Comparing the performance of a Quadrigram model and a Neural Language model for The Microsoft Research Sentence Completion Challenge

Author: Candidate number 215816
April 27,2022

1 Introduction

The aim of this report, is to compare the performance of two intelligent systems on The Microsoft Research Sentence (MRS) Completion Challenge (Zweig & Burges 2011). The MRS Completion Challenge, is a task for semantic systems, which consists of predicting the correct word, from a set of five possible options with similar occurrence statistics, to complete a sentence. An example of the task is shown in Figure 1.

That is his [generous | mother's | successful | favourite | main] fault , but on the whole he's a good worker.

Figure 1: An example sentence from The MRS Completion Challenge (Zweig & Burges 2011). The system must be able to predict the original word of the sentence from a set of five words with similar occurrence statistics.

The purpose of the MRS completion challenge, is to compare the performance of semantic models on a dataset task, which is not focused on isolated word pair similarity, but rather guessing the next most likely word, given a sequence of a sentence.

Knowing this, we believe that the best approach to solve this problem is to complete the sentence with each word option, in order to calculate the likelihood of each possible sentence, using a language model (LM). The motivation of our investigation, is to acquire the highest accuracy as possible, by using language models. We believe this is achievable, by combining them with two different features. Firstly, a quadrigram model combined with word similarity methods, as these models have been shown to achieve outstanding performances for a similar task, e.g, the combined n-gram and Latent Semantic Analysis model (Zweig et al. 2012). Secondly, a neural language model, which achieved results comparable with state of-the-art models on this task (Mirowski & Vlachos 2015).

2 Methods

The training data used to create the language models were taken from 19^{th} century novel data from Project Gutenberg (Stroube 2003).

Before creating the language models, we decided to remove all of the metadata from the training data, as the words in the metadata increase the frequency for the exact same sequence of words. For instance, the word "public" followed by "domain", has a certain probability of occurring in 19^{th} century novel data. However, because the sequence "public domain" appears many times in the metadata, the likelihood of this occurring increases, which leads to inaccurate predictions from the model, when it encounters the word "public". An illustration of this is shown in Figure 2.

Another data pre-processing technique we used, was converting the least frequent words in the data into Out-Of-Vocabulary (OOV) tokens. We decided to do this, as some words in the testing data don't appear during training, thus, the model will not consider the word

```
sort_orders = sorted(lm.bigram.get("public").items(), key=lambda x: x[1], reverse=True)
sort_orders[:3]
```

```
[('domain', 0.32380174291938996),
 ('___DISCOUNT', 0.1511437908496732),
 ('___END', 0.06236383442265795)]
```

Figure 2: A biagram's prediction, when we train the model on the data including the metadata. We can clearly observe that the model assigns a much higher probability to "domain" compared to "___DISCOUNT", when it encounters the word "public".

option when making a prediction. To avoid this, we added a threshold to the word frequency, meaning that if there is a word in the training data which occurred at a rate beneath the threshold, it would be converted into an OOV token. This is helpful when making predictions in the testing data, as instead of assigning a probability of 0% to an unseen word, the model would just make a prediction of seeing an OOV token, given a sequence of words.

Our language models, were both inspired by 2 different research papers. The 4-gram model from The Microsoft Research Sentence Completion Challenge (Zweig & Burges 2011) and the Recurrent Neural Network (RNN) model from Dependency Recurrent Neural Language Models for Sentence Completion (Mirowski & Vlachos 2015). The following sections (Section 2.1 and Section 2.2) explain why we chose to implement these models.

2.1 Quadrigram model

As demonstrated in The Microsoft Research Sentence Completion Challenge (Zweig & Burges 2011), a Good-Turing smoothed 4-gram model achieved 39% accuracy for this task. Nevertheless, for our model, we decided to use a combination of two smoothing techniques: Back-Off and Distributional smoothing. Although, Back-Off smoothing is designed for web scaled sized datasets (Brants et al. 2007), we still decided to implement this into our model, as we are working with a relatively large dataset, and it is simple to implement. We also used Distributional smoothing, as it uses word similarity methods, which have demonstrated to achieve an accuracy score of 49% (Zweig & Burges 2011).

A quadrigram or 4-gram model works by calculating the probability of the most possible word, given a sequence of three words from the sentence (See Figure 3).

$$P(w_i|w_{i-(n-1)},...,w_{i-1}) = \frac{freq(w_{i-(n-1)},...,w_{i-1},w_i)}{freq(w_{i-(n-1)},...,w_{i-1})}$$

Figure 3: The function of a quadrigram model, where given an observed sequence of words $(w_{i-(n-1)},...,w_{i-1})$, where $n = 3$, it predicts the most likely target word (w_i), which is calculated by dividing the frequencies of the observed sequence with the target word $(freq(w_{i-(n-1)},...,w_{i-1},w_i))$ by the frequencies of the observed sequence without the target word $(freq(w_{i-(n-1)},...,w_{i-1}))$.

The way we designed our quadrigram model, was by tokenising each line in the text from the dataset and adding padding, i.e dummy words, to the beginning ("___START") and end ("___END") of the line. We did this to avoid our model from going out of bounds, when taking the three previous words from the beginning of the sentence, e.g, for the sentence "I like chocolate", we calculate the probability of "like" given word "I" by calculating $P(like|I, \_\_\_START, \_\_\_START)$.

When testing the model's predictions, we wanted to maximise the probability of each possible

word option, in order to achieve better results. We did this by using Distributional smoothing, which uses word similarity methods to search for similar words of the option in the training data, and adds its probability of occurring to the probability of the word option. We show how this is calculated in Figure 4.

$$P_{SIM}(w_i|w_{i-(n-1)},...,w_{i-1}) = \sum_{w'=SIM(w_i)} P(w'|w_{i-(n-1)},...,w_{i-1}) \frac{SIM(w'|wi)}{\sum SIM(w'|wi)}$$

Figure 4: Calculation of Distributional smoothing. Distributional smoothing adds all of the probabilities of similar words $\sum_{w'=SIM(w_i)}$, to the probability of the possible word given the observed sequence of the sentence $P(w_i|w_{i-(n-1)},...,w_{i-1})$.

The similar words for the MSR word options, were acquired by obtaining the full vocabulary from the data files, and training a Word2Vec (Mikolov et al. 2013) neural network model with them. The Word2Vec model works by transforming each distinct word into a vector, and then determines their resemblance, by calculating the cosine similarity between the vectors. The Word2Vec neural network can be trained in two ways, using skip-gram and continuous bag-of-words (Rong 2014)

Finally, given a 4-gram model with distributional smoothing and training data with OOV tokens, when testing the model on the MRS sentences, if the possible word has still not been seen in the training data, in other words, the model predicts the option has a 0% probability of occurring, we apply Back-Off smoothing to our model.

Back-Off smoothing is a simple smoothing technique, consisting of "backing-off" to a lower order n-gram, e.g a trigram or bigram, as a means to assign a probability value to the word option.

2.1.1 Hyper-parameters for 4-gram model

To explore the impact of converting less frequent words into Out-Of-Vocabulary tokens. We decided to modify the **known** hyperparameter, which specifies the threshold of transforming the words.

For Back-Off smoothing, we explored the fixed weight hyper-parameter (λ), which is applied to the lower order n-gram probability.

As part of our Distributional smoothing, because we are using Word2Vec to get word similarities, we have many hyper-parameters which can all be seen in the Word2Vec API (*Gensim Word2vec embeddings API* 2021). The most important hyper-parameters however are **Number of epochs**, **Learning rate**, **Batch size**, **Window size** and **Training algorithm**. However, we will not be exploring these hyper-parameters due to time constraints, so we set them to their default values: **Number of epochs**=5, **Learning rate**=0.025, **Batch size**=10000, **Window size**=5 and **Training algorithm**=Continuous bag of words.

2.2 Recurrent Neural Network Language model

For our next model, we decided to use a Long Short-Term Memory (LSTM) Recurrent Neural Network (RNN) to study how well it performs on the challenge. Having observed the results from Mirowski & Vlachos (2015), their model achieved state of-the-art results on the MRS Completion Challenge. Our aim was to implement this model, as a means to see if we could achieve the same results.

The way LSTMs work in language models, is by using its long term memory, so that it is able to consider relationships of distant words, such as the word at the beginning of sentence and at the end. The inner workings of the LSTM, can be observed in Figure 5.

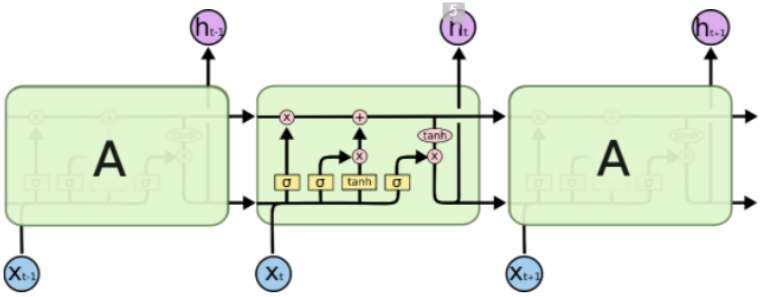


Figure 5: From Olah (2015). The core idea of LSTMs, is that at each timestep, the hidden layer activation values depend on: the curren input, activation values from the hidden layer at previous timestep, which retains *Short-term memory* and activation values from the cell state, which retain *Long term memory*.

Prior to creating the LSTM model, we first converted all of the training data into quadrigrams (where the observed sequence is the input data and the next word the target data), in addition to acquiring the vocabulary of the training data for word embeddings.

For word embeddings, we have to create a lookup table from the vocabulary first, in order to identify the index inside the embedding matrix. This was done by creating a dictionary for each word with an index as the key. The embedding matrix would have a size of (vocabulary size) x D (D is word embeddings dimension), where the i'th row in the matrix, would be the index for the lookup table.

In our LSTM model, the input size is a 3xD 1-d vector, where the 3 is the observed sequence size, and D the embedding dimensions.

