DD2417 - Mini-Project Report (Group 11)

Simon Döbele
`sidobele@kth.se`

Thomas Bouquet
`tobou@kth.se`

1 Introduction

Word prediction is used primarily in smartphones to enhance the user experience by providing a short list of words that are likely to follow the words already typed. The benefit is that it greatly speeds up writing on a phone, which can be much more tedious and time consuming than on a computer.

In this project, we aim to implement our own word predictor that could be used in a smartphone and thereby save a user as many keystrokes as possible. Such a system needs to have a language model at its core. We implement and compare three such language models: two that use n-grams to predict the next word and one that is based on a vanilla recurrent neural network (RNN) employing word embeddings. We find that the the n-gram models both outperform the RNN in terms of our evaluation metric: saved keystrokes.

In order to demonstrate the functioning of our models, we have implemented a simple user interface that allows to visualize and choose the predicted words.

2 Background

2.1 Language models

First, let us define what a language model is. Generally, a probabilistic language model returns a probability distribution for the next word w_t given the previous $t-1$ words, which we represent by $w_{1:t-1}$, and possibly given a context c [1]. In auto-regressive language modelling, which we will focus on in this paper, it is assumed that the probability of a whole sequence of words $w_{1:t-1}$ can be calculated by multiplying the probabilities of the individual word distributions:

$$P(w_t|w_{1:t-1}, c) = \prod_t P(w_t|w_{1:t-1}, c)$$

The context c differs depending on the language modelling task. For instance, in the dialogue task, c would contain the complete dialogue history of the interlocutors, including what either of the participants has said before. Regarding next word prediction task, which we investigate, we ignore such a context and simply predict the next word w_t given the previous $t-1$ words $w_{1:t-1}$, so:

$$P(w_t|w_{1:t-1}) = \prod_t P(w_t|w_{1:t-1})$$

The Recurrent Neural Network we implemented does this by passing the sequence of previous words through the network and the network outputs a probability distribution over the next words, from which we choose the top four words with the highest probability to present to the user.

In the following, we explain more in depth how each of both of these models work.

2.2 N-gram language models

The n-gram language model is frequently used in statistical text analysis to predict the next word to appear. An *n-gram* is a sequence of n contiguous words in a text. So for instance, a bigram is a sequence of two words.

In order to define a probability distribution on the n-grams of a text, we start by extracting all the sequences of n consecutive words present in our training corpus and we attribute to each one a probability of occurrence based on the frequency of appearance of this very n-gram. So for instance, in a bigram language model, we approximate all the previous words by just the last word and the above equation becomes:

$$P(w_t|w_{1:t-1}) \approx P(w_t|w_{t-1})$$

This assumption that we may approximate a word like that is called a Markov assumption [2]. Generally, the probability returned by a n-gram model can be interpreted as a $(n-1)$-order Markov model since given $n-1$ words, we estimate what will be the next words based on this probability and by only taking in account these very $n-1$ previous words.

Then, in order to get $P(w_t|w_{t-1})$, we use maximum likelihood estimation (MLE) in order to get an estimate for the respective probabilities. MLE is calculated by the relative frequency of the respective n-grams [2].

In general, we would like to make n as large as possible in order to get a better approximation. However, the higher n, the more data we would need to get significantly better results. So since our computation power is limited, we only compute all the n-grams up until 5-grams, i.e. we only consider (maximally) the four previous words to predict the next word. The way we combine those different n-gram model to a single combined n-gram model is described under 3.2.

2.3 RNN-based language models

Recurrent Neural Networks (RNNs) are a category of Artificial Neural Networks that can model time series data. They are particularly suited to model language, because language is inherently time-based in that we write (or speak) in a sequence of words. Likewise, it turns out that RNNs are particularly well suited to word prediction because a text can be seen as (arbitrarily long) time series: the first word of the sentence being the datapoint for $t = 1$, the second word for $t = 2$, etc.

Importantly, an RNN can take an arbitrarily long series of words as input. As we have previously seen, that is in practice no the case with n-gram models, due to limited amounts of data and limited amount of computing power to represent and compute a certain sequence length n.

RNNs are neural networks that use a cycle such that each hidden layer activation depends not only on the current input (e.g. a word), but also on the hidden layer activation from the previous timestep (and therefore also indirectly on the hidden layer activations of all previous timesteps) [2].

