Building Language Models in NLP

Koushiki Dasgupta Chaudhuri — Published On January 3, 2022

Beginner Model Deployment NLP Project

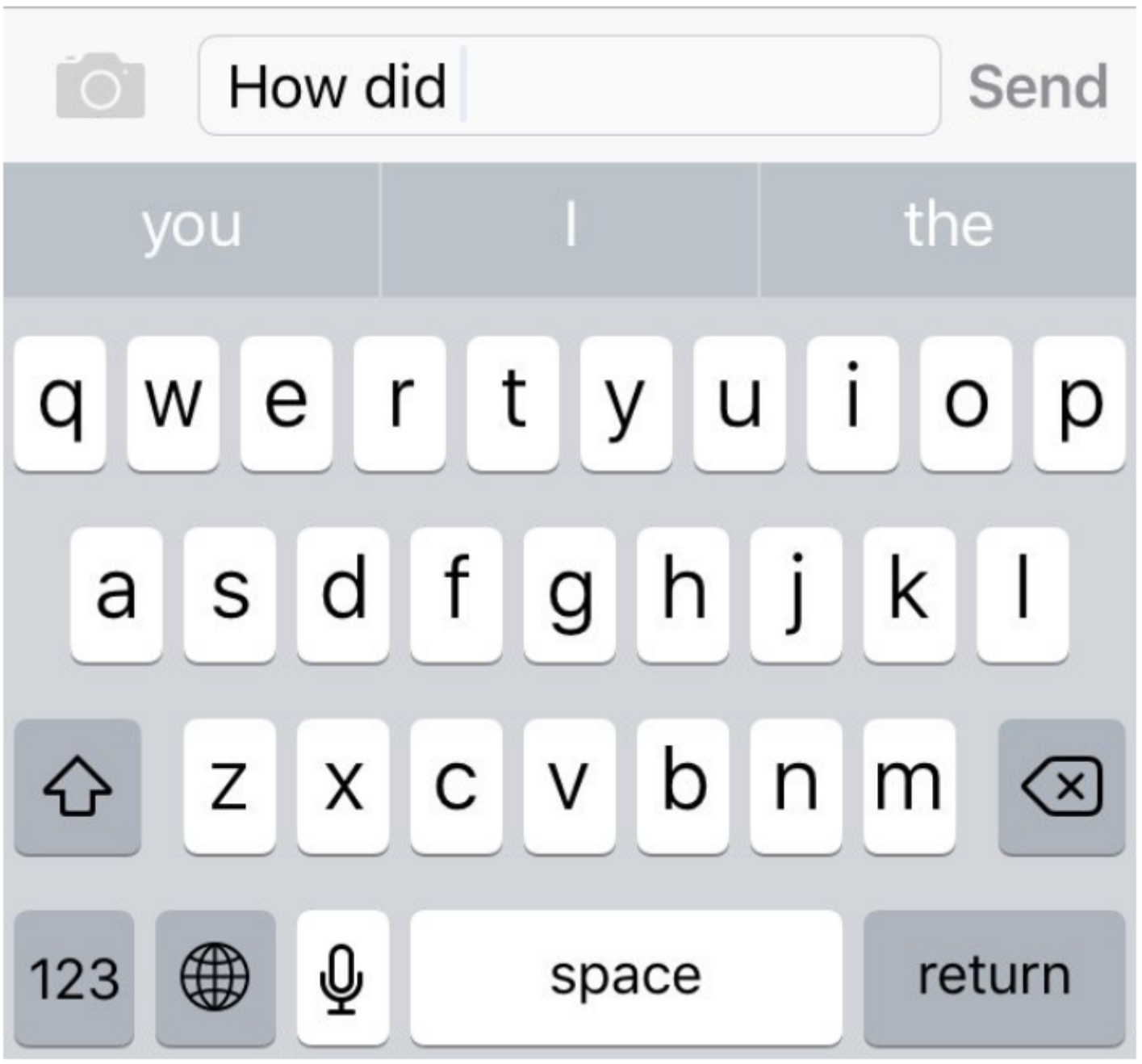
This article was published as a part of the Data Science Blogathon.