The vector would then be inserted into an LSTM model, whose output would then be the input for a fully connected (FC) network. The role of the FC network is to map the predictions made by the LSTM model, to the word target.

To train our model, we divided the data into batches and trained it for a fixed amount of epochs. Finally, for testing our model, we inserted the three left context words, and the possible word, into our model, which returned us the log probability of the possible word occurring given the observed sequence.

2.2.1 Hyper-parameters for Recurrent Neural Network Language model

Same as in Section 2.1.1, we can again explore the **known** hyper-parameter, to convert less frequent words into OOV words.

For the input of the LSTM language model, we have the **embedding dimensions** hyper-parameter, which is multiplied by the context (observed sequence) of the quadrigram, in addition to the **number of hidden neurons** and the **number of layers**.

When training the model, we have the **the number of epochs** and **learning rate** hyper-parameters.

Due to time constraints and lack of computational resources, we will only be exploring the **learning rate** and **the number of hidden neurons** hyper-parameters. The other settings were set to **known**=20, **number of epochs**=25, **number of layers**=1 and **embedding dimensions**=1000.

3 Results

To visualise the performance of our models, we calculated the **accuracy** of each model by dividing the correctly predicted sentences by all sentences in the MSR Completion Challenge, and averaged the result after 10 runs, in order to show more precise results.

For our 4-gram model we compared the results after exploring the **known** hyper-parameter for different values 100, 500 and 1000, and (λ) to 0.2, 0.5 and 1. Table 1 shows the average accuracy for each hyper-parameter for the 4-gram model.

known/weights	λ=0.2	λ=0.5	λ=1
100	29.310%	29.327%	29.282%
500	26.936%	26.423%	27.210%
1000	24.978%	25.101%	25.091%

Table 1: Accuracy for quadrigram model when exploring the **known** and weight (λ) hyper-parameter.

As we can clearly observe from our results, we get less accuracy when we get more Out-Of-Vocabulary (OOV) tokens in our training data.

Another method we used to evaluate the model, was to check how **random** it was, as this would show us if the model was making many random predictions on the test data (see Figure 6).

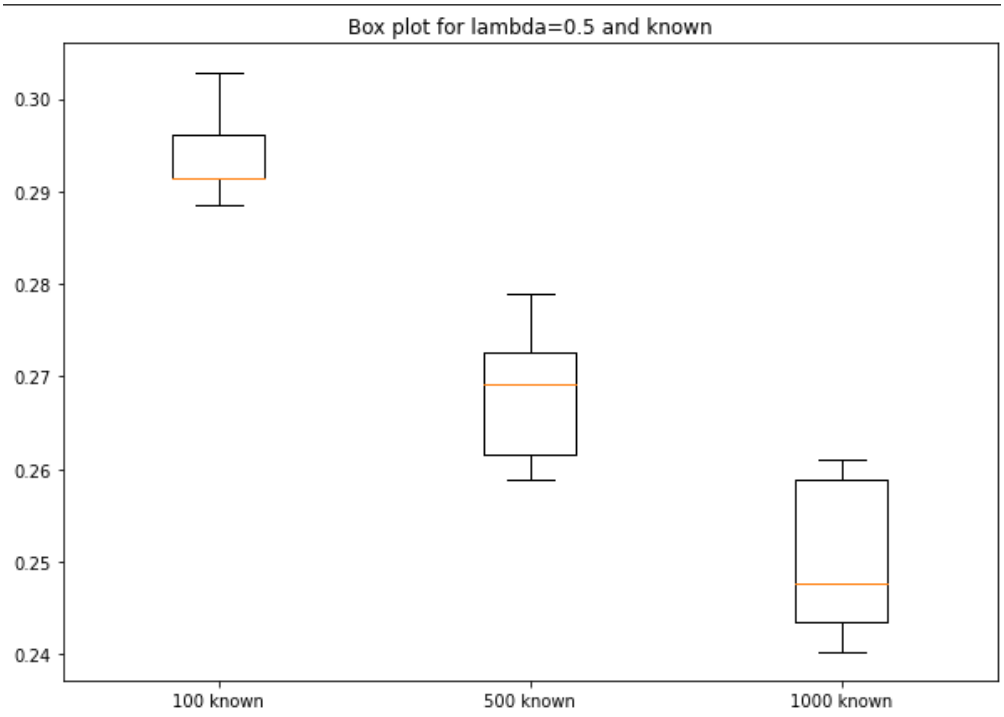


Figure 6: Changes in model accuracy, when averaged on 10 runs on different values of **known** hyper-parameter and λ=0.5. The model's accuracy is more random when there is more OOV tokens.

In the same way as for our quadrigram model, we calculated the **accuracy** when exploring different hyper-parameter settings in our LSTM model. Table 2 shows the average performance after 10 runs.

num of hidden neurons/learning rate	lr=0.001	lr=0.01
32	24.970%	22.502%
128	26.041%	22.900%
256	27.410%	23.650%

Table 2: Accuracy for RNN Language model for different number of hidden neurons and learning rate.

We can clearly see from the table that with more neurons and a smaller learning rate, the model is more accurate.

We also decided to observe the **loss rate**, as this would tell us if our model is training effectively. We can clearly observe from Figure 7 that by using a high learning rate, the loss converges too quickly to a sub-optimal solution, whereas with a lower learning rate, the model is able to reduce its loss faster.

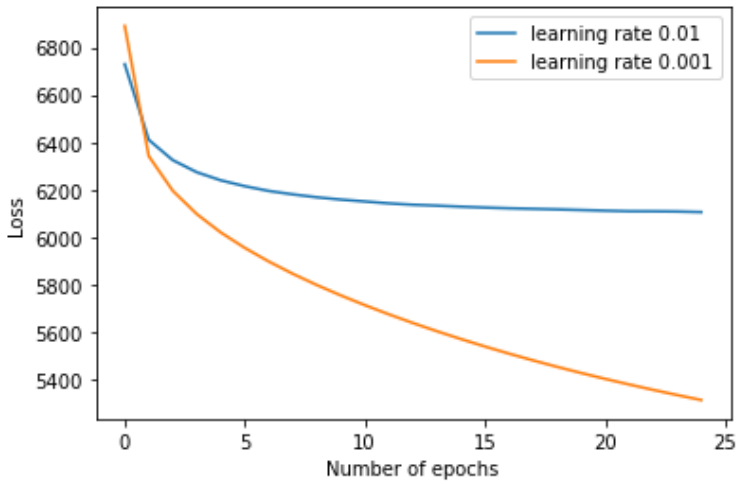


Figure 7: Loss for RNN language model with 25 epochs and learning rate 0.01 and 0.001

We decided not to evaluate the perplexity for any of our models for three reasons. The first reason is due to the fact that both training and testing data come from different corpora. Secondly, according to Brants et al. (2007) we cannot calculate perplexity for our smoothed 4-gram model, as the scores we receive from Back-Off smoothing are not normalized probabilities. Thirdly, didn't evaluate the perplexity of our LSTM model, as explained in Mirowski & Vlachos (2015), after training the RNN on a few epochs, change in perplexity is no longer a good predictor of change in word accuracy, as the validation accuracy rapidly reaches a plateau.

Finally, after taking our best models from the tuned hyper-parameters, we can see how well our model performs in comparison to unigram, bigram and trigram baselines, with different smoothing. These three models work exactly the same as a quadrigram, however, instead of observing the three previous words, a trigram observes two and a biagram one. The unigram just calculates the probability of the word appearing by dividing the frequency of the word by all of the words in the data.

As we can clearly observe from Table 3, our highest achieving model was the n-gram model when we applied both Back-Off and Distributional smoothing.The RNN language model scored 27.410%, which is less than our quadrigram model.

Models	Acccuracy
Only choosing the first possible word	19.903%
Unigram Discount smooth	24.780%
Bigram Discount smooth	27.077%
Bigram Back-Off smooth	26.730%
Trigram Back-Off smooth	29.087%
Quadrigram Back-Off smooth	28.423%
Bigram Distributional smooth	25.910%
Trigram Distributional smooth	26.630%
Quadrigram Distributional smooth	26.673%
Quadrigram both smooth	29.327%
RNN model	27.410%

Table 3: Comparing accuracy results between n-gram models with different smoothings and the Recurrent Neural Language model.

3.1 Failure cases

When analysing the errors made by both methods, while the 4-gram correctly predicted the answer in some sentences and the RNN language model in others, we discovered that there are some sentences where both models do badly at.

Figure 8 shows an example when the LSTM makes a mistake:

A low [courtship|struggle|blow|shrub|moan] *had fallen upon our ears.*

Answer: shrub
RNN prediction: struggle

Figure 8: Example of a sentence where RNN language model is not able to predict the correct answer.

We believe that the reason the LSTM model incorrectly predict these type of sentences, is because they have not seen some of the options in the training data, given the observed sequence, thus they assign these words these options as OOV tokens, and randomly choose from there.

Figure 9 illustrates a misprediction from the 4-gram model:

I do not think that I have ever seen so [accurate|numerous|thin|provocative|marked] *a man.*

Answer: thin
4-gram prediction: marked

Figure 9: Example of a sentence where 4-gram model is not able to predict the correct answer.

For this particular sentence, the 4-gram model showed to always choose the same word option. We believe that the cause of this, is that in the training data, the observed sequence and one the word option appears much more frequently compared to the other possibilities, hence it has a higher probability of being predicted.

Finally, Figure 10 illustrates an example where both models were mistaken in their prediction:

It was one of the main arteries which [supported|surrounded|sheltered|conveyed|contained] *the traffic of the city to the north and west.*

Answer: conveyed
4-gram prediction: contained
RNN prediction: surrounded

Figure 10: Example of a sentence where both models are not able to predict the correct answer.

We think that both models failed to predict the correct option for this particular sentence, due to the reason mentioned in Figure 8 and 9. To see more examples where our model make mistakes, please see the Appendix.

