Almir Aljic & Alexander Jakobsen
Royal Institute of Technology (KTH)
DD1418 Language Engineering with Introduction to Machine Learning
12 December 2019

Implementing a Word Predictor in Python

TASK

The objective of this project was to write a program in Python capable of predicting words, with functionality similar to that of word prediction software found on most smartphones. In brief, a good word prediction tool should, based on previous user input, 1) accurately predict potential completion words in a sentence 2) check for spelling errors and suggest corrections accordingly and 3) learn new words.

BACKGROUND

There are two primary domains of interest in implementing a word predictor as previously outlined; n-grams and the correction of spelling errors. The *Markov Assumption* states that a word only depends on the previous n words. Mathematically, we can express this as a conditional probability: $P(w_i \mid w_1, \ ..., \ w_{i-1}) \ = \ P(w_i \mid w_{i-n}, \ ..., \ w_{i-1})$, where $w_{i-n}, \ ..., \ w_{i-1}$ denotes the n words preceding the word given by w_i. For instance, $P(\text{"}you\text{"} \mid \text{"}how\ are\text{"})$ is the *trigram probability* that "you" will appear after "how are" $(n \ = \ 2)$.

Given a large *training corpus*, one may apply the *Maximum Likelihood Estimation (MLE)* method in estimating the sought n-gram probabilities. This is done by computing the relative frequency of the n- and $(n-1)$-grams as follows:
$P(w_i \mid w_{i-n}, \ ..., \ w_{i-1}) \ = c(w_1, \ ..., w_i) \div c(w_i, \ ..., w_{i-1})$, where c denotes a count function. Using the previous example, an estimation of the conditional probability $P(\text{"}you\text{"} \mid \text{"}how\ are\text{"})$, based on some training corpus, would be equal to the frequency of occurrence of "how are you" divided by the frequency of occurrence of "how are". Similarly, the following holds for

unigrams: $P(\text{"}how\text{"}) = c(\text{"}how\text{"}) / N$, where N is the total number of words in the training corpus.

Given a word from user input, how can a word prediction tool determine that the word has been misspelled and then suggest appropriate spelling corrections? By maintaining a *vocabulary*, generated from a training corpus, the word predictor can check if the user's input exists in the vocabulary. If not, the system may flag the input as a potentially misspelled word. How can the system then determine which word the user intended to type? Consider the following algorithm:

Premise: User has typed a non-existent word w.
for each word with edit distance less than or equal to m from w
    if word in vocabulary
        recommend word to user

The *edit distance* may be defined as the *Levenshtein distance*, also known as the *minimal edit distance*. Most commonly, an insertion/deletion has a cost of (edit distance) $m = 1$, whereas a substitution/transposition has a cost of 2. By generating all possible permutations of a misspelled word, such as "thye", with Levenshtein distance $m = 1$ (only one insertion/deletion), the following set would be obtained:
{"*hye*", "*tye*", "*the*", "*thy*", "*athye*", "*bthye*", ..., "*tahye*", "*tbhye*" ...}. From this set, one may conclude that the subset given by {"*the*", "*thy*"} contains the word the user intended to type. Note that the derived subset is a result of "the" and "thy" appearing in the vocabulary.

HYPOTHESIS

The word predictor program will predict words accurately, as long as a large training corpus containing text on different topics/from different sources/authors is used and a large enough maximal edit distance is chosen for the spelling correction algorithm.

IMPLEMENTATION

BigramTrainer.py from assignment 2 was extended to output a text file containing a *language model* including bigram *and* trigram probabilities. The main class, found in WordPredictor.py, reads the language model and stores the information in several different dictionaries. All unique

words are assigned a unique index number, and their frequency of occurrence is stored in the class variable unigram_count. Furthermore, all bigram- and trigram probabilities are found in the class variables bigram_prob and trigram_prob, respectively.

