Text Generation Using N-Gram Model

Oleg Borisov Oct 28, 2020 · 7 min read

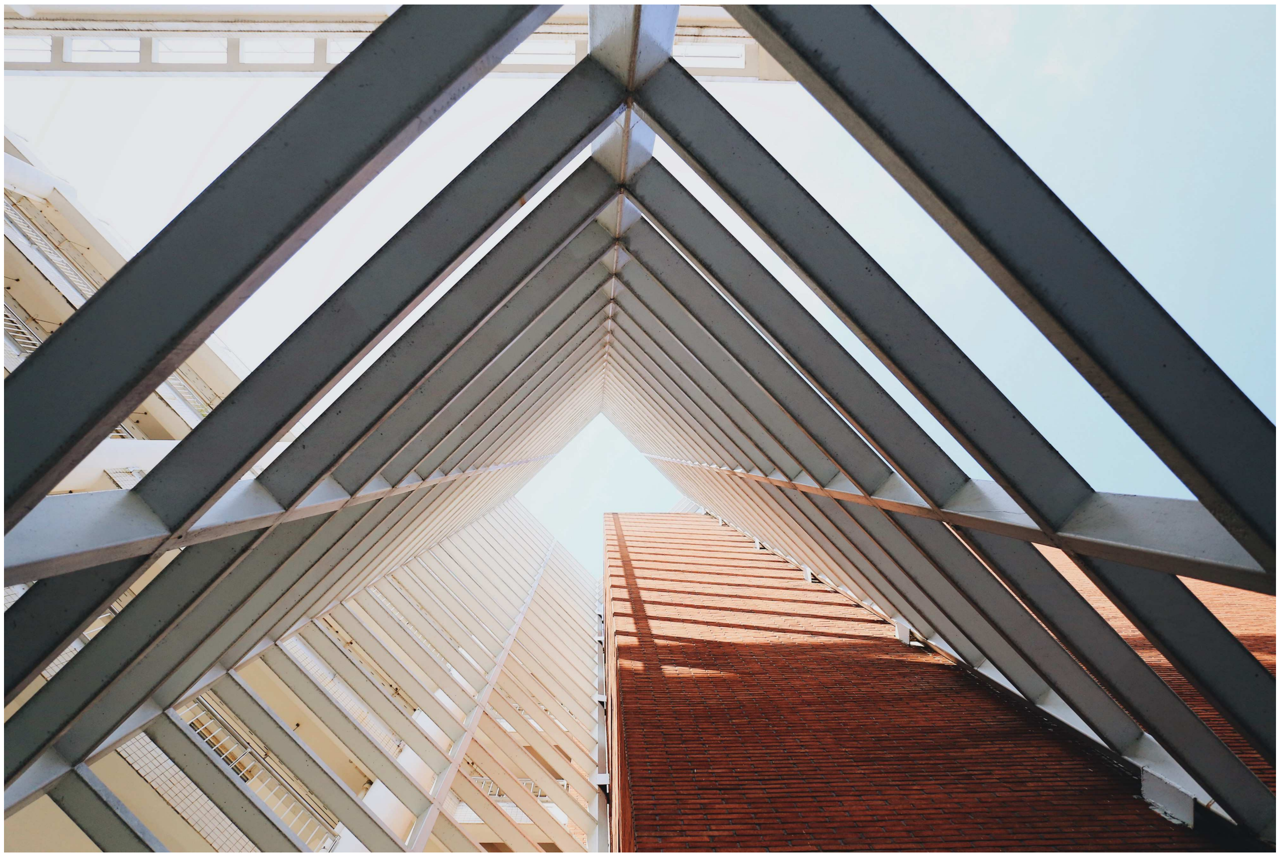


Photo by Dylan Lu on Unsplash

My interest in Artificial Intelligence and in particular in Natural Language Processing (NLP) has sparked exactly when I have learned that machines are capable of generating the new text by using some simple statistical and probabilistic techniques. In this article I wanted to share this extremely simple and intuitive method of creating Text Generation Model.