4 Conclusions & Further Work

In conclusion, while both models have shown to acquire a score higher than just choosing the same possible word (20%), the results are nowhere near as close as the results from Zweig et al. (2012) for the n-gram model, and Mirowski & Vlachos (2015) for the RNN language model. The reason for this for the 4-gram model, is because we did not use a suitable smoothing technique for this task, and for the RNN, because we did not leave it training long enough. In Mirowski & Vlachos (2015) the LSTM was trained for 300 hours in order to achieve state of-the-art results.

As further work to be done for the 4-gram model, we suggest converting the probabilities to log probabilities, as these are computationally more efficient, and may reduce training time when testing the model.

In terms of smoothing, we suggest not to use Back-Off smoothing, as these work better for Web-Scaled language model, which include billions of word sequences and millions of unique word. Instead we believe it's more efficient to use Knesser-Ney smoothing, as these work well for any scale.

We also suggest exploring the Word2Vec hyper-parameters, for instance, using a skip-gram instead of a continuous bag of words, as this may also result in higher accuracy for Distributional smoothing.

For the RNN language model, we believe that with a larger network (more hidden neurons and layers) and trained on enough epochs, the model will perform much better, as seen in our results where more hidden neurons achieve higher accuracy. In addition, we think that the model will perform more effectively, if dropout and batch normalisation layers are added to the model, as this will avoid the model from overfitting.

References

Brants, T., Popat, A. C., Xu, P., Och, F. J. & Dean, J. (2007), 'Large language models in machine translation'.

Gensim Word2vec embeddings API (2021).

Mikolov, T., Chen, K., Corrado, G. & Dean, J. (2013), 'Efficient estimation of word representations in vector space', *arXiv preprint arXiv:1301.3781* .

Mirowski, P. & Vlachos, A. (2015), 'Dependency recurrent neural language models for sentence completion', *arXiv preprint arXiv:1507.01193* .

Olah, C. (2015), 'Colah's blog'.

Rong, X. (2014), 'word2vec parameter learning explained', *arXiv preprint arXiv:1411.2738* .

Stroube, B. (2003), 'Literary freedom: Project gutenberg', *XRDS: Crossroads, The ACM Magazine for Students* **10**(1), 3–3.

Zweig, G. & Burges, C. J. (2011), The microsoft research sentence completion challenge, Technical report, Citeseer.

Zweig, G., Platt, J. C., Meek, C., Burges, C. J., Yessenalina, A. & Liu, Q. (2012), Computational approaches to sentence completion, *in* 'Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)', pp. 601–610.

Appendix

For a better view of our code, please visit: https://colab.research.google.com/drive/1QEFzy2pt2A7JicB-7AvyewPEFYeB7F95?usp=sharing or the .ipynb file included in the zip folder.

Advanced NLE Assignment Language Models

April 28, 2022

1 Converting .ipynb file into pdf

1.0.1 Mount drive

[ ]:
```
import sys
from google.colab import drive

#mount google drive
drive.mount('/content/drive/')

#update the system path so that Python knows to look in this folder for␣
↪libraries
sys.path.append('/content/drive/My Drive')
```

1.0.2 Installing latex libraries and converting to pdf

[18]:
```
%cd My\Drive/Colab\ Notebooks
```

```
[Errno 2] No such file or directory: 'MyDrive/Colab Notebooks'
/content/drive/MyDrive/Colab Notebooks
```

[3]:
```
!sudo apt-get install texlive-xetex texlive-fonts-recommended␣
↪texlive-generic-recommended
```

```
Reading package lists… Done
Building dependency tree
Reading state information… Done
texlive-fonts-recommended is already the newest version (2017.20180305-1).
texlive-generic-recommended is already the newest version (2017.20180305-1).
texlive-xetex is already the newest version (2017.20180305-1).
0 upgraded, 0 newly installed, 0 to remove and 41 not upgraded.
```

[4]:
```
!jupyter nbconvert --to pdf 'Advanced NLE Assignment Language Models.ipynb'
```

```
[NbConvertApp] Converting notebook Advanced NLE Assignment Language Models.ipynb
to pdf
```

```
[NbConvertApp] Support files will be in Advanced NLE Assignment Language
Models_files/
[NbConvertApp] Making directory ./Advanced NLE Assignment Language Models_files
[NbConvertApp] Making directory ./Advanced NLE Assignment Language Models_files
[NbConvertApp] Writing 455423 bytes to ./notebook.tex
[NbConvertApp] Building PDF
[NbConvertApp] Running xelatex 3 times: ['xelatex', './notebook.tex', '-quiet']
[NbConvertApp] Running bibtex 1 time: ['bibtex', './notebook']
[NbConvertApp] WARNING | bibtex had problems, most likely because there were no
citations
[NbConvertApp] PDF successfully created
[NbConvertApp] Writing 333276 bytes to Advanced NLE Assignment Language
Models.pdf
```

2 Useful packages & Set Training/Testing Files

2.0.1 Import packages

[ ]:
```
import os
import pandas as pd, csv
import random
import numpy as np
import math
import operator
from collections import Counter
import matplotlib.pyplot as plt

#For LSTM model
from torch.utils.data import Dataset, DataLoader
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
torch.manual_seed(1)


#For Word Similarity in Distributional smoothing
import gensim
from gensim.models import Word2Vec

#For word tokenization
import nltk
nltk.download('punkt')

from nltk import word_tokenize as tokenize
```

2.0.2 Train test split

[ ]:
```
#Get all of the training files from Homes_Training data for sentence completion
def get_training_files(training_dir,split=1):
  filenames=os.listdir(training_dir)
  print("There are {} files in the training directory: {}".
 ↪format(len(filenames),training_dir))
  random.shuffle(filenames)
  return filenames

parentdir="/content/drive/My Drive/lab2resources/lab2resources/
 ↪sentence-completion"  #'C:/Users/benat/Documents/sentence-completion'  ␣
 ↪Please enter your training data path
trainingdir=os.path.join(parentdir,"Holmes_Training_Data")

training = get_training_files(trainingdir)
```

3 Visualise the challenge & Create Sentence Completion Challenge Reader

[7]:
```
pd.options.display.max_colwidth = 500
questions=os.path.join(parentdir,"testing_data.csv")
answers=os.path.join(parentdir,"test_answer.csv")

#Visualise the sentences and their possibilities
with open(questions) as instream:
    csvreader=csv.reader(instream)
    lines=list(csvreader)
qs_df=pd.DataFrame(lines[1:],columns=lines[0])

#Visualise the answers to complete the sentence
with open(answers) as instream:
    csvreader=csv.reader(instream)
    lines=list(csvreader)
qs_df["answers"] = lines[1:]
qs_df.head()
```

[7]:
```
   id  \
0   1
1   2
2   3
3   4
4   5
```

```
                                   question  \
0                I have it from the same source that you are both an orphan and a
bachelor and are _____ alone in London.
1  It was furnished partly as a sitting and partly as a bedroom , with flowers
arranged _____ in every nook and corner.
2                 As I descended , my old ally , the _____ , came out of the room
and closed the door tightly behind him.
3                                      We got off , _____ our fare , and the trap
rattled back on its way to Leatherhead.
4                                       He held in his hand a _____ of blue paper
, scrawled over with notes and figures.

         a)               b)             c)         d)             e) answers
0    crying  instantaneously       residing    matched        walking  [1, c]
1  daintily        privately  inadvertently  miserably    comfortably  [2, a]
2      gods             moon        panther      guard  country-dance  [3, d]
3   rubbing         doubling           paid     naming       carrying  [4, c]
4    supply           parcel           sign      sheet         chorus  [5, d]
```

[8]:
```
class question:

    def __init__(self,aline):
      #Process sentence
      self.fields=aline

    def get_tokens(self):
      #Seperate sentence into tokens
      return ["__START"]+tokenize(self.fields[question.
↪colnames["question"]])+["__END"]

    def get_field(self,field):
      #Get the the field from the possible words to complete the sentence
      #Can be a), b), c), d) or e)
      return self.fields[question.colnames[field]]

    def add_answer(self,fields):
      #Get correct option
      self.answer=fields[1]

    def chooseA(self):
      #Always choose first possible word
      return("a")

    def chooseunigram(self,lm):
      #Method which uses the unigram language model
      choices=["a","b","c","d","e"]
```

```
    probs=[lm.unigram.get(self.get_field(ch+")"),0) for ch in choices]  #Get probability from dictionary (unigram) value from each possible word
    maxprob=max(probs)
    bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
    return np.random.choice(bestchoices)  #If multiple best options, choose randomly

  def get_left_context(self,window=1,target="_____"):
    #Get the left context from the target word
    found=-1
    sent_tokens=self.get_tokens()
    for i,token in enumerate(sent_tokens):
        if token==target:
            found=i
            break
    if found>-1:
        return sent_tokens[i-window:i]
    else:
        return []

  def choosebigram(self,lm,method="bigram",choices=[]):
    #Method which uses the bigram language model
    if choices==[]:
        choices=["a","b","c","d","e"]
    context=self.get_left_context(window=1)
    probs=[lm.get_prob(self.get_field(ch+")"),context,methodparams={"method": method}) for ch in choices]  #Get probability from language model which uses Discount smoothing
    maxprob=max(probs)
    bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
    return np.random.choice(bestchoices)

  def predict(self,lm,method):
    #Get the predicted word option
    if method=="chooseA":
        return self.chooseA()
    elif method=="random":
        return self.chooserandom()
    elif method=="unigram":
        return self.chooseunigram(lm=lm)
    elif method=="bigram":
        return self.choosebigram(lm=lm)

  def predict_and_score(self,method,lm):
    #compare prediction according to method with the correct answer
    #return 1 or 0 accordingly
    prediction=self.predict(method=method,lm=lm)
```

```
        if prediction ==self.answer:
            return 1
        else:
            return 0

class scc_reader:

    def __init__(self,qs=questions,ans=answers):
        self.qs=qs
        self.ans=ans
        self.read_files()

    def read_files(self):
        #read in the question file
        with open(self.qs) as instream:
            csvreader=csv.reader(instream)
            qlines=list(csvreader)

        #store the column names as a reverse index so they can be used to␣
↪reference parts of the question
        question.colnames={item:i for i,item in enumerate(qlines[0])}

        #create a question instance for each line of the file (other than␣
↪heading line)
        self.questions=[question(qline) for qline in qlines[1:]]

        #read in the answer file
        with open(self.ans) as instream:
            csvreader=csv.reader(instream)
            alines=list(csvreader)

        #add answers to questions so predictions can be checked
        for q,aline in zip(self.questions,alines[1:]):
            q.add_answer(aline)

    def get_field(self,field):
      #Get field a), b), c) or d) from word possibilities
      return [q.get_field(field) for q in self.questions]

    def predict(self,method,lm):
      #Get predicted word option
      return [q.predict(method=method,lm=lm) for q in self.questions]

    def predict_and_score(self,method,lm):
        #Used to predict model accuracy
        scores=[q.predict_and_score(method=method,lm=lm) for q in self.
↪questions]
```

```
        return sum(scores)/len(scores)
```

