Georgian Language Modelling

Demetre Pipia **Demetre uridia**

{dpipi17, durid17}@freeuni.edu.ge

Abstract

We have implemented a Georgian language model that can generate text according to the given context.

We tried several model types to solve the problem, but the best result was given by 2 layers bidirectional LSTM. Which takes into account both the previous and upcoming context in the learning process and gives good results despite the limited computational resources.

For comparison, we calculated the perplexity on the test data set, which was 415,000 on the static n-gram model, while it was 729 on the transformer model and just 24 on the 2 layers bidirectional LSTM.

As a result, you can use our model for Georgian text generation, as well as in the case of a large computing resource, you can train the model on a larger Georgian text to get better results.

1 Introduction

NLP problems for the Georgian language are not well researched because few engineers are interested in the specifics of this language. One of such important issues is the generation of the Georgian language.

That is why we decided to create a language model that would be able to continue the text according to the given context.

While building this model, we had to solve several important tasks:

- The Georgian language corpus is not big enough, for example, compared to the English language. Due to this fact, we had to manually search Georgian language books and modify them into the desired format. We also took the Wikipedia dump files of the Georgian language and the other dumps of Georgian texts available on the Internet. Turned out that the quality of these texts was not satisfactory and we had to filter them.
- As we have already mentioned the Georgian language is less mastered in the world of NLP, we did not already have acceptable word vectors. Because of this, we had to solve an independent task, which was to study word2vecs on the larger data and evaluate their quality. For evaluation, we used the methods of finding synonyms, solving analogies, and also looked at the clustering of words with similar meanings on the graph.
- Unlike English and other popular languages, Georgian does not have any known metrics for evaluating text generation. For that reason, we constructed a static n-gram model that we would take as a benchmark for evaluating our model.

After all, we started to choose a deep learning model for our task. We trained several variants of transformer and LSTM(Long short-term memory). We implemented several text processing algorithms such as beam-search, top-p in combination with top-k, and based on

experiments we can say that despite the generation algorithm, 2 layers bidirectional LSTM gave us the most adequate text, and also as already mentioned, it got the lowest perplexity score on the unknown text.

2 Data collection

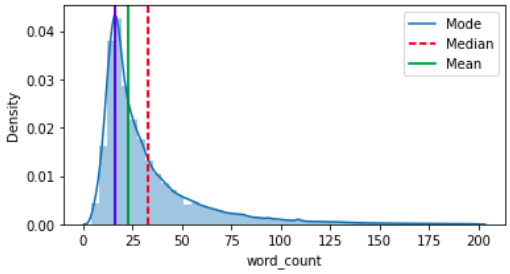
We started a data collection with Georgian and foreign (of course translated in Georgian) literature. But upon further observation, we found out that the data was too formal. The model could not learn about any of the modern words, such as a computer, phone, football… anything that rarely occurs in literature. So we started searching for much more modern data and of course, we used Wikipedia dump. But the data was still not enough, unfortunately, Wikipedia does not have huge data for such unpopular languages as is Georgian. But luckily for us, we found much more rich data, Oscar corpus which uses **Common Crawl** database.

3 Data analysis

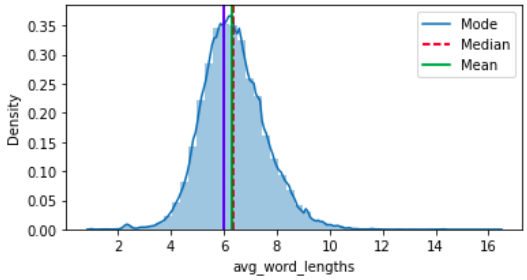
Because the data was mostly collected on the internet, it had many unnecessary data which would only confuse the model. So we needed to somehow filter data.

First of all, any words that contained anything except Georgian letters and punctuation marks were of no use. So we got rid of them. Also, paragraphs that were too small or too big were some kind of irregularities in the web and we need to filter those too. After analyzing data we saw that there were paragraphs that had repetitive words too many times and those paragraphs were not helpful.

Since we already filtered data let's check out what we got. Firstly, let's see the distribution of paragraph lengths.



