ML Project Report

December 6, 2017


```
01.112 ML Project
Done by :

Yin Ji Sheng(1001670)
Hilda Thian(1001776)
Teo Yang Rui(1001518)
```


1 Part 1

1.1 Annotation of raw data sets

1.1.1 Annotation Process

Every student of the course helped annotate 500 tweets/Weibo sentences each.

2 Part 2

2.1 Implementation of simple Sentiment Analysis using only emission parameters:

2.1.1 Loading File

To easily iterate through a desired file, we strip the white space in the string and put separate words with their tags in a whole list.

```
def load_original_train():
    with open("train", encoding="utf-8") as file:
        train_list = file.readlines()
        train_list = [x.strip() for x in train_list]
        return train_list

train_list = load_original_train()
```

2.1.2 Estimating emission parameters

- The method to estimate the emission parameters were as shown in the project description. First, we created an annotation dictionary to store counts of all possible state, {O, I-Positive, I-Negative, I-Neutral, B-Positive, B-Negative, B-Neutral}. Hence, we can call on count(y) anytime when estimating emission parameters without recounting them for every word.

- Then, to estimate emission, we created a function called estimate_emission which take in two variables, word and tag. In this function, the tag will call the annotation dictionary from the first part to yield us the relevant denominator named countBottom.
- For the numerator, a for loop through the loaded train_list above is employed. For each word with their tag, we further split them into a list with 2 elements, [word,tag]. This can only be done if the "word" is not an empty string "", found between different tweets. Creating an if statement to calculate only when we have a word and tag. We loop through the file and increase numerator count, named countTop when we have the specified word and tag. After this is done, the function returns the fraction countTop/countBottom as the estimated_emission parameter.

2.1.3 Replacing words that appear less than k times with #UNK

- To replace words that appear less than k times with #UNK#, the function modify_trainingset(k) was created. Firstly, we need to find a way to store the count of words so that we can know the words that have to be replaced. A dictionary called wordCountDict was created. We loop through the entire training set and counting the occurence of each word.
- Secondly, in order to iterate through the file just once and replace the words that needs to be replaced along the way, a list called wordToBeReplacedList was created. If the key in wordCountDict has a value less than k, it will be stored in wordToBeReplacedList.
- Thirdly, we iterate through train_list, and replace the word found in wordToBeReplacedList with #UNK#., then taking into account the original form of the file, the modified set of words and tags are written into a new file called modified_train

2.1.4 Implementing sentiment analysis system to produce tag

- In order to speed up the sentiment analysis algorithm, we created a giantEmissionDict to store all the possible emission parameters into it. This is done using the store_estimate_emission_fix function. The function is also done using the modified training set so we can count the emission of #UNK# as well.
- In sentiment analysis, we first check whether the word exists in the modified training set or not. If it does not, the word is treated as #UNK#. Then, we loop through all the possible tags and get it's emission with respect to that tag and storing them into scoreDict. The tag with the highest emission value is then picked.
- We then use the same sentiment_analysis function for all the words in the entire dev.in file and output the results to dev.p2.out.

2.1.5 Results for Part 2

CN

```
#Entity in gold data: 362
#Entity in prediction: 3318
```

```
#Correct Entity : 183
Entity  precision: 0.0552
Entity  recall: 0.5055
Entity  F: 0.0995

#Correct Sentiment : 57
Sentiment  precision: 0.0172
Sentiment  recall: 0.1575
Sentiment  F: 0.0310
```

EN

```
#Entity in gold data: 226
#Entity in prediction: 1201

#Correct Entity : 165
Entity  precision: 0.1374
Entity  recall: 0.7301
Entity  F: 0.2313

#Correct Sentiment : 71
Sentiment  precision: 0.0591
Sentiment  recall: 0.3142
Sentiment  F: 0.0995
```

FR

```
#Entity in gold data: 223
#Entity in prediction: 1149

#Correct Entity : 182
Entity  precision: 0.1584
Entity  recall: 0.8161
Entity  F: 0.2653

#Correct Sentiment : 68
Sentiment  precision: 0.0592
Sentiment  recall: 0.3049
Sentiment  F: 0.0991
```

SG

```
#Entity in gold data: 1382
#Entity in prediction: 6599

#Correct Entity : 794
Entity  precision: 0.1203
Entity  recall: 0.5745
```

```
Entity  F: 0.1990

#Correct Sentiment : 315
Sentiment  precision: 0.0477
Sentiment  recall: 0.2279
Sentiment  F: 0.0789
```

3 Part 3

3.1 Implementation of Viterbi

3.1.1 Estimating transition parameters using Maximum Likelihood Estimation

- We loop through the modified training set and we created a nested list to store tags of each sentence. We also append "START" and "STOP" to the list to allow easy counting later of the transition parameters.
- We then create a dictionary that contains the count of every possible transition in the modified training set. To calculate the transition parameter, we simply divide that count by the count of the start state, which is available to us from annotationDict that we have created earlier. The result is then stored into giantTransitionDict, with key being the transition.

3.1.2 Before implementing Viterbi

- In order to implement the Viterbi algorithm, we need to first split the whole list of words read by the file into a individual tweets. We use the split_into_sentences function to split the whole list of words dev_in_list into a nested list. We first create a list named sentences that is a collection of lists of individual tweets named each_sentence. each_sentence contains words from the same tweet as its elements. The end of each tweet is recognised by an "" empty string. In iterating through the every word in dev_in_list, we check if the word is the empty string "". If the word is the empty string "", we are at the end of a tweet so we stop appending words into each_sentence and instead, append each_sentence, now a list of words containing a single completed tweet into sentences. We then have to recreate a new empty list each_sentence to accomodate for the next tweet. If the iterated word is not an empty string "", the word still belongs to the same tweet and we continue to append it to each_sentence.

The following pseudocode will split the data set into sentences:

```
def split_into_sentences(dev_in_list):
    sentences = [] #List to store all tweets, tweet by tweet
    each_sentence = [] #List of each tweet, containing words from one tweet
    for every word in each line of the file:
        if word == "": #at end of a tweet
            # append each_sentence into list of all tweets
            # recreate a new empty list each_sentence to contain words for next tweet
        else: #still within the same tweet
            # append word into list containing words from this same tweet
    return sentences
sentences = split_into_sentences(dev_in_list)
```