Transformation of Russian prose to poem

Andrey Brykin

brand17@yandex.ru

Abstract

I describe an algorithm that transforms prose to poem minimizing changes to the initial text. My approach is to find the most appropriate words to insert/replace words using bidirectional recurrent neural network language model and then find the best path through different options. Poems obey rhythmic and rhyme constraints.

1. Introduction

Many poem generation algorithms were published last years. A majority of them generate poems based on user-supplied topic.

In this paper, I describe a backtracking algorithm that generates poem from any prose minimizing changes to the initial text.

Algorithm consists of the following steps:

1. First bidirectional recurrent neural network (RNN) language model (LM1) generates sets of words, which could be inserted between the words of initial text (including empty word). The words are pushed to priority queue (PQ1) with a weight equal to the probability returned by the LM1.
2. Second bidirectional RNN language model (LM2) generates sets of words, which could replace the words of initial text (including empty word). The words are pushed to priority queue (PQ2) with a weight equal to the probability returned by the LM2.
3. Array of sets of possible words is created as shown in Figure 1.
4. Words from initial text are added to the odd slots (1, 3, etc.) of the array.
5. On each step one word is extracted from priority queues – words from PQ1 are extracted to even slots (0, 2, etc.), from PQ2 – to odd slots (1, 3, etc.). The words are extracted until the two conditions are met:
    a. Pairs of rhymed words exist and ties to the rhyme scheme (e.g. for rhyme scheme ABAB – pair exists in slots 5 and 15 with another pair existing in slots 10 and 20).

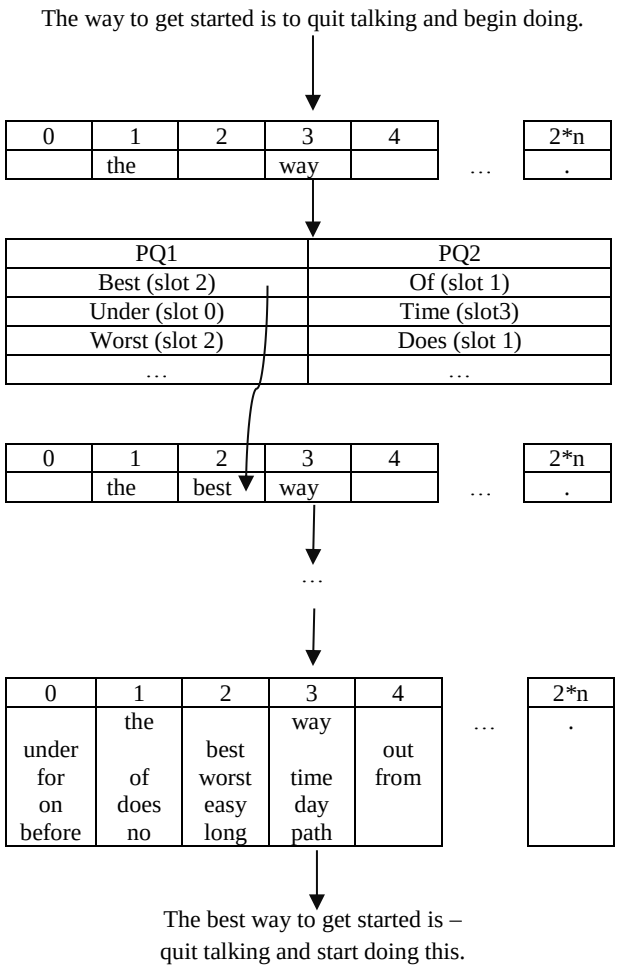

Figure 1: Overview of algorithm transforming prose to two-line iambic poem.

b. For every line ending by the corresponding end slot - sequence of words exists and obey to rhythm constraints (for the example above - sequences of words exists and obey to iambic pentameter for slots from 0 to 4, from 6 to 8, from 11 to 13, from 16 to 18).

2. Prior work

Last years automated poetry generation has been a popular research topic following progress in development of neural networks.

Andrej Karpathy developed a simple RNN which surprisingly was able to generate text samples looking like Shakespeare sonnets[1].

Marjan Ghazvininejad et al. [2]proposed a method based on building a graph of words related to topic and use RNN to score paths through the graph.

Zhe Wang et al.[3] proposed a similar technique to generate Chinese poetry.

John Benhardt et al.[4] proposed some improvements to this technique.

Hopkins and Kiela[5] proposed to use a pipepline comprising of generative model to produce content and discriminating model to represent form.