RNNs can then be used for language modelling in the following way, we retrieve word embeddings for each word of a sentence to pass into the RNN, and then a linear layer maps the RNN outputs to all possible words in the vocabulary. More information about the specific mathematics can be found under 3.3. Once an RNN is trained, it is sufficient to input a list of words and the network will be able to predict the next word as the most likely word by taking the argmax of the produced probability distribution.

3 Implementation description

3.1 Pre-processing the text

To begin with, we must ensure that the text has a proper format so that we can train on it. We chose to process the text by removing all characters that are not letters, and to use only lower case characters.

3.2 n-gram

The first model we implemented is the n-gram model. To get the frequency of each n-gram in the text, we just use the processed text (as presented above) and retrieve all the batch of n consecutive words and for each of them count their occurrences.

For our implementation, we limited the maximum dimension to the 5-gram model because we believe it is a good trade-off between the amount of required data to process (to have relevant statistical information) and the *quantity* of context taken in account. Besides, we also computed the probability distributions of all the n-grams for $n \in \{1, 2, 3, 4\}$. The reason is to deal with possible non-existing n-grams in test texts. This is what we are going to explain in the next section.

There is a problem when the dimension n increases. Indeed, the larger n is, the likelier it is that the probability that n consecutive words typed by a user correspond to an n-gram encountered in the model's training corpus is zero. To overcome this hurdle, there are mainly three techniques that we can use.

The first one called Laplace smoothing consists in slightly modifying the probabilities of occurrence to ensure that we never get zeros. However, this is not recommended because it tends to be too crude for many applications.

The second one is called linear interpolation and we looked at in assignment two. Thirdly, there is the method called stupid back-off [3] and we use a version of it. According to this method, if the highest n-gram is present (in our case if a 5-gram is present), we predict based on it. However, if it is not present, we recursively go to smaller n-grams (so we would predict based on four-grams next, if possible etc.)and return that until (in the worst case) we reach a unigram, which is just the relative frequency of a word occurring in our training corpus.

3.3 RNN

The second model we implemented was a vanilla RNN using pretrained GloVe word embeddings. That way we do not need to train them. Embeddings serve to represent words that have similar meaning similarly. For more information about GloVe, please refer to the respective paper [4].

Mathematically, we follow chapter 9.3 of [2]. So for the forward pass we take batches of input sequences $X = [x_1, ..., x_T]$, where each x_t is a word, possibly a padding word to ensure equal sequence lengths for the batches. Each x_t will the be converted into its pre-trained embedding e_t:

$$\mathbf{e_t} = \mathbf{E}\mathbf{x_t}$$

Next, we pass the embeddings through the vanilla RNN from Pytorch, which uses tanh as the activation function to get hidden vectors h_t (note that we use 100 dimensional hidden vectors):

$$\mathbf{h_t} = tanh(\mathbf{U}\mathbf{h_{t-1}} + \mathbf{W}\mathbf{e_t})$$

Finally, the hidden vectors are passed through a linear output layer and a softmax activation function to get the probabilities for each next word:

$$\mathbf{y_t} = softmax(\mathbf{V}\mathbf{h_{t-1}})$$

In our implementation, for training we use the cross-entropy loss as well as the Adam optimizer [5].

3.4 Prediction

We simulate a smartphone setting, whereby a user gets fixed number W of recommended next words. Those recommendations are the highest probability next words that are predicted by either of our two different language models. The higher W, the likelier it is that the correct next word will be part of the predictions. However, to keep it realistic in terms of screen size (as on a smartphone), it would be relevant to keep W lower or equal to 5.

What is more, our goal for prediction is not just next word prediction, but more precisely to save a user from typing as many keystrokes as possible. That is, the user does not just get a next word prediction whenever he presses the space bar. Rather, predictions are created "on-the-fly" while the user types. That is, we automatically update the predictions when the user types by filtering through the highest probability words the model predicts, given the characters the user already typed. So for instance, when the user types "How are", the user might get predictions such as "you / our / her / we", and now suppose the user was not looking to type either of these words, so the user continues to type so that we have "How are y". In that case, due to the user providing more information, the predictions update in order to only give plausible recommendations, such that we might get the following predictions "you / your". The ordering remains based on the probabilities but excludes implausible predictions. Since the user was looking for "your", the model then saved him three keystrokes.