[9]:
```
#Check if the reader is working
SCC = scc_reader()
print(SCC.get_field("a)")[:4])
print(SCC.predict(method="chooseA",lm=None)[:4])
SCC.predict_and_score(method="chooseA",lm=None)
```

```
['crying', 'daintily', 'gods', 'rubbing']
['a', 'a', 'a', 'a']
```

[9]: 0.19903846153846153

4 Language models

[10]:
```
class language_model():
    def __init__(self,trainingdir=parentdir,files=[]):
        self.training_dir=trainingdir
        self.files=files
        self.train()

    def train(self):
        self.unigram={}
        self.bigram={}

        self.start = False  #Starting point of when to get tokens from training␣
↪data
        self.end_tags = ["End of Project Gutenberg's Etext", "End of Project␣
↪Gutenberg Etext", "End of the Project Gutenberg Etext", "End of the Project␣
↪Gutenberg eText"]  #End mark for metadata
        self._processfiles()

        self._make_unknowns()
        self._discount()
        self._kneser_ney()

        self._convert_to_probs()


    def _processline(self,line):
      #Process each line in the file
      #Only process sentence if sentence is not in the metadata
      if "end of project gutenberg" not in line.lower() and "end of the project␣
↪gutenberg" not in line.lower():
```

```
        tokens=["__START"]+tokenize(line)+["__END"] #Add Start and End padding␣
↪to tokenized sentence
        previous="__END"
        for token in tokens:
            self.unigram[token]=self.unigram.get(token,0)+1
            current=self.bigram.get(previous,{})
            current[token]=current.get(token,0)+1
            self.bigram[previous]=current
            previous=token


    def _processfiles(self, remove_metadata=True):
      if remove_metadata:
        #Process each file
        for afile in self.files:
          print("Processing {}".format(afile))
          try:
              with open(os.path.join(self.training_dir,afile)) as instream:
                  for line in instream:
                      line=line.rstrip()
                      if len(line)>0 and self.start:
                          self._processline(line)
                      if "*END*" in line:
                        self.start = True
                  self.start = False
          except UnicodeDecodeError:
              print("UnicodeDecodeError processing {}: ignoring file".
↪format(afile))
      else:
        for file in training[:1]:
          print("Processing {}".format(file))
          try:
            with open(os.path.join(trainingdir,file)) as instream:
              for line in instream:
                line=line.rstrip()
                if len(line)>0:
                  tokens=["__START"]+tokenize(line)+["__END"] #Add Start and␣
↪End padding to tokenized sentence
                  previous="__END"
                  for token in tokens:
                    self.unigram[token]=self.unigram.get(token,0)+1
                    current=self.bigram.get(previous,{})
                    current[token]=current.get(token,0)+1
                    self.bigram[previous]=current
                    previous=token
          except UnicodeDecodeError:
```

```
            print("UnicodeDecodeError processing {}: ignoring file".
↪format(file))



    def _convert_to_probs(self):
        #To get the probabilities, divide frequency of a certain word and divided␣
↪by all occurring words
        self.unigram={k:v/sum(self.unigram.values()) for (k,v) in self.unigram.
↪items()}
        self.bigram={key:{k:v/sum(adict.values()) for (k,v) in adict.items()} for␣
↪(key,adict) in self.bigram.items()}
        self.kn={k:v/sum(self.kn.values()) for (k,v) in self.kn.items()}

        ###adjust __UNK probabilities to include probability of an individual␣
↪unknown word (1/number_unknowns)
        self.unigram["__UNK"]=self.unigram.get("__UNK",0)/self.number_unknowns
        self.bigram["__UNK"]={k:v/self.number_unknowns for (k,v) in self.bigram.
↪get("__UNK",{}).items()}
        for key,adict in self.bigram.items():
            adict["__UNK"]=adict.get("__UNK",0)/self.number_unknowns
            self.bigram[key]=adict
        self.kn["__UNK"]=self.kn.get("__UNK",0)/self.number_unknowns


    def get_prob(self,token,context="",methodparams={}):
        #Get probability from unigram/bigram, plus add some smoothing
        if methodparams.get("method","unigram")=="unigram":
            return self.unigram.get(token,self.unigram.get("__UNK",0))
        else:
            if methodparams.get("smoothing","kneser-ney")=="kneser-ney":
                unidist=self.kn
            else:
                unidist=self.unigram
            bigram=self.bigram.get(context[-1],self.bigram.get("__UNK",{}))
            big_p=bigram.get(token,bigram.get("__UNK",0))
            lmbda=bigram["__DISCOUNT"]
            uni_p=unidist.get(token,unidist.get("__UNK",0))
            p=big_p+lmbda*uni_p
            return p


    def nextlikely(self,k=1,current="",method="unigram"):
        #use probabilities according to method to generate a likely next␣
↪sequence
        #choose random token from k best
```

```
        blacklist=["__START","__UNK","__DISCOUNT"]

        if method=="unigram":
            dist=self.unigram
        else:
            dist=self.bigram.get(current,self.bigram.get("__UNK",{}))

        #sort the tokens by unigram probability
        mostlikely=sorted(list(dist.items()),key=operator.
↪itemgetter(1),reverse=True)
        #filter out any undesirable tokens
        filtered=[w for (w,p) in mostlikely if w not in blacklist]
        #choose one randomly from the top k
        res=random.choice(filtered[:k])
        return res

    def generate(self,k=1,end="__END",limit=20,method="bigram",methodparams={}):
      #Generate a sentence
        if method=="":
            method=methodparams.get("method","bigram")
        current="__START"
        tokens=[]
        while current!=end and len(tokens)<limit:
            current=self.nextlikely(k=k,current=current,method=method)
            tokens.append(current)
        return " ".join(tokens[:-1])


    def compute_prob_line(self,line,methodparams={}):
        #this will add _start to the beginning of a line of text
        #compute the probability of the line according to the desired model
        #and returns probability together with number of tokens

        tokens=["__START"]+tokenize(line)+["__END"]
        acc=0
        for i,token in enumerate(tokens[1:]):
            acc+=math.log(self.get_prob(token,tokens[:i+1],methodparams))
        return acc,len(tokens[1:])

    def compute_probability(self,filenames=[],methodparams={}):
        #computes the probability (and length) of a corpus contained in␣
↪filenames
        if filenames==[]:
            filenames=self.files

        total_p=0
        total_N=0
```

```
        for i,afile in enumerate(filenames):
            print("Processing file {}:{}".format(i,afile))
            try:
                with open(os.path.join(self.training_dir,afile)) as instream:
                    for line in instream:
                        line=line.rstrip()
                        if len(line)>0:
                            p,N=self.
↪compute_prob_line(line,methodparams=methodparams)
                            total_p+=p
                            total_N+=N
            except UnicodeDecodeError:
                print("UnicodeDecodeError processing file {}: ignoring rest of␣
↪file".format(afile))
        return total_p,total_N

    def compute_perplexity(self,filenames=[],methodparams={"method":
↪"bigram","smoothing":"kneser-ney"}):

        #compute the probability and length of the corpus
        #calculate perplexity
        #lower perplexity means that the model better explains the data

        p,N=self.
↪compute_probability(filenames=filenames,methodparams=methodparams)
        #print(p,N)
        pp=math.exp(-p/N)
        return pp

    def _make_unknowns(self,known=100):
      #Convert least occuring words in the language models into␣
↪Out-Of-Vocabulary "__UNK" tokens
        unknown=0
        self.number_unknowns=0
        for (k,v) in list(self.unigram.items()):  #In unigrams do for each key
            if v<known:
                del self.unigram[k]
                self.unigram["__UNK"]=self.unigram.get("__UNK",0)+v
                self.number_unknowns+=1
        for (k,adict) in list(self.bigram.items()): #In bigrams do for each key␣
↪and nested key
            for (kk,v) in list(adict.items()):
                isknown=self.unigram.get(kk,0)
                if isknown==0 and not kk=="__DISCOUNT":
                    adict["__UNK"]=adict.get("__UNK",0)+v
                    del adict[kk]
```

```
            isknown=self.unigram.get(k,0)
            if isknown==0:
                del self.bigram[k]
                current=self.bigram.get("__UNK",{})
                current.update(adict)
                self.bigram["__UNK"]=current

            else:
                self.bigram[k]=adict

    def _discount(self,discount=0.75):
        #discount each bigram count by a small fixed amount
        self.bigram={k:{kk:value-discount for (kk,value) in adict.items()}for␣
↪(k,adict) in self.bigram.items()}

        #for each word, store the total amount of the discount so that the␣
↪total is the same
        #i.e., so we are reserving this as probability mass
        for k in self.bigram.keys():
            lamb=len(self.bigram[k])
            self.bigram[k]["__DISCOUNT"]=lamb*discount

    def _kneser_ney(self):
        #work out kneser-ney unigram probabilities
        #count the number of contexts each word has been seen in
        self.kn={}
        for (k,adict) in self.bigram.items():
            for kk in adict.keys():
                self.kn[kk]=self.kn.get(kk,0)+1
```

