Indian Institute of Technology, Gandhinagar

Statistical Language Modeling Using N-grams

MA 202 PROJECT REPORT
1 APRIL, 2022

TEAM MEMBERS

Anuj Buch 20110019
Dhairya Shah 20110054
Dwip Dalal 20110063
Jinay Dagli 20110084
Madhav Kanda 20110104
Patel Kush 20110131
Pushpendra Pratap Singh 20110151
Ruchit Chudasama 20110172
Sahil Agrawal 20110178
Shruhrid Banthia 20110198
Yash Adhiya 20110235

Daniel Giftson 20110051
Dhyeykumar Thummar 20110059
Haikoo Khandor 20110071
Ksheer Agarwal 20110098
Mumuksh Tayal 20110116
Patel Vrajesh 2011134
R Yeeshu Dhurandhar 20110152
Saatvik Rao 20110175
Sanskar Sharma 20110185
Sutariya Mihirkumar 20110208

Under the guidance of
PROF. MAYANK SINGH

Contents

Abstract

Using Natural Language Processing, the model predicts the most probable next word and outputs the correctness of an input English sentence. To achieve the optimum accuracy, a large reliable dataset or corpus is extracted from Wikipedia, preprocessed, and then analyzed before using it to train the model. Analyzing the dataset and its visualization can be an insightful technique to understand the corpus before using it for the model's training. Choosing an appropriate model for any problem is a crucial step. In our case, using a trigram model to train the data proved to be the best trade-off. This trained model is finally used in the code to predict the next word and find the perplexity of a given sentence based on the trigram model.

Problem Statement

Computers were once thought of as “dumb terminals,” and human interactions were based on the principle of “garbage in, garbage out.” Computers could only communicate in sophisticated hand-coded rules. Natural Language Processing bridges the gap between humans and computers by enabling humans to interact with computers in human-developed languages. It can have various use-cases such as voice assistants, speech recognition, computer-assisted coding, and word & sentence prediction. The boundless possibilities in NLP, yet to be explored, motivate us to work in this field.

Data Pre-Processing and Visualisation

Team Members: Anuj Buch | Kush Patel | Yash Adhiya

Our task was to load the wiki corpus dataset in the Google Colab environment and do some data preprocessing before making the model. Downloading the Wikipedia data and then loading the data into the Colab was not a great choice since the data was of considerable size (nearly 18 GB). So, we decided to figure out a way of directly importing the data to them Colab to run the model.

Steps:

1. `pip install apache_beam mwparserfromhell`
   → We install the apache_beam_mwparserfromhell Python package to have an easy and powerful parser for MediaWiki wiki code.
2. `pip install datasets`
   → Then, we install the datasets to use present public data for preprocessing or running various ML models. Dataset package contains the load_dataset library to load any Wikipedia data by one line shown below:
   `load_dataset`("wikipedia," "20200501.en")
3. After that, we read the data using the Pandas package and analyzed some Exploratory data.

Frequency of a word in data: The frequency of words is an essential piece of information for natural language processing and further analysis. It helps us navigate the preprocessing steps and indicates the value and informativeness of the parts.

After loading the dataset, data visualizations are incredibly insightful techniques. To analyze the text statistics from datasets, we have used histograms. This is not just text analysis, but also it contains word frequency analysis, sentence length analysis, and average word length analysis. Our histogram is a little left-skewed in word length analysis, but it does not mean that people generally use short words. One reason why this may not be true is stopwords. Stopwords are the words that are most commonly used in any language, such as “the”, “a”, “an”, etc. As these words are probably small in length, these words may have caused the above graph to be left-skewed. Another analysis is also done in a very efficient manner.

N-grams are simply a contiguous sequence of n words. For example, “Positive,” “Million dollars,” etc. If the number of words is two, it is called bigram. For three words, it is called a trigram and so on. Looking at the most frequent N-grams can better understand the word's context.

Only analysis of data is not essential. If there are several synonym terms present, the system might take more time to load these data, but in fact, it is not needed because they have the same meaning, so some data cleaning is required. A combination of these words is necessary for this data cleaning. Sentiment analysis is a prevalent natural language processing task in which we determine if the text is positive, negative, or neutral. This is very useful for finding the sentiment associated with reviews and comments, which can get us some valuable insights from text data.