3.5 Interface

In order to have a visual interface to interact with our algorithm, we implemented a web application using the Dash framework of Python. Via this interface, the user can enter text which will be automatically processed. The platform will then suggest words to complete the current sentence. The user can click on a word to have it added to their text, just like on a regular phone. An illustration of the interface can be seen on Figure 3.

4 Datasets

We used the 50-dimensional glove word embeddings that we also used in assignment 4 from the following url: http://nlp.stanford.edu/data/glove.6B.zip.

For training and testing datasets, we used a subset of the News Crawl 2010 dataset, which was released as a part of ACL 2014 Ninth Workshop on Statistical Machine Translation, and which we also used in assignment three of the course.

We used the same training and test datasets for both n-gram and RNN models to ensure that we can compare them. This comparison can be seen on Figure 1.

5 Evaluation

As presented in the introduction, one of the main interests of word suggestion is to allow the user to save time and type faster by selecting a word rather than writing it out entirely. A consistent and effective metric for measuring this gain, unaffected by the average writing speed which varies from user to user, is to calculate the number of keystrokes saved by selecting a suggested word. We therefore evaluated the effectiveness of our models in terms of saved keystrokes. To perform this evaluation, we performed a train-test split of our data and evaluated using the test set.

We already gave an example that can serve as a high-level description of how we calculate the saved keystrokes exactly in section 3.4, but in the following we give the precise algorithmic steps to achieve this.

5.1 n-gram

For the n-gram model, the evaluation steps are described hereafter. Let us precise that we initialise a dictionary `dict` whose keys are the possible sizes for the suggestions list and the values are the accumulated number of saved keystrokes for this size of the list. Of course the values are initialised to zero. Besides, we initialise to zero a variable `totalKeystrokes` that will count, in the end of the evaluation, the total number of keystrokes that would have been required if the user had had to type everything entirely without selecting suggestions.

1. process the test set and retrieve the 5-grams
2. select a 5-gram
3. add the length of the fifth word of the 5-gram to `totalKeystrokes`
4. select a size for the suggestions list (between 1 and 5)
5. use the four first words as context
6. set the `firstCharacters` so the empty string

7. while the fifth word of the 5-gram is not in the prediction list, add one by one, from the beginning, the characters of this very fifth word to the `firstCharacters` string
8. update the corresponding value in `dict` by adding the difference between the length of the fifth word of the 5-gram and the length of `firstCharacters` (characters that were required to be typed so that the good next word would appear in the suggestions list)
9. repeat steps 4 to 8 for every possible list size between 1 and 5
10. repeat steps 2 to 9 for every 5-gram
11. normalise the values in the dictionary by dividing them all by `totalKeystrokes`

In the end, we then get a percentage of saved keystrokes for each suggestions list size. For visualisation sake, we chose to graphically represent these results that will be discussed later on.

5.2 RNN

For the RNN, the evaluation is quite similar except we do not rely on n-grams when processing the test set. Here again, we initialise `dict` and `totalKeystrokes` the same way as previously. Then, we process as follows:

1. process the test set by tokenising all the lines
2. select a tokenised line (with more than one token)
3. set a variable `indexToPredict` to 1
4. use the tokens with an index lower than `indexToPredict` as context and the token with index `indexToPredict` as the word to predict
5. add the length of the word to predict to `totalKeystrokes`
6. select a size for the suggestions list (between 1 and 5)
7. set the `firstCharacters` to the empty string
8. while the word to predict is not in the prediction list, add one by one, from the beginning, the characters of this very fifth word to the `firstCharacters` string
9. update the corresponding value in `dict` by adding the difference between the length of the word to predict and the length of `firstCharacters` (characters that were required to be typed so that the good next word would appear in the suggestions list)
10. repeat steps 6 to 9 for every possible list size between 1 and 5
11. repeat steps 3 to 10 for `indexToPredict` going from 2 to the length of the line - 1
12. repeat steps 2 to 11 for every line
13. normalise the values in the dictionary by dividing them all by `totalKeystrokes`

In the end, we then get a percentage of saved keystrokes for each suggestions list size. For visualisation sake, we chose to graphically represent these results that will be discussed later on.

