

Add all trigrams of remaining rows into a blank list(append row by row, because a trigram cannot be consist of words from two adjacent rows), and obtain the frequency distribution of trigrams and bigrams in the first 16,000 rows(using **nltk.ngram()**, **compute_freq_dist()**) to get ready for perplexity computation. In addition, Laplace Smoothing is used for dealing with those words never appear before(2.1)

$$\mathrm{P}(w_i|w_{i-2}, w_{i-1}) = \frac{count(w_{i-2}, w_{i-1}, w_i)+1}{count(w_{i-2}, w_{i-1})+V}, \tag{2.1}$$

*V* is the number of all words. And log transformation is used for making the perplexity readable(2.2).

$$\begin{gathered}\mathrm{PP(W)} = P(w_1 w_2 \ldots w_N)^{-\frac{1}{N}} = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{\mathrm{P}(w_i|w_{i-2}, w_{i-1})}} \\ \xrightarrow{\log\ transformation} \frac{\sum_{i=1}^{N} \log\left(\frac{1}{\mathrm{P}(w_i|w_{i-2}, w_{i-1})}\right)}{N} \\ = \frac{\sum_{i=1}^{N} \log\left(\frac{count(w_{i-2}, w_{i-1}) + V}{count(w_{i-2}, w_{i-1}, w_i) + 1}\right)}{N}\end{gathered} \tag{2.2}$$

## 3 Conclusion

The whole program takes about 60 seconds. After completing this exercise, I have acknowledged a better understanding of regular expression, n-grams and language models, with the skills of processing natural language using Python and related libraries.