Formation of a word cloud using the Wikipedia Dataset

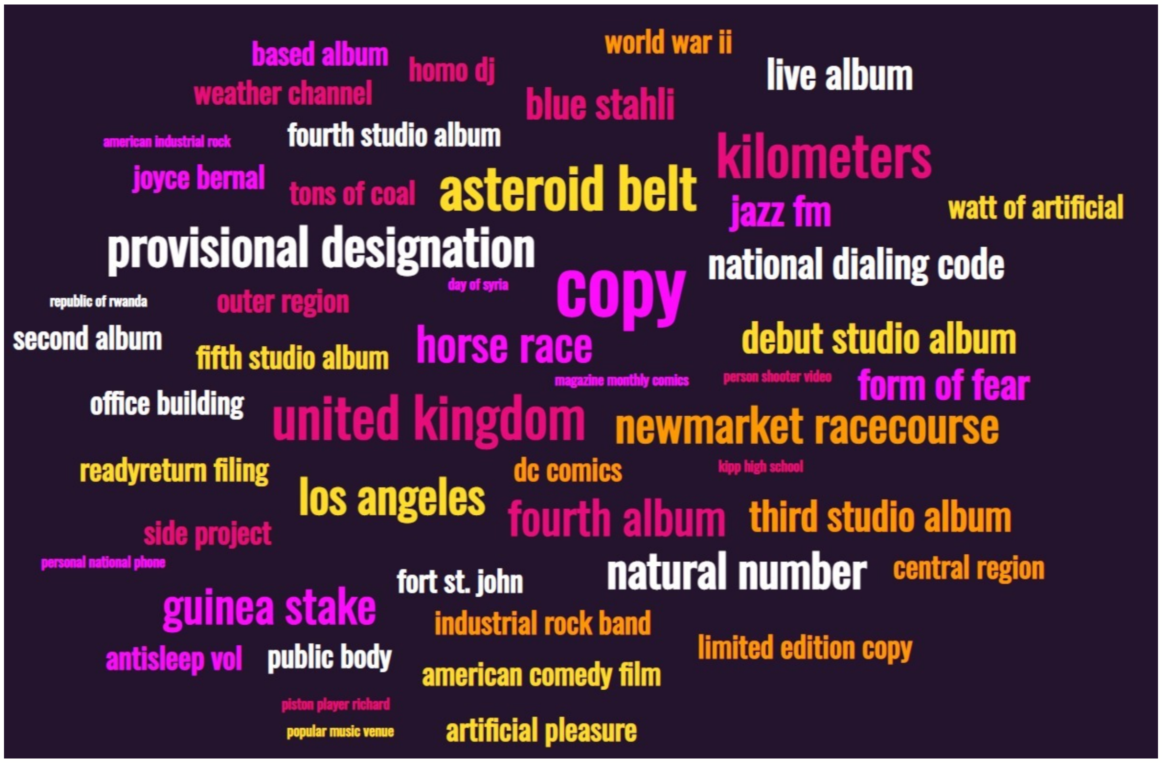


Fig1.1: Figure representation of a Word Cloud based on frequencies

Laplace Smoothing

Team Members: R Yeeshu Dhurandhar | Haikoo Khandor | Dhyeykumar Thummar | Mihir Sutariya | Daniel Giftson | Saatvik Rao

After building the model to predict the nth word, we encountered a problem. Since the dataset used to train the model is limited, all possible combinations, i.e., all n-grams, are not present in the training data. This presents a fundamental problem.

Let there be three words w1,w2, and w3. Let's suppose w1 and w2 are associated with each other, unlike w2 and w3 in the training data. Thus when the model predicts words, the probability for w2 and w3 to be predicted will be zero, but that is not true since it is sometimes possible for this combination to occur, even though its probability is very low but not zero. This problem also extends more when a big phrase consists of w2 and w3. The probability of that phrase would become zero then. So what options do we have at this point?

Option 1: Put the probability of w2 and w3 occurring together to be zero. This seems logical since, indeed, its probability is zero. But as discussed above, this is not a good option since it also assigns the probability zero to many possibilities.

Option 2: Ignore this instance. This means assigning it a probability of one, which seems highly illogical.

Hence we apply Laplace Smoothing to the training data so that the model predicts better.
We take the probabilities from the events which occurred one or more times and give them to N-grams whose probabilities were originally zeros. This is done by assigning a frequency of one to these N-grams.

