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 *text 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 our 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 text 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. For instance, consider the user input given by "thye", which is a non-existent word (i.e. does not exist in our hypothetical vocabulary) and thus means

that we may assume it to be misspelled. By generating all possible permutations of "thye" with Levenshtein distance $m = 1$ (only one insertion/deletion), we would get a set corresponding to {"*hye*", "*tye*", "*the*", "*thy*", "*athye*", "*bthye*", ..., "*tahye*", "*tbhye*" ...}. From this set, we 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 file containing the language model has the following structure:

<number of unique words in training corpus> <number of total words>

<i> <word$_i$> <frequency of occurrence of word$_i$>

...

<i> <j> <P(word$_j$ | word$_i$)>

...

<i> <j> <k> <P(word$_k$|word$_i$, word$_j$)>

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 implementation 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 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 our 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. 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. This indirectly implies that the word with the highest unigram_count is recommended at the beginning of a message the user is typing.

When the user inputs a space, or chooses a word from the list of recommended completion words, 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. However, 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).

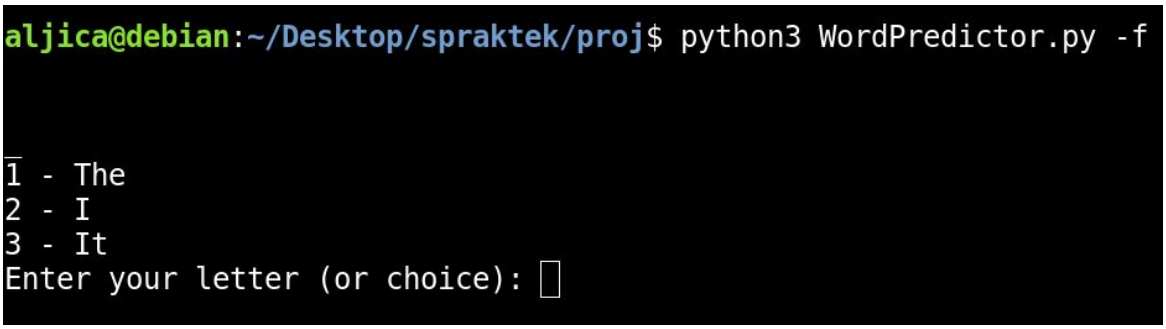
RESULTS & DATA

For this project, guardian_training.txt from assignment 2 was used to generate the language model. This is a relatively large set of data, containing close to 9 million total words and almost 200 000 unique words. The cross-entropy results from problem 3 in assignment 2 were used to assume that the language model would "learn well" from the training corpus. Note that this is not completely certain, because the results from that assignment were specific to bigrams. The cross-entropy of the test set on the language model using trigrams has not been computed! The reason why this information was not gathered is because this project was primarily about demonstrating the concept of how a word predictor could be designed and implemented, and not necessarily about determining the validity of a language model. In other words, *intrinsic evaluation* of the n-gram data was not explicitly performed for the sake of this project. Instead, *extrinsic evaluation* in the form of using the data in the word prediction tool was employed. The results from problem 3 in assignment 2 are outlined in table 1 below.

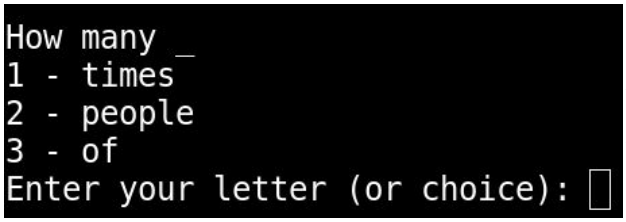
Language model/Test corpus	Guardian test	Austen test
Guardian model	**6.62**	**6.40**
Austen model	**9.75**	**5.72**

Attachment 1, table 1. The cross-entropy of test sets on language models.

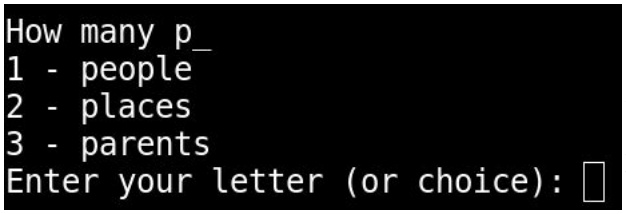
The results, which primarily consist of using the finished word prediction tool, are generally outlined by the following image attachments:



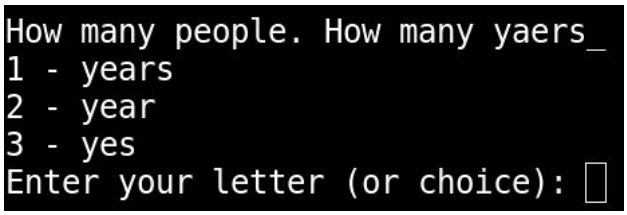
Attachment 2, image 1. Starting the word completion tool.



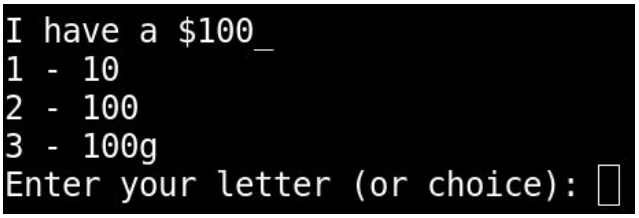
Attachment 3, image 2. List of recommended words given the user input “How many”.