6 Results

We used two differently sized training sets (both subsets from the same News Crawl 2010 dataset) to build two different n-gram models to compare to our RNN language model. We expected the RNN model to at least outperform the n-gram model that was trained on the smaller of both datasets. However, we find that the two n-gram models both outperform the RNN model.

The first n-gram model was trained on 1539 lines of text from the News Crawl 2010 dataset, which we also used to train the RNN on (see figure 1a and figure 1b for performance). This was because the RNNs take a long time to train and we wanted to compare the two models on the same basis. As we can see, we also used a training set with approximately 230 times the number of lines of text to create a second n-gram model. The performance of that model can be seen in figure 2. This required (comparatively) little time to compute. So since we are interested in saving the user as many keystrokes as possible, the rest of the description of the results in this section focus on this second ("bigger") n-gram model.

N-gram models can be said to overfit on the training corpus, as they only contain exactly those n-grams that are present in the training corpus. This also applies to the n-gram model(s) trained here. Hence, they inherently generalize badly. Conversely, that explains why the n-gram model(s) perform so well in our evaluation: we tested them on a different subset of the News Crawl dataset (than the training set). This whole corpus can be said to stem from the underlying distribution of language, one which is probably very different from a typical dialogue on a smartphone or also very different from, for instance, legal articles. This is because it exhibits, for instance, a different writing style and other style characteristics. Therefore, the achieved performance of the n-gram models should be taken with a grain of salt.

Nevertheless, we find that our "bigger" n-gram model can save almost up to 70% of the keystrokes, as can be seen in Figure 2. One can also notice in that figure that, as expected, the percentage of saved keystrokes increases with the number of recommended words up to 70% for five suggestions. Even when we choose to only suggest one single word, 49% of keystrokes are saved which means that we can almost type twice more characters per minute (if we neglect the time required to select a suggested word).

Interestingly, this lack of generalization ability may not be a problem to our use case, where a smartphone user wants next word predictions. In that case, we may want an individualized n-gram model that saves the keystrokes of this specific user. In other words, it would be promising because in that case we would want a model that is well suited to the user and does not need to generalize well.

Nevertheless, many other factors need to be taken into account, as a user might not just be interested in saving keystrokes, but also time, and it could very well be that the time it takes to look at the recommendations, process them and then click on them is slower (for many or even most words) than just typing that word.

Our RNN on the other hand, is not performing so well here. Part of the reason for this is that training it takes a long time (and needs much more data). However, applying it to a smartphone use case, an RNN will most likely still not be the model to choose for next word prediction. This is due to the reduced compute power of a smartphone and possibly also the reduced storage capacity. So even though in principle, RNNs might work on smartphone, in a real application, we would opt for n-gram models, especially since they could be individualized for each user (and update in real-time given what the user writes.)

7 Conclusions

In conclusion, we built built language models, two based on n-grams and one based on a vanilla RNN using GloVe embeddings. We find that the n-gram language models outperform the vanilla RNN in our chosen evaluation metric: percentage of saved keystrokes.

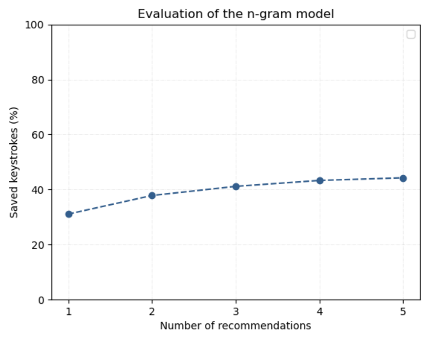
In a follow-up study, it would be interesting to use more data (and different test sets) and more computing power to improve the performance, e.g. by performing a grid search for different hyperparameters. Then we could also compare whether an RNN would generalize better, which, in principle, we would expect.