Harsh Jhamtani et al.[6] proposed using a discriminator that directly captures rhyming constraints.

Xiaoyuan Y et al.[7] proposed a reinforcement learning schema to generate poems with no rhythm/rhyme constraints.

Oliveira[8] surveyed different poetry generators and systematized existing approaches to this topic.

Novel contributions of my work are:

- Minimizing changes to the initial text improves meaningfulness of poem
- Results of my algorithm are more predictable comparing to building of poem from topic. Such results could be useful in areas like translation of poems and songs to other languages.
- The algorithm is fast (generates a poem within one second) because only two evaluations of neural network are required.
- I trained the model using prose corpora. It should add diversity and innovation to the poems.
- Russian language was not yet targeted by researchers.
- Backtracking algorithms were not targeted by researchers.

3. Vocabulary

Russian writing is quite phonetic and it is possible to retrieve rhymes directly from letter strings.

I tested only iambic and trochee masculine patterns.

For every word in my vocabulary, I precomputed stress pattern (total syllables and number of stressed syllable) and rhyme code.

To determine rhyme code I extracted the last syllable and replaced some letters by others which sounds similar: vowels "яеёию" by "аэоыу" correspondingly, consonants "бвгджзч" by "пфктшсщ" correspondingly. Then I removed consonants prior the vowel if there were consonants after the vowel.

I used word-level RNN language model comprising of one-layer recurrent network with long short-term memory (LSTM) units[9]. My vocabulary size is 50000. Because of a high number of output classes – I used sample softmax heuristic[10] to speed up training. For training RNN, I used different texts from electronic libraries.

There are many words with multiple syllable-stress patterns in Russian.

I used an accented National corpus of Russian language created by the Institute of Russian language[1] for training a separate bidirectional

[1] http://ruscorpora.ru/new/corpora-usage.html

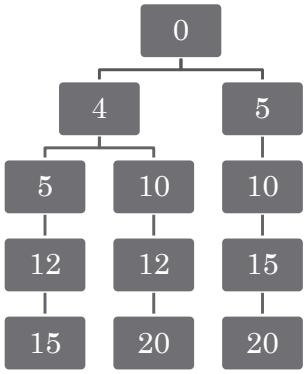

Figure 2: Tree of pair combinations

RNN determining an accented syllable based on the context.

Another challenge is specific for Russian writing: the letter “ё” is usually changed by “e” but sounds differently. As a result, sometimes “e” needs to be replaced by “e”.

I used the same National corpus of Russian language for training another bidirectional RNN determining whether such replacement should be made based on context.

I used the both RNNs to make changes to texts used for training RNN responsible for determining words to insert/replace to original prose. Accented National corpus of Russian language is not big enough for that. Then I checked the resulted vocabulary for errors and corrected them.

4. Backtracking

Assuming an average number of options for every slot is ~20 (40 good words generated by RNN with a half of which not obeying the rhythm), number of different combinations equals to 20^n. These options are generated by RNN – so none of them is meaningless. However, in practice – they are rarely looking good enough within the whole phrase and it is important to find the best of them and minimize changes to the initial text.

To do this – the words are added to phrase from priority queues starting from the best to the worse and for every new word the algorithm tests whether it is possible to compose a poem from the words already added. New words are added until a full phrase is possible.

Adding a new word solves the problem (makes the full phrase possible) only if it is rhymed with a word in another slot (creating a new combination of pairs for which a sequence of words exists), or its stress pattern differs from other words in this slot (creating a new sequence of words for certain combination of pairs).

Therefore, for every new word it is enough to check for every combination of pairs – whether a sequence of word obeying rhythm constraints exists for every line.

Checking that the stress pattern of the new word differs from other words in the slot can be done efficiently by storing stress pattern in a separate set for every slot.

Checking that the new word is creates a pair with another slots can be done efficiently by storing end codes for all words in a dictionary end-to-slot.

My approach to traversing pair combinations and testing of lines are described below.

4.1. Traversing through pairs of rhymed words

For a given combination of pairs, it does not make sense to test the next line if the previous line is not possible.

So traversing could be done quickly if the combinations of pairs are stored as a tree.

For example, if we have pairs in slots 4 and 12, 5 and 15, 10 and 20 – the possible pair combinations for rhyme scheme ABAB would be (4,5,12,15), (4,10,12,20), (5,10,15,20). If there is no sequence of words from the slot 0 to slot 4 – it does not make sense to check further for the first and second combinations. Figure 2 illustrates the shape of the tree.