As we can see Mode is around 20, mean is about 24 and median is around 35. Which was expected because most of the time paragraphs do not have huge word counts.



Another interesting thing we can see is the distribution of average word length in paragraphs. Average word length seems to be having a normal distribution with the mean, mode, and median all being close to 6.

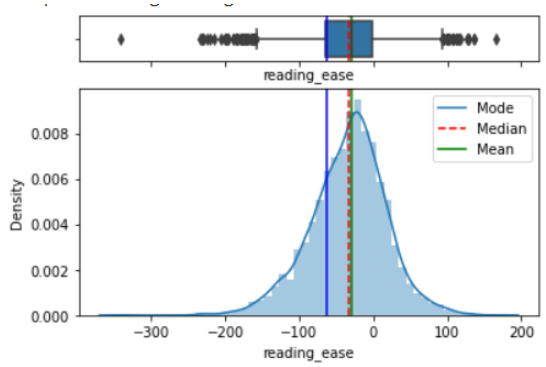
For analyzing how easily readable our data is we used Flesch–Kincaid readability tests which are originally defined for the English language but after testing it on the Georgian language it gave excellent results.

$$206.835 - 1.015\left(\frac{\text{total words}}{\text{total sentences}}\right) - 84.6\left(\frac{\text{total syllables}}{\text{total words}}\right)$$

The formula looks just like that and gives a score, how easily readable text is. The better the score is, the text is easier to read. (For more information we can see the picture below)

Score	Notes
90-100	very easy to read, easily understood by an average 11-year-old student
80-90	easy to read
70-80	fairly easy to read
60-70	easily understood by 13- to 15-year-old students
50-60	fairly difficult to read
30-50	difficult to read, best understood by college graduates
0-30	very difficult to read, best understood by university graduates

Now let's see how easily readable our data is.



As we can see most of the texts have really low scores, which should be no surprise, because we use different sources for our data, for example, Georgian literature which gives pretty difficult paragraphs.

Finally, we can conclude that data is well balanced and it satisfies all the criteria for language models to learn the Georgian language.

4 Models

The task is to assign a probability for the likelihood of a given word (or a sequence of words) to follow a sequence of words. A sequence of tokens is passed to the model which returns probability distribution over the whole vocabulary.

To optimize our models during training, we used the Cross-Entropy loss function.

$$\text{loss}(x, class) = -\log\left(\frac{\exp(x[class])}{\sum_j \exp(x[j])}\right) = -x[class] + \log\left(\sum_j \exp(x[j])\right)$$

We use perplexity as an evaluation metric for language models, which is the exponential of the cross-entropy.

4.1 Benchmark model (N-gram)

Background

As we all know, the Georgian language has a sequential nature, hence the order in which words appear in the text matters a lot. This feature allows us to understand the context of a sentence even if there are some words missing. Consider the example below:

პატარა გიორგის ძალიან უყვარს ბურთის _____

Although we do not know what the missing word is, from the context we can say that the word is likely to be "თამაში."

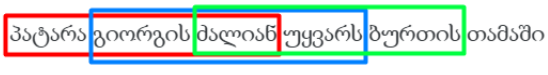
This brings us up to the idea behind the N-Grams, where the formal definition is "a contiguous sequence of n items from a given sample of text". The main idea is that given any text, we can split it into a list of unigrams (1-gram), bigrams (2-gram), trigrams (3-gram), etc. For example:

Text: პატარა გიორგი თამაშობს

Unigrams: [(პატარა), (გიორგი), (თამაშობს)]

Bigrams: [(პატარა გიორგი), (გიორგი თამაშობს)]

A more visual example of Trigrams:



Theory

The main idea of generating text using N-Grams is to assume that the last word of the n-gram can be inferred from the other words that appear in the same n-gram which is called context.

