GRAMMATICAL CORRECTNESS OF WORD PREDICTORS

Author:
Wouter Zwarts
Student number:
6292526

Supervisor:
Tejaswini Deoskar
Second reader:
Jos Tellings

Utrecht University

Artificial Intelligence

Faculty of Humanities

A 7.5 ECTS Thesis

2 JULI 2021

Abstract

Word predictors are frequently used software helping millions of people on a daily with typing on their mobile devices. They do however have their flaws, as they are not always grammatically correct in their predictions. The aim of this research is to create a way to increase the grammatical correctness of other models and see how that affects the performance and whether they can be implemented on a mobile device. Two models have been created, an n-gram model and an RNN model. They are compared to one another and then a POS tagger has been added to both models and compared to each other and those results with the that of the initial models. A dataset of sentences extracted from resources on social media services has been used to train the models on. The base models' performances on predictions are unfortunately quite disappointing, mainly due to a too small of a dataset. Additionally, it is found that adding the POS tagger does not improve the main performance issue. Furthermore, the created models are not optimized and thus not suited for mobile devices due to the large size and high response times. The grammatical correctness of the predictions, however, did increase with the usage of the POS taggers, albeit to a much lesser degree than was initially hoped for.

Contents

1. INTRODUCTION **4**

1.1 RELATED WORKS 4

1.2 RESEARCH QUESTION 4

1.3 OVERVIEW 5

2. METHOD **6**

2.1 DATASET 6

2.2 PRE-PROCESSING 6

2.3 TRAINING 7

2.4 POS-TAGGING 8

2.5 TESTING 9

2.6 EVALUATION 9

3. RESULTS **11**

3.1 N-GRAM MODEL 11

3.2 RNN MODEL 12

4. CONCLUSION **14**

5. DISCUSSION **15**

6. FUTURE WORKS **16**

BIBLIOGRAPHY **17**

ATTACHMENTS **19**

1. Introduction

In recent years, the usage of smart devices has grown rapidly, and the amount of people using smartphones for daily communications with others has grown with that. Writing messages on a smartphone can be frustrating at times due to the keyboard being relatively small and thus resulting in making more mistakes than necessary. This is where an auto-correct software jumps in and helps to correct these mistakes. Most of these types of software also include a word prediction algorithm that suggests words while you are typing. Word prediction is an intelligent word processing feature that looks at your current writing and based on that, predicts what words could be typed next and suggests those words to you. The use of these types of software may assist on-screen keyboard users to improve typing speed [1] which is beneficial for the users, making them an essential part of Artificial Intelligence in the modern world.

The vast majority of these predictions and suggestions are perfectly fine. However more often than we desire some of the given predictions can be grammatically incorrect, mainly because only *n* amount words are taken into consideration. This is due to the use of n-gram based statistical language models in most of these predictors [2] where only a range of *n* words is used for the prediction. Taking Google's Gboard as example. The way the grammatical constraints were given was by using n-grams [3]. They predict it using the current best (partial) sentence and then finding the highest order n-gram state and selecting the best fit from these. Which are the most occurring sentences, meaning they do not necessarily take grammar into account [3]. Thus, if you want to make the suggested words more grammatically correct there are more things that need to be considered when looking at the next word.

The problem with doing that is that you are bound to run into efficiency issues when doing so, since when considering multiple different factors for what word can be picked, there are more calculations to be done, meaning more resources used and thus less efficiency. This is troublesome on mobile devices, since you are limited on memory and response times to prevent the keyboard from taking over all the processing powers and draining the battery of the device.

1.1 Related works

Within this field there have been several studies done [2, 4, 5, 6, 7], though seemingly not as much as for closely related fields such as speech recognition and also recognition of handwriting [7]. A few models worth mentioning are those from Ouyang et al [5] and from Hard et al [7]. The complete models use Finite State Transducers and Recurrent Neural Networks (RNN) respectively. The first one using n-grams for the grammatical constraints for predictions, as has been mentioned before, whereas the second one the grammatical constraints are purely based on the RNN's knowledge of the language. The RNN builds a language model (LM) based on the given inputs and takes those input examples as grammatical constraints. Another model is that from Yu et al [2], where they also use a RNN to train their language model. Another well-known keyboard used on mobile devices, SwiftKey [8], is reportedly also using a neural network for their prediction system, combined with some sort of tagging of words [9, 10, 11, 12], though there seem to be little details publicly available about what it exactly does.