Introduction

A language model in NLP is a probabilistic statistical model that determines the probability of a given sequence of words occurring in a sentence based on the previous words. It helps to predict which word is more likely to appear next in the sentence. Hence it is widely used in predictive text input systems, speech recognition, machine translation, spelling correction etc. The input to a language model is usually a training set of example sentences. The output is a probability distribution over sequences of words. We can use the last one word (unigram), last two words (bigram), last three words (trigram) or last n words (n-gram) to predict the next word as per our requirements.

Why Language Models?

Language models form the backbone of Natural Language Processing. They are a way of transforming qualitative information about text into quantitative information that machines can understand. They have applications in a wide range of industries like tech, finance, healthcare, military etc. All of us encounter language models daily, be it the predictive text input on our mobile phones or a simple Google search. Hence language models form an integral part of any natural language processing application.



In this article, we will be learning how to build unigram, bigram and trigram language models on a raw text corpus and perform next word prediction using them.

Reading the Raw Text Corpus

We will begin by reading the text corpus which is an excerpt from Oliver Twist. You c
is downloaded, read the text file and find the total number of characters in it.

```
file = open("rawCorpus.txt", "r")
rawReadCorpus = file.read()
print ("Total no. of characters in read dataset: {}".format(len(rawReadCorpus)))
```

We need to import the nltk library to perform some basic text processing tasks which we will do with the help of the following code :

```
import nltk
nltk.download()
from nltk.tokenize import word_tokenize,sent_tokenize
```

Preprocessing the Raw Text

Firstly, we need to remove all new lines and special characters from the text corpus. We do that by the following code :

```
import string
string.punctuation = string.punctuation +'“'+'”'+'-'+'’'+'‘'+'—'
string.punctuation = string.punctuation.replace('.', '')
file = open('rawCorpus.txt').read()
#preprocess data to remove newlines and special characters
file_new = ""
for line in file:
    line_new = line.replace("n", " ")
    file_new += line_new
preprocessedCorpus = "".join([char for char in file_new if char not in string.punctuation])
```

After removing newlines and special characters, we can break up the corpus to obtain the words and the sentences using sent_tokenize and word_tokenize from nltk.tokenize. Let us print the first 5 sentences and the first 5 words obtained from the corpus :

```
sentences = sent_tokenize(preprocessedCorpus)
print("1st 5 sentences of preprocessed corpus are : ")
print(sentences[0:5])
words = word_tokenize(preprocessedCorpus)
print("1st 5 words/tokens of preprocessed corpus are : ")
print(words[0:5])
```

The output looks something like this :

```
1st 5 sentences of preprocessed corpus are :
[' CHAPTER I.', 'TREATS OF THE PLACE WHERE OLIVER TWIST WAS BORN AND OF THE  CIRCUMST
public buildings in a certain town which for many reasons  it will be prudent to ref
ssign no fictitious name there is one anciently common to most towns  great or small
as born on  a day and date which I need not trouble myself to repeat inasmuch as  it
eader in this stage of  the business at all events the item of mortality whose name
'For a long time after it was ushered into this world of sorrow and  trouble by the
iderable  doubt whether the child would survive to bear any name at all in which  ca
hese memoirs would never  have appeared or if they had that being comprised within a
the inestimable merit of being the  most concise and faithful specimen of biography
ntry.', 'Although I am not disposed to maintain that the being born in a  workhouse
e circumstance  that can possibly befall a human being I do mean to say that in this
for Oliver Twist that could  by possibility have occurred.', 'The fact is that there
liver to take upon himself the office of  respirationa troublesome practice but one
easy existence and for some time he lay gasping on a  little flock mattress rather u
next the balance being decidedly in favour of the latter.']
1st 5 words/tokens of preprocessed corpus are :
['CHAPTER', 'I', '.', 'TREATS', 'OF']
```


