An Exploration of Predictive Typing and Its Possible Implementations

George Tolkachev

Abstract

The task of predicting the next word following a particular sequence of words - generally known as *predictive typing* - is highly relevant to applications that require the user to type text into an interface. This paper explores one possible implementation of predictive typing, which uses a Markov model based on *n*-grams. The corpora for training, developing, and testing the algorithm include news articles, blog entries, and tweets. We shall evaluate the system using a standard measure which calculates the percentage of keystrokes that the algorithm saves for the user (that is, the keystrokes that the user does not need to type to get the desired result).

1. Introduction

Many modern technologies, including search engines and chat applications on mobile phones, require users to type text in order to obtain search results or to send a message. Therefore, to save time for the user, such programs can attempt to predict the next word that follows the sequence of words that he or she has typed. Within the realm of computational linguistics, this discipline is widely known as *predictive typing*. The task has many practical applications, one of which is to assist physically handicapped individuals in the process of writing a sentence or generating speech (Boggess, 1988). Such systems fall into the category of Augmentative and Alternative Communication (AAC) devices (Matiasek and Baroni, 2002), (Trnka and McCoy, 2007). In any application, the primary objective of predictive typing is to save effort for the user by automatically suggesting the next word that they would like to type.

In order to generate accurate predictions for the next word, predictive typing systems must collect data on the frequency of words typed by the user over time and their possible orderings within a particular sentence. A good system will take into account not only the previous word when predicting the next one, but the context of the entire sequence typed so far. Such a predictor should also observe how parts of speech link together to create meaning, and should thus ensure that the word options that it offers to the user satisfy these semantic rules. The system described in this paper attempts to account for several of the above considerations when predicting the following word, and uses statistical methods in order to decide which word options make the most sense within the context of the given sentence.

2. Theoretical Background

Consider the task of predicting the ith word following a sequence consisting of i - 1 words (Carlberger, 1998). Then, the probability that this ith word is, say, w_i, can be expressed as $P(w_i | w_1, w_2, \ldots, w_{i-1})$. Therefore, a good predictor will take advantage of the relative probabilities in the process of deciding the value of w_i.

To make the system's predictions more accurate, we can utilize a Markov model, which consists of a set of states and the probabilities of transitions between them. For our purposes, an mth order Markov model stores the last m words in a given sequence. Another relevant concept that we

can apply to improve the accuracy of our predictor is that of *n*-grams, contiguous sequences of *n* words (Gendron, 2015). In effect, an *m*th order Markov model keeps track of the last *m* - 1 grams that the user has typed into the interface. For instance, a second order Markov model can be thought of as a trigram model, in which each state is uniquely determined by the previous two text inputs (Boggess, 1988). Using such a model, we can calculate that $P(w_i | w_{i-1}, w_{i-2}) = C(w_{i-2}, w_{i-1}, w_i)/C(w_{i-2}, w_{i-1})$, where C is the count of *n*-grams. Consequently, an *m*th order Markov model requires *m*+1-grams to be extracted from the training texts to calculate the above probability.

(Carlberger, 1998) draws a distinction between two language models that have been used by predictive typing systems in the past. The old language model considered only the previous word before any letters of the next word were typed. When the user began typing the next word, the model completely disregarded the previous word and simply offered the most common words that started with the first letter typed by the user. As a result of these drawbacks, the old language model was not particularly reliable for making predictions for the next word. Meanwhile, the new language model does take the previous word into consideration, even after the user has begun typing the next word. Furthermore, this model pays attention to the relationships between the various parts of speech within the sentence, and can thus predict a word that would make sense within the given context.

Our system will experiment with both of the language models that Carlberger described in his paper. As a way of imitating the old language model, we shall consider unigrams (i.e. individual words) and attempt to predict each following word in the sequence without considering the context of the sentence. For the new language model, meanwhile, our system will deal with bigrams and trigrams, thus using either one or two of the previous words in order to predict the next word. Implementing both language models will allow us to compare and contrast their effectiveness at suggesting a new word for the given sentence. In the end, we shall draw conclusions about which model was the most effective, as well as suggest areas for further research that can shed more light on the impact that the language model that we choose has on the accuracy of the word predictor.