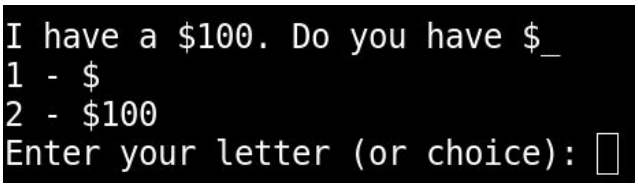
Attachment 4, image 3. Inputting a letter narrows the possible completion words.



Attachment 5, image 4. Misspelling a word also yields possible completion words.



Attachment 6, image 5. Demonstration of how the system learns new words.



Attachment 7, image 6. Demonstration of how the system learns new words, continued.

DISCUSSION & CONCLUSIVE REFLECTION

The choice of n significantly impacts the time, computing power and training data size required to accurately estimate the n-gram probabilities, which are vital to any word prediction system. It goes without saying that the larger the n, the more language structure is captured. The trade-off is that for 4-grams and higher, one must have an incredibly vast training corpus, comparable to the size used to generate Google's N-gram viewer. Note, however, that regardless of the choice of n, there is the lingering problem of *long-distance dependencies*. With that said, the estimation of trigram probabilities using the Guardian training corpus, as was done in this project, is actually somewhat questionable. Ideally, one would like to have as large a corpus as possible, but this was not possible due to computational (and time) constraints. Additionally, it is advisable to compute the cross-entropy of the test set on the language model, to ensure that the language model accurately reflects the data extracted from the training corpus.

On another note, there is an important flaw in the use of the backoff technique in the context of handling missing trigram probabilities. Take, for example, the following beginning of a sentence: "*We like jumping*". Assume that the user has written "*We like jum*" and the word prediction system finds no trigram probabilities in the given context, with words starting with "*jum*". Then, the system falls back to examining potential bigrams, but finds none. The system finally examines all individual words, and notes that "*jump*" and "*jumping*" both exist in the vocabulary. However, it turns out that $P(\text{"}jump\text{"}) > P(\text{"}jumping\text{"})$, i.e. the word "*jump*" had a higher relative frequency of occurrence in the training corpus. This means that the word "*jump*" will be recommended as the top completion word, despite the obvious grammatical error. One way of dealing with this problem is by assigning a *part-of-speech* (POS) tag to each word in the text, and then using those tags to determine the grammatical validity of a sentence. There are other ways of dealing with these kinds of grammatical inconsistencies, for instance through employing rule-based methods such as *finite state parsing* etc.

In this word prediction tool, misspelled words are corrected by computing all possible permutations of the word (using a predetermined tolerance in edit distance), and recommending the word with the highest relative frequency of occurrence. Note that this means the resulting words are ranked solely by their unigram count, and not by the context in which they appeared. Ideally, the system should retrieve the probability of the corrected word appearing in the context given by the n previous words written by the user. Should such n-gram probabilities not exist, an appropriate method such as backoff can be used. In the same way, the context in which new

words were added to the vocabulary is not recorded. Also, each time the user types a word, its frequency of occurrence (unigram count) is incremented by one. It is important to mention this because the more times a word is written, the higher is its probability of appearing as a unigram, according to the system. Ideally, however, the corresponding bigram and trigram probabilities should also be updated, but are not, for the sake of simplicity. This functionality ought to be improved in later versions of the program.

There is an important issue to address in regards to computing all possible permutations with a predetermined edit distance of a misspelled word. The higher the tolerated edit distance, the greater the number of possible permutations, which requires more time and/or computing power. In other words, choosing too large a value for the edit distance m runs the risk of rendering the word prediction tool inefficient. By choosing $m = 2$, the most common spelling mistakes are identified and amended while maintaining a manageable number of permutations, because according to Kann (2019), most mistakes (in text written using a keyboard) are due to 1) two neighboring characters being confounded 2) one character being left out from the word 3) an additional character that does not belong being inserted 4) one correct character being replaced by an incorrect character.

Information is not saved across sessions, which means new words the user has added to the vocabulary are not saved for future reference. This is easily accomplished by structuring a text file according to some pattern, perhaps similar to the structure of the output file produced by TrigramTrainer.py, and writing to the file as the user inputs new words, and then of course

reading the data into appropriate data structures as a new session is started. One way of potentially storing not only new data, but also data given by the language model, is by using a database management system (DBMS) such as PostgreSQL.

WORKS CITED

Boye, Johan. 2019. *Statistical Properties of Languages.* Lecture 4. [PowerPoint]. (Accessed: 2019-12-11).

Boye, Johan. 2019. *Part-of-speech Tagging and Hidden Markov Models*. Lecture 5. [PowerPoint]. (Accessed: 2019-12-11).

Boye, Johan. 2019. *Basic Text Processing and String Alignment*. Lecture 2. [PowerPoint]. (Accessed: 2019-12-11).

Kann, Viggo. 2019. *Svensk stavnings- och grammatikkontroll.* [PowerPoint]. (Accessed: 2019-12-11).

Norvig, Peter. 2007. *How to Write a Spelling Corrector*. [Blog]. https://www.norvig.com/spell-correct.html. (Accessed: 2019-12-11).