Practically Testing Context Dependent Language Models and Context Independent Language Models Using Differing String Comparator Metrics From the Perspective of Autocorrection

Ignacio Arruza, Matthew Kazan, Matthew Kugler

CS5100 Spring 2022, Northeastern University

{arruza.i, kazan.m, kugler.m} @northeastern.edu

Github Repository

Excel Data Sheet

Abstract

Our goal for this project was to research and develop comparative results for a class of language model algorithms which seek to recognize and correct commonly misspelled words. We have implemented two main methods of autocorrect in order to study the accuracy and relative time requirement in seconds for each. These python implemented methods involve the language model algorithms of edit distance and N-gram evaluation. We utilized public Twitter [1] and Reuters [2] datasets for this research. These sources contained numerous sentences that we were able to train our models to use and thus identify misspelled words in order to provide a correction. The comparisons we decided to make in our research were between a "Naive" pure edit distance comparison, a "Bigram" or two word algorithm which predicts the intended word based on the word directly before or directly after, and a "Trigram" or three word algorithm which predicts the intended word based on two words before or after the misspelled word. Our results indicated that the highest accuracy method was the Trigram implementation using the Twitter dataset, and that accuracy was improved when running Trigram evaluations compared to Bigram and Naive for this dataset. We note that the Naive implementation did have the fastest runtime, yet with lower accuracy than the Twitter Trigram method. In addition we saw a significant increase in runtime when using the Trigram implementations of Twitter and Reuters compared to their Bigram counterparts. However the goal of this research was accuracy, and optimizing the runtime of the program was not seriously pursued. Though on a larger scale, time may be a more important consideration.

Introduction

Autocorrect is a commonly implemented feature in many messaging services and word processors. In the past decade the majority of apps that require a user input have some kind of an autocorrect implementation. We set out to explore some basic autocorrect development processes by utilizing concepts of edit distance and N-gram and comparing their effectiveness against each other. When approaching our research question we wanted to present a range of potential implementations of autocorrect in order to paint a wide picture of the benefits and disadvantages of each. Including a Naive algorithm alongside more advanced implementations such as Bigram and Trigram algorithms was meant to show what kinds of improvements and results can be expected from N-gram compared to edit distance. Improving the quality and consistency of these autocorrect algorithms represents a large benefit to the overall productivity of millions of potential users across a broad range of applications [3].

.

Background

The two main algorithms we used to develop our autocorrect comparison were edit distance and N-gram probability (specifically Bigram and Trigram probabilities). Edit distance is simply a way of determining how dissimilar two strings are to each other. This is determined by counting the number of operations that are required to transform one string into another. In the scope of our autocorrect program, we implemented functionality that uses edit distance to determine possible words that would replace a misspelled word from an input, and return the most likely intended word based off of the lowest edit distance. The most basic definition of an N-gram is a

sequence of N words. We're able to use these sequences to create a model of probability for what words are expected to be found together. The probability of an N-gram is modeled by the following expression, which compares the probabilities of words that could potentially follow each other based on a learned dataset of sentences:

$$\text{Probability of N-gram:} \quad P(w_N|w_1^{N-1}) = \frac{C(w_1^{N-1}\, w_N)}{C(w_1^{N-1})}$$

By breaking an input sentence into a series of words, we're able to apply the following chain rule of probability to determine which words are expected to be found next to each other and thus determine what the intended word provided by the input was:

$$\begin{aligned} P(w_{1:n}) &= P(w_1)P(w_2|w_1)P(w_3|w_{1:2})\ldots P(w_n|w_{1:n-1}) \\ &= \prod_{k=1}^{n} P(w_k|w_{1:k-1}) \end{aligned}$$

This is the foundation of autocorrect by the concept of N-grams. There are parallels with this method to an auto-complete implementation which requires a more advanced language model outside the scope of this research.

Related Work

The widespread use of autocorrect means that there are a lot of implementations and variations of these algorithms. Most recently, the development of Neural Language Models has had a profound impact on the field. This class of algorithms use continuous representations or embeddings of words to make their predictions. They function similarly to how the brain stores information and creates new connections, by creating new nodes for words and establishing common connections between those words in a training manner [4]. This method of autocorrection significantly outperforms the N-gram and word edit distance based models, however implementing such an algorithm was beyond the scope of this project. These algorithms require rigorous training on a very diverse dataset in order to establish the deep neural network connections that are required. Future research surrounding language models and autocorrect should definitely look to improve on the accessibility of these neural models and push them as the standard for this field of work due to their efficiency and accuracy.

Approach

For our first algorithm, we wanted to try a simple, naive approach to autocorrection. We would not consider the word's context, only the misspelled word itself. To do this, we implemented our own version of the Levenshtein distance [5] to compute the minimum edit distance between the misspelled term and a given word. Both algorithms begin in the same manner. They are given a sentence or list of words, and the autocorrect program scans through it and looks for any words that are not in its vocabulary. If it finds one, it will send that word to the desired autocorrect algorithm and any further information the algorithm might need. Both algorithms need the dictionary of terms to term counts, while the N-gram algorithm also needs the context the word was in and a precomputed model for probabilities. The algorithm will then get a list of similar words by scanning through all terms in the vocabulary and saving all words within some cutoff criteria.