So the main simplification of the model is that we do not need to keep track of the whole sentence in order to predict the next word, we just need to look back for n-1 tokens. Meaning that the main assumption is:

$$P(\boldsymbol{x}^{(t+1)}|\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(1)}) = P(\boldsymbol{x}^{(t+1)}|\overbrace{\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)}}^{n\text{-1 words}})$$

in order to calculate the probability above we just need to apply a simple conditional probability rule.

$$\text{prob of a n-gram} \rightarrow P(\boldsymbol{x}^{(t+1)},\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)}) \Big/ P(\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)}) \leftarrow \text{prob of a (n-1)-gram}$$

The question is how do we get this n-gram and (n-1)-gram probabilities? A good answer for this question is that We can get good statistical approximation By counting n-grams in some large corpus of text.

$$\approx \frac{\text{count}(\boldsymbol{x}^{(t+1)},\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)})}{\text{count}(\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)})}$$

Implementation

We wanted to keep the code as simple as possible so that there is no need to install any external package to run the N-gram model.
In the N-gram model class, we have two simple dictionaries to index every word in the read text. The first one translates word two index and the second one translates index to the word.
As we mentioned before, to predict the token at position n, we would have to look at all previous n-1 tokens of our N-gram. So the main dictionary in our N-gram model class has such structure: keys are (n-1)-grams, (n-1) tokens tuple which represents context, and values are also dictionaries, where keys are n-th token, which comes after this (n-1) tokens, and values are counts of how many times occurs this n-th token, after given context.
The problem arises when we need to generate one of the first tokens of a sentence, as there is no preceding context available. To address this issue, we can introduce some leading tags like <sos> which will be making sure that at any point of inference we always working with correct N-Grams.

For example, for text: „პატარა გიორგი თამაშობს“ and for n=3, dictionary structure will be:

```
{
    ("<sos>", "<sos>") : {
        "პატარა" : 1
    },

    ("<sos>", "პატარა") : {
        "გიორგი" : 1
    },

    ("პატარა", "გიორგი") : {
        "თამაშობს" : 1
    },
}
```

After train this N-gram model on large text, we can generate text, using this model. It has one main method which gets context and returns probabilities distribution on the whole vocabulary. If the model does not know such context, it returns uniform distribution.

To resolve sparsity problems, we use smoothing, which gives little probability to words, when these words are not in values for a given context.

4.2 Pre-trained word vectors

We trained word2vecs using gensim library function gensim.models.Word2Vec on 10 times larger data than we used in the language model.

Words with Multiple Meanings

```
word2vec.wv.most_similar("სილა")
```

```
[('გააწნა', 0.729386568069458),
 ('გავაწანი', 0.7147606015205383),
 ('გამაწნა', 0.6759832501411438),
 ('გაერტყა', 0.6513462066650391),
 ('უთავაზა', 0.5788317918777466),
 ('წიხლი', 0.5784150958061218),
 ('გამარტყა', 0.569734513759613),
 ('ჩასჩურჩულა', 0.5666040778160095),
 ('დააყარა', 0.5660226345062256),
 ('ამოიკვნესა', 0.564782977104187)]
```

"სილა" has two meanings in Georgian, as a slap in the face (the corresponding words are found: "გააწნა", "გავაწანი", "გაერტყა") and also as sand, so the word "დააყარა" has a similar context.

```
word2vec.wv.most_similar("წელი")
```

```
[('წელიწადი', 0.8397859334945679),
 ('წელიც', 0.8122638463973999),
 ('თვე', 0.7607247233390808),
 ('წლისა', 0.7437652945518494),
 ('დღე', 0.6658779978752136),
 ('წლები', 0.664325475692749),
 ('წლით', 0.6505492329597473),
 ('წლისას', 0.6498648524284363),
 ('კვირა', 0.6388163566589355),
 ('საუკუნე', 0.6368224620819092)]
```

"წელი" also has two meanings(year, waist), although in the given text it was mainly used as a meaning of time, and therefore only synonyms of one of its meanings or related words can be found.