3. Hypothesis

Clearly, the number of words that the language model considers will ultimately effect how accurate our system will be. The reason for this lies in the fact that a model that considers a greater number of words will have a wider selection of possible phrases that the user could possibly type, and this selection would allow the system to make more educated guesses when predicting a given word. Thus, a reasonable hypothesis would be that the more words the model takes into consideration, the more accurate the system; for our system in particular, this means that the trigram model will be more accurate than the bigram model, which in turn will be more accurate than the unigram model. Indeed, this is the hypothesis that we shall adopt while conducting the prediction experiment, and based on the results that we obtain, we shall see whether this hypothesis holds weight, or whether the accuracy of the predictor relies on other factors besides the number of words in its associated language model.

4. Corpora

The training, development, and test corpora for this project were downloaded from a source created by a collaboration between Johns Hopkins University and the company SwiftKey, and can be found at https://d396qusza40orc.cloudfront.net/dsscapstone/dataset/Coursera-SwiftKey.zip. The training file, called "en_US.news.txt", consists of articles from a variety of American news sources; the development file, called "en_US.blogs.txt", contains blog entries, and the test file, called "en_US.twitter.txt", is comprised of tweets. These files are extremely large in size; as such, due to time restrictions, we have chosen to work with only the first half of the training corpus of news articles, and only the first 30 lines of both the development corpus of blog entries and the test corpus of tweets.

5. System Description

The program, written in Python, takes two command-line arguments, the first being the training file and the second either the development or test file. The program reads information from the training file and collects the frequencies of all occurring unigrams, bigrams, and trigrams. This will form the basis of a Markov model that the program will use to generate the necessary predictions.

It is important to formalize that, as far as our system is concerned, a *word* is a contiguous sequence of characters followed by a whitespace or tab.

The program has three dictionaries, one for each of the possible *n*-grams. It parses the training file line by line and places every unigram (individual word), bigram (two adjacent words), and trigram (three adjacent words) into the corresponding dictionary, and counts their frequencies within the training file. The first step that the program takes when encountering a new word in the line that it is parsing is to "clean" the word, i.e. to convert it to a standardized format before storing it in the appropriate dictionary. This process is handled by the `cleanup(word)` function defined at the beginning of the program. `cleanup` performs the following sequence of operations on the `word` that it receives as input:

1. Convert the word to lowercase;
2. Strip the word of all preceding or succeeding punctuation, such as colons, question marks, etc.;
3. If the word is now of length 0 (meaning it only consisted of punctuation), return "INVALID", the signal that the word should not be considered by the predictor or placed into the dictionary;
4. Strip the word of preceding or succeeding quotation marks (these occasionally possess different forms depending on the encoding of the file, and thus must be handled separately);
5. Iterate through every character in the lowercase, punctuation-free word and return "INVALID" if a non-alphabetic character is encountered (with the exception of a dash (-) or possessive quotation mark ('));
6. Finally, return the resulting word.

This procedure effectively eliminates all words that contain non-alphabetic characters, such as numbers or punctuation marks. This helps to standardize each word within the training file so that the three dictionaries contain only the necessary words.

Once the program has made sure that the word is valid, it checks if it is a contraction (e.g. "don't", "let's", "you've"). If so, the program splits the contraction into each of its individual words and adds them to the appropriate dictionary. This was the reason why `cleanup` counted the ' character as an exception when parsing the word.

Once the program has stored all of the unigrams, bigrams, and trigrams in their respective dictionaries, it sorts them in the following order:

1. Alphabetically
2. For the first letter of each word, by frequency from greatest to smallest
3. If two or more words have the same first letter and the same frequency, sort them alphabetically

This sorting operation is accomplished for the unigrams by the line `alpha_unigrams = sorted(unigrams, key=lambda x: (x[0], -unigrams[x], x))`. What is the purpose of sorting the *n*-grams in this way? When the predictor encounters a new unigram, it uses the first letter to find the appropriate place in the dictionary at which it can begin suggesting options for the next word. First sorting the unigrams alphabetically enables the predictor to find this index more efficiently. When it has done so, it begins going through the dictionary and looking for words that contain the characters that have been typed so far. The more frequent a word was in the training file, the more likely it is to occur again in the development or test file, so the more frequent words are placed earlier in the dictionary than less frequent ones. Finally, if two words have the same first letter and the same frequency, their order in the dictionary does not necessarily impact the overall accuracy of the predictor, but we have chosen to sort them alphabetically for the sake of consistency.

