# DD2417 Language Engineering - Word Prediction

Axel Larsson - Kamil Mellouk

May 20, 2022

**Abstract**

This project assignment aims to implement a word prediction program, similar to what can be found on most smartphones. We will implement this with Python and using two different language models: n-gram and neural networks. We will then evaluate the performance of our predictor by computing the proportion of saved keystrokes.

## 1 Introduction

As you are typing text on your device, the word predictor displays a shortlist of the most probable completions of the current word, or prediction for the next one. The suggestions need to be updated for each keystroke the user makes, as it helps narrow down the options. If one of the suggestions turns out to be right, the user can type the word just by pressing it, saving keystrokes and time.

We will implement such a word predictor using two separate languages models: a trigram distribution and a neural network based model. We will then build a program that computes the average number of keystrokes saved thanks to the predictions, which allowing us to compare these two models, as well as different choices of hyper-parameters.

## 2 Background

### 2.1 n-grams

With the Markov assumption, i.e. assuming that a word only depends on the previous couple of words, we can define the n-gram model which takes the $n-1$ preceding words into account:

$$P(w_i|w_1, \cdots, w_{i-1}) = P(w_i|w_{i-n+1}, \cdots, w_{i-1})$$

This means that for the unigram model ($n = 1$) we do not take the context into account and only consider word frequency; for the bigram model ($n = 2$) we use information gained with the previous word to predict the next and so on...

### 2.2 Neural networks

As natural language processing often deals with varying length of sequences of data, the classic feed-forward neural networks have certain limitations in this domain. Instead, a recurrent neural network (RNN) is commonly used. The RNN network contains loops in its connections. This way, the current state of the network can be regarded as memory, which it has gathered from prior outputs from the sequence. [DJ21]

Long short-term memory (LSTM) is a type of RNN with a more sophisticated structure. Due to the vanilla RNN's simple cell structure, it is prone to "forgetting" input in the past as the gradient vanishes when trained with back-propagation. LSTMs however mitigate this problem by the addition of three gates; an input gate, an output gate, and a forget gate, to be able to use information further back in time.

## 3 Data

Two datasets are used for training and testing our networks; NewsCrawl 2010 and Blog Post dataset. Our trainingsets consist of 2.1 million lines of the NewsCrawl 2010 dataset and 900k lines of the Blog Post dataset, while our testsets consist of 2,000 lines of each dataset.

The datasets consist of a lot of special characters and phrases that are not commonly used in real life scenarios for which the word predictor is intended, like texting. Therefore, we evaluate the models on cleaned test data, where only the space character, apostrophe, and letters are present in the cleaned test data. The reason for this is to get a more accurate estimate of how well the models can predict actual words.

## 4 Implementation

### 4.1 Trigram model

As we are training our model on a large corpus, and have a rather limited time and computational resources, we will limit ourselves to using a trigram model for our prediction task. This is done in two steps: first process the data in order to build the statistical model with the unigram, bigram and trigram probabilities, before using this model to predict words from user input. We therefore naturally split our code into two modules: a `TrigramTrainer` and a `TrigramPredictor`.

For the `TrigramTrainer` we adapted the `BigramTrainer` code from the assignments so that it also collects the trigram counts. The code iterates through cleaned input from the training data and increments the counts in the right entries of the dictionaries holding the n-gram counts. It then saves the log probabilities to a model file, applying the Laplace smoothing method for transferring some of the probability mass from the seen sequences to the unseen sequences, if specified in the arguments.

The `TrigramPredictor` reads the model, sorts it by decreasing probabilities and uses it to predict words from user input, or from test data in case it is specified that we want to compute the proportion of saved keystrokes. This done by iterating through a small test set sampled from the original data set, and comparing the correct label with the model predictions given each words, as well as each character inside the word.