Synonyms & Antonyms

```
word2vec.wv.most_similar("მშვენიერი")
```

```
[('ლამაზი', 0.8420426845550537),
 ('არაჩვეულებრივი', 0.8214867115020752),
 ('საოცარი', 0.7867075204849243),
 ('დიდებული', 0.7265579700469971),
 ('გასაოცარი', 0.7260792255401611),
 ('საუცხოო', 0.7253862619400024),
 ('კარგი', 0.7209930419921875),
 ('შესაწიმწავი', 0.7135215401649475),
 ('ულამაზესი', 0.7125148177146912),
 ('მხიარული', 0.6875621676445007)]
```

As you can see perfectly found synonyms.

```
word2vec.wv.most_similar("მაღალი")
```

```
[('დაბალი', 0.9115073680877686),
 ('მაღლი', 0.762312114238739),
 ('მარალი', 0.7011095285415649),
 ('ფართო', 0.6374143362045288),
 ('დაბალია', 0.6370543241500854),
 ('მაღალია', 0.6355365514755249),
 ('უმაღლესი', 0.6288471221923828),
 ('მიწიმალური', 0.623460054397583),
 ('დაბალ', 0.6142180562019348),
 ('მაღალ', 0.600111722946167)]
```

In this case, the most similar word to "მაღალი" (high) was thrown out "დაბალი" (low). It is likely that "მაღალი"(high) and "დაბალი"(low) in sentences of the same context often alternate. Probably in any sentence they can be replaced with each other so that the sentence stays meaningful.

Solving Analogies

```
word2vec.wv.most_similar(positive=['გოგო', 'ქმარი'], negative=['ბიჭი'])
```

```
[('ცოლი', 0.8019391298294067),
 ('ქმარივით', 0.7068041563034058),
 ('შვილი', 0.7023085951805115),
 ('დედამთილი', 0.7009946107864380),
 ('ქმარ', 0.7006902694702148),
 ('დაქალი', 0.6978672742843628),
 ('ცოლიც', 0.6914170980453491),
 ('მეუღლე', 0.6692677140235901),
 ('მამამთილი', 0.6644603013992310),
 ('შეყვარებული', 0.6641018390655518)]
```

The analogy is as follows: “ბიჭი“(boy)-“ქმარი“ (husband): “გოგო“ (girl) - ?. The correct answer is „ცოლი“ (wife).

```
[ ]  word2vec.wv.most_similar(positive=['ქალი', 'მეფე'], negative=['კაცი'])

    [('დედოფალი', 0.7543380260467529),
     ('პრინცესა', 0.6602231860160828),
     ('მეუღლე', 0.6583357453346252),
     ('დედოფლად', 0.6401000618934631),
     ('ერცჰერცოგი', 0.6390016674995422),
     ('უფლისწული', 0.6362884044647217),
     ('ლიზელოტა', 0.633010745048523),
     ('კროწპრინც', 0.632745087146759),
     ('ელისაბედი', 0.6322320699691772),
     ('დოროთეა', 0.6304565668106079)]
```

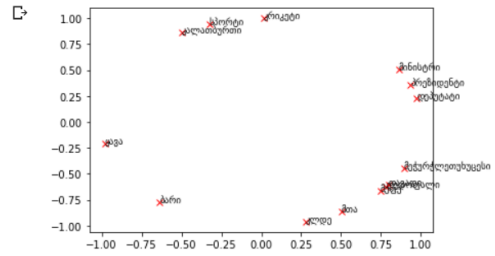
The analogy is as follows: “კაცი“(man)-“მეფე“(king): “ქალი“(woman) - ?. The correct answer is „დედოფალი“(queen).

```
[ ]  word2vec.wv.most_similar(positive=['თევზი', 'დარბის'], negative=['ძაღლი'])

    [('ცურავს', 0.6150200963020325),
     ('ცურავს', 0.6113138198852539),
     ('დაცურავს', 0.5872666835784912),
     ('დაცურავს', 0.5579556226730347),
     ('მიგრირებს', 0.5552483797073364),
     ('დასრიალებენ', 0.5440467000007629),
     ('იჭრება', 0.5435172319412231),
     ('ხტის', 0.5343250036239624),
     ('იყრება', 0.5333105325698853),
     ('იკვეთება', 0.5328798890113831)]
```

The analogy is as follows: “ძაღლი“(dog)-“დარბის“(is running): “თევზი“(fish) - ?. The correct answer is „ცურავს“ or „დაცურავს“(is swimming). In this case, it was slightly difficult for word2vec to match the verb in the correct form.

```
plot_embeddings(get_normilized_embeddings(word2vec, words), words)
```



we see from the graph that several clusters are separated, for example: „მინისტრი“(Minister), „პრეზიდენტი“(President), „დეპუტატი“(Deputy). probably from a similar political context.

