NLP Term Project: Team 17

B01902028 周書禾 B01902044 林于智 B01902128 張鈞堯

Introduction:

This semester's NLP term project requires us to come up with first with a detector for redundant words, then requires us to come with a locator to detect the location of the redundant words.

Phase I:

Observations:

We immediately thought that such a problem could be solved using some sort of n-gram method. However, it became apparent that performing n-gram on words wasn't a good idea. Instead, the logically way to solve this problem was to perform POS tagging. Then, we perform n-gram method on the POS tags. We also discovered that each training data came in pairs, one correct and one incorrect. This became a basis for knowing whether or not our solution is good or not. We discuss our methods below.

Execution Overview:

Step #1: Finding Pairs w/ Levenshtein Distance

```
python nlp.py | tr -s $'\n' > tmp1.txt
```

Step #2: Change Sentences into POS Tags

```
gcc -I/usr/local/scws/include/scws/ -L/usr/local/scws/lib/ train_cut.c -lscws -o a.out
```

Step #3: Get Useful Information

```
./a.out < tmp1.txt | grep -B 1 "===" | grep -E "0|1|POS" > tmp_train.txt
```

Step #4: Process Test Data

```
cat p1.test.txt | cut -d $'\t' -f 2 > tmp1.txt
```

Step #5: POS Tagging on Test Data

```
./a.out < tmp1.txt | grep -B 1 "===" | grep -E "0|1|POS" > tmp_test.txt
```

Step #6: Build Up Knowledge Base and Give Answers

```
python llh.py tmp_train.txt tmp_test.txt 2.0 3.0 > tmp_result.txt
```

Step #7: Add Sentence ID to tmp_result.txt

```
python recover.py > p1.result.txt
```

Step #8: Remove Files

```
rm -f a.out tmp1.txt tmp_result.txt tmp_test.txt tmp_train.txt
```

Source Code

Methodology

- Pos Tags
- Use different mixture models for Shannon Game

- Bigram
- Trigram
- Special Trigram - look at word before and after, instead of the two following words

llh.py
- Derive likelihood for both positive and negative models
- Compare the likelihood then decide
- Utilize parameters Epsilon and Epsilon_S
  - Default is 1.5, 2.0, but can be tuned

Self-Evaluation
- Use 5-fold cross validation
- Smoothing didn't lead to better result, perhaps smoothing method is not very good
- Results are as follow

Phase 1 Evaluation	Score
Precision	54
Recall	88
F1 Score	67
Accuracy	56

Discussion

Despite utilising many different n-gram models and even smoothing methods, our accuracy remains in the 50's. Tuning our epsilon and epsilon_s parameters did yield some improvement, but it was not significant enough to be noteworthy. Another issue we noted is that our smoothing technique did not yield a better result. Perhaps the reason this occurred is that our smoothing method is not very refined, and a better smoothing technique may have solved this issue.

Phase II:

Observations

Similar to Phase I, Phase II is also about redundant words. But instead of just evaluating whether a sentence contains such a redundancy, we need to locate the position of the word in the sentence. Even though the tasks are different, we decided to go with a similar approach, by using the Stanford Segmented and Stanford POS Tagger as the basis of our analysis. Furthermore, we also use n-gram to do error detection.

Execution Overview

crossvalid.sh
- Perform cross validation on results of main.py

run.sh
- Perform normal run on main.py

Source Code

slice_train.sh
- Slice all sentences in training data into 5 sections, one for test other four for train

slice_answer.sh
- Slice training data's answer into 5 sections

demo.sh
- Try Stanford Segmenter and Tagger

get_test_tagged.sh
- Use Stanford Tagger to tag test data

get_test_tagged2.sh
- Use SCWS Tagger to tag test data

get_train_tagged.sh
- Use Stanford Tagger to tag train data

get_train_tagged2.sh
- Use SCWS Tagger to tag train data

answer_for_training_data.py
- Calculate training data's correct answers

main.py
- Class NGRAM
  - Build up the n-gram model
  - Count total occurrences of all gram instances
  - Check if the tuple already exists in grumbliest
    1. Return the GRAM instance if found
    2. Return false if not found
  - Add a gram instance of the tuple
    1. Return the GRAM instance if found
    2. Return false if not found
  - Return the count corresponding to the given tuple
  - Return a probability indicating how much the taglist fits this NGRAM model
- process_raw_line()
  - Return list of pos tags, given rawline
- guess()
  - Find out the most-likely redundant tag
- Rest of code
  - Building models, and also other initiating and terminating functions

---

Discussion

A main different between Phase I and Phase II is that Phase II we used a different tagger. Phase II we decided to go with the Stanford Tagger, which runs slower but gives a more accurate result.