### 4.2 Neural network model

For our neural network model to handle longer sequences of data and leverage that into predicting the next word, we use an LSTM. Here we take advantage of PyTorch's LSTM class to initialize and train a stacked 2-layer LSTM, with a size of 256 nodes for the hidden layer and a dropout probability of 20%. The LSTM takes as input a sequence of embedded characters, where each character is embedded and trained using PyTorch's Embedding class. The size of the embeddings are 16. Further, the outputs of the LSTM are passed through a linear layer to an output size of the number of character we have an embedding for. Lastly, we pass the outputs from the linear layer to a softmax layer, were we finally compute the cross-entropy loss. Thus, the trainable parameters for our network consist of the character embedding parameters, the LSTM parameters, and the final linear layer parameters. As optimizer for gradient descent we use Adam.

Furthermore, we train 2 different LSTM separately for this task. The core structure for both LSTMs are the same, the only difference is the number of characters they are trained on. The first network, LSTM-lower, is only trained on data where all characters are lowercase letters, and where no special characters exist except for apostrophe and period. The other network, LSTM-all, is trained on all characters that exist in the dataset. Thus, LSTM-lower is trained with 29 different characters, while LSTM-all is trained on 102 different characters.

## 5 Results

### 5.1 Trigram model

To gather the proportion of keystrokes saved by our predictor, we perform the testing process described at the end of section 4.1 with different values for the following:

- train set: blogs or news data
- test set: smaller sample from the blogs or news data
- Laplace smoothing: whether we to transfer some of the probability mass from the seen sequences to the unseen sequence
- only lowercase: whether we lowerise the train and test data when cleaning it

| Training data | blogs | | news | |
|---|---|---|---|---|
| Test data | blogs | news | blogs | news |
| raw | 36.60% | 25.66% | 26.28% | 31.63% |
| laplace smoothing | 36.60% | 25.66% | 26.29% | 31.63% |
| lowercase | 43.24% | 26.42% | 26.58% | 31.76% |

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

We can observe on table 1 that as expected, the predictor generally performs better when testing it on a test sample taken from the same training set, with the blogs/blogs configuration performing slightly better than the news/news one. Applying Laplace smoothing to the probabilities when building the statistical model doesn't seem to noticeably improve performance across all train/test combinations. Lowerising the text when processing it seems to generally improve the proportion of saved keystrokes, and especially so for the model trained and tested on the blogs dataset. The reason behind the increase in performance in this scenario is for now unknown to us and would require us to further investigate the data.

### 5.2 Neural network model

The proportion of saved keystrokes for all models on the raw- and cleaned test data are shown in Table 1. The models are trained for 5 epochs, with a learning rate of 0.002.

| Training data | blogs | | news | |
|---|---|---|---|---|
| Test data | blogs | news | blogs | news |
| LSTM-all | 51.44% | 49.97% | 49.49% | 54.83% |
| LSTM-lower | 54.13% | 51.04% | 51.25% | 55.74% |

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

As expected, the results in table 2 show that the models perform better when training on the same type of data as the test data. We can also see that the best predictors were the models that trained on the news dataset, with LSTM-lower having slightly better performance than LSTM-all, which is a trend over all combinations of tests. Namely that only training on lower case characters yields a better overall prediction capability than training on all characters does, even though it is unable to predict capitalised words. Having too many possible characters to predict, like LSTM-all does, it would require more training to reach LSTM-lower's performance, will is clearly indicated by the results.

Although both models varies in performance depending on which training dataset was used, their generalisation capabilities seem to be similar across all combinations. Looking at the cases where the training dataset is different from the test dataset, both the news and blogs test cases perform very similar to each other.

## 6 Conclusion

The most obvious takeaway from this project is that the Neural Network Model performs clearly better than the Trigram one and seems more adapted for the word prediction task. The results obtained also highlighted the importance of the training data, which needs to either be very diversified or tailor-made for a specific word prediction context.

An interesting feature to add to such a prediction software could be to use word embeddings to suggest emojis linked to words semantically similar to the one being typed by the user.

## References

[DJ21] James H. Martin Dan Jurafsky. *Speech and Language Processing*, chapter 9. 2021.