<u>Laplace Smoothing Applied to the Bigram:</u>

<u>Add-one method or Laplace Smoothing:</u>

$$P^*(w_n|w_{n-1}) = \frac{c(w_{n-1}w_n)+1}{c(w_{n-1})+V}$$

<u>The generalization of Add-one smoothing is Add-k smoothing where k is a parameter:</u>

$$P^*_{Add-k}(w_i|w_{i-1}) = \frac{c(w_i,w_{i-1})+k}{c(w_{i-1})+kV}$$

P* = Probability of the Laplace Smoothed N-grams

Wi = i'th word

$c(w_i, w_{i-1})$ = count of word sequence Wi-1,Wi

V = total number of words in vocabulary

<u>Advantages of Laplace Smoothing:</u>

1. It helps to encounter the problem of zero probability instances.
2. This improves the model significantly compared to the original model.

<u>Disadvantages of Laplace Smoothing:</u>

1. It takes a lot of weight from the occurring instances and assigns it to the ones with zero probability. For some instances, there may be reporting of imprecise probabilities.

<u>Example of change of probabilities of next word before and after smoothing:</u>

Input word: company is

Words	**Probability without Laplace Smoothing**	**With LaplaceSmoothing**
'not'	0.0707070707070707	0.0003389256058295204
'a'	0.06060606060606061	0.0002965599051008304
'currently'	0.050505050505050504	0.0002541942043721403
'now'	0.04040404040404041	0.00021182850364345027
'making'	0.020202020202020204	0.00012709710218607016
'registered'	0.010101010101010102	8.47314014573801e-05
'comfortable'	0.010101010101010102	8.47314014573801e-05

'awarded'	0.010101010101010102	8.47314014573801e-05
'stronger'	0.010101010101010102	8.47314014573801e-05

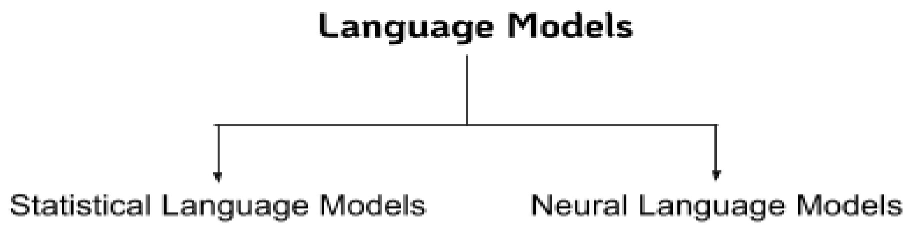
Inference: The words which come one or more times have significant probabilities. After smoothing is done, their probabilities have decreased, as observed in the above table. Before smoothing, some weights from their probabilities have been given to those with zero probability.

Predicting Next Word Using N-grams and Markov Chain

Team Members: Ruchit Chudasama | Madhav Kanda | Ksheer Agarwal | Patel Vrajesh| Pushpendra Pratap Singh | Jinay Dagli

The task of our sub-group was to code and implemented the next word prediction. We were supposed to determine the next word based on the frequency of that word after the previous words. For the same, we have tried and used the n-grams approach based on the concept of the Markov chain.

Before starting with the ideas of n-grams and the Markov Chain, we briefly describe language models. A language model is used to predict the probability of a sequence of words based on statistics.
There are two types of language models: Statistical language models and neural language models.



Essential terms to note

- N-grams approach/model: An n-gram is a sequence of N words. They are a continuous sequence of words or characters. For example, "Next Word" is a bi-gram (2-gram), "Next Word Prediction" is a tri-gram (3-gram), etc.

Uni-gram:

Where	The	Mind	Is	Without	Fear

Bi-gram:

Where The	Mind Is	Without Fear

Tri-gram:

Where The Mind	Is Without Fear

For the project purpose, we will be using the trigram model.

- Markov Chain: Markov chain is a process that models a finite set of states and assigns conditional probabilities of going from one state to another. These probabilities of jumping from one state to another are generally expressed in the form of a matrix called the transition matrix.