The base implementation, i.e. running the program without the '-s' flag, is best illustrated using an example: assume the user has typed the sequence of words given by $w_1, ..., w_{n-1}$ and is about to type w_n (although has not begun typing w_n). First, all trigram probabilities are examined. These probabilities are given by the dictionary 'self.trigram_prob[w_{n-2}][w_{n-1}]'. For each word (i.e. each word that succeeded the sequence [w_{n-2}, w_{n-1}] in the training corpus), the word with the highest probability of appearing after the preceding two words are recommended to the user as completion words. For each new letter the user types, the list of recommended completion words is updated accordingly; only those words that begin with the same letters the user has typed are considered. The number of words to recommend is also known as the *prediction window size.* For cases where the language model lacks trigram probabilities, the *backoff* technique was implemented. In other words, if there are no trigrams, then all possible bigrams are examined. If there are no bigrams, then all unigrams are examined. *Immediate prediction* is used, which means that even before the user has typed their first letter, the top words from the trigrams or bigrams or unigrams are recommended.

When the user inputs a space, or chooses a word from the list of recommended completion words (a space is then automatically inserted), the word completion tool determines it is the end of the word. If there are no recommended words given a set of user-inputted characters for word w_n, the system suspects that the user might have misspelled. At this point, the program calculates all possible permutations of the user-inputted characters with edit distance less than or equal to $m = 2$. Note that a single character insertion, deletion, transposition or replacement has been defined as having a cost (edit distance) of 1. Of the possible permutations, only the words that appear in the vocabulary are considered. Of those words, the ones with the highest unigram counts are recommended as potential completion words. Preferably, however, the corrected words should be recommended based on their bigram and trigram probabilities. If the user at this point inputs a space instead of choosing one of the recommended words, then the word is added

to the vocabulary as a new word, and its unigram count is set to 1. Note that, for the sake of simplicity, the context in which new words were written is not recorded, which means that bigram_prob and trigram_prob are not updated (although ideally should be).

The '-s' flag enables the user to supply it with a test file. The test file ought to be a .txt file, and should contain, for instance, a newspaper article or some other text one might wish to compose. The program will then use the method employed in the base implementation described above (although without spelling correction) to compose the text found in the test file, while counting the number of total keystrokes required to compose the text, and also the total number of keystrokes the user would have had to make with the aid of this word prediction tool.

DATA

For this project, guardian_training.txt from assignment 2 was used to generate the language model. Two data sets were collected and used as simulations to determine how many keystrokes a hypothetical user would have saved while using the word prediction tool. One of the data sets was a corpus containing text messages, which included lots of misspelled words, abbreviations and slang. The other data set contained articles from BBC Politics. Additionally, a test set containing misspelled words was collected and used to determine the accuracy of the spelling correction algorithm. **For information on how to retrieve & compile the aforementioned data, please consult the file README.md.**

RESULTS

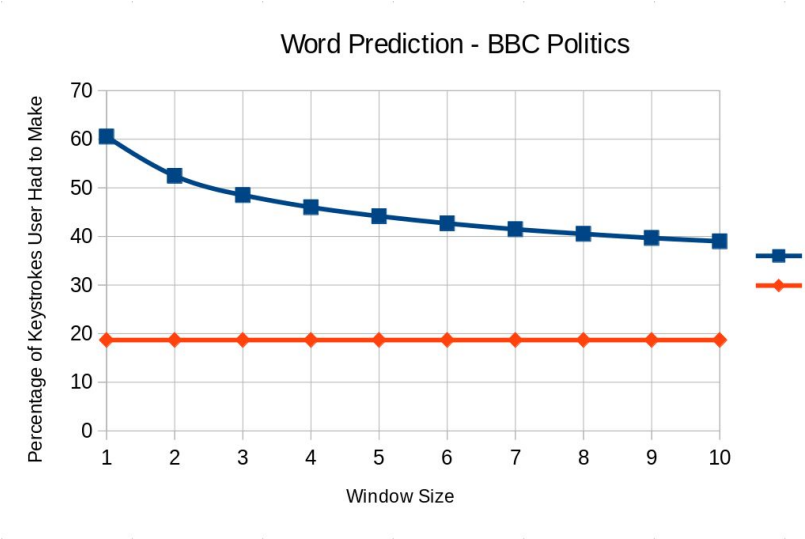

Figure 1. Graph shows the percentage of keystrokes the user had to make for the test corpus 'BBC Politics'. The red line shows the theoretical limit, and the blue line shows the actual percentages.

