Chapter 2

Noisy-Channel Spelling Check

But it must be recognized that the notion "probability of a sentence" is an entirely useless one, under any known interpretation of this term. —Noam Chomsky (1969)

2.1 What is a Noisy Channel?

The noisy channel model (Shannon 1948) has been applied to a wide range of NLP tasks, including machine translation, speech recognition, optical character recognition, and spelling correction. A noisy channel consists of a language model and a channel model. Interestingly, while each task has its own channel mode, all these English-generation tasks can use the same English language model. Although there is much NCM work in the literature, very little has been done to improving the channel model for spelling correction.

For a starter, Church and Gale (1993) propose a channel model for spelling correction based on **Levenshtein edit distance**. This channel model consists of (non-equiprobable) probability functions related to four kinds of edit operation:

- Add a letter,
- Delete a letter.

- Replace one letter with another,
- Transpose two adjacent letters,

Brill and Moore (2000) extend the scope of edit operations to include general **string to string** transformation. Intuitively, a spelling error such as 'confusedment' can be more appropriately described as two edits (DEL 'd' and REPL 'ment' with 'sion') instead of five edits (DEL 'd', REPL 'm' with 's', REPL 'e' with 'i', REPL 'n' with 'o', REPL 't' with 'n').

The string to string model is a more powerful model, since it can better explain both simple typing mistakes and cognitive errors (e.g, misuse of the suffix -ment for the word 'confuse'). Using this model to learning general string to string edits (with the probabilities) leads to a more powerful model that gives significant improvements in accuracy over previous edit-distance based models.

2.2 N-gram Language Model

In the fields of computational linguistics, an n-gram (e.g., "my sincere hole" and "my sincere hope") is a contiguous sequence of n words. But, sequences of characters or parts of speech are also called n-grams. The n-grams are either obtained from a text corpus, or generated by an NLP system.

N-grams are often grouped together by size. N-grams of size 1 are called unigrams, size 2 bigrams, size 3 trigrams, size 4 four-grams, and so on and so forth.

An n-gram model is a type of probabilistic language model for predicting the last item in an n-gram sequences. For example, $Prob(hole|my\ sincere)$ is the probability of having the word "hole" after the preceeding words "my sincere." Using n-grams to predict the next word this way is called **(*n*-1) order Markov model**. So, $Prob(hole|my\ sincere)$ is an example of **second order Markov model**.

Because they are simple and scalable, N-gram models are now widely used in **proba-**

bility, statistical natural language processing, genetic sequence analysis, and data compression. Previously, researcher typically use **trigram models** trained on **tens of millions of words**. Nowadays, it is common to use billions, or even trillions of words to train five-gram (or even higher-order) models.

2.3 Channel Model

The Chanel Model consists of probability function $P(x|w)$ to model how likely an error x would be typed for the intended word w. To cope with data sparseness, this probability is typically broken down to the character level. Intuitively, we can get a pretty reasonable estimate of P(x | w) just by looking at local character context: the misspelled, missing, or unnecessary character, and the counterpart or surrounding letters (e.g, the preceding character). Kernighan, Church, and Gale (1990) propose using four probability functions to model spelling errors on the character level:

- $P_{del}(y|x) = count$ (xy typed as x) / $count(x)$
- $P_{ins}(y|x) = count$ (x typed as xy) / $count(x)$
- $P_{sub}(y|x) = count$ (x typed as y) / $count(x)$
- $P_{trans}(y|x) = count$ (xy typed as yx) / $count(x)$

This is so-called maximum likelihood estimation (MLE). For zero counts, to avoid the dreaded problem of $zero probability$, we have to use a small number instead of zero.

Note that for insertion and deletion, the probabilities are conditioned on the previous character. The choice is somewhat arbitrary. Brill and Moore (2000) propose using a variable-length string (e.g., *ent*) conditioned on a variable-length string (e.g., *ant*) for better results.

We can estimate the character level channel model, by using lists of misspellings like the following training data:

- additional: addional, additonnal
- environments: enviornments, e nviorments,
- enviroments preceded: preceeded.

Where do we get such training data? There are lists available on Wikipedia and from Roger Mitton (http://www. dcs.bbk.ac.uk/ ÌČROGER/corpora.html) and Peter Norvig (http://norvig. com/ngrams/). Norvig also gives the counts for each single-character edit that can be used to directly create the channel model probabilities.

We proceed by first *aligning* x and w on the character level. For example, with the entries of ('televisin', 'telivision') in the training data, we obtain the alignment,t e l i | e v i s | si o n' which accounts for the following events:

- (unchanged): t e l e v i o n
- (insert): i | io
- (replace): i | e

2.4 Work Sheet

Write a program (spell.ncm.py) to perform spelling check with channel probabilities in addition to word probability. For this, you either use

- Existing edit data from Peter Norvig (`norvig.com/ngrams/count_1edit.txt`)
- Write a function to generate edit instances and estimate the channel probability from wrong-correct word pairs (download **gecSpellDict.py** from course website).