1.2 Academic Relevance & Research question

The goal of this research is to attempt to find a way to decrease the amount of grammatically incorrect suggestions in the predictions, while still retaining an efficiency that is high enough such that the word prediction could potentially be used on smaller and less powerful devices, like older and cheaper smartphones. This in order to gain insight in the field and help with possible future implementations of intelligent word prediction software.

The question that will be answered in this research is if the occurrence of grammatically incorrect suggestions in word predictors can be decreased using a combination of the models with a POS tagger, whilst retaining efficiency.

To answer that question there are two parts that will be looked at. The first one being the possibility of making an algorithm that can predict and suggest a next word that complies to the grammatical constraints. To answer that question, current methods that are used for predicting the next words will be looked into, such as the methods used by Ghosh et al (2017), Silva et al (2019), Ouyang et al (2017), and Yu et al (2018) [4, 6, 5, 2], and the way they take grammatical constraints into consideration and why they could be lacking in some sense resulting in the occurrence of grammatically incorrect suggestions and what possibilities there are to improve them.

1.3 Overview

This is done by taking two different models, the first one being an n-gram language model and the second one being a recurrent neural network (RNN) language model. Both models will be evaluated and compared to each other and then a part of speech tagger (POS) will be added to see if that positively influences the grammatical correctness of the suggested words by the models, and if that has any other implications, be it positive or negative, for the suggested words.

The second part is whether it is possible to make it efficient enough to have such a system running on an embedded device. For this the findings of the first question will be taken and attempted to apply in such a way that it fits on embedded systems, taking into account all the current restrictions on size and response times. This makes it so the model that the study has brought forth has to be limited as well.

This paper is structured in the following way. Firstly, in section 2 the method will be presented, in which the dataset, the pre-processing of it and the training method will be explained after which the testing and evaluation part will be laid out. Secondly, the results for each of the models will be presented in section 3 and then compared to one another. Afterwards the conclusion will be given in section 4 and then the research question will be answered. Then in section 5 the findings will be discussed and limitations of the model will be spoken about. Finally, in section 6 possible future works and changes are discussed. The used datasets and programs are attached on the final page of this paper.

2. Method

Here the dataset, the ways of processing the data and the methods used in this research will be presented. The two base models that were created are an n-gram LM with Kneser-Key smoothing and an RNN LM with two LSTM layers and two activation layers.

2.1 Dataset

The dataset[1] used for this research is one previously used in the model created by Yu et al (2018) [2]. It is a dataset that was built out of a large 8 billion words containing set of sentences extracted from resources on social media services. The dataset used by Yu et al [2] is a subset of that containing 196 million words, split up in a training set (60%), a validation set (10%), and a test set (30%).

In this research only a subset of that subset was used, due to hardware limiting the amount of input that could be used. This ended up in a training dataset of 320 thousand words being used for the research.

The choice for this dataset was made because it has been used in previously done research and therefore deemed good enough to use again in this research.

2.2 Pre-processing

The pre-processing of this dataset was pretty straight forward, as most pre-processing is done generally the same. For the dataset the following steps have been taken to pre-process the data:

First of all, the whole dataset is converted to lowercase characters, this is done to make all words equal in meaning, no matter the position or if written in caps. This means that words at the start of a sentence that are usually capitalized will be regarded as the same word as the non-capitalized word somewhere else in the sentence, which is exactly what we want. It comes with a minor drawback, though. Some words are always written with in all caps. For most of them (e.g., ASAP, BTW, DIY) there are no other identical words that have a different meaning, however, some words that do have a different meaning when they are written in capital letters are now no longer written that way. Words such as ‘AM’, ‘SEA’, or ‘US’, and thus these words may lose their semantic meaning.

Secondly, all the numbers that were originally in the text have been replaced with ‘num’, since the actual number that was originally there does not have any semantic meaning in a sentence and only carries a quantitative value. Thus, a general replacement for all numbers indicating there is supposed to be some sort of quantitative value at that position will be used to indicate just that.

Then all non-alphanumeric characters that are in the text are removed and replaced with an empty space ‘ ’. This is done to make word contractions that use apostrophes, such as “it is” becoming “it’s”, appear as “it s”, making them two separate words again. Furthermore, this is done to get dispose of all the excess dots and commas that may be present and of course all other non-alphanumeric characters that are without any real meaning in a sentence.

After that the words are tokenized. This means the text is split into individual words, such that the text “the quick brown fox jumps over the lazy dog” becomes a list containing each individual word: [“the”, “quick”, “brown”, “fox”, “jumps”, “over”, “the”, “lazy”, “dog”]. This is done using a Python Library provided by NLTK [13].