Worst-case complexity equals to number of possible ranges and equals to l*(l-1), where l is a number of slots. On average, it is much less because of pruning.

4.2. Checking that sequence of words exists for line

Testing of the line can be done quickly using dynamic programming algorithm. For this purpose, I created array of sets to store possible accumulated lengths for every slot in every possible range of slots. Figure 5 illustrates the idea for the first line (which always starts from slot 0). For example - to check whether the sequence of words exist for the first line ended by word "this" in slot 5 - it is enough to check whether the corresponding accumulated length exist in the set in slot 4. It does not exist for iambic pentameter – there is no accumulated length of nine in slot 4. However, it exists for iambic tetrameter because the accumulated length of seven exists in the slot 4.

It is necessary to have n arrays - for every starting slot: first array starts from slot 0, second – from slot 1, etc.

These arrays could be updated quickly in a process of testing a new word. A new word affects only the arrays starting before its slot. Moreover, it affects only the slots next to the added word. For example, if a new word "its" is added to slot 1 – it does not affect any array because other words with the same stress pattern already exist in the slot (and therefore were already tested). If a new word "under" is added to the slot 0 – to does not affect arrays because it does not obey our rhythm constraint (the second syllable is stressed). If a new word "easiest" is added to slot 2 – it does not affect arrays for starting slot more than 2. For the array with starting slot 0 – it does not affect slots 0 and 1.

Another observation important for efficient pruning – if a new word did not affect set in the slot – then it will not affect the slots next to it. For example, if we add a word "their" to the slot 2 – it does not affect a set in this slot (despite the stress pattern of this word is new for the slot - all possible accumulated lengths are already included in the set). Therefore, it does not make sense to check the slots next to the slot 2.

It is also important that it does not make sense to add accumulated length exceeding a number of syllables in the line. For example – if we creating a poem with a form $(10)^3$ – lengths of 7 and 8 are impossible – so we should not add them.

Based on the above observations – updating of arrays could be efficiently done recursively. Worst-case complexity of this task is n arrays * n slots * m (number of lengths in the previous slot). However, after adding a few words in every slot it becomes linear on average.

This simple algorithm generates bad poems in the case when it replaces the initial text. In my example - if you change the initial word "the" by word "of" – you get such options as "under of way" which is impossible. To avoid it I prevented insertion of words into the slots that are neighbors to the replaced word. So you get only options "Of way …", "Does way …", "No way …" in our example.

It affects accumulated lengths calculation – certain accumulated lengths are not possible now because some paths through possible words are not permitted now. In our example – accumulated length of four is impossible for slot 1 because combination "under does" is not permitted more. Figure 3 illustrates a graph of possible ways. Initial words are marked.

To calculate possible accumulated lengths I need to keep separately those accumulated lengths, which can add up only to one length for the next slot: zero for odd slot, length of initial word for even slot. To calculated accumulated lengths for slot, I first added this "universal" length to all accumulated lengths from the previous slot and added the result to the first set. Then I added other word lengths to accumulated lengths from the first set of the previous slot and added the result to the second slot. Figure 4 illustrates the idea.

0	1	2	3	4
for on before	the of does no	 best worst easy long	way time day path	 out from

0 1 2	0 1 2 3 4	0 1 2 3 4 5 6	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7 8

Figure 5: Array of accumulated lengths for the sets of possible words.

4.3. Choosing priority queue to extract the next word

To minimize changes made to the initial text it is necessary to extract more words from PQ1 than from PQ2. I observed that RNN1 generates ~40 good words to insert. So I extracted first 40*n words from PQ1.

After that, I extracted from the both slots. Replacing of word usually results in worse meaningfulness comparing to insertion of word. So I prioritized PQ1.

4.4. Punctuation marks

Algorithm does not insert punctuation marks.

It deals with marks in the original text based on the following restrictions:

- they are permitted in the end of line,
- they are permitted in the middle of the line only if the line starts from mark and ends by mark.

4.5. Words not included into vocabulary

Such words are posted to RNN as a special "universal" code. As a result, RNN generates the nearest words with a bit lower quality.