After the dictionaries have been sorted, the program reads the development or test file line by line and attempts to predict the next word in each sequence of words. The program iterates through every word in the line and identifies the current word as a unigram, the current word and the next word as a bigram, and the current word and the two following words as a trigram. For example, if the line consists of 14 words, there will be 14 unigrams, 14 - 1 = 13 bigrams, and 14 - 2 = 12 trigrams. In the case of the unigram, the program does not take into account the words that immediately precede it. Meanwhile, for the bigram, the program knows what the current word is, and attempts to predict the second word in the bigram. Similarly, for the trigram, the program is aware of the current word and the next word, and attempts to predict the third word.

The program first reads a character in the word that it is trying to predict. It then checks the appropriate dictionary and collects a certain number of possible suggestions that it can give to complete the current word. An important factor to consider is what number of suggestions to choose for any given character. For unigrams, we have arbitrarily chosen 3 possible suggestions, as this is the number that many modern texting applications suggest when the user begins typing in a new word. For bigrams and trigrams, the number of suggestions must be much larger, due to the fact that the more words an *n*-gram contains, the more possible combinations of the individual words within the *n*-gram. Thus, if the bigram and trigram models offered only 3 suggestions, they would be far less effective than the

unigram model, due to the greater variety of bigrams and trigrams in comparison to unigrams. For our training corpus in particular, we have found that there are approximately 17 times as many bigrams within the training file as there are unigrams, and about twice as many trigrams as bigrams. As such, since we initially chose to make 3 unigram suggestions, we decided to expand the number of bigram suggestions to 60, and the number of trigram suggestions to 120. This allows the program to account for the greater number of bigrams and trigrams within the training file, and consider equally many possibilities for each model.

Every time that the user types a new character of the following word, the program searches through the dictionaries and obtains the corresponding number of suggestions. If the actual next word or phrase is not one of the suggestions, the user types in a new character, and the program retrieves new suggestions. This process repeats until either 1) the next word or phrase is found within the suggestion list, or 2) the user has typed the next word in full, and thus the predictor proved to be ineffective in that particular instance.

This brings us to the process of evaluating the performance of the predictor, which is described in the following section.

6. System Evaluation

We need to have a useful way of measuring the performance of our predictor. (Matiasek and Baroni, 2002) suggest a measure called the *keystroke savings rate* (*ksr*), which computes the rate of keystrokes that the user did not have to type using the predictor. Mathematically, it is defined as

$$ksr = (1 - \frac{k_i + k_s}{k_n}) * 100\,\%,$$

where k_I = the number of input characters typed, k_S = the number of keystrokes needed for the program to begin generating predictions, and k_n = the number of keystrokes that would be necessary to type the entire sequence without the help of a predictor.

Previous papers, (Trnka and McCoy, 2008) chief among them, have claimed that the *ksr* is an imperfect measure for evaluating the performance of the predictor. According to these authors, one of the most glaring issues with the *ksr* is that "the equation alone does not provide much in the way of interpretation — is 60% keystroke savings good? Can we do better?" Trnka and McCoy attempt to devise practical and theoretical limits for prediction systems. Importantly, they distinguish between two purposes for which predictors are used: word *completion*, in which the user must enter the first character of the next word in order for the system to generate predictions, and word *prediction*, which attempts to predict the next word before the user enters any of its characters. Trnka and McCoy find that the *ksr* in word prediction is higher than that in word completion, both in theory and in practice. Specifically, the theoretical *ksr* limit for word prediction is around 77%, while that for word completion is approximately 58%.

Our system implements word completion by way of the unigram model, and word prediction using the bigram and trigram models. Thus, for the unigram model, $k_S = 1$, while for the bigram and trigram models, $k_S = 0$. These values are important, as they will impact the calculation of the *ksr* for each of the lines within the development and test files. We shall use Trnka and McCoy's theoretical limits as baselines with which to compare the results of our predictor system, and observe which of the three *n*-gram models attains the highest average value of the *ksr*.