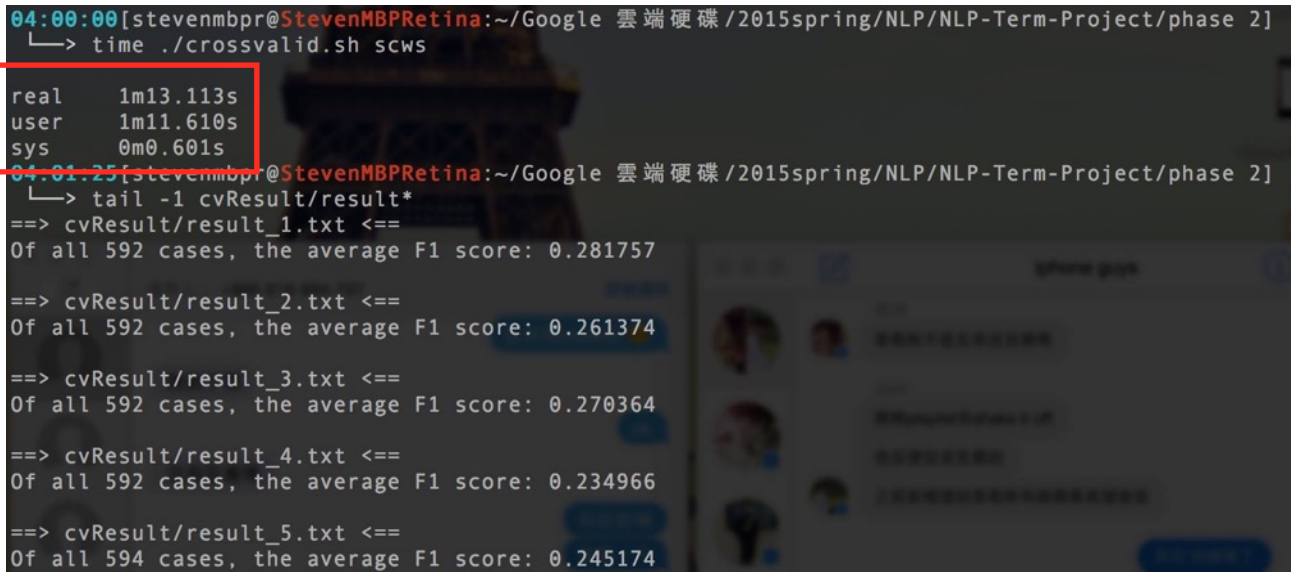
Our method is to look at each POS tag individually. Meaning, if the removal of that particular POS tag leads to a better probability of the entire sentence, we consider the word that is represented by that particular POS as the redundant word. But first we consider all tags in the sentences, then we make our choice. But there are cases in which we may have to consider the removal of multiple words represented by POS's.

First Run
Our original run of F1 scores resulted in 1999/2962 (roughly 33% correct) cases that were completely off. And in those 963 correct cases, roughly 1/9 of such cases were partially correct. Upon closer inspection, however, we noticed that there are cases in which we were completely correct and also cases in which we needed to remove more POS tags (F1 score between 0 and 1). We also noted that the tagger sometimes returned cases in which redundant words were bunched

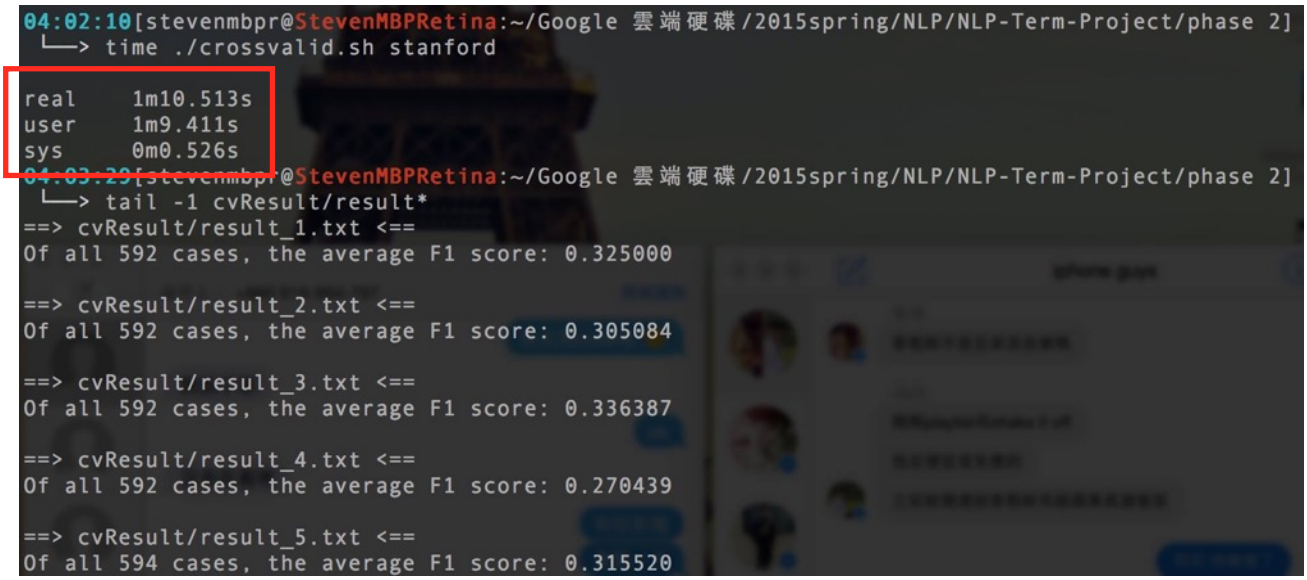
together with non-redundant words. (之後的 was bunched together as one tag, but the correct case identifies only the first word 之 as redundant).