4.6. Traversing the results

When adding a new word makes a phrase possible – it usually results in enormous number of options. It happens because of many words

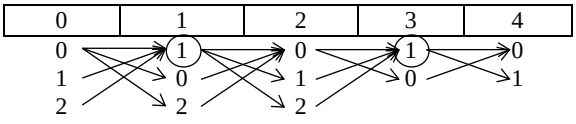

Figure 3: Directed graph of possible paths through different word lengths

0	1 2 3	0 1 2 3	1 2 3 4 5 6	0 1 2 3 4 5 6
1 2	0	4 5	0	7

Figure 4: Array of accumulated lengths with two sets

usually obey rhythm pattern for a given slot (in Figure 1 the word "best" could be replaced by "worst" or "long"). Another reason is that there could be many possible ways through different rhythms within every line (in Figure 1 the sequence "The best" can be replaced by "Before").

Therefore, it is important to choose the best option and give an opportunity to see other possible options. I weighted every word with a counter of words (weight 0 – for the first word added, 1 – for second, etc.). The words with the least weight should be used for a given rhythm. I weighted rhythm sequence by concatenating the best weights for every slot and sorting the resulted array of weights in decreasing order. The least arrays of weights should be used. I developed a functionality to traverse through rhythm sequences for every line and through different rhymes. Weight arrays should be calculated for the both cases.

5. Complexity

Worst-case complexity of every step is $O(n^2*m)$, where n – number of slots, m – number of syllables in every line.

Worst-case number of steps is n*l, where l – size of dictionary. So worst-case complexity of algorithm is $O(n^3*m*l)$.

Actually the shortest path could be found using Dijkstra algorithm (with a better complexity) as follows.

1. Calculate arrays of sets to store possible accumulated lengths for every slot in every possible range of slots for all words (every word can be extracted in every slot). Sets of syllables are the same for every slot and can be created from the dictionary in O(l) time. Arrays can be created in $O(n^2*m)$ time.
2. Create array of ending word rhythm patterns for every slot: for every pattern of every slot of every array – check whether a proper accumulated length exist in prior slot. Complexity is $k*n^2$, where k – number of patterns. List of patterns can be created from a dictionary in O(l) time.
3. Create array of records (possible ends, end slots, weights) for every slot. For example, if range slot0-slot5 is possible with end0 – then we add record (end0, 5, w0) to slot0, where w0 – weight of the best word (paragraph 4.6). If all the words are added to dictionaries rhythm-end-words for every slot – it could be done in O(log(e)), where e – number of ends.
4. Now we have a list of edges of weighted directed graph. And we can use Dijkstra algorithm to find the shortest path in O(|E|+|V|*log|V|) time.

The problem is that |E| and |V| are extremely high. For example, if we have 30 slots and 600 ends, |V| approximately equals to 30 * 30 * 600 = 540000. If we create a poem of 4 lines - |E| approximately equals to $(30 * 600)^4$.

6. Results and analysis

The very first poem generated by algorithm is usually not good. It is necessary to look through a few options to find something interesting.

Sample outputs are presented in Figures 6-11.

Meaningfulness of poems is preserved wellin a course of insertion of additional words. But it quickly degrades after a few of initial words are replaced: usually the model replaces word not by synonym but by the word with a different meaning.

Иногда момент настал,
На который так уже
Долго терпеливо ждал,
Иногда приходит же…

Figure 6: Poem generated from "Иногда момент, который ты так долго ждал, приходит в самое неподходящее время…"

Если каждый нравится –
Он как будто хочет их,
И расплачивается
Наказание за них.

Если каждый гражданин.
Это как работает.
И последовал один
Исключением сам нет.

Figure 7: Poems generated from "Каждый живет, как хочет, и расплачивается за это сам."

The results are rarely looking as good poetry. Usually the poems need further improvements. However, sometimes they can be used as a starting point to create a poem.

For example, the second poem from Figure 6 can be slightly adjusted and transformed to:

Если этот гражданин
Хорошо работает -
Он получит все один -
Исключения здесь нет.

The poem from Figure 9 may be adjusted to:

Никогда не совершай
Ты ошибку всех детей:
Никогда не полагай,
Что умнее всех людей.

The major idea of initial text is preserved in the both cases.

Other poems could be improved as well.

To speed up looking through low quality options

Человека так ничто
Никогда не выдает,
Знаете как только то,
Что смешит его народ.

Figure 8: Poem generated from "Человека так ничто не выдает, как то, что смешит его."

I added a functionality of removing a "bad" word from results.

If the syntax of initial text is too complicated – probability of getting a good option falls. It happens because RNN doesn't recognise rare sintax pattern and the suggested words may not be correct. In this case manual simplification of syntax helps. Another way to deal with complicated syntax is removing "bad" words. In example, illustrated in Figure 8, it is better to preserve sequense "то, что". So it is useful to manually remove all the words inserted within this sequence.