Source: Screenshot from my Jupyter Notebook

We also need to remove stopwords from the corpus. Stopwords are some commonly used words like ‘and’, ‘the’, ‘at’ which do not add any special meaning or significance to a sentence. A list of stopwords are available with nltk, and they can be removed from the corpus using the following code :

```
nltk.download('stopwords')
from nltk.corpus import stopwords
stop_words = set(stopwords.words('english'))
filtered_tokens = [w for w in words if not w.lower() in stop_words]
```

Creating Unigram, Bigram and Trigram Language Models

We can create n-grams using the ngrams module from nltk.util. N-grams are a sequence of n consecutive words occurring in the corpus. For example, the sentence “I love dogs” – ‘I’, ‘love’ and ‘dogs’ are unigrams while ‘I love’ and ‘love dogs’ are bigrams. ‘I love dogs’ is itself a trigram i.e. a contiguous sequence of three words. We obtain unigrams, bigrams and trigrams from the corpus using the following code :

```
from collections import Counter
from nltk.util import ngrams
unigrams=[]
bigrams=[]
trigrams=[]
for content in (sentences): # *** Write code ***
    content = content.lower()
    content = word_tokenize(content)
    for word in content:
        if (word =='.'):
            content.remove(word)
        else:
            unigrams.append(word)
    bigrams.extend(ngrams(content,2))
    ##similar for trigrams
    # *** Write code ***
    trigrams.extend(ngrams(content,3))
print ("Sample of n-grams:n" + "-------------------------")
print ("--> UNIGRAMS: n" + str(unigrams[:5]) + " ...n")
print ("--> BIGRAMS: n" + str(bigrams[:5]) + " ...n")
print ("--> TRIGRAMS: n" + str(trigrams[:5]) + " ...n")
```

The output looks like this :

```
Sample of n-grams:
------------------------
--> UNIGRAMS:
['chapter', 'i', 'treats', 'of', 'the'] ...

--> BIGRAMS:
[('chapter', 'i'), ('treats', 'of'), ('of', 'the'), ('the', 'place'), ('place', 'wh

--> TRIGRAMS:
[('treats', 'of', 'the'), ('of', 'the', 'place'), ('the', 'place', 'where'), ('plac
'twist')] ...
```


Source: Screenshot from my Jupyter notebook

Next, we obtain those unigrams, bigrams and trigrams from the corpus which do not contain stopwords or determiners in them. For example, we remove bigrams like 'in the' and we remove unigrams like 'the', 'a' etc. We use the following code for the removal of stopwords from n-grams.

```
def stopwords_removal(n, a):
    b = []
    if n == 1:
        for word in a:
            count = 0
            if word in stop_words:
                count = 0
            else:
                count = 1
            if (count==1):
                b.append(word)
        return(b)
    else:
        for pair in a:
            count = 0
            for word in pair:
                if word in stop_words:
                    count = count or 0
                else:
                    count = count or 1
            if (count==1):
                b.append(pair)
        return(b)
unigrams_Processed = stopwords_removal(1,unigrams)
bigrams_Processed = stopwords_removal(2,bigrams)
trigrams_Processed = stopwords_removal(3,trigrams)
print ("Sample of n-grams after processing:n" + "-------------------------")
print ("--> UNIGRAMS: n" + str(unigrams_Processed[:5]) + " ...n")
print ("--> BIGRAMS: n" + str(bigrams_Processed[:5]) + " ...n")
print ("--> TRIGRAMS: n" + str(trigrams_Processed[:5]) + " ...n")
```

The unigrams, bigrams and trigrams obtained in this way look like :

```
Sample of n-grams after processing:
-------------------------
--> UNIGRAMS:
['chapter', 'treats', 'place', 'oliver', 'twist'] ...

--> BIGRAMS:
[('chapter', 'i'), ('treats', 'of'), ('the', 'place'), ('place', 'where'), ('where', 'oliver')] ...

--> TRIGRAMS:
[('treats', 'of', 'the'), ('of', 'the', 'place'), ('the', 'place', 'where'), ('place', 'where', 'oliver'), ('where', 'oliver',
'twist')] ...
```