For the naive algorithm, the criteria chosen was that the min edit distance must be less than the length of the misspelled word minus one. This was chosen somewhat arbitrarily on the notion that if more than half of the letters are different, it is unlikely to be an accurate correction. We decided to scan through the list of words instead of simply computing all mutations within a certain edit distance because our vocabulary was small enough that it was faster to scan through the entire list than to get a list of mutations that would be equivalent in accuracy. Also, it was far easier to use threads to speed up the task of scanning through a list and independently saving certain words that match a criterion than it would have been to use threads with the mutation algorithm. Once the similar word list was compiled, each term was assigned a probability which was simply its term frequency. The final result was computed based on three cases. First, it was chosen if any word was the most likely and had the smallest edit distance to the misspelled word. If no such word existed, and none of the similar words occurred significantly in the corpus, the best word was the word with the smallest edit distance. Finally, if no other option were available, the best word would be chosen by finding the highest value when the probability of a term was divided by its edit distance raised to itself. This formula was chosen to value high probabilities and very low edit distances. Higher edit distances make the term exponentially less likely to be chosen.

For the N-gram algorithm, we used the Jaro-Winkler similarity [6] between a term and the misspelling had to be less than .8. This was chosen entirely based on testing and finding the maximum value that rarely excluded the actual correction. Once a similar word list was found, a score was computed for each term to find the best final choice. This score was computed by summing the log score of each term using the nltk language model library [2] given all possible contexts. For example, if three were chosen for n, then the last three terms need to be given for context, and the algorithm would sum the log score for the term given the previous term and the term given the previous two terms. Once this sum was found, it was normalized, multiplied by an arbitrary weight, and then summed with the Jaro-Winkler similarity score. The overall score for each term was maximized to find the best choice.

Once the best choice was found, it was then appended to a list of tuples containing the previous best choices and their scores. Then the term gets substituted into the sentence so future N-gram autocorrections would not have misspelled words in their given context. In vocabulary words would be added to the corrected list with a score of 1.

Experiments and Results

We performed comparison experiments for Bigram (2 word) and Trigram (3 word) N-gram algorithms of both the Twitter and Reuters datasets alongside the Naive word edit distance algorithm. An input file containing ~50 phrases with scattered spelling mistakes that varied in complexity and length was used with these datasets and algorithms. Accuracy comparisons were composed based on the number of correct predictions by the algorithm divided by the total number of incorrect words in the input file phrases. Time was measured by an embedded method in the program to output the time spent in seconds correcting the spelling mistakes.

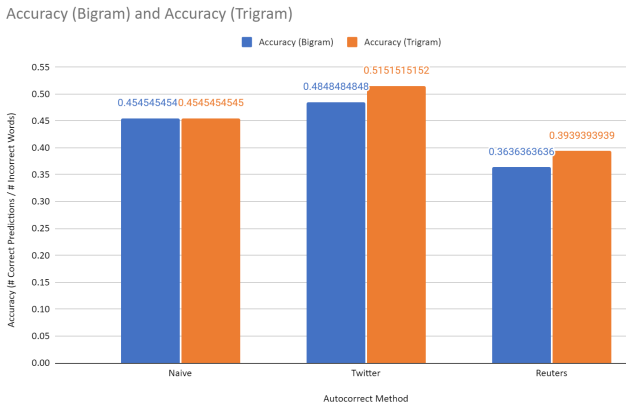

Figure 1: Comparisons of the accuracy for Naive (word edit distance - Twitter dataset), Bigram and Trigram algorithms using the Twitter and Reuters datasets. The graph denotes accuracy as the number of correct predictions divided by the number of incorrect words in the input for each algorithm.

Figure 1 shows the accuracy results of comparing our Naive, Bigram and Trigram algorithms on the Twitter and Reuters datasets. We see that Reuters had the lowest accuracy of the three methods, with a 36.364% result for Bigram and 39.394% for the Trigram algorithm. This is likely due to the Reuters dataset being less complete than the Twitter algorithm and not having as many words to reference when trying to correct the misspelling. The Naive algorithm done via word edit distance showed a 45.455% correction accuracy, however this value is still lower than the Bigram and Trigram implementations of the same Twitter dataset. This is expected due to the more complex nature of the N-gram algorithms compared to word edit distance. We see the Twitter Bigram accuracy value of 48.485% and the Twitter Trigram accuracy of 51.515%. The Bigram and Trigram implementations on the Twitter dataset yielded the most accurate results from our experimentation, and outperformed the Naive algorithm by a significant margin. This improvement in performance was expected and is exciting so see, as we've shown that implementing a more complex language model that uses N-gram prediction is more accurate than a simpler word edit distance model.