With all words tokenized the next step can be taken in making n-grams. To do that all the sentences are gone through and *n* words that follow each other are taken and the n-grams are made from those words. In

[1] See attachments

this case n = 4, this means if the previously mentioned sentence is taken into account, the n-grams ends up looking like following:

> [[“the”, ”quick”, ”brown”, ”fox”], [“quick”, “brown”, “fox”, “jumps”], [“brown”, “fox”, “jumps”, “over”], [“fox”, “jumps”, “over”, “the”], [“jumps”, “over”, “the”, “lazy”], [“over”, “the”, “lazy”, “dog”]].

With the processing of the n-grams two slightly different steps are taken for the models. For the simple n-gram model a library from NLTK was used that takes the tokenized sentences for us and splits them into n-grams with a max size of 4 after adding a start and end symbol for every sentence. This creates a large array with all n-grams in there and also returns the vocabulary as a result, which consists of all words that have been used for making the model.

For the RNN-LM the n-grams were made by going through the text input one by one and taking the current word and adding the next three words to that word. This way an n-gram is created after which it was added to a list. After that all the words were converted into unique integers, such that every word that is the same has the same integer and every different word has a different integer, and by doing that it is also the case that all possible words have been gone through, thus a vocabulary can be created at the same time, in which unknown words are marked as ‘oov’. This was done with the help of TensorFlow’s Library [14]. With that done all the n-gram words are transformed into integers as well. For example, if the sentence of the quick brown fox is taken once again, it will be converted from a list with n-grams consisting of words to a list with integers:

> [[“the”, ”quick”, ”brown”, ”fox”], [“quick”, “brown”, “fox”, “jumps”], [“brown”, “fox”, “jumps”, “over”], [“fox”, “jumps”, “over”, “the”], [“jumps”, “over”, “the”, “lazy”], [“over”, “the”, “lazy”, “dog”]] → [[1, 2, 3, 4], [2, 3, 4, 5], [3, 4, 5, 6], [4, 5, 6, 1], [5, 6, 1, 7], [6, 1, 7, 8]]

Then, using Python Library NumPy [15], an array is created with a shape that has the length of the number of n-grams made and a width of the length of the n-grams (n=4) and it is filled with the created n-grams. Then, using that array, the input labels and target labels are created. The input labels are all the first three elements in the array, so it would look like this for the quick brown fox example:

> [[1, 2, 3], [2, 3, 4], [3, 4, 5], [4, 5, 6], [5, 6, 1], [6, 1, 7]]

And then for the target labels the last element of the n-grams is taken and then turned into one-hot vectors. For the quick brown fox example, it would look like the following:

> [[4], [5], [6], [1], [7], [8]] → [[0. 0. 0. 0. 1. 0. 0. 0. 0.], [0. 0. 0. 0. 0. 1. 0. 0. 0.], [0. 0. 0. 0. 0. 0. 1. 0. 0.], [0. 1. 0. 0. 0. 0. 0. 0. 0.], [0. 0. 0. 0. 0. 0. 0. 1. 0.], [0. 0. 0. 0. 0. 0. 0. 0. 1.]]

With that the pre-processing for both models have been completed and the training phase is up next.

2.3 Training

Since both models use a different method for training they will be talked about separately. Because the n-gram model is the simpler one of the two, it will be the one that is started with.

For the n-gram model a LM was made by creating a probability distribution over all n-grams with maximum likelihood estimation. A smoothed model using interpolated Kneser-Key smoothing was used, which is part of the NLTK package. Smoothing is modifying a dataset in such a way that there is less noise and to make sure that there are no probabilities of zero present that we have to work with in case the model is looking at a sequence with words that it has never seen before. Interpolated Kneser-Ney smoothing is currently praised as the best smoothing method for larger as well as smaller datasets. It was fit on the training data and using

the vocabulary that was gained from the earlier function that was used to make the n-grams. The model is then saved in order for it to be used again for testing later.

For the RNN model, a model was instantiated with the required embedding for the training data. Next, two long short-term memory (LSTM) layers were added, each with 256 units, where the second layer receives a sequence output from the first layer, after it has gone through the data such that the second layer can learn from what the first layer has learned and continue with that, which should increase the performance of the model as a whole. After that a dense layer was added with an activation function to speed up the process. It uses the relu-function, which in this model means we change all negative values (if there are any) to 0, such that we end up with only non-negative values which helps the model learn faster and perform better. Then there is one last layer which is the output layer after which the output is normalized using a softmax-function. This is done to make the network resemble a probability distribution, just like the n-gram model is. Then the model is compiled using the adam-optimizer after which it is fit in on the data input and runs for a total of 1000 epochs. Lastly, it is saved in order to use it again for testing later.