Appendix

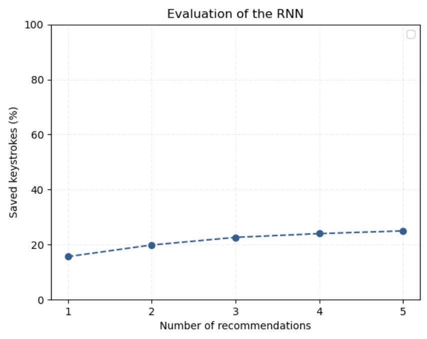
A Code

Our code is available on this GitHub: `https://github.com/bqth29/word-predictor`

B Illustrations



(a) Evaluation of the n-gram model



(b) Evaluation of the RNN

Figure 1: Evaluation of the models in terms of saved keystrokes

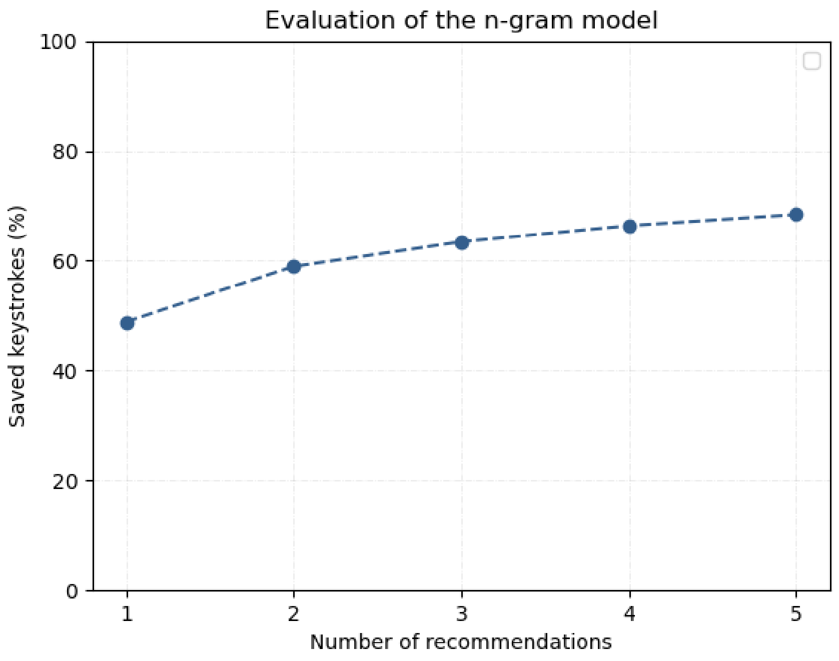
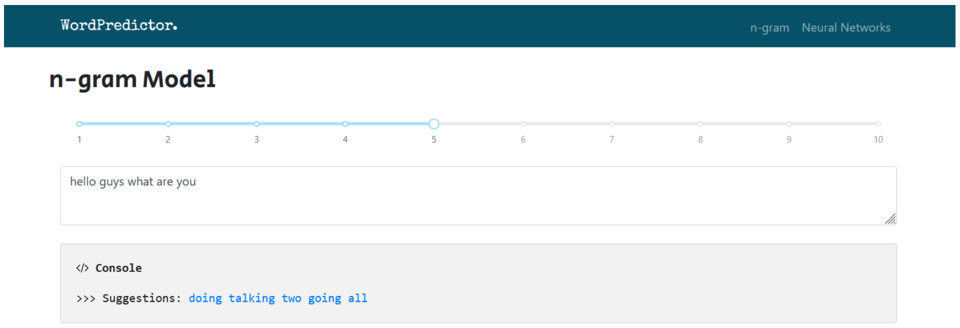


Figure 2: Evaluation of the n-gram model for a bigger training set



hello guys what are you

```
</> Console
>>> Suggestions: doing talking two going all
```

Figure 3: Screenshot of the interface

References

[1] Abigail See, Stephen Roller, Douwe Kiela, and Jason Weston. What makes a good conversation? how controllable attributes affect human judgments. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 1702–1723, 2019.

[2] Dan Jurafsky and James H Martin. Speech and language processing (3rd (draft) ed.), 2022.

[3] Thorsten Brants, Ashok C. Popat, Peng Xu, Franz J. Och, and Jeffrey Dean. Large language models in machine translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 858–867, Prague, Czech Republic, June 2007. Association for Computational Linguistics.

[4] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[5] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[6] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Proceedings of the IEEE international conference on computer vision*, pages 2223–2232, 2017.

[7] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[8] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep learning face attributes in the wild. In *Proceedings of International Conference on Computer Vision (ICCV)*, December 2015.

[9] Philip TG Jackson, Amir Atapour Abarghouei, Stephen Bonner, Toby P Breckon, and Boguslaw Obara. Style augmentation: data augmentation via style randomization. In *CVPR Workshops*, volume 6, pages 10–11, 2019.