[ ]:
```
lm = language_model(trainingdir=trainingdir,files=training)
```

4.0.1 Unigram & Bigram models

4.0.2 Scores

[12]:
```
def get_left_context(sent_tokens,window,target="_____"):
    found=-1
    for i,token in enumerate(sent_tokens):
        if token==target:
            found=i
            break

    if found>-1:
        return sent_tokens[i-window:i]
    else:
```

```
        return []


qs_df['tokens']=qs_df['question'].map(tokenize)
qs_df['left_context']=qs_df['tokens'].map(lambda x: get_left_context(x,2))
```

[14]:
```
#Check if it is working
SCC_unigram = scc_reader()
SCC_bigram = scc_reader()

uni_average = []
bi_average = []
for i in range(10):
  acc = SCC_unigram.predict_and_score(method="unigram",lm=lm)
  uni_average.append(acc)

  acc = SCC_bigram.predict_and_score(method="bigram",lm=lm)
  bi_average.append(acc)

print("Score for unigram",sum(uni_average)/len(uni_average))
print("Score for bigram",sum(bi_average)/len(bi_average))

qs_df["bigram_pred"]=SCC.predict(method="bigram",lm=lm)
qs_df["unigram_pred"]=SCC.predict(method="unigram",lm=lm)
qs_df[:5]
```

```
Score for unigram 0.20798076923076922
Score for bigram 0.2021153846153846
```

[14]:

	id	question	left_context	bigram_pred	unigram_pred
0	1	I have it from the same source that you are both an orphan and a bachelor and are _____ alone in London.	[I, have]	[I, have]	[I, have]
1	2	It was furnished partly as a sitting and partly as a bedroom , with flowers arranged _____ in every nook and corner.	[It, was]	[It, was]	[It, was]
2	3	As I descended , my old ally , the _____ , came out of the room and closed the door tightly behind him.	[the, ,]	[the, ,]	[the, ,]
3	4	We got off , _____ our fare , and the trap rattled back on its way to Leatherhead.	[off, ,]	[off, ,]	[off, ,]
4	5	He held in his hand a _____ of blue paper , scrawled over with notes and figures.	[a, of]	[a, of]	[a, of]

```
         a)               b)             c)         d)             e) answers  \
0    crying  instantaneously       residing    matched        walking  [1, c]
1  daintily        privately  inadvertently  miserably    comfortably  [2, a]
2      gods             moon       panther      guard  country-dance  [3, d]
3   rubbing         doubling          paid     naming       carrying  [4, c]
4    supply           parcel          sign      sheet         chorus  [5, d]

                                                                tokens  \
0              [I, have, it, from, the, same, source, that, you, are, both, an,
orphan, and, a, bachelor, and, are, _____, alone, in, London, .]
1  [It, was, furnished, partly, as, a, sitting, and, partly, as, a, bedroom, ,,
with, flowers, arranged, _____, in, every, nook, and, corner, .]
2              [As, I, descended, ,, my, old, ally, ,, the, _____, ,, came, out,
of, the, room, and, closed, the, door, tightly, behind, him, .]
3                                        [We, got, off, ,, _____, our, fare, ,,
and, the, trap, rattled, back, on, its, way, to, Leatherhead, .]
4                                          [He, held, in, his, hand, a, _____,
of, blue, paper, ,, scrawled, over, with, notes, and, figures, .]

          left_context bigram_pred unigram_pred
0           [and, are]           a            c
1  [flowers, arranged]           c            d
2             [,, the]           e            d
3             [off, ,]           b            c
4            [hand, a]           e            a
```

Metadata probability Training without metadata

```
[ ]: sort_orders = sorted(lm.bigram.get("public").items(), key=lambda x: x[1],␣
     ↪reverse=True)
     sort_orders[:3]
```

```
[ ]: [('__DISCOUNT', 0.22330097087378642),
      ('__END', 0.08940129449838188),
      (',', 0.06351132686084142)]
```

Training with metadata

```
[ ]: class language_model():
         def __init__(self,trainingdir=parentdir,files=[]):
             self.training_dir=trainingdir
             self.files=files
             self.train()

         def train(self):
```

```
        self.unigram={}
        self.bigram={}

        self.start = False  #Starting point of when to get tokens from training␣
↪data
        self.end_tags = ["End of Project Gutenberg's Etext", "End of Project␣
↪Gutenberg Etext", "End of the Project Gutenberg Etext", "End of the Project␣
↪Gutenberg eText"]  #End mark for metadata
        self._processfiles()

        self._make_unknowns()
        self._discount()
        self._kneser_ney()

        self._convert_to_probs()


    def _processline(self,line):
      #Process each line in the file
      #Only process sentence if sentence is not in the metadata
      if "end of project gutenberg" not in line.lower() and "end of the project␣
↪gutenberg" not in line.lower():
        tokens=["__START"]+tokenize(line)+["__END"] #Add Start and End padding␣
↪to tokenized sentence
        previous="__END"
        for token in tokens:
            self.unigram[token]=self.unigram.get(token,0)+1
            current=self.bigram.get(previous,{})
            current[token]=current.get(token,0)+1
            self.bigram[previous]=current
            previous=token


    def _processfiles(self, remove_metadata=False):
      if remove_metadata:
        #Process each file
        for afile in self.files:
          print("Processing {}".format(afile))
          try:
              with open(os.path.join(self.training_dir,afile)) as instream:
                  for line in instream:
                      line=line.rstrip()
                      if len(line)>0 and self.start:
                          self._processline(line)
                      if "*END*" in line:
                        self.start = True
                  self.start = False
```

```
            except UnicodeDecodeError:
                print("UnicodeDecodeError processing {}: ignoring file".
↪format(afile))
        else:
          for afile in self.files:
            print("Processing {}".format(afile))
            try:
              with open(os.path.join(self.training_dir,afile)) as instream:
                for line in instream:
                  line=line.rstrip()
                  if len(line)>0:
                    tokens=["__START"]+tokenize(line)+["__END"] #Add Start and␣
↪End padding to tokenized sentence
                    previous="__END"
                    for token in tokens:
                      self.unigram[token]=self.unigram.get(token,0)+1
                      current=self.bigram.get(previous,{})
                      current[token]=current.get(token,0)+1
                      self.bigram[previous]=current
                      previous=token
            except UnicodeDecodeError:
              print("UnicodeDecodeError processing {}: ignoring file".
↪format(file))



    def _convert_to_probs(self):
      #To get the probabilities, divide frequency of a certain word and divided␣
↪by all occurring words
      self.unigram={k:v/sum(self.unigram.values()) for (k,v) in self.unigram.
↪items()}
      self.bigram={key:{k:v/sum(adict.values()) for (k,v) in adict.items()} for␣
↪(key,adict) in self.bigram.items()}
      self.kn={k:v/sum(self.kn.values()) for (k,v) in self.kn.items()}

      ###adjust __UNK probabilities to include probability of an individual␣
↪unknown word (1/number_unknowns)
      self.unigram["__UNK"]=self.unigram.get("__UNK",0)/self.number_unknowns
      self.bigram["__UNK"]={k:v/self.number_unknowns for (k,v) in self.bigram.
↪get("__UNK",{}).items()}
      for key,adict in self.bigram.items():
          adict["__UNK"]=adict.get("__UNK",0)/self.number_unknowns
          self.bigram[key]=adict
      self.kn["__UNK"]=self.kn.get("__UNK",0)/self.number_unknowns
```

```
    def get_prob(self,token,context="",methodparams={}):
      #Get probability from unigram/bigram, plus add some smoothing
      if methodparams.get("method","unigram")=="unigram":
          return self.unigram.get(token,self.unigram.get("__UNK",0))
      else:
          if methodparams.get("smoothing","kneser-ney")=="kneser-ney":
              unidist=self.kn
          else:
              unidist=self.unigram
          bigram=self.bigram.get(context[-1],self.bigram.get("__UNK",{}))
          big_p=bigram.get(token,bigram.get("__UNK",0))
          lmbda=bigram["__DISCOUNT"]
          uni_p=unidist.get(token,unidist.get("__UNK",0))
          p=big_p+lmbda*uni_p
          return p


    def nextlikely(self,k=10,current="",method="unigram"):
        #use probabilities according to method to generate a likely next␣
↪sequence
        #choose random token from k best
        blacklist=["__START","__UNK","__DISCOUNT"]

        if method=="unigram":
            dist=self.unigram
        else:
            dist=self.bigram.get(current,self.bigram.get("__UNK",{}))

        #sort the tokens by unigram probability
        mostlikely=sorted(list(dist.items()),key=operator.
↪itemgetter(1),reverse=True)
        #filter out any undesirable tokens
        filtered=[w for (w,p) in mostlikely if w not in blacklist]
        #choose one randomly from the top k
        res=random.choice(filtered[:k])
        return res

    def generate(self,k=1,end="__END",limit=20,method="bigram",methodparams={}):
      #Generate a sentence
        if method=="":
            method=methodparams.get("method","bigram")
        current="__START"
        tokens=[]
        while current!=end and len(tokens)<limit:
            current=self.nextlikely(k=k,current=current,method=method)
            tokens.append(current)
        return " ".join(tokens[:-1])
```

```
   def compute_prob_line(self,line,methodparams={}):
       #this will add _start to the beginning of a line of text
       #compute the probability of the line according to the desired model
       #and returns probability together with number of tokens

       tokens=["__START"]+tokenize(line)+["__END"]
       acc=0
       for i,token in enumerate(tokens[1:]):
           acc+=math.log(self.get_prob(token,tokens[:i+1],methodparams))
       return acc,len(tokens[1:])

   def compute_probability(self,filenames=[],methodparams={}):
       #computes the probability (and length) of a corpus contained in␣
↪filenames
       if filenames==[]:
           filenames=self.files

       total_p=0
       total_N=0
       for i,afile in enumerate(filenames):
           print("Processing file {}:{}".format(i,afile))
           try:
               with open(os.path.join(self.training_dir,afile)) as instream:
                   for line in instream:
                       line=line.rstrip()
                       if len(line)>0:
                           p,N=self.
↪compute_prob_line(line,methodparams=methodparams)
                           total_p+=p
                           total_N+=N
           except UnicodeDecodeError:
               print("UnicodeDecodeError processing file {}: ignoring rest of␣
↪file".format(afile))
       return total_p,total_N

   def compute_perplexity(self,filenames=[],methodparams={"method":
↪"bigram","smoothing":"kneser-ney"}):

       #compute the probability and length of the corpus
       #calculate perplexity
       #lower perplexity means that the model better explains the data

       p,N=self.
↪compute_probability(filenames=filenames,methodparams=methodparams)
       #print(p,N)
```

```
        pp=math.exp(-p/N)
        return pp

    def _make_unknowns(self,known=100):
      #Convert least occuring words in the language models into␣
↪Out-Of-Vocabulary "__UNK" tokens
        unknown=0
        self.number_unknowns=0
        for (k,v) in list(self.unigram.items()):  #In unigrams do for each key
            if v<known:
                del self.unigram[k]
                self.unigram["__UNK"]=self.unigram.get("__UNK",0)+v
                self.number_unknowns+=1
        for (k,adict) in list(self.bigram.items()): #In bigrams do for each key␣
↪and nested key
            for (kk,v) in list(adict.items()):
                isknown=self.unigram.get(kk,0)
                if isknown==0 and not kk=="__DISCOUNT":
                    adict["__UNK"]=adict.get("__UNK",0)+v
                    del adict[kk]
            isknown=self.unigram.get(k,0)
            if isknown==0:
                del self.bigram[k]
                current=self.bigram.get("__UNK",{})
                current.update(adict)
                self.bigram["__UNK"]=current

            else:
                self.bigram[k]=adict

    def _discount(self,discount=0.75):
        #discount each bigram count by a small fixed amount
        self.bigram={k:{kk:value-discount for (kk,value) in adict.items()}for␣
↪(k,adict) in self.bigram.items()}

        #for each word, store the total amount of the discount so that the␣
↪total is the same
        #i.e., so we are reserving this as probability mass
        for k in self.bigram.keys():
            lamb=len(self.bigram[k])
            self.bigram[k]["__DISCOUNT"]=lamb*discount

    def _kneser_ney(self):
        #work out kneser-ney unigram probabilities
        #count the number of contexts each word has been seen in
        self.kn={}
        for (k,adict) in self.bigram.items():
```

```
            for kk in adict.keys():
                self.kn[kk]=self.kn.get(kk,0)+1
```