2.4 POS-tagging

Another thing that is done, is adding a part of speech tagger. What a POS tagger does is going through a sentence and telling what type of word every word is in the sentence. Taking "the quick brown fox jumps over the lazy dog" as example, the POS tagger would go through the sentence and tag the words as follows:

> [('the', 'DT'), ('quick', 'JJ'), ('brown', JJ), ('fox', 'NN'), ('jumps', 'VBZ'), ('over', 'IN'), ('the', 'DT'), ('lazy', 'JJ'), ('dog', 'NN')]

DT	JJ	NN	VBZ	IN
Determiner	Adjective	Noun, singular or mass	Verb, 3rd person singular present	Preposition or subordinating conjunction

The POS tagger that has been used is NLTK's currently recommended one which is imported from their Library. The input text has been pre-processed the exact same way up until after we tokenized the text, except that numbers are not substituted with 'num'. Once the text has been tokenized it will be put into the POS tagger and all of the input sentences will be individually tagged, which means every sentence will be tagged independently from all the others. After that has been done a list with every word and their POS tag is created, like in the example above. With that list n-grams of the POS tags are made, an example is shown below, and added to a dictionary, while keeping track of how many of each combination of n-grams we have seen.

> [('DT', 'JJ', 'JJ', 'NN'), ('JJ', 'JJ', 'NN', 'VBZ'), ('JJ', 'NN', 'VBZ', 'IN'), ('NN', 'VBZ', 'IN', 'DT'), ('VBZ', 'IN', 'DT', 'JJ'), ('IN', 'DT', 'JJ', 'NN')]

This dictionary is created with as first key three words of an n-gram, as second key the last word of the n-gram and as value the number of times we have seen the second key appear as word after the first three words of the n-gram. That means the dictionary can be used to look up what POS tag is most likely to come after a sequence of three other POS tags and that can be utilized to predict what the POS tag of the next word chosen should be.

2.5 Testing

The way the models are tested is in multiple different ways. They will be addressed one by one and for each of them it will be explained how the model chooses the next word.

First model to be addressed is the n-gram model, which is fairly straight forward. The next word in this model is chosen by looking at the current input, up till the last 3 words in total. The input is pre-processed by changing all numbers into 'num' once again and by removing any non-alphanumeric characters. Then up to three words are taken and then the model looks which words have the highest probability of following those three words and returns one of those words as the prediction for the next word.

Next up is the RNN language model. For this model the basics are the same. The next word is once again chosen by looking at the current input and taking the last three words to see what the highest probable following word could be. The input is pre-processed slightly differently, the first part is the same where the numbers are substituted with 'num', and all non-alphanumeric characters are removed. After that it changes a little, because the words need to be integers again for the input. Therefore, the same method as before is applied where the words are taken, and it is looked up what integer corresponds to that word. That way a sequence of three integers is the resulting input which is exactly our models input. Using those integers, the model is asked what the next best word could be and it returns the most probable next word. However, the word is still in hot-one vector form, thus it has to be translated to be used as a normal word again, which is done using the tokenizer vocabulary after which the word is returned and the best next word prediction to the given input is presented.

Then lastly is the combination with the POS tagger. The POS tagger is used as extra check for the next predicted word, where the input is looked at and from that it searches what POS tag would be expected next. This is done by taking the current input as a whole, pre-processing it by removing all non-alphanumeric characters, adding the currently predicted word, and POS tagging the complete sentence. Then the dictionary is used to look up the POS tags of the three words before the predicted word and a list is created containing the POS tags that are expected to be next. The predicted POS tags are compared to the given POS tag of the predicted word and if there is a match it proceeds and counts the predicted word as a good prediction. If the POS tag is not matched by the predicted POS tags, the next best word is taken, and the process will be repeated. This is done up to five times, if after all five predictions the POS tags are still not matching the first predicted word is used.

2.6 Evaluation

Evaluation is done using a manually curated dataset[2], again the same one as was used in earlier performed experiments [2]. It is a set created by taking a collection of formal and informal utterances. It consists of 926 words in a total of 102 sentences.

