Final Project

*Python Programming-UCSC*

*06-19-2018*

**Overview**

In this project, several language models have been trained to build a spell checker with a given likelihood term or edit model. Then a test data will be generated where each sentence only contains exactly one typing error. The language models are evaluated for both accuracy (number of valid correction divided by the number of test sentences) and speed. Data used for training and testing comes from the writings of secondary-school children, collected from David Holbrook.

All the codes/data list below are in the zip file as well as the output log.

**Requirement**

Modules needs to be imported (installed): re, collections and math.

Some of the codes are used from Stanford University Natural Language Processing [Coursera], please see detailed source codes and how to use them in **Code Given** section.

For evaluation test bench (spellCorrect.py), used Peter Norvig’s spelling corrector as reference (<https://norvig.com/spell-correct.html>).

Here are the Python elements I think included in the code implemented.

1. Use any data structure list, dictionary, set or tuple
2. Functions
3. Classes
4. Importing external modules
5. File input and output
6. Decorators
7. Error checks using try-except
8. List comprehension
9. Dictionary comprehension

**Description**

**Code Implemented**

* Language Models with different algorithms:

1. UnigramLM.py: an unsmoothed unigram model. Treat out-of-vocabulary items as a word which was seen zero times in training.
2. LaplaceUnigramLM.py: a unigram model with add-one smoothing.
3. LaplaceBigramLM.py: a bigram model with add-one smoothing.
4. StupidBackoffLM.py: use an unsmoothed bigram model combined with back-off to an unsmoothed unigram model
5. StupidBackoffAddoneSmoothingLM.py: use an unsmoothed bigram model combined with back-off to a add-one smoothed unigram model
6. ModifiedKneserNeyLM.py: used an absolute-discounting smoothed bigram interpolated with unigram model, modified part is for optimal discounts depend on frequency of frequency of words
7. KatzBackoffLM.py: used a Katz-smoothed bigram model combined with back-off to a Katz-smoothed unigram model.

* Each Language Model is implemented with at least two functions:

1. Train: takes a corpus and trains the language models. Generate ngram model tokens and counts.
2. Score: takes a list of strings and returns the numerical score, which will be the log-probabilities of the sentence using the language model.
3. Some discounting calculation functions are implemented depends on the language model type.

* Evaluation Test bench

1. SpellCorrect.py: This is the top level evaluation code. Computes the most likely correction given a language model and edit model. The main () function will load all the language models and create an output file “ComparisonLM.log” to store the performance on the development data.
2. SpellResult.py: Computes accuracy with number of valid correctness divided by number of total test sentences.

[**DATA**](http://ota.ox.ac.uk/headers/0643.xml)

1. Holbrook-tagged-train.dat: a corpus to train the language models
2. Holbrook-tagged-dev.dat: a corpus of spelling errors for development (test)
3. Count\_1edit.txt: a table listing counts of edits x|w, taken from Wikipedia.
4. Evaluation is on the development data set
5. Inside the data set, spelling errors are already tagged: <ERR targ=That's> Thats </ERR>

**Code Given**

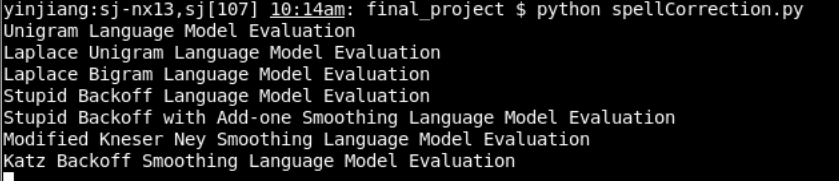
1. EditModel.py: Reads the Count\_1edit.txt file and computes the probability of corrections. The candidate corrections are all strings within Damerau-Levenshtein edit distance <= 1. The probability of no correction is set at 0.9.
2. HolbrookCorpus.py: Reads in the corpus and generates test cases from misspellings.
3. Sentence.py: Holds the data for a given sentence, which is a list of Datums. Contains helper functions for generating correct sentence and the sentence with the spelling error.
4. Datum.py: Contains two strings, word and error. The word is the corrected word, and error contains the spelling error. For tokens which are spelled correctly in the corpus. Error = “”.

**Screenshots**

How to use:

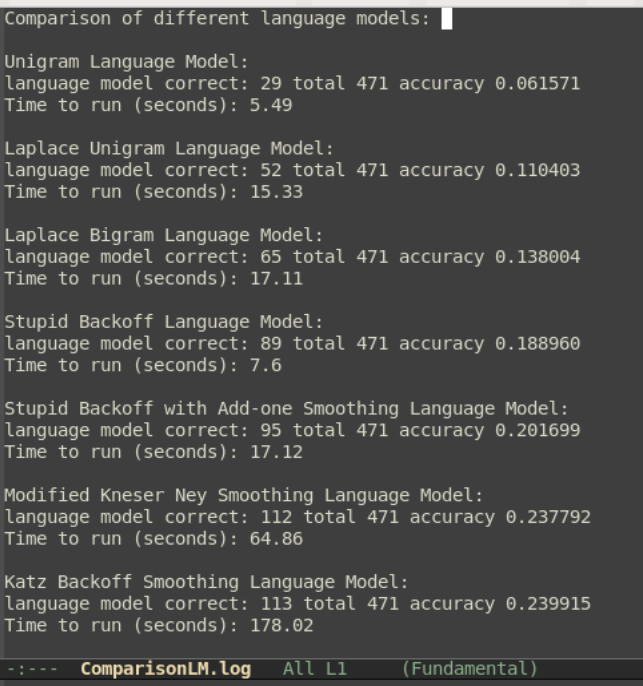
Python spellCorrection.py

It will print out which language model is in evaluation and let you know the progress.



Comparison log file:

The accuracy and speed results are all written to the log file.



**Conclusion**

* Modified interpolated Kneser Ney Smoothing gives the best Accuracy but the speed is slowest. This speed might be further improved by using efficient modules like Counter or nltk.
* Stupid Back off (lambda set to 0.4) gives the best Accuracy and Speed tradeoff. This is why Stupid Back off is popular in large-scale statistical language modeling in machine translation, it is inexpensive to train on large data sets and approaches the similar quality of Kneser-Ney Smoothing as the amount of training data increases. The performance can be further improved by using add-one smoothing technique without sacrifice the speed too much.
* Katz Back off Smoothing is not reliable for small data set. There are actually “holes” of counts of counts. It will perform better if we have larger training data set. The speed is not desirable, that’s due to n-grams model is defined in terms of (n-1) gram model. It can be definitely further improved by using recursive algorithm. This will leave to future work.