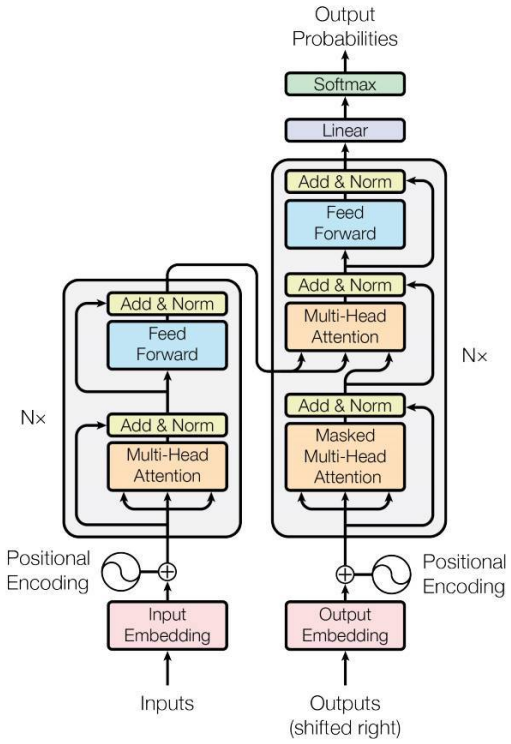
Accordingly, we meet „თავადი“(Lord), “მეფე“(King), „დედოფალი(Queen).

The most interesting is the position of the word "ბარი" on the graph because a bar has two meanings. One as a cafe-bar and the other as a field, respectively the bar is roughly in the middle between the coffee and the mountain.

With model training experiments, we found that initialization with pre-trained vectors helps the model learn faster, which means loss and perplexity decrease relatively quickly.

4.3 Transformer model

For this model we use a PyTorch transformer module which is based on popular paper **Attention is all you need.**



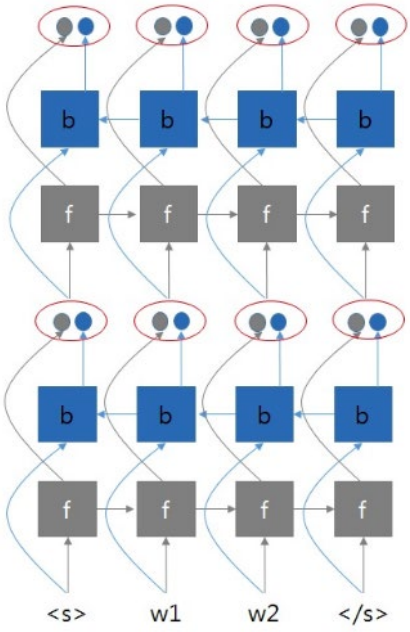
A sequence of tokens is passed to the embedding layer first, followed by a positional encoding layer to account for the order of the word. The Pytorch TransformerEncoder consists of multiple layers of TransformerEncoderLayer. Along with the input sequence, a square attention mask is calculated because the self-attention layers in TransformerEncoder are only allowed to attend the earlier positions in the sequence. For the language modeling task, any tokens on the future positions should be masked. To have the

actual words, the output of the TransformerEncoder model is sent to the final Linear layer, which is followed by a log-Softmax function.

The PositionalEncoding module injects some information about the relative or absolute position of the tokens in the sequence. The positional encodings have the same dimension as the embeddings so that the two can be summed. Here, we use sine and cosine functions of different frequencies.

4.4 LSTM model

For this model, we wanted to use recurrent neural networks. First, we started with regular lstm which had good results but not good enough. Then we decided to add bidirectionality into the model which dramatically improved model performance. The last step was to add multiple layers into lstm. To test that each model was behaving sensibly we tested models with minimal data, and they were able to memorize each one of them. 2 layers bidirectional LSTM has a pretty interesting architecture as we can see from the picture.



A sequence of tokens is passed to the embedding layer first, followed by a multilayer bidirectional LSTM layer and then a simple linear layer that returns distribution over the whole vocabulary.

Bidirectional LSTM uses backward and forwards paths so the model could learn not just about previous words but also about the whole context. And we were not disappointed with the outcome. It showed promising results on test data, with perplexity being as low as 24.11... (We should also take the fact into consideration that we did not have access to huge computation resources which could only make our model better).