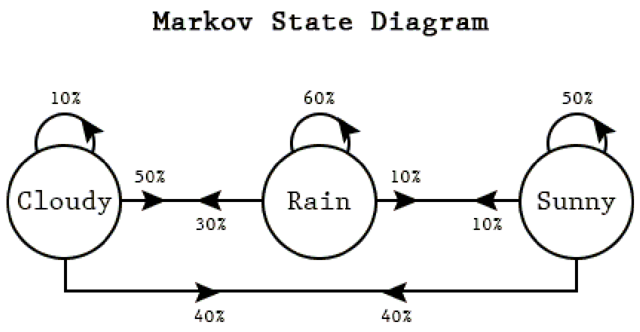


Fig1.2 Figure representation of Markov Chain

The above-shown state diagram is an example of a Markov chain, where weather prediction on the following day is being predicted. For example, let us say we have rain on a particular day. Then the probability that it would be sunny the next day is 0.1.
Similarly, the probability that it would rain the next day is 0.6. The critical point to note here is that the probability of any event on a given day is dependent only on the immediate previous day.

Detailed descriptions and mathematical definitions:

N-gram model:
The task of predicting the next word using an n-gram model is equivalent to the probability of getting a specific word at the nth place given the previous n-1 words. The probability of the same is given by the Bayes formula,

$$P(w \ |w_1,.....,w \ _{-1})$$

Here, we use previous words(the history) to predict the next word. The number of parameters contributing to our next word prediction depends on the value of n as

$$Number\ of\ parameters\ =\ (Number\ of\ distinct\ words)^n *\ (Number\ of\ distinct\ words\ -\ 1$$

We need to choose wisely the n. Too high a value of n will lead to too many parameters leading to high training time. However, we cannot simply make our decision based on parameters since we need a decent prediction of our next word.

Considering this tradeoff, the most accepted model in the present situation is either bigram(n=2) or a trigram(n=3) model. Hence, we pursued our task by implementing a trigram model.

Four-gram model is also used, but it is generally implemented along with stemming. In stemming, we remove the inflectional endings. For example, we have words clustering, clustered, cluster, and so on in our vocabulary. In this case, all these words are grouped under the subgroup of the base words. The base word here is the cluster. However, we have not applied this method in our model.

N-gram model with Markov chain:

The n-gram model could go as complex as possible. To reduce the complexity of the n-gram models, we use the Markov property. The Markov property as mentioned earlier, states that the probability of the future state depends only on the present state and not on the events that preceded it. To implement the Markov property in N-grams, we use a Markov chain to store the probability of transitioning from one state to another. It can be explained better using an example:
Suppose that the input corpus is the following:
data = "I am Sam. Sam, I am. I do not like green eggs and ham."

To build a trigram model for the data, we first have to list all the possibilities:
->I am Sam　*->am Sam. Sam*　*->Sam. Sam I*　*->Sam I am*
-> I am. I　*->am. I do*　*-> I do not*　*-> do not like*
->not like green　*->like green eggs*　*->green eggs and*
->eggs and ham.　*->and ham.eggs.*
Transitions from **I** :
The transitions from ***I*** for our trigram model using the Markov chain can be shown in the state machine:

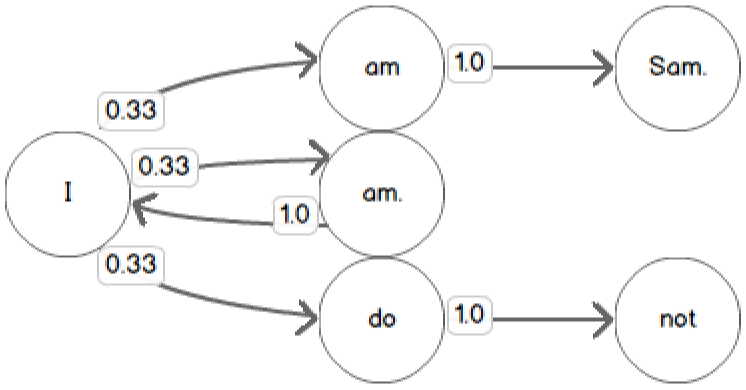
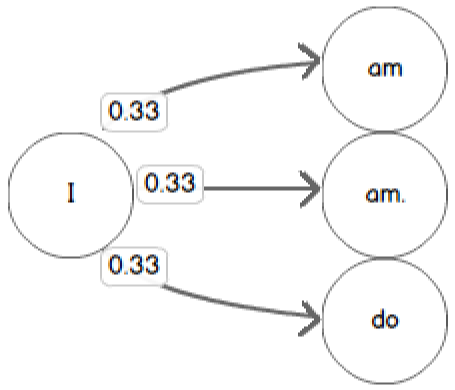


Fig1.4 Figure representation of N-gram model using Markov chain

The numbers/weights represent the probability of transition. Since ***I*** can transition to only three states, i.e., *am, am.,* and *do*, the probability of each transition is 0.33.