The way the models are evaluated is by using three different types of measurements. The first measurement that is used is the accuracy of the models used. Accuracy in this case is the number of times the model correctly predicts the next word in the evaluation dataset. The next words it tries to predict are those in the sentence after the currently typed partially sentence. The measurement is a binary measurement, it either is a correct prediction or it is just wrong, since there is only one try. This way a total amount of correct predictions and a total amount of incorrect predictions are accumulated and from that a percentage of correct predictions can be calculated which gives the accuracy of the model.

[2] See attachments

The second measurement is the perplexity of the model, which is a commonly used evaluation method for language models and is seen in nearly every major language model evaluation. The perplexity of a model is essentially a number that shows how well it is at predicting the word. Intuitively it can be explained as saying the model would be correct about as often as an x-sided dice would have been [16], where x is the perplexity score. This means the lower the score is, the better it is, since the number of times you are correct by throwing an x-sided dice increases when the number of sides decrease, because a 1/80 chance is obviously worse than a 1/20 chance.

Perplexity in a model is calculated by calculating the entropy and for the n-gram model the perplexity (PP) can be written as $PP = 2^{H(p)}$ where *H(p)* is the entropy, whereas for the RNN model it is calculated as $PP = e^{H(p)}$, because the entropy is calculated using the natural log in that model. The reason they are different is because for the n-gram we use NLTK's Library to calculate the perplexity, for which they use the entropy with a 2-base log, because our model is made using NLTK. Whereas with the RRN-LM we use TensorFlow's Library, which, as said before, calculates the entropy using the natural log, after which the perplexity has to be calculated by hand with the formula above.

The last measurement is the grammatical correctness of the model, for which there are two different scores given. The way this is done is by hand. An alteration of the evaluation set is used[3] consisting of 100 unfinished sentences.

The first one is to determine whether or not the model predicted a word that was grammatically fitting in the sentence that was given to it. Here the word itself does not matter, as long as it could be a possible next word that would comply to normal English rules. After doing this there will be a score of 0-100% based on how many words predicted were grammatically correct out of all the predictions, this is the grammatical correctness score (GCS) of the model.

The second one of the two is done by looking if the word actually makes sense in the given context, that is, if it is a word a human could have also placed following up the given sentence. The results of this test will also be expressed with a 0-100% score and this is the grammatical contextual correctness score (GCCS).

[3] See attachments

3. Results

Here the results of the evaluations done by the different models are shown and discussed, first going through them one by one and then comparing them all together.

3.1 N-gram model

First up is the n-gram model. The results can be seen in table 1. The n-gram model is a fast model with, after 100 generated words, a max response time of 2.31 ms and an average response time of 0.93 ms per word.

The n-gram model generates a random word that is considered to be highly likely to be the next best word. That means when calculating the accuracy there will be different results every time it is tested on the evaluation text. To account for that a total of 500 tests have been done and the average of all of those has been taken to calculate the accuracy.

For perplexity we use a built-in function provided by NLTK, which returns the results and gives us the perplexity. This is a deterministic function and thus returns the same results every time it is called.

And then for the grammatical correctness score (GCS). The model scored pretty well on the grammatical correctness test, but there are a lot of words that could potentially fit after a sequence, so 80% is a relatively low score.

Lastly the grammatical contextual correctness score (GCCS), this is as expected lower than the GCS, since not every word fits in every sequence, even when grammatically it could be allowed there. It really is hit or miss, half of the time the model produces a word that would fit in the sentence, the other half the word just does not make any sense.

	N-gram model
Accuracy	0.46%
Perplexity	1888.64
Max response time	2.31 ms
Size	13.7 MB
GCS	80%
GCCS	51%

Table 1: Evaluation of the n-gram model

Next up is the n-gram model with the added POS tagging. The results can be seen in table 2. The speed of the model has slowed down significantly by adding the POS tagging as extra check. The average response time is now 7.11 ms which is almost 8 times slower than the original model, with the max response time of 14.17 ms.

The perplexity is still calculated the same way and therefore has not changed. The accuracy however did change, but not for the better. The same as for the n-gram model holds true for testing accuracy, therefore an average over 500 tests for the accuracy has been taken again.

For the grammatical correctness test the model scored slightly better than the model without the POS tagger, it scored 2% higher than the model without a POS tagger, which brings it to 82%. This is however still relatively low.

Finally, the GCCS, which is almost 10% better in this model with POS tagging. It has a score of 55%, which is 4% higher.