Background

As we all know, 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 (or in case if stumble across a word which meaning is unknown). Consider example below:

> *“Mary was scared because of the terrifying noise emitted by Chupacabra.”*

Without some previous context we have no idea what “Chupacabra” indeed is, but we probably can say that we would be not happy to encounter this creature in real life.

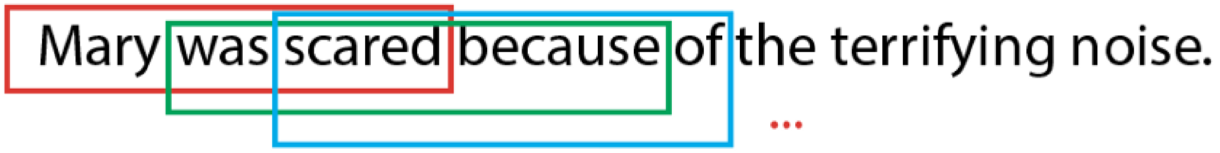
This dependency of words inside the sentence can give us some clues about the nature of the missing word and sometimes we do not even need to know the whole context. In the example above, by looking only at “*noise emitted by*” we on the intuitive level can say that the following word should be a **noun** and not some other part of speech.

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: “I went running”*
>
> *Unigrams: [(I), (went), (running)]*
>
> *Bigrams: [(I, went), (went, running)]*

As you can notice word “went” appeared in 2 bigrams: (I, **went**) and (**went**, running).

The other, more visual, way to look at it



Example of Trigrams in a sentence. Image by Oleg Borisov.

Theory

The main idea of generating text using N-Grams is to assume that the **last word** (x^{n}) of the n-gram can be inferred from the other words that appear in the same n-gram (x^{n-1}, x^{n-2}, … x^1), which I call **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:

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

Great! And the most beautiful thing is that in order to calculate the probability above we just need to apply simple conditional probability rule

$$P(x^{t+1}|x^t,...,x^{t-n+2}) = \frac{P(x^{t+1},x^t,...,x^{t-n+2})}{P(x^t,...,x^{t-n+2})}$$

For example: using trigram model (n=3)

> *Text: “Mary was scared because of ___”*
>
> *Since, we use a trigram model we drop the beginning of the sentence: “Mary was scared”, and we would need to only calculate only the possible continuation from “because of”. Assume from the dataset that we know that there are following possible continuations: “me”, “noise”. Therefore, we would need to compute:*
>
> *P(noise | because of) and P(me| because of)*

After the probabilities are computed, there exist multiple ways of selecting the final word given all candidates. **One** way would be to produce the word which had the highest conditional probability (this option might be not the best one as it tends to get stuck in a loop if *n* is small).

The **other** (and better) option would be to output the word ‘*semi-randomly*’ with regards to its conditional probability. So that the words that have a higher probability will have higher chances of being produced, while still other words with lower probability have a chance of being generated.

Photo by Clément H on Unsplash

Code

Enough of a theory, let's get to the implementation part. Source code is available on my github.

First of all, we need some source text, from which we are going to train our *Language Model*. Ideally we would like to have some large book (or even multiple books), because we not only want to have large vocabulary but also we are interested to see as many different permutations or words as possible. Here I will use Frankenstein book written by Mary Shelley, but you can use any other book of your choice (for convenience you can use Project Gutenberg link).

I want to keep the code as simple as possible, so that there is no need to install any external package. In addition to that I have also made some code optimization in order to reduce computation time and to make efficient inference procedure.

Before we implement the N-gram language model let’s implement some helper functions: one to perform tokenization (splitting words of a sentence), another one to merge consequent tokens into the N-Grams.

For tokenization it would be better to use some external library like NLTK or spaCy, but for our purposes custom tokenizer would be sufficient.