7. Sample Input and Output

The program takes a line of words as input and prints 1) the word or phrase that the system is trying to predict; 2) the portion of the word or phrase that the user had to type before the system was able to predict it; and 3) the *ksr* for this particular word for each of the *n*-gram models. (See Figure 1.)

```
UNIGRAM MODEL:
The word is: in
The word so far is: i
The keystroke savings rate for in is: 50.0 %
----------------
BIGRAM MODEL:
The words are: in the
The words so far are: in
The keystroke savings rate for in the is: 100.0 %
----------------
TRIGRAM MODEL:
The words are: in the years
The words so far are: in the y
The keystroke savings rate for in the years is: 80.0 %
----------------
UNIGRAM MODEL:
The word is: the
The word so far is: t
The keystroke savings rate for the is: 66.66666666666667 %
----------------
BIGRAM MODEL:
The words are: the years
The words so far are: the y
The keystroke savings rate for the years is: 80.0 %
----------------
TRIGRAM MODEL:
The words are: the years thereafter
The words so far are: the years thereafter
The keystroke savings rate for the years thereafter is: 0.0 %
```

Figure 1: Sample output for the phrase "in the years thereafter", which appears in the first line of the development file.

```
The average unigram ksr for block 1 is: 45.00090876045076 %
The average bigram ksr for block 1 is: 66.48825921553194 %
The average trigram ksr for block 1 is: 26.376733876733876 %
-----------------
The average unigram ksr for block 2 is: 42.51790591996776 %
The average bigram ksr for block 2 is: 72.35905856595511 %
The average trigram ksr for block 2 is: 37.15213358070501 %
-----------------
The average unigram ksr for block 3 is: 49.602137151156754 %
The average bigram ksr for block 3 is: 80.75480075480074 %
The average trigram ksr for block 3 is: 47.10526315789474 %
```

Figure 2: Output obtained for the 3 blocks within the test file.

To calculate the performance of the predictor as a whole, the program considers "blocks" of 10 consecutive lines each, and takes the average of the *ksr* values computed within that block for each of the three models (we shall analyze only the first 3 blocks of the test corpus due to the time that it takes for the program to train and generate results). These values represent the accuracy of the system for predicting words within the given passage of text.

8. Test Results

The results of the predictor as applied to the test corpus of tweets are shown in Figure 2. We can see that for each of the 3 blocks, the bigram *ksr* is consistently the highest, while the trigram *ksr* is always the lowest, with the unigram *ksr* falling somewhere in-between the two. The lowest value among the 9 presented is around 26.38% for the trigram model, and the highest is approximately 80.75% for the bigram model. Extrapolating from these results, we infer that the bigram model appears to be the most accurate at predicting the next word in the given phrase, while the trigram model is the least accurate.

9. Discussion of Test Results

Recall that our original hypothesis in Section 3 was that the trigram model would be the most accurate among the 3 due to the greater variety of possible combinations stored within the trigram dictionary. However, our results undermine this hypothesis by illustrating that, in fact, the bigram model appears to be the most efficient. What is a plausible explanation for this phenomenon?

Most likely, our prediction that the bigram model would be more accurate than the unigram model was indeed correct because of the larger amount of word combinations that the former offers. As a concrete example, consider the first two words within the test file: "how are". The unigram model only takes into account the word "how" itself when predicting the letters that come after "h", and is thus less likely to predict the word quickly enough, especially considering that it has only 3 letters, the first of which must already be typed in order for the program to start generating predictions. Meanwhile, the bigram model stores all of the possible occurrences of the word "how" in pairs of words, and as a result is better equipped to predict the word "are" that follows immediately after. This theory is supported by the test results, as the unigram *ksr* for "how" is 33% while the bigram *ksr* for "how are" is 100%, meaning that the program was able to predict the next word right away. This result is replicated for the majority of the text in the passage, as indicated by the higher average bigram *ksr* value overall.

Given this outcome, it is reasonable to expect that the trigram model would be even more accurate than the bigram model, and yet, this does not end up being the case. The reason likely lies in the fact that the number of combinations within the trigram model is so large that most trigrams within the dictionary will never be accessed by the predictor. As a result, while the system does benefit from having access to a larger number of phrases, this vast amount simultaneously lowers the system's chances of finding the correct word amid all of the other possible suggestions. In the end, this major disadvantage outweighs the benefits of accessing a wider pool of phrases, reducing the accuracy of the model and leading to a lower *ksr* value.