	N-gram model + POS
Accuracy	0.14%
Perplexity	1888.64
Max response time	14.17 ms
Size	13.7 MB + 0.6 MB
GCS	82%
GCCS	55%

Table 2: Evaluation of the n-gram model with POS tagger. The size is that of the model used and we add the size of the look-up table for the POS tagger

3.2 RNN model

Then for the RNN model. The results can be seen in table 3. The RNN model is also not the fastest model out there. The model had a response time that averaged at 48.67 ms, with a max response time of 77.62 ms, tested the same way as with the n-gram model. This means the base RNN model is almost 34 times slower than the base n-gram model, which is a significantly large decrease in speed.

The accuracy for the RNN model can be measured in a deterministic way, since it always gives us a list with the best options, in a sorted way. Thus, taking the first one of that list results in the same accuracy every time.

Secondly, for the perplexity TensorFlow's built-in function for models was used, which returns the cross entropy (which was 33.96) and from which the perplexity is calculated. This is also a deterministic process and therefore has to be done only once as well. It is, however, surprisingly high, unrealistically high even.

Lastly for the grammatical correctness. We see roughly the same numbers for this, just slightly lower than the n-gram models. With a GCS of 77% it is just a few percentages away, which can also be said about the GCCS with 48% score.

	RNN model
Accuracy	0.65%
Perplexity	560906354294784.0
Max response time	77.62 ms
Size	74.2 MB
GCS	77%
GCCS	48%

Table 3: Evaluation of the RNN model

Finally, the RNN model with the added POS checking. The results can be seen in table 4. The speed is once again not great, and as expected it is around 10 ms slower than the base model, just like the POS tagger on top of the n-gram model. The average tested response time was 52.43 ms, so around 4 ms slower than the base LM, but the max recorded response time was 87.05 ms.

Once again, the perplexity of the model stays the same. And surprisingly enough, so does the accuracy. Since this is also calculated in a deterministic way only one test was performed to get to the number.

At last, we have the grammatical correctness. Again, the same trend can be seen for the GCS, it being around 80%. Furthermore, the GCCS is once again around 10% better in the model with POS tagging with a score of 55%.

	RNN model + POS
Accuracy	0.65%
Perplexity	560906354294784.0
Max response time	87.05 ms
Size	74.2 MB + 0.6 MB
GCS	81%
GCCS	55%

Table 4: Evaluation of the RNN model with POS tagger. The size is that of the model used and we add the size of the look-up table for the POS tagger

If all the models are compared it is apparent that the results are fairly different, especially the difference between the RNN and the n-gram model are apparent. The full results can be seen in table 5. The biggest difference can be seen in the perplexity, which is extremely high for the RNN model. Another large difference in the models is the response times, both n-gram models are significantly faster than the RNN models.

An observation that can be made between the models with POS tagger and those without is the difference of the GCCS and also the relative response times. The POS-models are slower but score slightly higher.

	N-gram	N-gram + POS	RNN	RNN + POS
Accuracy	0.46%	0.14%	0.65%	0.65%
Perplexity	1888.64	1888.64	560906354294784.0	560906354294784.0
Max response time	2.31 ms	14.17 ms	77.62 ms	87.05 ms
Size	13.7 MB	13.7 MB + 0.6 MB	74.2 MB	74.2 MB + 0.6 MB
GCS	80%	82%	77%	81%
GCCS	51%	55%	48%	55%

Table 5: Evaluation of all models

4. Conclusion

Here the findings in this researched will be concluded and the research questions posed will be answered.

Starting with the second part of the question:

> "*whilst retaining efficiency."*

It can be safely said that this is not the case. Both models used performed worse in terms of response times when a POS tagger was applied to the models, the n-gram model far worse even, relatively speaking. And even though the size increase was marginal, it did increase slightly, but that was to be expected. That is however not a problem in itself, considering both models were already well over the size limit requirements for mobile devices. But that does sum up the conclusion for that part: efficiency was not retained, and the models are definitely not suited to be implemented on and used for mobile devices.

Then for the main question:

> *"Did the occurrence of grammatically incorrect suggestions in word predictors decrease using a combination of models with a POS tagger?"*

The short answer is yes. However, looking at the accuracy, it can be noted that either there was no change at all, which is the case for the RNN models, or less of the words supposed to be predicted were actually predicted and the model ended up with a lower accuracy, which was the case for the n-gram models. That is however not necessarily a bad thing, since multiple different words can fit after a sequence of words. Therefore, the grammatical correctness tests were conducted. The tests show that the POS tagging did deliver what was hoped for, however the increase was not significant. Nevertheless, there was a slight decrease in grammatically incorrect predictions, especially regarding the GCCS, which is a positive result and means that the use of a POS tagger in LM could definitely help as an extra check for LM used for intelligent word predictions.