Similarly, there is only one possible state given I am, I am, and I do. Hence, the probability of obtaining *Sam, I,* and *not*, as the next word, is 1, respectively.

Calculating probabilities:

Since calculating the probability for the trigram model by hand can be complex, we will use the bigram model to demonstrate the calculations. It can be easily extended to a trigram using a computer.

Corpus for this case:

<s> I am Sam </s>

<s> Sam I am </s>

<s> I do not like green eggs and ham </s>

Bi-grams:

[<s> I, I am, am Sam, Sam </s>, <s> Sam, Sam I, I am, am </s>, <s> I, I do, do not, not like, like green, green eggs, eggs and, and ham, ham </s>]

Probability:

$$P(w_{1:n}) \approx \prod_{k=1}^{N} P(w_k|w_{k-1})$$

Hence, from the bi-grams we observe that:

P(*I* | <s>) = ⅔ = 0.66

P(*Sam* | <s>) = 1/3 = 0.33

P(*am* | *I*) = ⅔ = 0.33

P(*</s>* | *Sam*) = ½ =0.5

P(*<s>* | *Sam*) = ½ = 0.5

P(*Sam* | *am*) = ½ = 0.5

P(*do* | *I*) = ⅓ = 0.33

Tri-grams:
[<s> I am, I am Sam, am Sam </s>, <s> Sam I, Sam I am, I am </s>, <s> I do, I do not, do not like, not like green, like green eggs, green eggs and, eggs and ham, and ham </s>]

Probability:

$$P(w_1^N) = \prod_{i=1}^{N} P(w_n | w_{n-2}, w_{n-1})$$

Hence, from the tri-grams we observe that:
P(*Sam* | *I am*) = ½ = 0.5

P(*not*| *I do*) = 1

P(*</s>* | *and ham*) = 1

P(*do* | *<s> I*) = ½ = 0.5

Perplexity

Team Members: Dwip Dalal | Dhairya Shah | Shruhrid Banthia | Sanskar Sharma | Mumuksh Tayal | Sahil Agarwal

Introduction

We assigned our subgroup to determine the perplexity of a sentence to be given as input. The first subgroup downloaded the English text data from Wikipedia, also known as "Corpus." The second subgroup did the data smoothing so that important patterns could more clearly stand out. The two main tasks assigned to us are "The next word predictor" and finding out the "perplexity of a given sentence."

What exactly is Perplexity?

Assume we have a model that takes an English sentence as input and outputs a probability score indicating how likely it is to be a legitimate English sentence. We're trying to figure out how good this model is. A good model should give valid English sentences a high score and invalid English sentences a low score. Perplexity is a commonly used metric for determining how "excellent" a model is. The perplexity is given by the following relation when the sentence contains n-words:

$$\mathrm{PP(s)} = p(w1,\ldots\ldots, wn)^{-\frac{1}{n}}$$

Perplexity is a statistic for evaluating language models. But why would we want to utilize it in the first place? Why can't we just look at our final system's loss/accuracy on the job that matters?

To answer this question, let us first look at the two approaches to evaluating and comparing language models:

1) Extrinsic evaluation: This entails putting the models to the test in a real-world situation (such as machine translation) and calculating the resulting results loss/accuracy.
2) Intrinsic evaluation entails devising a metric for evaluating the language model rather than the individual activities for which it will be employed. While intrinsic evaluation as a final metric isn't as "excellent" as extrinsic evaluation, it's a quick approach to comparing models. Perplexity is a built-in grading system.

How did we find out the Perplexity of Corpus?

The process of first finding the perplexity of a sentence
First, we figured out if we could find the perplexity of the Corpus. If we want to know the perplexity of the whole corpus C that contains m sentences and N-words, we have to determine how well the model can predict all the sentences together. This can be done after discovering the individual perplexities of every sentence that the Corpus is made up for.

The perplexity of the corpus, per word, is given by:

$$Perplexity(C) = \frac{1}{P(s_1, s_2, ..., s_N)^{\frac{1}{N}}} = \sqrt[N]{\frac{1}{P(s_1, s_2, ... , s_N)}}$$

If the probabilities are considered independent of one another, then the cumulative chance of these phrases being in C is given by:

$$P(s_1, ... , s_m) = \prod_{i=1}^{m} p(s_i)$$

$$Perplexity(C) = \sqrt[N]{\frac{1}{\prod_{i=1}^{m} p(s_i)}}$$

This equation provides us with the perplexity of the corpus to the number of words after replacing the multiplications with additions using logarithmic identities.