Of course, one possible solution to this problem would be to increase the number of allowed suggestions accessed by the system after the user types a new character. Yet we allowed this limit to be twice as large for the trigram model than for the bigram (120 compared to 60), and the former is still much more inefficient. Therefore, it is unclear how much further we would have to expand this limit until the two models would be equally as accurate.

10. Conclusion and Further Research

Overall, our hypothesis that the larger the *n*-gram model, the more accurate it would be at predicting the next word was only partly correct. As we have discovered, this notion holds weight only up until a certain extent, as a greater number of possible suggestions possesses the effect of decreasing the probability that one particular suggestion will occur within the passage of text.

Though the experiment cast doubt on our initial proposition, we can say that we have found evidence in support of Trnka and McCoy's claim that the *ksr* for word prediction systems (such as the bigram model) tends to be higher on average than for word completion systems (in our case, the unigram model). Furthermore, the *ksr* values that we obtained during the experiment largely matched up with those that the two authors suggested are most commonly found in practice. Thus, the results provided us with more insight into the relative advantages and disadvantages of each *n*-gram model, and how such models can be utilized to improve (or worsen) the performance of a predictor.

To improve the system in the future, we could consider the relationships between parts of speech within a given sentence. This additional training would enable the system to only suggest parts of speech that make sense following a particular sequence of words, and would eliminate unnecessary suggestions. Additionally, we can explore other methods of "cleaning" the words in a passage of text, such as creating equivalence classes for verbs with "ing" or "ed" endings, and how such techniques impact the accuracy of our predictor. We could investigate larger *n*-gram models and observe the connection between the number of suggestions that they can access and their resulting *ksr* values. We can also look into current research happening within the discipline of predictive typing, such as the use of counterfactual learning for word suggestions (Arnold, Chang, & Kalai, 2017). Finally, we can breach out into other various applications of predictive typing, such as mobile text composition (Arnold, Gajos, & Kalai, 2016) and text prediction for foreign language translators (Foster, Langlais, & Lapalme, 2002).

References

Kenneth C. Arnold, Kai-Wi Chang, & Adam T. Kalai. Proceedings of the The 8th International Joint Conference on Natural Language Processing, pages 49–54, Taipei, Taiwan. http://www.aclweb.org/anthology/I17-2009.

Kenneth C. Arnold, Krzysztof Z. Gajos, and Adam T. Kalai. 2016. On suggesting phrases vs. predicting words for mobile text composition. In Proceedings of UIST '16. https://www.microsoft-.com/en-us/research/wp-content/uploads/2016/12/arnold16suggesting.pdf.

Boggess, Lois, and P. O. Drawer. "Two simple prediction algorithms to facilitate text production." *Proceedings of the second conference on Applied natural language processing*. Association for Computational Linguistics, 1988, http://aclweb.org/anthology/A88-1005.

Gendron Jr, Gerald R., and V. A. Chesapeake. "Natural Language Processing: A Model to Predict a Sequence of Words." http://www.modsimworld.org/papers/2015/Natural_Language_Processing.pdf.

George Foster, Phillipe Langlais, & Guy Lapalme. Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP), Philadelphia, July 2002, pp. 148-155. Association for Computational Linguistics. http://aclweb.org/anthology/W02-1020.

J. Carlberger. 1998. Design and Implementation of a Probabilistic Word Prediction Program, Royal Institute of Technology (KTH). http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=C32A03DC87A6A4E82F9DD0DAC28712A3?doi=10.1.1.55.2073&rep=rep1&type=pdf.

J. Matiasek and M. Baroni. 2002. Exploiting Long Distance Collocational Relations in Predictive Typing. http://www.aclweb.org/anthology/W03-2501.

Trnka, Keith, and Kathleen F. McCoy. "Corpus studies in word prediction." *Proceedings of the 9th international ACM SIGACCESS conference on Computers and accessibility*. ACM, 2007, *https://www.eecis.udel.edu/~mccoy/sig-nlp-fall07/Trnka-corpus_study.pdf*.

———. "Evaluating word prediction: Framing Keystroke Savings." Proceedings of ACL-08: HLT, Short Papers (Companion Volume), pages 261–264, Columbus, Ohio, USA, June 2008. *http://aclweb.org/anthology/P08-2066*.

T. Wandmacher and J. Antoine. 2007. *Methods to Integrate a Language Model with Semantic Information for a Word Prediction Component*. *http://www.aclweb.org/anthology/D07-1053*.