DD2417 Language Engineering Word Prediction

Adam Ezzaim, Houcine Hassani idrissi, Kaan Ucar

June 2023

Abstract

This assignment project aims to develop a word prediction program, similar to the ones commonly found on smartphones. We will be using Python to implement this program, employing two different language models: n-gram and neural networks. Subsequently, we will evaluate the performance of our predictor by calculating the proportion of keystrokes saved.

1 Introduction

While you type text on your device, the word predictor feature displays a list of the most likely word completions or predictions for the next word. This feature constantly updates the suggestions with each keystroke, helping to narrow down the options. If one of the suggestions matches the intended word, the user can simply select it, saving both keystrokes and time. In this project, we will be implementing a word predictor using two distinct language models: a trigram distribution model and a neural network model. By developing a program that calculates the average number of saved keystrokes through predictions, we can compare the performance of these two models, as well as explore different hyperparameter choices.

2 Background

2.1 n-grams

By applying the Markov assumption, we can establish the n-gram model, which considers that a word's probability depends solely on the previous n-1 words. The conditional probability of a word w_i given the preceding words $w_1, ..., w_{i-1}$ can be expressed as: $P(w_i|w_1, ..., w_{i-1}) = P(w_i|w_{i-n+1}, ..., w_{i-1})$. This means that for the unigram model (n = 1), the context is not taken into account, and only the word frequency is considered. In the case of the bigram model (n = 2), information from the previous word is utilized to predict the next word. This pattern continues for higher n-gram models, where the prediction is based on a sequence of preceding words.

2.2 Neural networks

Natural language processing often grapples with sequences of data that can vary significantly in length, which can pose a challenge for traditional feed-forward neural networks. In light of this, recurrent neural networks (RNNs) are frequently employed as a more efficient alternative. The unique architecture of an RNN features loops within its network connections. This loop structure allows the network to maintain a form of "memory" by retaining the state of the network, which is informed by previous sequence outputs.

Long short-term memory (LSTM) networks are a specialized type of RNN, characterized by a more intricate structure. A fundamental issue with basic RNNs is their inherent tendency to "forget" past input, primarily because the gradient tends to disappear during the back-propagation training process due to their relatively simple cell structure. LSTMs address this issue by incorporating three distinct gates into their structure: an input gate, an output gate, and a forget gate. These gates allow LSTMs to preserve and access information from further back in the sequence, thereby enhancing their memory and predictive capabilities in the field of natural language processing.

3 Data

For the purpose of training and evaluating our models, we employ two distinct datasets: the Eng News and the Eng Twitter datasets. Our training data is considerably large, encompassing 2 million lines from the Eng Twitter dataset and an additional 1 million lines from the Eng News dataset. In order to gauge our models' effectiveness, we utilize test sets drawn from each dataset, comprising 2,000 lines each.

An important point to remember is that these datasets feature a range of special characters and phrases that might not commonly appear in everyday usage contexts such as messaging, which is our primary target application for the word prediction tool. Therefore, to simulate a more authentic testing environment, we perform model evaluations on sanitized test data. In this refined data, we retain only fundamental characters such as spaces, apostrophes, and alphabetic letters. This data-cleansing process aids in achieving a more precise evaluation of the models' practical application performance.

4 Implementation

4.1 Trigram model

Given that we are training our model on a substantial corpus, we are mindful of time and computational resource constraints. Consequently, we have opted for a trigram model for our word prediction task. This task is bifurcated into two segments: Firstly, we process the data to construct a statistical model inclusive of unigram, bigram, and trigram probabilities. Following this, we use this model to anticipate words based on user input. As such, our code naturally segregates into two components: a TrigramTrainer and a TrigramTester.