Sometimes RNN replaces important initial word and generates a big number of low quality poems. It this case it is useful to remove "emply" word for the corresponding slot.

Sometimes long words in initial text are ordered so that it is difficult for algorithm to find a good word to insert while obeying rhythm constraints. It is always useful to help algorithm by changing order of words. For example, a phrase "Ничт`о так не выда`ёт челов`ека, как то, над чем он сме`ётся" is difficult because the stress pattern for sequence of long words "выда`ёт челов`ека" is "0010010". It is difficult to find good words obeying iambic tetrameter. But if we change the order to "Человека так ничто не выда`ёт, как то, над чем сме`ётся он" – the number of bad options will be much lower. If we further change the initial text – we get a good result – refer to Figure 8.

Никогда не совершай
Ли ошибку всех детей:
Никогда не забывай,
Что умнее всех людей.

Figure 9: Poem generated from "Не совершай классическую ошибку всех умников: не думай, что нет людей умнее тебя."

Настроение идет,
Это и когда рад вам
Видеть даже тех господ,
Кто ошибся дверью сам

Figure 11: Poem generated from "Новогоднее настроение – это когда рад видеть даже тех, кто ошибся дверью"

If the initial text is too short – probability of getting a good option falls. It happens because RNN usually prioritize short words over long words, and adding a big number of words usually damages meaningfulness. It is a problem because a number of such bad options could be enourmous. When it happened – I manually added words to the initial text. Usually such words could be taken from the options suggested by the algorithm.

Sometimes it could be useful to change position of punctuation marks. For example – it is difficult to get a good poem from short text "Будь счастлив в этот миг. Этот миг и есть твоя жизнь". But if we change the text to "Будь счастливым, в этот самый миг, он и есть, жизнь твоя" – we have a good result (Figure 11). It could be improved manually.

After short training I usually was able to find a poem in minutes for any text given. The algorithm generates enormous number of options.

I tried to improve quality in the following ways:

1) by predicting the best slot for inserting a word (even slots)
2) by predicting the best slot for replacement

Будь счастливым стариком,
В этот самый миг всегда,
Так он так и то есть дом,
И жизнь и твоя мечта.

Figure 10: Poem generated from "Будь счастливым, в этот самый миг, он и есть, жизнь твоя"

of initial word (odd slots)

3) by predicting the best slot for deleting of initial word (odd slots).

I trained RNN models in each case. All these model didn't improve quality.

7. Conclusion

I have described an algorithm of transformation prose to poem minimizing changes to the initial text. The algorithm uses RNN to generate possible words to adjust initial text.

References

[1] Andrej Karpathy. "The Unreasonable Effectiveness of Recurrent Neural Networks"

[2] Marjan Ghazvininejad, Xing Shi, Yejin Choi, and Kevin Knight. http://52.24.230.241/poem/index.html

[3] Zhe Wang, Wei He, Hua Wu, Haiyang Wu, Wei Li, Haifeng Wang, Enhong Chen. 2016. Chinese Poetry Generation with Planning based Neural Network. arXiv preprint arXiv:1610.09889.

[4] John Benhardt, Peter Hase, Liuyi Zhu, Cynthia Rudin. Shall I Compare Thee to a Machine-Written Sonnet? An Approach to Algorithmic Sonnet Generation. arXiv preprint arXiv:1811.05067.

[5] Jack Hopkins and Douwe Kiela. "Automatically Generating Rhythmic Verse with Neural Networks." ACL (2017).

[6] Harsh Jhamtani, Sanket Vaibhav Mehta, Jaime Carbonell, Taylor Berg-Kirkpatrick. Learning Rhyming Constraints using Structured Adversaries. 2019. arXiv preprint arXiv:1909.06743.

[7] Xiaoyuan Yi, Maosong Sun, Ruoyu Li , Wenhao Li. Automatic Poetry Generation with Mutual Reinforcement Learning.

[8] Goncalo Oliveira. A Survey on Intelligent Poetry Generation: Languages, Features, Techniques, Reutilisation and Evaluation. 2017.

[9] Sepp Hochreiter and Jurgen Schmidhuber. 1997. Long short-term memory. Neural Computation, 9(8).

[10] Yoshua Bengio and Jean-Sebastien Senecal. 2007. Adaptive Importance Sampling to Accelerate Training of a Neural Probabilistic Language Model. Dept. IRO, Universite de Montreal.