Source: Screenshot from my Jupyter Notebook

We can obtain the count or frequency of each n-gram appearing in the corpus. This will be useful later when we need to calculate the probabilities of the next possible word based on previous n-grams. We write a function get_ngrams_freqDist

```
def get_ngrams_freqDist(n, ngramList):
    ngram_freq_dict = {}
    for ngram in ngramList:
        if ngram in ngram_freq_dict:
            ngram_freq_dict[ngram] += 1
        else:
            ngram_freq_dict[ngram] = 1
    return ngram_freq_dict
unigrams_freqDist = get_ngrams_freqDist(1, unigrams)
unigrams_Processed_freqDist = get_ngrams_freqDist(1, unigrams_Processed)
bigrams_freqDist = get_ngrams_freqDist(2, bigrams)
bigrams_Processed_freqDist = get_ngrams_freqDist(2, bigrams_Processed)
trigrams_freqDist = get_ngrams_freqDist(3, trigrams)
trigrams_Processed_freqDist = get_ngrams_freqDist(3, trigrams_Processed)
```


Predicting Next Three words using Bigram and Trigram Models

The chain rule is used to compute the probability of a sentence in a language model. Let w1w2…wn be a sentence where w1, w2, wn are the individual words. Then the probability of the sentence occurring is given by the following formula :

$$P(w_1 w_2 \ldots w_n) = \prod_i P(w_i | w_1 w_2 \ldots w_{i-1})$$

For example, the probability of the sentence “I love dogs” is given by :

P(I love dogs) = P(I)P(love | I)P(dogs | I love)

Now the individual probabilities can be obtained in the following way :

P(I) = Count(‘I’) / Total no. of words

P(love | I) = Count(‘I love’) / Count(‘I’)

P(dogs | I love) = Count(‘I love dogs’) / Count(‘I love’)

Note that Count(‘I’), Count(‘I love’) and Count(‘I love dogs’) are the frequencies of the respective unigram, bigram and trigram which we computed earlier using the get_ngrams_freqDist function.

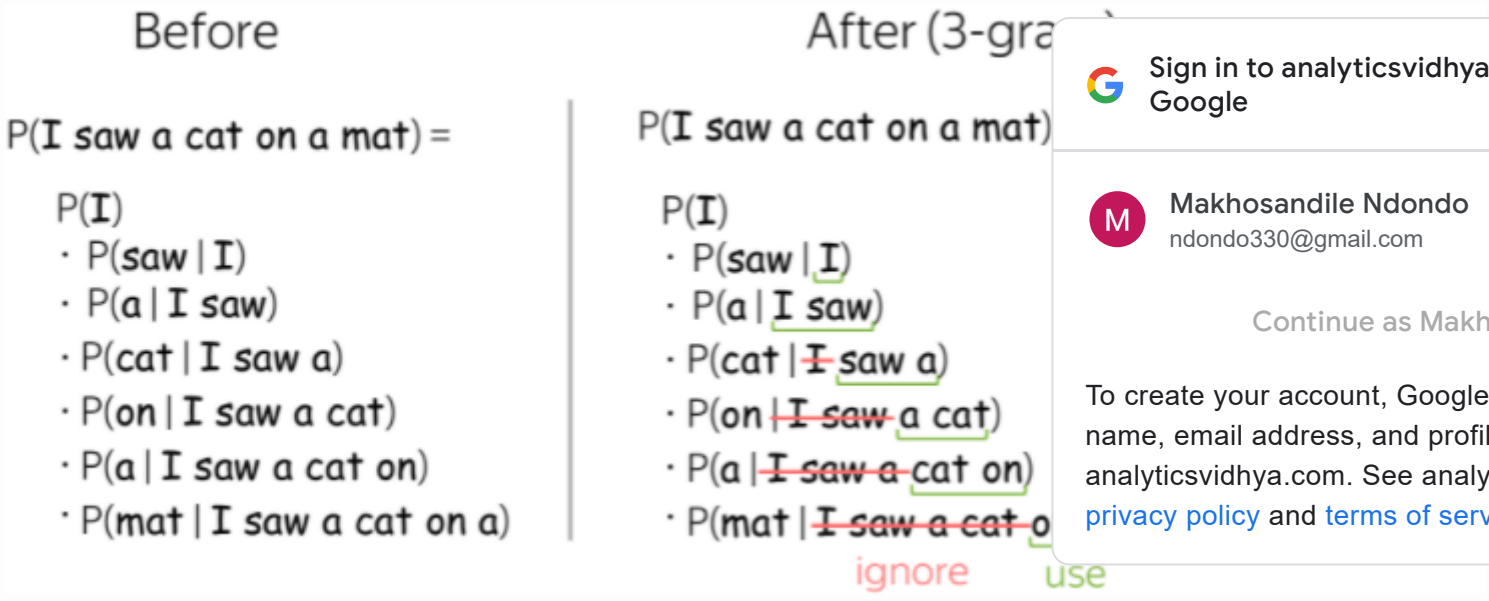
Now, when we use a bigram model to compute the probabilities, the probability of each new word depends only on its previous word. That is, for the previous example, the probability of the sentence becomes :