[ ]:
```
lm = language_model(trainingdir=trainingdir,files=training)
```

[ ]:
```
sort_orders = sorted(lm.bigram.get("public").items(), key=lambda x: x[1],␣
↪reverse=True)
sort_orders[:3]
```

[ ]:
```
[('domain', 0.32380174291938996),
 ('__DISCOUNT', 0.1511437908496732),
 ('__END', 0.06236383442265795)]
```

4.0.3 Quadrigram with Distributional and Back-Off smoothing

[ ]:
```
class language_model(language_model):
      def train(self):
        self.unigram={}
        self.bigram={}
        self.trigram={}
        self.quadrigram={}
        self.data = []

        self.start = False
        self.end_tags = ["End of Project Gutenberg's Etext", "End of Project␣
↪Gutenberg Etext", "End of the Project Gutenberg Etext", "End of the Project␣
↪Gutenberg eText"]
        self._processfiles()

        self.wordsim_model = gensim.models.Word2Vec(self.data, min_count = 1,␣
↪vector_size = 100, window = 5)

        self._make_unknowns()

        self._convert_to_probs()

      def _processline(self,line):
        if "end of project gutenberg" not in line.lower() and "end of the␣
↪project gutenberg" not in line.lower():
          self.data.append(tokenize(line))

          tokens=["__START"]+tokenize(line)+["__END"]
          previous="__END"
          for token in tokens:
            self.unigram[token]=self.unigram.get(token,0)+1
```

```
            current=self.bigram.get(previous,{})
            current[token]=current.get(token,0)+1
            self.bigram[previous]=current
            previous=token

        tokens.insert(0,"__END")
        for i in range(len(tokens)-2):
            prev=(tokens[i],tokens[i+1])
            current=self.trigram.get(prev,{})
            current[tokens[i+2]]=current.get(tokens[i+2],0)+1
            self.trigram[prev]=current

        for i in range(len(tokens)-3):
            prev=(tokens[i],tokens[i+1],tokens[i+2])
            current=self.quadrigram.get(prev,{})
            current[tokens[i+3]]=current.get(tokens[i+3],0)+1
            self.quadrigram[prev]=current

    def _make_unknowns(self,known=100):
        unknown=0
        self.number_unknowns=0
        for (k,v) in list(self.unigram.items()):
            if v<known:
                del self.unigram[k]
                self.unigram["__UNK"]=self.unigram.get("__UNK",0)+v
                self.number_unknowns+=1

        for (k,adict) in list(self.bigram.items()):
            for (kk,v) in list(adict.items()):
                isknown=self.unigram.get(kk,0)
                if isknown==0 and not kk=="__DISCOUNT":
                    adict["__UNK"]=adict.get("__UNK",0)+v
                    del adict[kk]
            isknown=self.unigram.get(k,0)
            if isknown==0:
                del self.bigram[k]
                current=self.bigram.get("__UNK",{})
                current.update(adict)
                self.bigram["__UNK"]=current
            else:
                self.bigram[k]=adict

        for (k,adict) in list(self.trigram.items()):
            for (kk,v) in list(adict.items()):
                isknown=self.unigram.get(kk,0)
                if isknown==0 and not kk=="__DISCOUNT":
```

```
                    adict["__UNK"]=adict.get("__UNK",0)+v
                    del adict[kk]
            for word in k:
              isknown=self.unigram.get(word,0)
              if isknown==0:
                del self.trigram[k]
                k = list(k)
                k[k.index(word)] = "__UNK"
                k = tuple(k)
                current=self.trigram.get(k,{})
                current.update(adict)
                self.trigram[k]=current
              else:
                self.trigram[k]=adict

        for (k,adict) in list(self.quadrigram.items()):
          for(kk,v) in list(adict.items()):
            isknown=self.unigram.get(kk,0)
            if isknown==0 and not kk=="__DISCOUNT":
              adict["__UNK"]=adict.get("__UNK",0)+v
              del adict[kk]
          for word in k:
            isknown=self.unigram.get(word,0)
            if isknown==0:
                del self.quadrigram[k]
                k = list(k)
                k[k.index(word)] = "__UNK"
                k = tuple(k)
                current=self.quadrigram.get(k,{})
                current.update(adict)
                self.quadrigram[k]=current
            else:
                self.quadrigram[k]=adict

      def _convert_to_probs(self):
        self.unigram={k:v/sum(self.unigram.values()) for (k,v) in self.unigram.
↪items()}
        self.bigram={key:{k:v/sum(adict.values()) for (k,v) in adict.items()}␣
↪for (key,adict) in self.bigram.items()}
        self.trigram={key:{k:v/sum(adict.values()) for (k,v) in adict.items()}␣
↪for (key,adict) in self.trigram.items()}
        self.quadrigram = {key:{k:v/sum(adict.values()) for (k,v) in adict.
↪items()} for (key,adict) in self.quadrigram.items()}
        if self.number_unknowns > 0:
          self.unigram["__UNK"]=self.unigram.get("__UNK",0)/self.number_unknowns
          self.bigram["__UNK"]={k:v/self.number_unknowns for (k,v) in self.
↪bigram.get("__UNK",{}).items()}
```

```
        for key,adict in self.bigram.items():
          adict["__UNK"]=adict.get("__UNK",0)/self.number_unknowns
          self.bigram[key]=adict

        for key, value in self.trigram.items():
          if "__UNK" in key:
            self.trigram[key]={k:v/self.number_unknowns for (k,v) in self.
↪trigram.get(key,{}).items()}

        for key,adict in self.trigram.items():
            adict["__UNK"]=adict.get("__UNK",0)/self.number_unknowns
            self.trigram[key]=adict

        for key, value in self.quadrigram.items():
          if "__UNK" in key:
            self.quadrigram[key]={k:v/self.number_unknowns for (k,v) in self.
↪quadrigram.get(key,{}).items()}

        for key,adict in self.quadrigram.items():
            adict["__UNK"]=adict.get("__UNK",0)/self.number_unknowns
            self.quadrigram[key]=adict

    def get_prob(self,token,context,methodparams,smoothing,lmbda):
      if smoothing=="back_off":
        if methodparams.get("method","unigram")=="unigram":
          return lmbda*lmbda*lmbda*self.unigram.get(token,self.unigram.
↪get("__UNK",0))
        else:
          if methodparams.get("method","bigram")=="bigram":
            bigram=self.bigram.get(context[0],self.bigram.get("__UNK",{}))
            big_p=bigram.get(token,bigram.get("__UNK",0))
            if big_p == 0:
              big_p = self.
↪get_prob(token=token,context=context,methodparams={"method":
↪"unigram"},smoothing="back_off",lmbda=lmbda)
              return big_p
            else:
              return big_p*lmbda*lmbda

          elif methodparams.get("method","trigram")=="trigram":
            trigram=self.trigram.get(tuple(context),self.trigram.
↪get("__UNK",{}))
            big_p=trigram.get(token,trigram.get("__UNK",0))
            if big_p == 0:
              big_p = self.
↪get_prob(token=token,context=context,methodparams={"method":
↪"bigram"},smoothing="back_off",lmbda=lmbda)
```

```
                return big_p
              else:
                return big_p*lmbda

            else:
              quadrigram=self.quadrigram.get(tuple(context),self.quadrigram.
↪get("__UNK",{}))
              big_p=quadrigram.get(token,quadrigram.get("__UNK",0))
              if big_p == 0:
                big_p = self.
↪get_prob(token=token,context=context,methodparams={"method":
↪"trigram"},smoothing="back_off",lmbda=lmbda)
              return big_p

        elif smoothing=="distributional":
          if methodparams.get("method","unigram")=="unigram":
            return self.unigram.get(token,self.unigram.get("__UNK",0))
          else:
            big_p = 0
            if methodparams.get("method","bigram")=="bigram":
              bigram=self.bigram.get(context[0],self.bigram.get("__UNK",{}))
              if token in self.wordsim_model.wv.key_to_index:
                similar_words = self.wordsim_model.wv.most_similar(token)
                for word,prob in similar_words:
                  small_p=bigram.get(word,bigram.get("__UNK",0))*(prob/
↪sum([a_tuple[1] for a_tuple in similar_words]))
                  big_p+=small_p
              else:
                big_p = bigram.get(token,bigram.get("__UNK",0))
            elif methodparams.get("method","trigram")=="trigram":
              trigram=self.trigram.get(tuple(context),self.trigram.
↪get("__UNK",{}))
              if token in self.wordsim_model.wv.key_to_index:
                similar_words = self.wordsim_model.wv.most_similar(token)
                for word,prob in self.wordsim_model.wv.most_similar(token):
                  small_p=trigram.get(word,trigram.get("__UNK",0))*(prob/
↪sum([a_tuple[1] for a_tuple in similar_words]))
                  big_p+=small_p
              else:
                big_p = trigram.get(token,trigram.get("__UNK",0))
            else:
              quadrigram=self.quadrigram.get(tuple(context),self.quadrigram.
↪get("__UNK",{}))
              if token in self.wordsim_model.wv.key_to_index:
                similar_words = self.wordsim_model.wv.most_similar(token)
                for word,prob in self.wordsim_model.wv.most_similar(token):
```

```
                small_p=quadrigram.get(word,quadrigram.get("__UNK",0))*(prob/
↪sum([a_tuple[1] for a_tuple in similar_words]))
                big_p+=small_p
            else:
              big_p = quadrigram.get(token,quadrigram.get("__UNK",0))
          return big_p

      else:
        if methodparams.get("method","unigram")=="unigram":
          return lmbda*lmbda*lmbda*self.unigram.get(token,self.unigram.
↪get("__UNK",0))
        else:
          big_p = 0
          if methodparams.get("method","bigram")=="bigram":
            bigram=self.bigram.get(context[0],self.bigram.get("__UNK",{}))
            if token in self.wordsim_model.wv.key_to_index:
              for word,prob in self.wordsim_model.wv.most_similar(token):
                small_p=bigram.get(word,bigram.get("__UNK",0))*(prob/
↪sum([a_tuple[1] for a_tuple in lm.wordsim_model.wv.most_similar(token)]))
                big_p+=small_p
            if big_p == 0:
                big_p = self.
↪get_prob(token=token,context=context,methodparams={"method":
↪"unigram"},smoothing="both",lmbda=lmbda)
                return big_p
            else:
                return big_p*lmbda*lmbda
          elif methodparams.get("method","trigram")=="trigram":
            trigram=self.trigram.get(tuple(context),self.trigram.
↪get("__UNK",{}))
            if token in self.wordsim_model.wv.key_to_index:
              for word,prob in self.wordsim_model.wv.most_similar(token):
                small_p=trigram.get(word,trigram.get("__UNK",0))*(prob/
↪sum([a_tuple[1] for a_tuple in lm.wordsim_model.wv.most_similar(token)]))
                big_p+=small_p
            if big_p==0:
              big_p = self.
↪get_prob(token=token,context=context,methodparams={"method":
↪"bigram"},smoothing="both",lmbda=lmbda)
              return big_p
            else:
              return big_p*lmbda
          else:
            quadrigram=self.quadrigram.get(tuple(context),self.quadrigram.
↪get("__UNK",{}))
            if token in self.wordsim_model.wv.key_to_index:
```

```
                for word,prob in self.wordsim_model.wv.most_similar(token):
                  small_p=quadrigram.get(word,quadrigram.get("__UNK",0))*(prob/
↪sum([a_tuple[1] for a_tuple in lm.wordsim_model.wv.most_similar(token)]))
                  big_p+=small_p
              if big_p == 0:
                big_p = self.
↪get_prob(token=token,context=context,methodparams={"method":
↪"trigram"},smoothing="both",lmbda=lmbda)
              return big_p
```