Modelling Probability Distribution P (Building the model)
A statistical model that assigns a probability to words and sentences is a language model. In most cases, we're attempting to estimate the next word w in a sentence based on the preceding words, referred to as the "history." We're often curious about the probability that our model gives to a whole sentence W made up of the words ((*w_1,w_2,...,w_N*).

So, we can determine the aforementioned conditional probabilities given some data (called train data). However, this is not feasible since it would need a large amount of training data. Then we conduct a calculation based on our assumptions to find:

$$P(W) = P(w_1, w_2, ... , w_N)$$

Only individual words are considered in a unigram model. A unigram model would output the probability given a sequence of words W. Individual probabilities P(w i) might be computed, for example, using the frequency of words in the training corpus.

$$P(w_1, w_2, \ldots , w_N) = P(w_1) * P(w_2) *\ldots* P(w_N)$$

Bigram-First order Markov Assumption: Next word depends only on the previous words.

$$P(w_1, w_2, \ldots , w_N) = P(w_1) * P(w_2|w_1) *\ldots* P(w_N|w_{N-1})$$

An n-gram model, on the other hand, uses the previous (n-1) words to predict the next. We can expand it using the chain rule of probability.

$$P(w_1, w_2, \ldots , w_N) = P(w_1) * P(w_2|w_1) * P(w_3|w_1, w_2) *\ldots* P(w_k|w_1, \ldots, w_{k-1})$$

Perplexity as the normal inversed probability

Probability of test set:

First and foremost, what constitutes a good language model? We want our model to give actual and syntactically accurate sentences with high probability and false, erroneous, or uncommon sentences with low probabilities. Intuitively, if a model gives a high probability to the test set, it indicates it is not startled to find it, implying that it understands how the language works.

Normalizing:

The datasets can contain various phrases, and sentences can have a variety of terms. Adding more phrases adds more uncertainty. Therefore, a more significant test set is likely to have a lower probability than a smaller one, all other things being equal. We would have a statistic unaffected by dataset size in an ideal world. We can get this by dividing the test set's likelihood by the total number of words, which gives us a per-word measure.
We could divide the sum of some terms by the number of words to generate a per-word measure if what we intended to normalize was the total of specific terms. On the other hand, a product determines the probability of a series of words. What is the best way to normalize this probability? It is simpler if you look at the log probability, which converts the product into a sum.

For example, let's take a unigram model:

$$P(W) = P(w_1, w_2, \ldots , w_N) = P(w_1) * P(w_2) *\ldots* P(w_N) = \prod_{i=1}^{N} P(w_i)$$

$$ln(P(W)) = ln(\prod_{i=1}^{N} P(w_i)) = \sum_{i=1}^{N} ln(P(w_i))$$

We can now normalize this by dividing by N to obtain the per-word log probability:

$$\frac{ln(P(W))}{N} = \frac{\sum_{i=1}^{N} ln(P(w_i))}{N}$$

Raising both LHS and RHS to the power of e to remove the logarithmic terms:

$$e^{\frac{ln(P(W))}{N}} = e^{\frac{\sum_{i=1}^{N} ln(P(w_i))}{N}}$$

$$(e^{ln(P(W))})^{\frac{1}{N}} = (e^{\sum_{i=1}^{N} ln(P(w_i))})^{\frac{1}{N}}$$

$$P(W)^{\frac{1}{N}} = (\prod_{i=1}^{N} P(w_i))^{\frac{1}{N}}$$

We can see that we have obtained normalization by taking the N-th root.

Returning to our initial perplexity equation, we can deduce that it is the inverse probability of the test set, normalized by the number of words in the test set. Because we use the inverse probability, the smaller the perplexity, the better the model. We may even compute the perplexity of a single sentence if we desire, in which case W would be that single sentence.

$$PP(W) = \frac{1}{P(w_1, w_2, \ldots\ldots, w_N)^{\frac{1}{N}}} = \sqrt[N]{\frac{1}{P(w_1, w_2, \ldots. , w_N)}}$$

Interface on Huggingface and Challenges faced

Team Member: Shruhrid Banthia

Instructions

1) Please do not use space after the last word.
2) The entire model is probabilistically structured; hence be careful about the case sensitivity of the letters. It is advisable to use a Capital letter as the first letter of the sentence.