P(I love dogs) = P(I)P(love | I)P(dogs | love)

Similarly, for a trigram model, the probability will be given by :

P(I love dogs) = P(I)P(love | I)P(dogs | I love) since the probability of each new word depends on the previous two words.

Trigram modelling can be better explained by the following diagram :



Source: ImageLink

However, there is a catch involved in this kind of modelling. Suppose there is some bigram that does not appear in the training set but appears in the test set. Then we will assign a probability of 0 to that bigram, making the overall probability of the test sentence 0, which is undesirable. Smoothing is done to overcome this problem. Parameters are smoothed (or regularized) to reassign some probability mass to unseen events. One way of smoothing is Add-one or Laplace smoothing, which we will be using in this article. Add-one smoothing is performed by adding 1 to all bigram counts and V (no. of unique words in the corpus) to all unigram counts.

$$\text{Old estimate:} \quad P_{MLE}(w_i \mid w_{i-1}) = \frac{c(w_{i-1}, w_i)}{c(w_{i-1})}$$

$$\text{Add-1 estimate:} \quad P_{Add-1}(w_i \mid w_{i-1}) = \frac{c(w_{i-1}, w_i) + 1}{c(w_{i-1}) + V}$$

Now that we have understood what smoothed bigram and trigram models are, let us write the code to compute them. We will be using the unprocessed bigrams and trigrams (without articles, determiners removed) for prediction.

```
smoothed_bigrams_probDist = {}
V = len(unigrams_freqDist)
for i in bigrams_freqDist:
    smoothed_bigrams_probDist[i] = (bigrams_freqDist[i] + 1)/(unigrams_freqDist[i[0]]+V)
smoothed_trigrams_probDist = {}
for i in trigrams_freqDist:
    smoothed_trigrams_probDist[i] = (trigrams_freqDist[i] + 1)/(bigrams_freqDist[i[0:2]]+V)
```

Next, we try to predict the next three words of three test sentences using the computed smoothed bigram and trigram language models.

```
testSent1 = "There was a sudden jerk, a terrific convulsion of the limbs; and there he"
testSent2 = "They made room for the stranger, but he sat down"
testSent3 = "The hungry and destitute situation of the infant orphan was duly reported by"
```

First, we tokenize the test sentences into component words and obtain the last unigrams and bigrams appearing in them.

```
token_1 = word_tokenize(testSent1)
token_2 = word_tokenize(testSent2)
token_3 = word_tokenize(testSent3)
ngram_1 = {1:[], 2:[]}
ngram_2 = {1:[], 2:[]}
ngram_3 = {1:[], 2:[]}
for i in range(2):
    ngram_1[i+1] = list(ngrams(token_1, i+1))[-1]
    ngram_2[i+1] = list(ngrams(token_2, i+1))[-1]
    ngram_3[i+1] = list(ngrams(token_3, i+1))[-1]
print("Sentence 1: ", ngram_1,"nSentence 2: ",ngram_2,"nSentence 3: ",ngr
```