2.4.1 Channel Probability

Write a function to read a list of wrong-correct edits, and estimate channel probability function. Notice the channel probability is condition on **correct** rather than *wrong* due to the manipulation of Baye's Theorem of conditional probability. In other words, estimate the probability function P(wrong, correct) as follows:

$$argmax_c P(c|w) = argmax_c P(w|c)P(c)/P(w)$$

$$= argmax_c P(w|c)P(c) \qquad \text{because } P(w) \text{ is constant}$$

$$N = sum(count(w,c)\ for\ count(w,c) >= 1)$$

$$N_r = distinct - number(w,c\ for\ count(w,c) = r >= 1)$$

$$N_{all} = N_1 + N_2 + N_3 + ...$$

$$N_0 = 26*26*26*26 + 2*26*26*26 + 26*26 - N_{all}$$

$$count_1(w,c) = (r+1)*N_{r+1}/N_r \qquad if\ 0 <= count(w,c) = r <= 10$$

$$Pedit(w,c) = count_1(w,c)/count(c)$$

Note that this is a simple version Good-Turing smoothing (see `en.wikipedia.org/wiki/GoodâĂŞTuring_frequency_estimation`). As for N_r, the distinct numbers of events with a certain count r, you can compute the size **keys()** to calculate them. So, N_{all} is simply the size of **count.keys()**.

```
count = dict( [  ______________ for ___________ in open('count_1edit.txt', 'r') ] )

N = _________________________________

N1 = ________________________________

N0 =  26*26*26*26+2*26*26*26+26*26 - 2*N1

def Pedit(w, c):

  if (w, c) in count:

      _________________________

  else:

      _________________________
```

2.4.2 Combining channel probability with word probability to score states

Rewrite the probability of state, Ps(state) to factor in the edit cost. First, change the representation of state as [L, R, edits, Pw, Pedit] where

- L and R are the edited and unedited parts of 'wrong'
- edits are the accumuated edits
- Pw is the probabilities of L+R
- Pedit is the accumulated edit probability

Then, revise the next_states() function to work with the new representation of state. However, to avoid underflow (when the probability get dangerous close to zero), use log(probability) and addition of logprob instead of multiplication of prob.

2.4.3 BONUS: Alignment and Error Counts

Write a function called $spellAlign(word, correction)$ which returns the minimal edit-distance state-transition leading from *word* to *correction*. You may want to modify the correct() function in Worksheet #1. Following the notation of $count_1 edit.txt$, use 'wrong | correct' to represent edits and ignore 'no edit'. Also use the previous character as part of the edit, in the case of INSERT and DELETE. For example, the two edits to transform 'appearant' correctly to 'apparent' are 'pe|p' and 'a|e'. Note that the vertical is just a separator and has nothing to do with conditional probability.

Also, to make your code shorter and keep debugging time to a minimum, use **Counter** from the **collections** module.

You should do the following to obtain the training data of word-correction pairs (with counts).

```
from gecSpellDict import spelldict
```

```
def spellAlign(word, correct):

    L, R, edits = '', word, ''

    states = [ [L, R, ''] ]

    for i in range(len(word)+______ ):

        states = [ state for states in map(next_states, states) for state in states ]

        states = [ state for state in states if state[0] _____________________   )


    states = [ state for state in states if state[0] _____________________ )

    states.sort( ________________________________________________ )

    return _______________________


$ python -i spell.align.py

>>> spellAlign( 'appearant', 'apparent' )

pe|p e|a

>>> spellAlign( 'aesy', 'easy' )

ae|ea

>>> spellAlign( 'languge', 'language' )

u|ua

>>> spellAlign( 'noisy', 'noise' )

y|e

>>>
```

References

1. Daniel Jurafsky and James H. Martin. 2017. Speech and Language Processing An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition, 3rd Edition. Chapter 5 Spelling Correction and the Noisy Channel.

(draft available at https://web.stanford.edu/~jurafsky/slp3/ed3book.pdf on the book website https://web.stanford.edu/~jurafsky/slp3/

2. Brill, Eric; Moore, Robert C. (Jan 2000). "An Improved Error Model for Noisy Channel Spelling Correction". Proceedings of ACL 2000.
3. Kukich, Karen. 1992. Techniques for automatically correcting words in text. ACM Computing Surveys 24(4):377-437.
4. Shannon, Claude. 1949. Communication in the presence of noise. Proceedings of the IRE. 37(1):10-21.
5. Peter Norvig. 2007. How to Write a Spelling Corrector. On http://norvig.com.
6. Tom White. Can't beat Jazzy. IBM Developerworks.