To sum up the answer to the main question: the occurrence of grammatically incorrect suggestions was decreased when models are used in combinations with a POS tagger, however the accuracy of predicting the actual expected word did not change or was even decreased. Thus, as complete conclusion the following can be said:

> *Can the occurrence of grammatically incorrect suggestions in word predictors be decreased using a combination of models with a POS tagger, whilst retaining efficiency?*

No, it cannot in its current form. While we can indeed slightly decrease the occurrence of grammatically incorrect predictions, this is at the cost of the efficiency of the model, in particular costly for the response time.

5. Discussion

The results that were obtained in testing the models are far from ideal. Especially when looking at the perplexity scores. The perplexity score for the RNN model seems unrealistically high with a score far exceeding that of the n-gram model: $5.61 * 10^{14}$. However, it seems to be correct, considering it was calculated with the built-in function from TensorFlow. Nevertheless, the perplexity from the n-gram model was also far from what would be considered a good model. Taking the base line model that was used in the paper from Yu et al, it had a perplexity of 56.55 [2], which is an enormous difference with the score obtained here on the evaluation set. The reason for this can be boiled down to essentially two factors.

The first one being the size of the training dataset. Even though the same dataset was used to for the training, in this research the dataset was sized down substantially up to a point where it might have been too small a dataset to yield any significant results. The training data used in this research was with a word count of 320 thousand words, compared to the 117 million words is the other model, almost 366 times smaller. That means that consequently the amount of training sentences was way lower too, which in turn makes the number of possible n-grams the model is exposed to when learning much lower as well. All of that means that when a test is performed the chance for unknown words and never before encountered n-grams is much higher, and the model will have a harder time predicting next words, which obviously results in it performing way worse.

Most of this all could have been prevented to some extend if a larger training set was used for the model, but as stated previously the implementation did not allow for that on the hardware used for the task. A larger dataset would have improved the initial models, which in turn would make the alternative models with the extra implementation of the POS tagger more useful and the testing might have given different results in using that.

One way to potentially make use of a bigger dataset would have been to opt out of one-hot encoding, since it produces matrices with enormous dimensions, which is part of the reason why a larger dataset could not be used. Thus, finding a different way to represent the words could be helpful.

The seconds reason why the models, mainly why the RNN model might not have been an ideal model is the implementation of the model. With some testing the current numbers were chosen, but they might not have been the ideal numbers. For example, the amount of epoch chosen is based on a few tests. With 100 and 500 epoch the accuracy during training was still increasing substantially right before the training ended and the cross-entropy loss was still decreasing as well, however with 1000 epochs total, at the last 100-50 of epochs there are barely any improvements at all, and more often slight dips in accuracy and rises in the cross-entropy loss can even be observed, which could indicate that the model might be starting to overfit on the training data, so some fine-tuning for that might increase the performance of the model.

Furthermore, for the number of units in the LSTM layers not all possibilities that could have been used were tested, since the time required for that was not available (the time needed for the current model to be trained was almost 64 hours). The RNN was tested with 64, 128 and 256 units used. The performance significantly increased when switching from 64 to 128 and increased even more when switching to 256 units.

Units	100 epochs	500 epochs	1000 epochs
64		Loss: 3.73; accuracy: 0.2488	
128		Loss: 2.95; accuracy: 0.3475	
256	Loss: 3.08; accuracy: 0.3205	Loss: 1.63; accuracy: 0.5912	Loss: 1.39; accuracy: 0.6483

Table 6: performance comparison of different model settings during testing

Another point for discussion is the grammatical correctness of predictions. Currently all words that could grammatically fit in the context were accepted. However, some of those words were articles (the/an/a) which fit way easier in context since they can be used in nearly all different sorts of context, even though other words would have been a much better fit. One way to overcome this would be to only count them where you would expect articles to appear. This will probably reduce the overall score, but might paint a different picture and give more realistic results.

6. Future works

For future works and research there are a couple of things that can be done and looked into. There are a few things to go over that would be relevant and useful.

The first important change for a future work would be to have it work with a larger dataset. As mentioned in the discussion the size of the dataset has been the main bottleneck for this research. Therefore, working with and getting a larger dataset running would be the ideal scenario to improve this research and in a future work that would be the one thing to work on the most.