Next, we write functions to predict the next word and the next 3 words respectively of the three test sentences using the smoothed bigram model.

```
def predict_next_word(last_word,probDist):
    next_word = {}
    for k in probDist:
        if k[0] == last_word[0]:
            next_word[k[1]] = probDist[k]
    k = Counter(next_word)
    high = k.most_common(1)
    return high[0]
def predict_next_3_words(token,probDist):
    pred1 = []
    pred2 = []
    next_word = {}
    for i in probDist:
        if i[0] == token:
            next_word[i[1]] = probDist[i]
    k = Counter(next_word)
    high = k.most_common(2)
    w1a = high[0]
    w1b = high[1]
    w2a = predict_next_word(w1a,probDist)
    w3a = predict_next_word(w2a,probDist)
    w2b = predict_next_word(w1b,probDist)
    w3b = predict_next_word(w2b,probDist)
    pred1.append(w1a)
    pred1.append(w2a)
    pred1.append(w3a)
    pred2.append(w1b)
    pred2.append(w2b)
    pred2.append(w3b)
    return pred1,pred2
print("Predicting next 3 possible word sequences with smoothed bigram model : ")
pred1,pred2 = predict_next_3_words(ngram_1[1][0],smoothed_bigrams_probDist)
print("1a)" +testSent1 +" "+ '33[1m' + pred1[0][0]+" "+pred1[1][0]+" "+pred1[2][0] + '33[0m')
print("1b)" +testSent1 +" "+ '33[1m' + pred2[0][0]+" "+pred2[1][0]+" "+pred2[2][0] + '33[0m')
pred1,pred2 = predict_next_3_words(ngram_2[1][0],smoothed_bigrams_probDist)
print("2a)" +testSent2 +" "+ '33[1m' + pred1[0][0]+" "+pred1[1][0]+" "+pred1[2][0] + '33[0m')
print("2b)" +testSent2 +" "+ '33[1m' + pred2[0][0]+" "+pred2[1][0]+" "+pred2[2][0] + '33[0m')
pred1,pred2 = predict_next_3_words(ngram_3[1][0],smoothed_bigrams_probDist)
print("3a)" +testSent3 +" "+ '33[1m' + pred1[0][0]+" "+pred1[1][0]+" "+pred1[2][0] + '33[0m')
print("3b)" +testSent3 +" "+ '33[1m' + pred2[0][0]+" "+pred2[1][0]+" "+pred2[2][0] + '33[0m')
```

The predictions from the smoothed bigram model are :

```
Predicting next 3 possible word sequences with smoothed bigram model :
1a)There was a sudden jerk, a terrific convulsion of the limbs; and there he had been expected
1b)There was a sudden jerk, a terrific convulsion of the limbs; and there he was a very
2a)They made room for the stranger, but he sat down the old gentleman
```