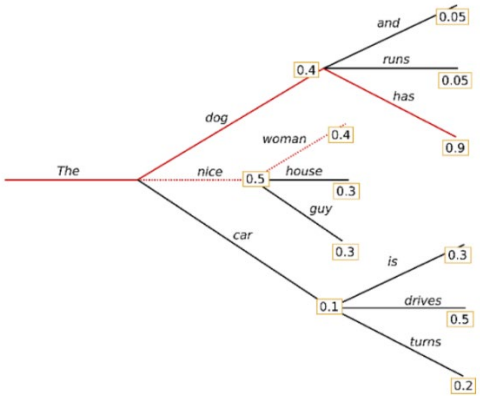
5 Model evaluation

5.1 language generation

We tried several methods for generating text from our model. These methods are beam-search and top-p in combination with top-k. We will try to briefly describe the principle of their work and show what results they had on our best model.

Beam search

Beam search reduces the risk of missing hidden high probability word sequences by keeping the most likely num_beams of hypotheses at each time step and eventually choosing the hypothesis that has the overall highest probability. Let's illustrate with num_beams=2:

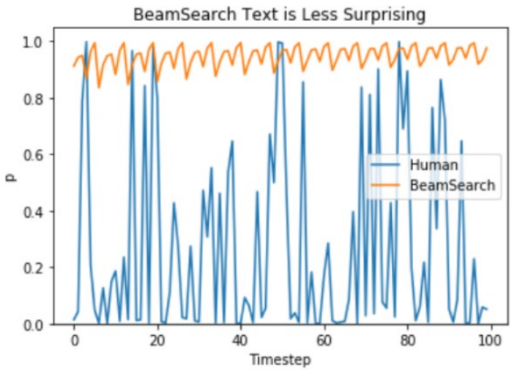


At time step 1, besides the most likely hypothesis ("The", "woman"), beam search also keeps track of the second most likely one ("The", "dog"). At time step 2, beam search finds that the word sequence ("The", "dog", "has"), has with 0.360.36 a higher probability than ("The", "nice", "woman"), which has 0.20.2.

Beam search will always find an output sequence with a higher probability than greedy search, but We have seen that beam search heavily suffers from repetitive generation on our model.

Testing has shown that beam search is very slow if we increase the depth (because the algorithm has exponential growth) and if the depth is small, it does not show nice results.

It is also a known fact that high-quality human language does not follow a distribution of high probability next words. In other words, as humans, we want generated text to surprise us and not to be boring/predictable.



For these reasons, we decided to try another method.

Top-p in combination with Top-k

In Top-K sampling, the K most likely next words are filtered and the probability mass is redistributed among only those K next words.

In Top-p sampling, we choose from the smallest possible set of words whose cumulative probability exceeds the probability p. The probability mass is then redistributed among this set of words. (for redistribution, we had two options: 1. to divide every element to the sum of remaining probabilities. 2. to just pass through softmax function. Testing has shown that first approach was better) This way, the size of the set of words can dynamically increase and decrease according to the next word's probability distribution.

In our implementation of text generation, we use Top-p in combination with Top-K, which helps us to avoid very low ranked words while allowing for some dynamic selection and also improves the performance of top-p algorithm, because we sort distribution in descending order.

Base on the experience we chose k as 1000 and p as 0.8.

5.2 Lstm and N-gram comparison

Let's compare Lstm and N-gram models. It should be no surprise that the lstm model is much more superior than the n-gram. For example, if we compute perplexity on the test set, N-gram has a perplexity of 415551.411 while Lstm has 24.111.

This is what n-gram generates when we give input მდინარე(river) and length 10.

```
მდინარე თრიაღეთის სხეულს ქორწილის
დაცვის გვიანდელი სასუნთქ იხსნა
ხიფათი სადაც მოპყრობისა
```

It hard for n-gram to catch context and generated sentence has no meaning.

Let's see how lstm behaves when we give input პოლიტიკა (politics) and length 10.