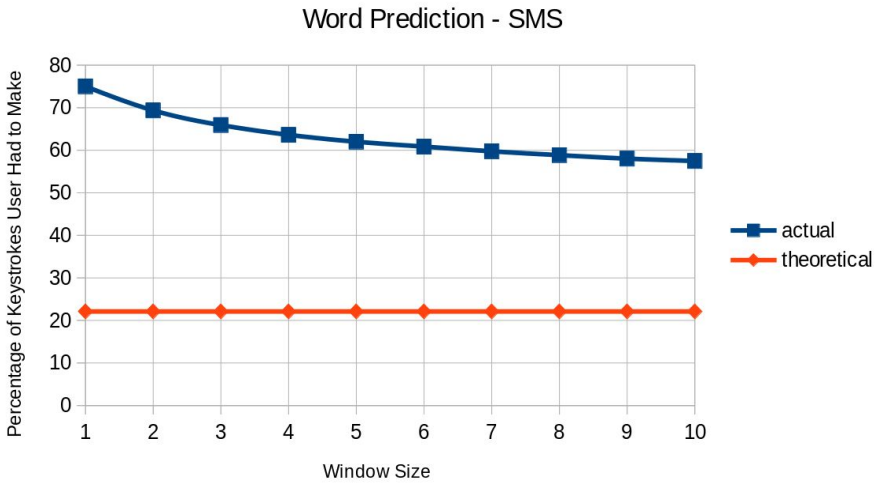

Figure 2. Graph shows the percentage of keystrokes the user had to make for the test corpus 'SMS'.

The results in the graphs above were generated by running the main class found in WordPredictor.py, with values of self.num_words_to_recommend varying from 1 to 10 (window size), using the '-s' flag to supply the BBC Politics test corpus and the SMS test corpus, respectively. The program then calculated, for each window size, the number of keystrokes the user would have had to make and the total number of keystrokes (without word prediction) required to compose the test corpus being examined. By dividing these two key values, the graphs above were able to be produced. The theoretical limit assumes that the word the user intended to type immediately shows up as a recommended word (without the user inputting a single character), which means the number of keystrokes required to compose the entire test corpus would be equal to the total number of words it consists of (remember, spaces are automatically inserted when choosing a recommended word!). As for the accuracy of the spelling correction algorithm, it was found to be equal to roughly **42%** when using a window size of 3, on the test data given by missp.dat. **Out of 36 133 misspelled words, 15 201 were recommended** to the user.

DISCUSSION, EVALUATION & CONCLUSIVE REFLECTION

If the chosen window size is too small, then accurate word predictions may be forsaken, which could lead to sub-par performance. In this implementation of a word predictor, choosing a larger

window size does not cost more in terms of computing power; the same computations are made regardless. With that said, it is not a valid reason to maintain a narrow window size. However, smartphones for instance generally have smaller screens, and a common limit in that context seems to be a window size of 3, which does not provide optimal performance but is certainly better than recommending just one word. Increasing the window size by one step for sizes less than or equal to five generally provides better improvement than increasing a window size already larger than 5. For instance, the results from figure 1 show that increasing the window size from 2 to 3 decreases the number of required keystrokes by roughly 6%, whereas an increase from 7 to 8 is an improvement of less than 2%.