For the TrigramTrainer, we have revised the BigramTrainer code from the assignments, adjusting it to accumulate trigram counts. This piece of code traverses through the sanitized input from the training data, escalating the counts in the corresponding entries of the dictionaries that maintain the n-gram counts. It then stores the log probabilities to a model file, employing the Laplace smoothing technique to redistribute some probability mass from observed sequences to unseen sequences, if specified in the arguments.

On the other hand, the TrigramTester accesses the model, organizes it according to descending probabilities, and employs it to predict words from user input. This is also used when dealing with test data, particularly when we aim to compute the proportion of saved keystrokes. This is accomplished by navigating through a small test set extracted from the original dataset, and subsequently comparing the correct label with the model predictions for each word, as well as each character within the word. By doing so, we gain a comprehensive understanding of the model's performance and efficiency.

By dividing the code into these two modules, TrigramTrainer and TrigramTester, we ensure a clear separation of responsibilities and facilitate the training and prediction processes within the trigram model framework.

4.2 Neural network model

To enable our neural network model to effectively handle longer sequences and make accurate predictions, we employ Long Short-Term Memory (LSTM). We leverage the LSTM class provided by PyTorch to initialize and train a stacked 2-layer LSTM. The hidden layer size is set to 256 nodes, and a dropout probability of 20% is applied. The LSTM model takes as input a sequence of embedded characters. Each character is embedded and trained using PyTorch's Embedding class, with an embedding size of 16.

The outputs from the LSTM layers are then passed through a linear layer, which maps them to a size equivalent to the number of characters we have embeddings for. Finally, these linear layer outputs are fed into a softmax layer, where the cross-entropy loss is computed. The trainable parameters of our network include the character embedding parameters, LSTM parameters, and the parameters of the final linear layer. To optimize the training process using gradient descent, we utilize the Adam optimizer.

In our approach, we train two different LSTMs separately for this task, each with a slightly different focus. Both LSTMs share the same core structure, but differ in the characters they are trained on. The first LSTM, named LSTM-lower, is trained exclusively on data containing lowercase letters and only two special characters: apostrophe and period. This results in a total of 29 different characters used for training. The second LSTM, LSTM-all, is trained on all characters present in the dataset, totaling 102 different characters.

By training two LSTMs with different character sets, we can explore the impact of character diversity on the performance of our models. LSTM-lower focuses on the most commonly used characters, while LSTM-all considers a wider range of characters present in the dataset. This allows us to compare their performance and evaluate the influence of character variation on the word prediction task.

5 Results

5.1 Trigram model

In our study, we employed a trigram model to estimate the proportion of keystrokes saved by our word predictor. To evaluate its performance, we conducted a series of tests outlined in section 4.1, considering various factors that may affect the results. Specifically, we explored different combinations of the following variables:

- Train set: we trained the model using either twitter or news data.
- Test set: we tested the model on a smaller sample taken from the same twitter or news data.
- Laplace smoothing: we investigated whether incorporating Laplace smoothing, which redistributes probability mass from seen sequences to unseen sequences, improves prediction accuracy.
- Only lowercase: we examined the impact of converting the train and test data to lowercase during the cleaning process.

Training data	Test data	raw	laplace smoothing	lowercase
twitter	twitter	24.33%	24.33%	26.62%
twitter	news	17.94%	17.95%	19.77%
news	twitter	21.11%	22.13%	24.24%
news	news	23.31%	23.32%	28.00%

Table 1: Proportion of saved keystrokes using different train/test data and hyperparameter combinations

The outcomes of these tests are summarized in Table 1, presenting the proportion of saved keystrokes for each combination of variables. Our evaluation demonstrates that the predictor's performance aligns with our expectations and is influenced by different factors.

Firstly, when the predictor is tested on a sample from the same training set, it generally achieves higher accuracy. Notably, the configuration using twitter as both the training and testing dataset slightly outperforms the news configuration in the raw and Laplace smoothing setup. However, the use of the lowercase in the news configurations outperforms the other models.