Now let’s take a look at the `get_ngrams` function.

```python
def tokenize(text: str) -> List[str]:
    """
    :param text: Takes input sentence
    :return: tokenized sentence
    """
    for punct in string.punctuation:
        text = text.replace(punct, ' '+punct+' ')
    t = text.split()
    return t

def get_ngrams(n: int, tokens: list) -> list:
    """
    :param n: n-gram size
    :param tokens: tokenized sentence
    :return: list of ngrams

    ngrams of tuple form: ((previous wordS!), target word)
    """
    tokens = (n-1)*['<START>']+tokens
    l = [(tuple([tokens[i-p-1] for p in reversed(range(n-1))]), tokens[i]) for i in range(n-
    return l
```

NGram_helper_functions.py hosted with ❤ by GitHub view raw

As we mentioned before, to predict token at position *n*, we would have to look at all previous *n-1* tokens of our N-gram. So the way we are going to represent our N-gram is a tuple of type `((context), current_word)`:

$$((x_1, x_2, \ldots, x_{n-1}), x_n)$$

The problem arises when we need to generate one of the first token(-s) of a sentence, as there is no preceding context available. To address this issue, we can introduce some leading tags like `<START>` which will be making sure that at any point of inference we always working with correct N-Grams. For example if we want to get all 3-grams from the text "*I bought a red car*":

> *[((‘<START>’, ‘<START>’), ‘I’),*
>
> *((‘<START>’, ‘I’), ‘bought’),*
>
> *((‘I’, ‘bought’), ‘a’),*
>
> *((‘bought’, ‘a’), ‘red’),*
>
> *((‘a’, ‘red’), ‘car’),*

Great, now let's create our N-gram model:

In the initialization we must specify, what is our *n* value for n-grams, in addition to that I also create *context* and *ngram_counter* dictionaries. **Context** dictionary as keys it has context, and as values stores the list of possible continuations given a context. For example:

```
>> self.context
>> {(red, car): [names, wallpapers, game, paint ...],
    (weather, in): [London, Bern, Paris, ...]}
```

So in a way this *context* dictionary immediately shows us what candidate words we can use in order to generate relatively random text that is going to make some sense (still better than blindly producing bunch of words).

Ngram_counter dictionary just counts how many times we have seen a particular N-gram in our training set before.

To update our Language Model we will supply each individual sentence from the book, and will update the dictionaries with information corresponding to our N-grams.

As was mentioned in theoretical part of this article, to compute probability of the next word given the context, we just need to apply simple conditional probability rule.

Now let’s move to the text generation part.

Before we can start text generation, we first of all must provide our system with some **context** which in our case will be starting token `<START>` repeated *n–1* times.

After that we can generate our first `random_token` by using our “semi-random” approach. Then we repeat our procedure until a certain number of tokens has been generated (specified by `token_count` variable).

Note: when we reached the full stop `.` in generation, the system would be not aware of how to proceed, because of the way how we were updating our Language Model. Therefore, we would need to reinitialize the contextual queue `context_queue` every time our model generates a full stop.

Great! Everything is set up, so let’s run our model and see what sentences we can produce!

Some of the results are presented below:

> *n=1, “care I of . its destined , in , from . . for the Felix an not which measure excited”*

> *n=2, “I compassionated him , how heavily ; but I had arrived , but they averred , my unhallowed damps and”*
>
> *n=3, “I continued their single offspring . At length I gathered from a man who , born in freedom , spurned”*
>
> *n=4, “I continued walking in this manner , during which I enjoyed the feeling of happiness . Still thou canst listen”*
>
> *n=5, “I continued walking in this manner for some time , and I feared the effects of the dæmon’s disappointment .”*

It is quite cool to see how the generated text improves as we increase *n*, however this is not surprising, because with the increase the size of the N-gram model, we start copying the original text from the training set, book in our case.

Indeed, for *n>5* generated text remains the same as “I continued walking in this manner for some time , endeavouring by bodily exercise to ease the load that weighed”.

Final remarks

What I still find quite impressing is that our over-simplified Language Model which uses N-grams and simple probability rules is still capable of producing some new text that can make some sense for a reader.

Of course such naive model is not going to be able to write some human-like articles or perform like GPT-3 as it has many drawbacks. Such system can get stuck in a loop, or is not capable of keeping track of longterm contextual relationships. But it is a nice first possible introduction to Natural Language Processing and to Language Modelling as it emphasise on the importance of the context in a sentence.

Stay tuned for more articles about Artificial Intelligence and Natural Language Processing.

Naturallanguageprocessing   Artificial Intelligence   Language Modeling   Ngrams