Challenges faced

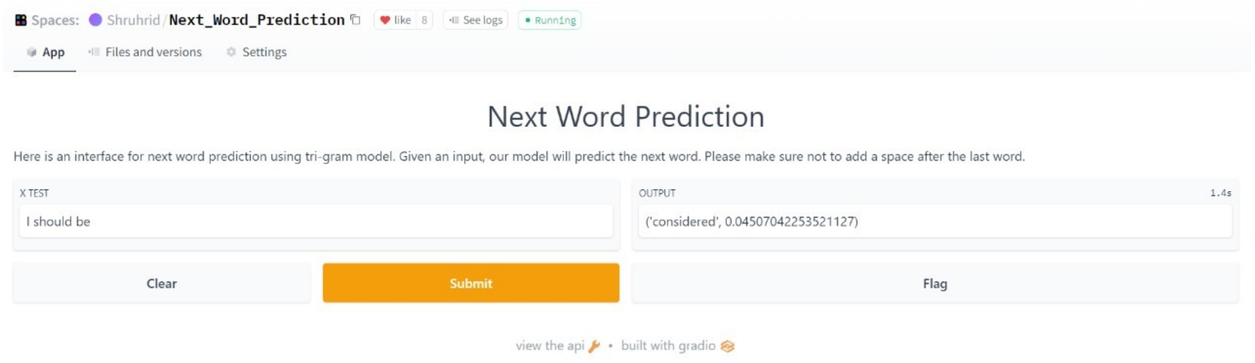
1. In the app.py file we had to import the class of the model and thus had to upload this along with the app.py.

2. We initially uploaded our original model to Hugging Face, but its size was not supported by the server of Hugging Face (1.01 GB). We faced pickle unpickling errors. We then uploaded another model (600MB) trained on a smaller dataset supported by the server.

3. To submit the large files we had to use git-LFS. This was a new challenge that we encountered.

Huggingface model

https://huggingface.co/spaces/Shruhrid/Next_Word_Prediction


Spaces: Shruhrid / Next_Word_Prediction
like 8
See logs
Running
App
Files and versions
Settings
Clear
Submit
Flag
view the api • built with gradio

Next Word Prediction

Here is an interface for next word prediction using tri-gram model. Given an input, our model will predict the next word. Please make sure not to add a space after the last word.

X TEST: I should be

OUTPUT: ('considered', 0.04507042253521127) 1.4s

Acknowledgements

We, as a team, would like to extend our gratitude to all the three instructors of MA202 who gave us this opportunity to work on such an exciting topic. We would also like to thank Prof Mayank Singh for providing valuable guidance and assistance whenever needed.

Team Management

We are a team of several great friends, so we had a mutual understanding, and our project journey was smooth. We were late in meeting Prof Mayank Singh, and hence we started working on the tasks immediately after the meeting. We divided ourselves into four sub-groups according to the tasks assigned. Everyone contributed at least the bare minimum required for the project, but the following members did an exemplary job. Dwip, Shruhrid, Dhyey, Ruchit, and Mihir coded the complete model. Shruhrid Banthia also performed impressively by building the interface for presenting the model. Pushpendra and Yeeshu analyzed the code and added appropriate comments where needed. Dhairya formulated everything into a compiled document with Vrajesh and Mumuksh. Yeeshu, Pushpendra, Haikoo, Yash, Sahil, Sanskar, Ksheer, Madhav, Jinay researched and wrote the content of their respective subgroups in the document. Anuj and Saatvik designed the presentation file and Dhairya, Madhav, Mihir, Ksheer, Yeeshu did the presentation. Lastly, no team can win without good leaders, and here Dhairya and Yeeshu lead our team of 21 members and make sure that everything goes as planned.

References

[1] [CH] Christopher D. Manning and Hinrich Schütze. 1999. Foundations of Statistical Natural Language Processing. MIT Press, Cambridge, MA, USA

[2] Perplexity Intuition (and its derivation) | by Aerin Kim | Towards Data Science

[3] How can I calculate perplexity using nltk - Stack Overflow

[4] Perplexity in Language Models. Evaluating language models using the… | by Chiara Campagnola | Towards Data Science

[5] How to find the perplexity of a corpus - Cross Validated

[6] AttributeError: Can't get attribute on <module '__main__' from 'manage.py'> - Stack Overflow

[7] Dataset: https://www.kaggle.com/datasets/mikeortman/wikipedia-sentences