Source: Screenshot from my Jupyter Notebook

We obtain predictions from the smoothed trigram model similarly.


```
def predict_next_word(last_word,probDist):
    next_word = {}
    for k in probDist:
        if k[0:2] == last_word:
            next_word[k[2]] = probDist[k]
    k = Counter(next_word)
    high = k.most_common(1)
    return high[0]
def predict_next_3_words(token,probDist):
    pred = []
    next_word = {}
    for i in probDist:
        if i[0:2] == token:
            next_word[i[2]] = probDist[i]
    k = Counter(next_word)
    high = k.most_common(2)
    w1a = high[0]
    tup = (token[1],w1a[0])
    w2a = predict_next_word(tup,probDist)
    tup = (w1a[0],w2a[0])
    w3a = predict_next_word(tup,probDist)
    pred.append(w1a)
    pred.append(w2a)
    pred.append(w3a)
    return pred
print("Predicting next 3 possible word sequences with smoothed trigram model : ")
pred = predict_next_3_words(ngram_1[2],smoothed_trigrams_probDist)
print("1)" +testSent1 +" "+ '33[1m' + pred[0][0]+" "+pred[1][0]+" "+pred[2][0] + '33[0m')
pred = predict_next_3_words(ngram_2[2],smoothed_trigrams_probDist)
print("2)" +testSent2 +" "+ '33[1m' + pred[0][0]+" "+pred[1][0]+" "+pred[2][0] + '33[0m')
pred = predict_next_3_words(ngram_3[2],smoothed_trigrams_probDist)
print("3)" +testSent3 +" "+ '33[1m' + pred[0][0]+" "+pred[1][0]+" "+pred[2][0] + '33[0m')
```

The output looks like this :

```
Predicting next 3 possible word sequences with smoothed trigram model :
1)There was a sudden jerk, a terrific convulsion of the limbs; and there he had been too
2)They made room for the stranger, but he sat down to a branchworkhouse
3)The hungry and destitute situation of the infant orphan was duly reported by the side of
```

Source: Screenshot from my Jupyter Notebook

Conclusion

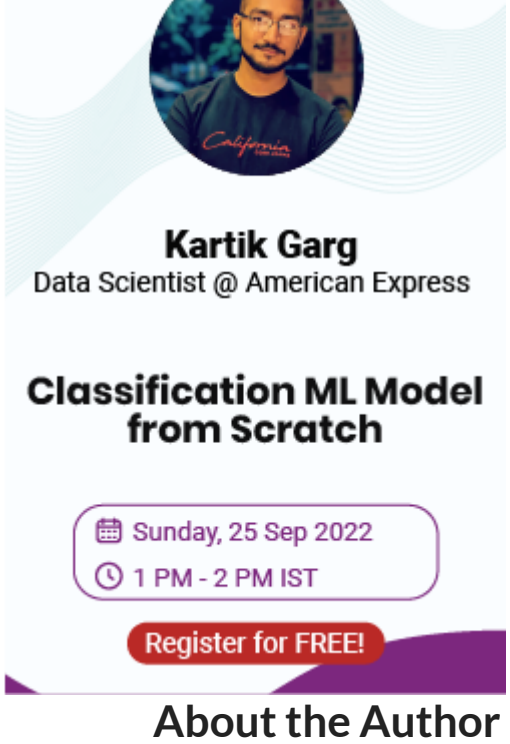
I hope by now you have understood what language models are and how to perform next word prediction using them. Go ahead and try to explore other applications of language models like part of speech tagging, syntactic parsing and sentiment analysis.

Thank you for reading! Read more on language models in NLP, here.

Feel free to connect with me over email: koushikidasguptachaudhuri@gmail.com

The media shown in this article is not owned by Analytics Vidhya and are used at the Author's discretion.