Another key feature is to try and reduce the size of the models we got. To get them working on mobile the devices the size has to be reduced significantly for especially the RNN model. Maybe in the future it will be less of a problem if the devices will increase in performance and thus the restrictions would be less of a restriction, but for now the size has to be decreased. There are already multiple ways to decrease model sizes [2], so it is definitely possible but something that might take a lot of time.

Speaking of mobile devices, the response time of the models was also not enough for them to be used, thus for a future work those have to be improved as well, if possible. Because even with the eventual increase in performance of devices, the response time should always be as low as possible to guarantee a seemingly instant response of the model when typing. Possible ways for improvement are to implement the POS tagging differently, and only make it check for longer sequences in order for the POS tags to be more accurate, or to integrate it in the models themselves by training the LM with POS tags as well.

Bibliography

[1] D. Anson, P. Moist, M. Przywara, H. Wells, H. S. and H. Maxime, "Keyboards, The Effects of Word Completion and Word Prediction on Typing Rates Using On-Screen," *Assistive Technology,* no. 18, pp. 146-154, https://doi.org/10.1080/10400435.2006.10131913, 2006.

[2] S. Yu, N. Kulkarni, H. Lee and J. Kim, "On-Device Neural Language Model based Word Prediction," *Proceedings of the 27th International Conference on Computational Linguistics: System Demonstrations,* pp. 128–131, https://www.aclweb.org/anthology/C18-2028, August 2018.

[3] F. Beaufays and M. Riley, "The Machine Intelligence Behind Gboard," 17 May 2017. [Online]. Available: https://ai.googleblog.com/2017/05/the-machine-intelligence-behind-gboard.html. [Accessed June 2021].

[4] S. Ghosh and P. O. Kristensson, "Neural Networks for Text Correction and Completion in Keyboard Decoding," p. arXiv:1709.06429 [cs.CL], 19 September 2017.

[5] T. Ouyang, D. Rybach, F. Beaufays and M. Riley, "Mobile keyboard input decoding with finite-state transducers," p. arXiv:1704.03987v1 [cs.CL], 13 April 2017.

[6] J. Silva, *Grammatical Error Correction System with Deep Learning,* 2018, p. http://hdl.handle.net/10211.3/208022.

[7] A. Hard, K. Rao, R. Mathews, S. Ramaswamy, F. Beaufays, S. Augenstein, H. Eichner, C. Kiddon and D. Ramage, "Federated learning for mobile keyboard predictions," p. arXiv:1811.03604v2 [cs.CL], 28 February 2019.

[8] Microsoft, "SwifKey Application," [Online]. Available: https://www.microsoft.com/en-us/swiftkey.

[9] B. Medlock, "Why Turing's legacy demands a smarter keyboard," 8 October 2015. [Online]. Available: https://medium.com/@Ben_Medlock/why-turing-s-legacy-demands-a-smarter-keyboard-9e7324463306. [Accessed June 2021].

[10] N. Sharma, "How does SwiftKey predict your next keystrokes?," 12 January 2017. [Online]. Available: https://medium.com/@curiousNupur/how-does-swiftkey-predict-your-next-keystrokes-b048ef67267d. [Accessed June 2021].

[11] M. Reporter, "SwiftKey has just taken a huge step towards predicting every word you text," 28 September 2016. [Online]. Available: https://news.microsoft.com/en-gb/2016/09/28/swiftkey-just-taken-huge-step-towards-predicting-every-word-text/. [Accessed June 2021].

[12] A. Souppouris, "SwiftKey for Android is now powered by a neural network," 15 September 2016. [Online]. Available: https://www.engadget.com/2016-09-15-swiftkey-android-neural-network-update.html. [Accessed June 2021].

[13] "NLTK," [Online]. Available: https://www.nltk.org/.

[14] "TensorFlow," [Online]. Available: https://www.tensorflow.org/.

[15] “NumPy,” [Online]. Available: https://numpy.org/.

[16] A. Schumacher, 23 September 2013. [Online]. Available: https://planspacedotorg.wordpress.com/2013/09/23/perplexity-what-it-is-and-what-yours-is/. [Accessed June 2021].

Attachments

The dataset used for training and evaluation was made publicly available by Yu et al [2] and can be found at https://github.com/Meinwerk/WordPrediction

For the evaluation for the grammatical correctness, the edited file and results, together with the core code of the thesis, can be found at https://github.com/Wouterz1/word-prediction