[ ]:
```
class question(question):
    def chooseunigram(self,lm):
        choices=["a","b","c","d","e"]
        probs=[lm.get_prob(self.get_field(ch+")"),0,methodparams={"method":
↪"unigram"},smoothing="both",lmbda=0.5) for ch in choices]
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
        return np.random.choice(bestchoices)


    def choosebigram(self,lm,choices=[]):
        if choices==[]:
            choices=["a","b","c","d","e"]
        context=self.get_left_context(window=1)
        probs=[lm.get_prob(self.
↪get_field(ch+")"),context,methodparams={"method":
↪"bigram"},smoothing="both",lmbda=0.5) for ch in choices]
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
        return np.random.choice(bestchoices)

    def choosetrigram(self,lm,choices=[]):
        if choices==[]:
            choices=["a","b","c","d","e"]
        context=self.get_left_context(window=2)
        probs=[lm.get_prob(self.
↪get_field(ch+")"),context,methodparams={"method":
↪"trigram"},smoothing="both",lmbda=0.5) for ch in choices]
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
        return np.random.choice(bestchoices)

    def choosequadrigram(self,lm,choices=[]):
        if choices==[]:
            choices=["a","b","c","d","e"]
        context=self.get_left_context(window=3)
```

```
        probs=[lm.get_prob(self.
↪get_field(ch+")"),context,methodparams={"method":
↪"quadrigram"},smoothing="both",lmbda=0.5) for ch in choices]
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
        return np.random.choice(bestchoices)

    def predict(self,lm,method):
        if method=="chooseA":
            return self.chooseA()
        elif method=="random":
            return self.chooserandom()
        elif method=="unigram":
            return self.chooseunigram(lm=lm)
        elif method=="bigram":
            return self.choosebigram(lm=lm)
        elif method=="trigram":
            return self.choosetrigram(lm=lm)
        elif method=="quadrigram":
          return self.choosequadrigram(lm=lm)
```

Scores Both smoothing techniques with known=100 and lambda=0.5

[ ]:
```
lm = language_model(trainingdir=trainingdir,files=training)
```

[ ]:
```
#Both smoothing techniques
SCC_test = scc_reader()
bo_100 = []
for i in range(10):
  acc = SCC_test.predict_and_score(method="quadrigram",lm=lm)
  print(acc)
  bo_100.append(acc)


print("Score for quadrigram",sum(bo_100)/len(bo_100))
```

Both smoothing techniques with known=500 and lambda=0.5

[ ]:
```
class language_model(language_model):
  def _make_unknowns(self,known=500):
        unknown=0
        self.number_unknowns=0
        for (k,v) in list(self.unigram.items()):
            if v<known:
                del self.unigram[k]
                self.unigram["__UNK"]=self.unigram.get("__UNK",0)+v
```

```
                self.number_unknowns+=1

        for (k,adict) in list(self.bigram.items()):
            for (kk,v) in list(adict.items()):
                isknown=self.unigram.get(kk,0)
                if isknown==0 and not kk=="__DISCOUNT":
                    adict["__UNK"]=adict.get("__UNK",0)+v
                    del adict[kk]
            isknown=self.unigram.get(k,0)
            if isknown==0:
                del self.bigram[k]
                current=self.bigram.get("__UNK",{})
                current.update(adict)
                self.bigram["__UNK"]=current
            else:
                self.bigram[k]=adict

        for (k,adict) in list(self.trigram.items()):
            for (kk,v) in list(adict.items()):
                isknown=self.unigram.get(kk,0)
                if isknown==0 and not kk=="__DISCOUNT":
                    adict["__UNK"]=adict.get("__UNK",0)+v
                    del adict[kk]
            for word in k:
              isknown=self.unigram.get(word,0)
              if isknown==0:
                del self.trigram[k]
                k = list(k)
                k[k.index(word)] = "__UNK"
                k = tuple(k)
                current=self.trigram.get(k,{})
                current.update(adict)
                self.trigram[k]=current
              else:
                self.trigram[k]=adict
```

[ ]:
```
lm = language_model(trainingdir=trainingdir,files=training)
```

[ ]:
```
#Both smoothing techniques
SCC_test = scc_reader()
bo_500 = []
for i in range(10):
  acc = SCC_test.predict_and_score(method="quadrigram",lm=lm)
  print(acc)
  bo_500.append(acc)
```

```
print("Score for quadrigram",sum(bo_500)/len(bo_500))
```

Both smoothing techniques with known=1000 and lambda=0.5

[ ]:
```
class language_model(language_model):
  def _make_unknowns(self,known=1000):
        unknown=0
        self.number_unknowns=0
        for (k,v) in list(self.unigram.items()):
            if v<known:
                del self.unigram[k]
                self.unigram["__UNK"]=self.unigram.get("__UNK",0)+v
                self.number_unknowns+=1

        for (k,adict) in list(self.bigram.items()):
            for (kk,v) in list(adict.items()):
                isknown=self.unigram.get(kk,0)
                if isknown==0 and not kk=="__DISCOUNT":
                    adict["__UNK"]=adict.get("__UNK",0)+v
                    del adict[kk]
            isknown=self.unigram.get(k,0)
            if isknown==0:
                del self.bigram[k]
                current=self.bigram.get("__UNK",{})
                current.update(adict)
                self.bigram["__UNK"]=current
            else:
                self.bigram[k]=adict

        for (k,adict) in list(self.trigram.items()):
            for (kk,v) in list(adict.items()):
                isknown=self.unigram.get(kk,0)
                if isknown==0 and not kk=="__DISCOUNT":
                    adict["__UNK"]=adict.get("__UNK",0)+v
                    del adict[kk]
            for word in k:
              isknown=self.unigram.get(word,0)
              if isknown==0:
                del self.trigram[k]
                k = list(k)
                k[k.index(word)] = "__UNK"
                k = tuple(k)
                current=self.trigram.get(k,{})
                current.update(adict)
                self.trigram[k]=current
              else:
                self.trigram[k]=adict
```