Different Approaches…
We tried different approaches, such as bigram only, trigram only, only negative, only positive, bigram and trigram. However, our results have become worse and worse for some unidentified reason. Thus we reverted back to our original approach, which gives us the best results as of right now.

Looking at Different Taggers



scws tagger with bigram/trigram ratio of 0.7 and 0.3



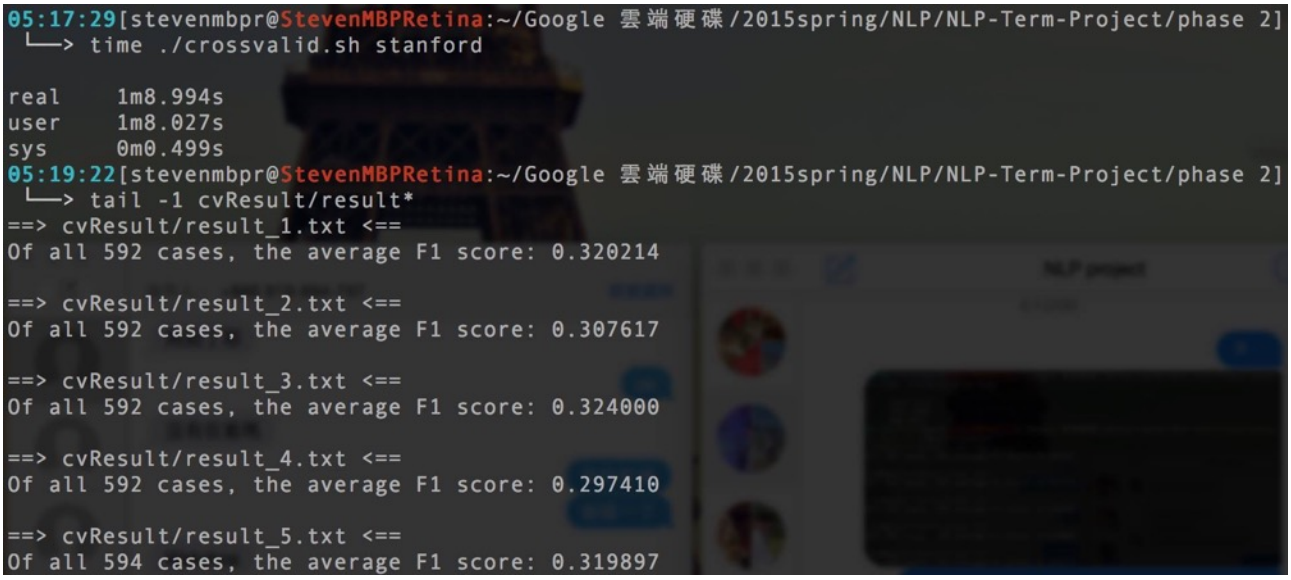
stanford tagger with bigram/trigram ratio of 0.7 and 0.3

-> Confirms our observation that Stanford is a more accurate, although slower method

Making Things Smoother?

Although the improvement is so minute, we still thought it'd be worth a mention. After lowering the Laplace Smoothing Factor from 1 down to 0.5, we discovered a slim improvement. Although not very noteworthy, still an improvement nevertheless.

Smoothing screen capture is on the following page.



Thoughts…

Definitely learned a lot from this project in terms of implementation of theories taught in class. Furthermore this project has helped me appreciate how refined some NLP services are right now. The level of accuracy some of these services have reached can truly be appreciated now that we know how difficult NLP is.