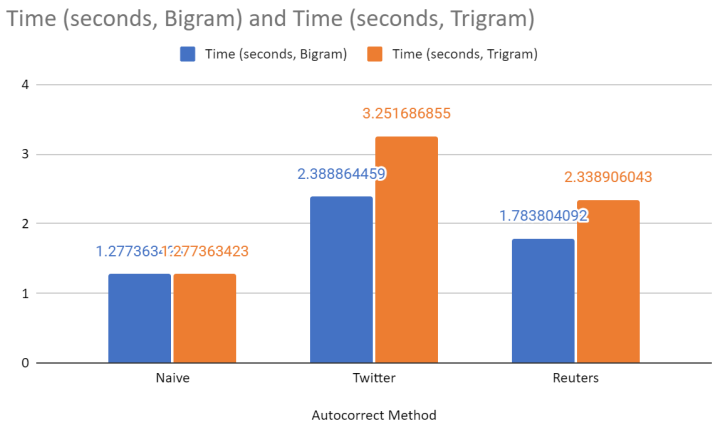

Figure 2: Comparisons of the time in seconds for Naive (word edit distance - Twitter dataset), Bigram and Trigram algorithms using the Twitter and Reuters datasets. The graph denotes time in seconds spent running the respective algorithms on the same input and trial from Figure 1.

Figure 2 shows the runtime results of comparing our Naive, Bigram and Trigram algorithms on the Twitter and Reuters datasets. We see that the Naive algorithm had the lowest runtime of the three methods, with a time of 1.277 seconds to process the data using 1 thread. This is likely due to the relative simplicity of the word edit distance algorithm compared to the Bigram and Trigram implementations. The Reuters dataset had a Bigram runtime of 1.784 seconds, and a Trigram runtime of 2.339 seconds. The twitter dataset had the highest time values with a Bigram runtime of 3.889 seconds and a Trigram runtime of 3.252 seconds. It was expected that the Twitter dataset running the Bigram and Trigram implementations would have the highest runtime since it is processing a large amount of dataset reference data in order to find the most likely matching word. It's important to note that there was a significant difference between the runtimes of these three algorithms, which is reflected in the complexity of the code and the depth of the datasets used in each experimentation method.

After interpreting the accuracy and runtime results for each experimental method, we present the Twitter database Trigram implementation as the best language model from our research. This method had the highest accuracy percentage when compared to the other results. We note that this method had the highest runtime by a significant margin, which may be important to consider when working with massive datasets and input values. The Naive implementation of our word edit distance algorithm surprisingly showed a relatively high accuracy value with a low runtime, suggesting that future improvements for this method may be worth pursuing. The Reuters dataset was likely not as extensive as the Twitter dataset and thus fell behind the other methods in both Bigram and Trigram accuracy. For our research purposes we set out to find the most accurate language model, which is shown in the Trigram Twitter dataset model.

In addition to the experimental methods we present above, there is a space to explore space complexity comparison for these methods. Optimizing space is a crucial topic in the field of language modeling and artificial intelligence as a whole, so it would definitely be worth looking into. There are also neural network based language models on the cutting edge of this field that would be fascinating to implement and compare to our accuracy and runtime metrics of our provided methods.

Conclusion

Our results showed that the Trigram Twitter dataset model was the most impressive of our language model experimental methods. The Trigram Twitter model outperformed the Twitter Bigram model in accuracy, as well as the Naive Twitter dataset and Reuters dataset models. The other models were shown to have a significantly lower runtime than the Trigram Twitter implementation, with the Naive model notedly resulting in a low runtime with surprisingly high relative accuracy of correction. Further research into neural network based language models would possibly yield interesting results to build off of this research and move towards a more optimized auto correction implementation in terms of accuracy and runtime. We had the opportunity through this research to explore common language modeling methods and implement them ourselves to see the effects of varying algorithms on accuracy and runtime when correcting misspelled words. We proudly present these results and claim that the Trigram Twitter dataset model was expectedly the highest performing implementation by the criteria presented in this paper.

References

[1] Vasil'ev, Vitalij. “Exploratory Analysis of Text Data.” RPubs, RStudio, 26 Oct. 2017, https://www.rpubs.com/lbind/EA1.

[2] Bird, S., Klein, E., & Loper, E. (2009). Natural language processing with Python: analyzing text with the natural language toolkit. " Reilly Media, Inc."

[3] K. Kukich, “Techniques for automatically correcting words in text,” ACM Computing Surveys, 24(4), 377–439, 1992.

[4] Yoon Kim, Yacine Jernite, David Sontag, and Alexander M. Rush. 2016. Character-aware neural language models. In Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI'16). AAAI Press, 2741–2749.

[5] Black, Paul E., ed. (14 August 2008), "Levenshtein distance", Dictionary of Algorithms and Data Structures [online], U.S. National Institute of Standards and Technology, retrieved 2 November 2016

[6] Winkler, William. (1990). String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage. Proceedings of the Section on Survey Research Methods.