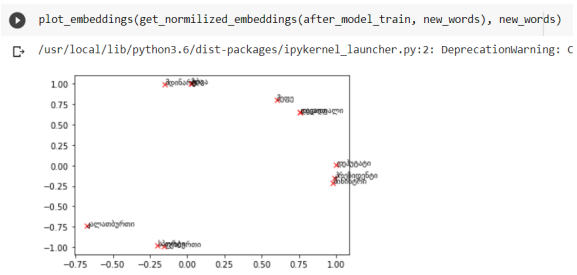
```
პოლიტიკა ვისაც გვაქვს არასწორი
ან ამის მიუხედავად პრემიერ
ბრიუსელში პოლიტიკური ინტერესები
```

Even though lstm catches the sentence context greatly it still fails to construct sentences that would be grammatically correct. Of course, N-gram has the same problem.

When there is an OOV (out of vocabulary) word in the sentence, lstm still manages to catch context based on other words, while n-gram completely breaks because it uses probability methods and has never seen such a word, it does not know how to behave.

5.3 Word2vecs after model trains

After language model training, word2vecs quality was still great.



We see following clusters on the graph:

- „მდინარე“(River), „ზღვა“(Sea), “ტბა“(Lake).
- “მეფე“(King), “დედოფალი“(Queen), „თავადი“ (Lord).
- “დეპუტატი“(Deputy), “მინისტრი“(Minister), “პრეზიდენტი“(President).
- “კალათბურთი“(Basketball), „სპორტი“(Sport), “ფეხბურთი“(Football).

6 Conclusion

Finally, we can assume that our goal has been achieved, because we have created a deep learning model that can generate Georgian language text. Despite the problems caused by the pandemic, both authors worked together online and came up with a product that can analyze the existing initial context and accordingly continue the given sentence.

References

- Natural Language Processing with Deep Learning (CS224N/Ling284) Lecture 6: Language Models and Recurrent Neural Networks: http://web.stanford.edu/class/cs224n/slides/cs224n-2019-lecture06-rnnlm.pdf
- Text Generation Using N-Gram Model: https://towardsdatascience.com/text-generation-using-n-gram-model-8d12d9802aa0
- Word2vec embeddings: https://radimrehurek.com/gensim/models/word2vec.html
- Ultimate guide to deal with Text Data (using Python) – for Data Scientists and Engineers: https://www.analyticsvidhya.com/blog/2018/02/the-different-methods-deal-text-data-predictive-python/
- How to generate text: using different decoding methods for language generation: https://huggingface.co/blog/how-to-generate
- SAVING AND LOADING MODELS: https://pytorch.org/tutorials/beginner/saving_loading_models.html?fbclid=IwAR3_pnInEceed4hXzNoROk7FFNt0TorFu-IynEqX5evp0s-HG_3_CJLA-hw
- SEQUENCE-TO-SEQUENCE MODELING WITH NN.TRANSFORMER AND TORCHTEXT: https://pytorch.org/tutorials/beginner/transformer_tutorial.html?fbclid=IwAR1udzv35GMPWnUOA4ksdI3YX01uQvd4jP3cI8YJI7M-jRKSWEUwWuyaeC8
- Attention Is All You Need: https://arxiv.org/pdf/1706.03762.pdf
- CROSSENTROPYLOSS: https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html
- Perplexity in Language Models: https://towardsdatascience.com/perplexity-in-language-models-87a196019a94
- Georgian texts on internet dumbs: https://oscar-corpus.com/
- Books on Georgian language: https://drive.google.com/drive/folders/0BwCMyTECcJPnSnhzblRydjJPd2M
- Wiki dumps of Georgian language corpus: https://dumps.wikimedia.org/kawiki/latest/?fbclid=IwAR1fEJ5YzoWZt6BDXu4mms6UMvomM__rLJMdnpvRPrZ3MpQhf-vlt9S5ljE
- Flesch–Kincaid readability tests: https://en.wikipedia.org/wiki/Flesch%E2%80%93Kincaid_readability_tests
- Understanding LSTM Networks: https://colah.github.io/posts/2015-08-Understanding-LSTMs/#:~:text=Long%20Short%20Term%20Memory%20networks,many%20people%20in%20following%20work.
- Mosestokenizer: https://pypi.org/project/mosestokenizer/