Secondly, incorporating Laplace smoothing into the probability calculation during the model construction does not significantly enhance performance across all training and testing dataset combinations. This suggests that Laplace smoothing may not provide noticeable improvements in word prediction accuracy.

Thirdly, applying lowercase conversion during text processing has a positive impact on the proportion of saved keystrokes, particularly for the model trained and tested on the twitter dataset. The exact reason for this improvement remains unknown and warrants further investigation into the underlying data.

Moreover, before concluding, we observe that the model performs better when it is trained on the news data and tested on the twitter data rather than the other way around. This could be explained by the larger size of the news' corpus or maybe it simply covers a wider variety of words.

In conclusion, our findings indicate that the performance of the trigram model is affected by various factors, including the choice of training and testing datasets, the use of smoothing techniques, and the text processing techniques employed. Further research and analysis are necessary to gain a deeper understanding of these observations and identify the underlying factors driving the observed performance variations.

5.2 Neural network model

We also investigated the performance of neural network models in terms of keystroke savings. Table 2 presents the results for two LSTM models, trained for 5 epochs with a learning rate of 0.002, using raw and cleaned test data.

Training Data	**Test Data**	**LSTM-all**	**LSTM-lower**
Twitter	Twitter	48.75%	50.29%
Twitter	News	42.68%	45.20%
News	Twitter	42.60%	44.46%
News	News	51.93%	53.16%

Table 2: Proportion of saved keystrokes by LSTM models using different train/test data combinations.

The findings in Table 2 align with our expectations, showing that the models perform better when trained on the same type of data as the test data. Specifically, the models trained on the news dataset outperform those trained on the twitter dataset across almost all test combinations.

Among the two models, LSTM-lower demonstrates slightly superior prediction capability compared to LSTM-all. Despite its limitation to lowercase characters, LSTM-lower achieves better prediction accuracy. This indicates that focusing on lowercase letters during training can yield improved results compared to considering a broader set of characters, including uppercase letters and special characters. The performance of LSTM-all is hindered by the increased complexity of predicting from a larger character set, highlighting the need for more extensive training to match the performance of LSTM-lower.

Interestingly, despite the variations in performance depending on the training dataset, both models exhibit similar generalization capabilities across all test combinations. When the training dataset differs from the test dataset, both news and twitter test cases show comparable performance.

These findings underscore the importance of training models on data that closely aligns with the target domain and highlight the benefits of leveraging lowercase characters for word prediction. However, further investigation and experimentation are required to gain deeper insights into the observed performance variations and identify optimal training strategies for different datasets and prediction tasks.

6 Conclusion

The key finding from this project is that the Neural Network Model outperforms the Trigram model and appears to be better suited for the word prediction task. This suggests that the neural network approach, particularly the LSTM-based model, can effectively capture the patterns and dependencies in the data, leading to improved prediction accuracy.

The results also emphasize the significance of the training data. The dataset used for training should either be diverse, covering a wide range of language patterns, or tailored specifically for the target domain. This highlights the importance of data preparation and selection to ensure the model's effectiveness.

To further enhance the models' performance, several avenues can be explored. First, training the models for a longer duration or increasing the training data size may lead to better results by allowing the models to capture more intricate patterns and dependencies. Additionally, experimenting with different hyper parameters, such as the network architecture, embedding size, or learning rate, can help fine-tune the models and improve their performance.

Another potential avenue for improvement is to combine the strengths of both the Trigram and Neural Network models. The Trigram model excels at capturing common word sequences, while the Neural Network model can capture more nuanced patterns. Integrating these models could potentially yield more accurate predictions by leveraging the strengths of each approach.

In summary, this project has provided valuable insights into the implementation and evaluation of word prediction models using statistical and neural network approaches. It highlights the superiority of the Neural Network Model, the importance of training data, and suggests potential avenues for further improvements. By continuously refining these models and exploring new techniques, word prediction systems can be enhanced to better assist users in various applications, such as text completion, typing assistance, and language generation.