blogathon build language model data preprocessing NLP



About the Author

Koushiki Dasgupta Chaudhuri

Our Top Authors

Download

Analytics Vidhya App for the Latest blog/Article

Previous Post

Classification of Tweets using SpaCy

Next Post

Knowledge Distillation: Theory and End to End Case Study

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

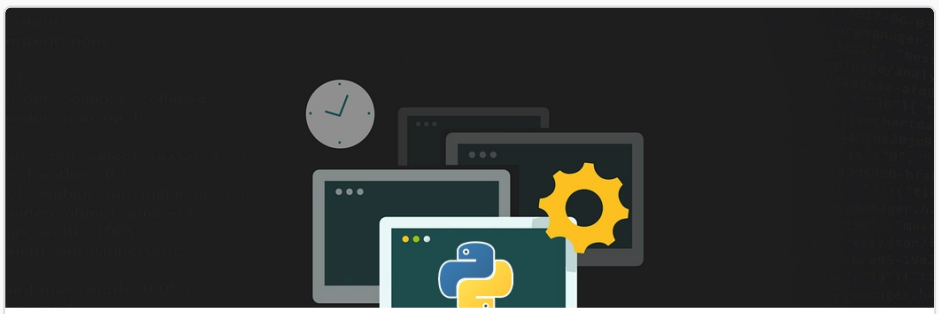
Name*

Email*

Website

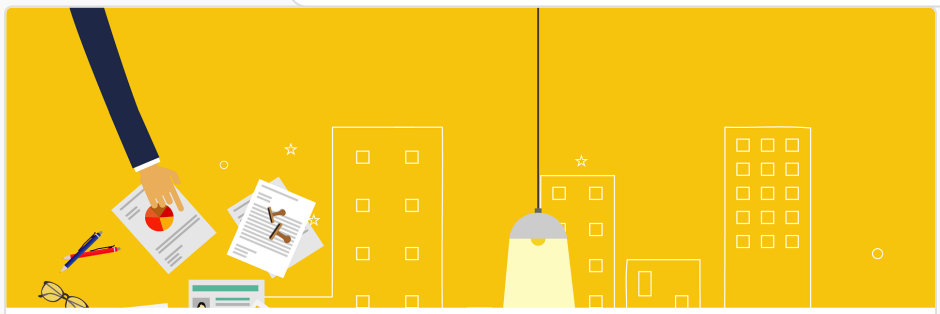
Submit

Top Resources



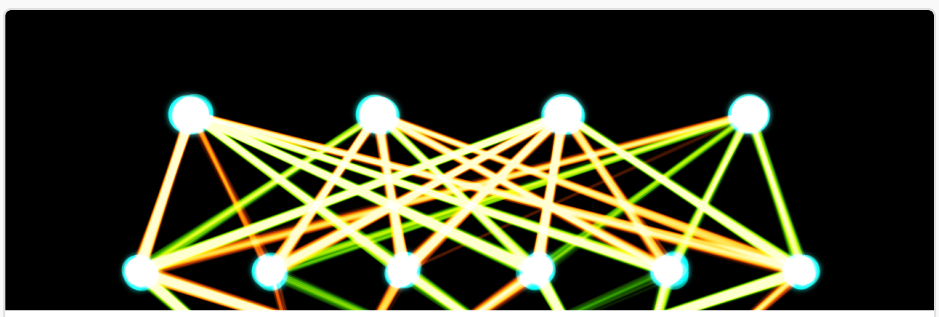
Python Tutorial: Working with CSV file for Data Science

Harika Bonthu - AUG 21, 2021



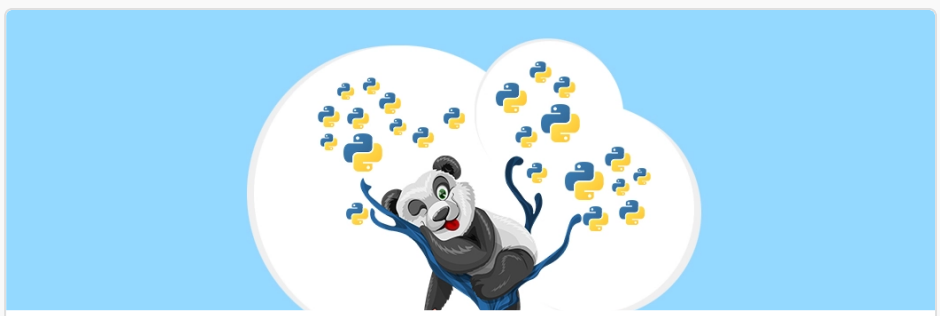
Python Coding Interview Questions for Freshers

Saumyab271 - JUL 23, 2022



Boost Model Accuracy of Imbalanced COVID-19 Mortality Prediction Using GAN-based..

Bala Gangadhar Thilak Adiboina - OCT 07, 2020



Joins in Pandas: Master the Different Types of Joins in..

Abhishek Sharma - FEB 27, 2020