The actual percentages of keystrokes the user had to make, for both test corpuses, deviate significantly from the theoretical values. This is not to say that this word prediction system performs poorly, because the theoretical values assume that the system immediately guesses the next word the user intends to type, without the user inputting a single character. This would mean that the only keystrokes the user has to make are selections of the recommended words. But, there is a distinct difference between the performance of this word predictor when tested on the BBC Politics test corpus as compared to when tested on the SMS test corpus. The reason for this is because the language model was generated from The Guardian, which is a British daily newspaper, similar to BBC, which is a British public service broadcaster. Perhaps most importantly, this implies that both the language model and the test corpus stem from British English. It would be interesting to compare the word predictor's performance when using an American daily newspaper for the test corpus (this was not done in this paper because of time constraints). This could potentially shed light on the differences and importance of distinguishing between different dialects in word prediction. Additionally, the fact that the training corpus and the test corpuses were written by British journalists is probably another factor not to be overlooked. The word predictor performed significantly worse when tried on the test corpus 'SMS'. The reason is quite clear; the text messages in SMS contain lots of abbreviations, slang and misspelled words, which simply does not occur in the articles written by professional journalists, which constituted the training corpus.

Using a fixed window size of 3, the spelling correction algorithm performed quite poorly on the given test data set. Two potential explanations, which are highly interrelated, are found in 1) the vastness of the training corpus and 2) the nature of the test data set. The larger the training corpus, and the more diverse sources it contains, the more likely it is that the resulting language model will consist of a vast and diverse vocabulary. This minimizes the risk of a correctly spelled word from the test data set not existing in the vocabulary. In other words, this is perhaps what caused the low accuracy presented in this paper. If a correctly spelled word does not exist in the system's vocabulary, then even if the spelling correction algorithm generates the correctly spelled word for all misspelled variants of the word, the system will not be able to recommend it to the user. With that said, the nature of the test data set used was such that, for some correctly spelled words, there are lots of misspelled variants of it. So, if it turns out that the correctly spelled word does not exist in the vocabulary, the resulting accuracy may be disproportionately affected. Additionally, it might be that the word exists in the vocabulary but is capitalized or in some other way slightly modified, which would skew the accuracy as described. According to Peter Norvig, a good spelling corrector can undoubtedly score a 90% accuracy, if it were to use n-gram contexts for evaluation. In other words, there are likely much better implementations of spelling correctors elsewhere.

In conclusion, it appears to be of utmost importance that the training corpus used to generate language models from is vast and contains data from different sources/authors, or is specifically designed to be used in a specific domain. An example of this is clearly illustrated by the results of this paper, which indicate that generating a language model from a training corpus consisting of professional journalist's articles is more likely to produce better results in word prediction if tested on other professional journalist's writings. Similarly, employing a word predictor for text messages ought to produce better results if the language model is generated using a training corpus consisting of other text messages. Alternatively, perhaps, it is possible to use a training corpus consisting of both professionally written texts and text messages so that the resulting word predictor may be used for *both* newspaper articles and text messages. This is left as a

pointer for future work. Lastly, a good spelling corrector ought to take into consideration the context in which misspelled words were written, on-top of having a vast vocabulary to determine when a misspelled word has been accurately corrected.

WORKS CITED

Trnka K., Yarrington D., McCoy K. & Pennington C. *The Keystroke Savings Limit in Word Prediction for AAC*. University of Delaware & AgoraNet Inc, Newark DE. Link: https://pdfs.semanticscholar.org/ff4d/c78e6016c47ed0f5d347a9944b8da8443236.pdf

Almeida, T.A., Gómez Hidalgo, J.M., Yamakami, A. *Contributions to the Study of SMS Spam Filtering: New Collection and Results*. Proceedings of the 2011 ACM Symposium on Document Engineering (DOCENG'11), Mountain View, CA, USA, 2011.

D. Greene and P. Cunningham. *Practical Solutions to the Problem of Diagonal Dominance in Kernel Document Clustering*, Proc. ICML 2006.

Boye, Johan. 2019. *Statistical Properties of Languages.* Lecture 4. [PowerPoint].

Boye, Johan. 2019. *Part-of-speech Tagging and Hidden Markov Models*. Lecture 5. [PowerPoint].

Boye, Johan. 2019. *Basic Text Processing and String Alignment*. Lecture 2. [PowerPoint].

Kann, Viggo. 2019. *Svensk stavnings- och grammatikkontroll.* [PowerPoint].

Norvig, Peter. 2007. *How to Write a Spelling Corrector*. [Blog]. Link: https://www.norvig.com/spell-correct.html.