[ ]:
```
lm = language_model(trainingdir=trainingdir,files=training)
```

[ ]:
```
#Both smoothing techniques
SCC_test = scc_reader()
bo_1000 = []
for i in range(10):
  acc = SCC_test.predict_and_score(method="quadrigram",lm=lm)
  print(acc)
  bo_1000.append(acc)


print("Score for quadrigram",sum(bo_1000)/len(bo_1000))
```

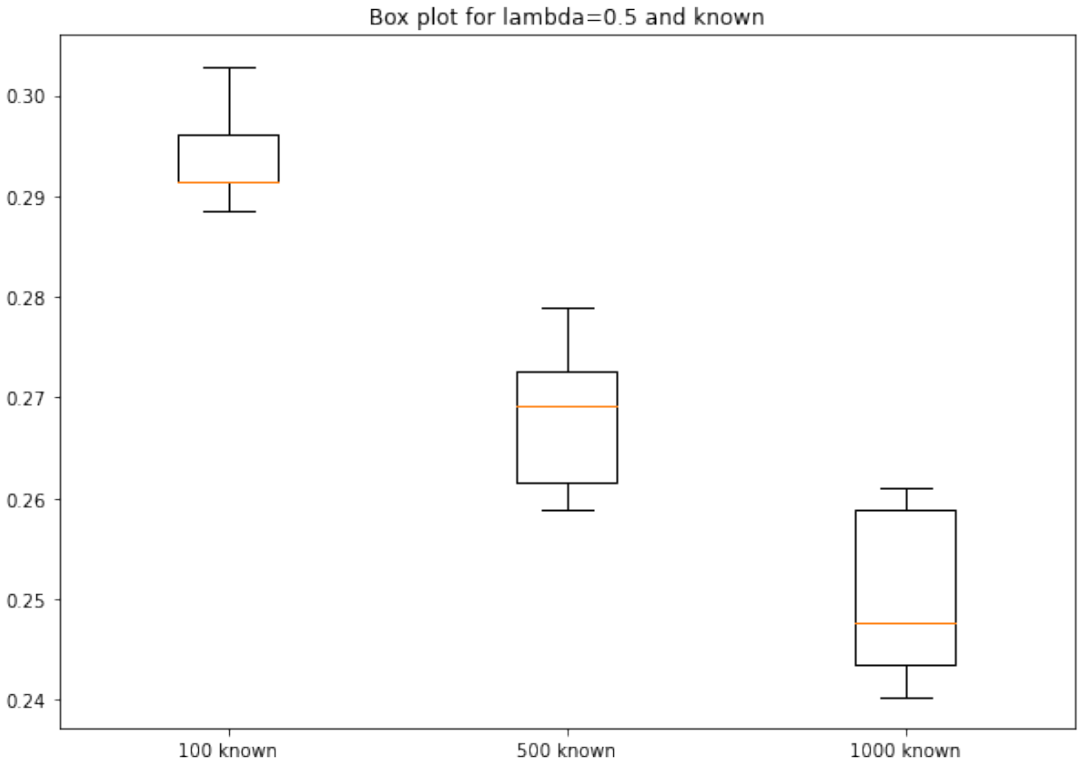
[ ]:
```
# Creating dataset
np.random.seed(10)
data = [bo_100 , bo_500, bo_1000]

fig = plt.figure(figsize =(10, 7))
ax = fig.add_subplot(111)

ax.set_xticklabels(['100 known', '500 known','1000 known'])

# Creating plot
plt.boxplot(data)
plt.title("Box plot for lambda=0.5 and known")
# show plot
plt.show()
```

```
C:\Users\benat\AppData\Local\Temp/ipykernel_21152/113886431.py:13: UserWarning:
FixedFormatter should only be used together with FixedLocator
  ax.set_xticklabels(['100 known', '500 known','1000 known'])
```



Distributional smoothing techniques with known=100

[ ]:
```
class language_model(language_model):
  def _make_unknowns(self,known=100):
        unknown=0
        self.number_unknowns=0
        for (k,v) in list(self.unigram.items()):
            if v<known:
                del self.unigram[k]
                self.unigram["__UNK"]=self.unigram.get("__UNK",0)+v
                self.number_unknowns+=1

        for (k,adict) in list(self.bigram.items()):
            for (kk,v) in list(adict.items()):
                isknown=self.unigram.get(kk,0)
                if isknown==0 and not kk=="__DISCOUNT":
                    adict["__UNK"]=adict.get("__UNK",0)+v
                    del adict[kk]
            isknown=self.unigram.get(k,0)
            if isknown==0:
                del self.bigram[k]
                current=self.bigram.get("__UNK",{})
```

```
                current.update(adict)
                self.bigram["__UNK"]=current
            else:
                self.bigram[k]=adict

        for (k,adict) in list(self.trigram.items()):
            for (kk,v) in list(adict.items()):
                isknown=self.unigram.get(kk,0)
                if isknown==0 and not kk=="__DISCOUNT":
                    adict["__UNK"]=adict.get("__UNK",0)+v
                    del adict[kk]
            for word in k:
              isknown=self.unigram.get(word,0)
              if isknown==0:
                del self.trigram[k]
                k = list(k)
                k[k.index(word)] = "__UNK"
                k = tuple(k)
                current=self.trigram.get(k,{})
                current.update(adict)
                self.trigram[k]=current
              else:
                self.trigram[k]=adict
```

[ ]:
```
class question(question):
    def chooseunigram(self,lm):
        choices=["a","b","c","d","e"]
        probs=[lm.get_prob(self.get_field(ch+")"),0,methodparams={"method":
↪"unigram"},smoothing="distributional",lmbda=0.5) for ch in choices]
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
        return np.random.choice(bestchoices)


    def choosebigram(self,lm,choices=[]):
        if choices==[]:
            choices=["a","b","c","d","e"]
        context=self.get_left_context(window=1)
        probs=[lm.get_prob(self.
↪get_field(ch+")"),context,methodparams={"method":
↪"bigram"},smoothing="distributional",lmbda=0.5) for ch in choices]
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
        return np.random.choice(bestchoices)

    def choosetrigram(self,lm,choices=[]):
        if choices==[]:
```

```
            choices=["a","b","c","d","e"]
        context=self.get_left_context(window=2)
        probs=[lm.get_prob(self.
↪get_field(ch+")"),context,methodparams={"method":
↪"trigram"},smoothing="distributional",lmbda=0.5) for ch in choices]
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
        return np.random.choice(bestchoices)

    def choosequadrigram(self,lm,choices=[]):
        if choices==[]:
            choices=["a","b","c","d","e"]
        context=self.get_left_context(window=3)
        probs=[lm.get_prob(self.
↪get_field(ch+")"),context,methodparams={"method":
↪"quadrigram"},smoothing="distributional",lmbda=0.5) for ch in choices]
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
        return np.random.choice(bestchoices)

    def predict(self,lm,method):
        if method=="chooseA":
            return self.chooseA()
        elif method=="random":
            return self.chooserandom()
        elif method=="unigram":
            return self.chooseunigram(lm=lm)
        elif method=="bigram":
            return self.choosebigram(lm=lm)
        elif method=="trigram":
            return self.choosetrigram(lm=lm)
        elif method=="quadrigram":
          return self.choosequadrigram(lm=lm)
```

[ ]:
```
lm = language_model(trainingdir=trainingdir,files=training)
```

[ ]:
```
#Distributional smoothing bigram
SCC_test = scc_reader()
distri_bi = []
for i in range(10):
  acc = SCC_test.predict_and_score(method="bigram",lm=lm)
  print(acc)
  distri_bi.append(acc)


print("Score for quadrigram",sum(distri_bi)/len(distri_bi))
```

[ ]:
```
#Distributional smoothing trigram
SCC_test = scc_reader()
distri_tri = []
for i in range(10):
  acc = SCC_test.predict_and_score(method="trigram",lm=lm)
  print(acc)
  distri_tri.append(acc)


print("Score for quadrigram",sum(distri_tri)/len(distri_tri))
```

[ ]:
```
#Distributional smoothing quadrigram
SCC_test = scc_reader()
distri_quad = []
for i in range(10):
  acc = SCC_test.predict_and_score(method="quadrigram",lm=lm)
  print(acc)
  distri_quad.append(acc)


print("Score for quadrigram",sum(distri_quad)/len(distri_quad))
```

Back-Off smoothing techniques with known=100

[ ]:
```
class question(question):
    def chooseunigram(self,lm):
        choices=["a","b","c","d","e"]
        probs=[lm.get_prob(self.get_field(ch+")"),0,methodparams={"method":
 ↪"unigram"},smoothing="back_off",lmbda=0.5) for ch in choices]
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
        return np.random.choice(bestchoices)


    def choosebigram(self,lm,choices=[]):
        if choices==[]:
            choices=["a","b","c","d","e"]
        context=self.get_left_context(window=1)
        probs=[lm.get_prob(self.
 ↪get_field(ch+")"),context,methodparams={"method":
 ↪"bigram"},smoothing="back_off",lmbda=0.5) for ch in choices]
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
        return np.random.choice(bestchoices)

    def choosetrigram(self,lm,choices=[]):
        if choices==[]:
```

```
            choices=["a","b","c","d","e"]
        context=self.get_left_context(window=2)
        probs=[lm.get_prob(self.
↪get_field(ch+")"),context,methodparams={"method":
↪"trigram"},smoothing="back_off",lmbda=0.5) for ch in choices]
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
        return np.random.choice(bestchoices)

    def choosequadrigram(self,lm,choices=[]):
        if choices==[]:
            choices=["a","b","c","d","e"]
        context=self.get_left_context(window=3)
        probs=[lm.get_prob(self.
↪get_field(ch+")"),context,methodparams={"method":
↪"quadrigram"},smoothing="back_off",lmbda=0.5) for ch in choices]
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
        return np.random.choice(bestchoices)

    def predict(self,lm,method):
        if method=="chooseA":
            return self.chooseA()
        elif method=="random":
            return self.chooserandom()
        elif method=="unigram":
            return self.chooseunigram(lm=lm)
        elif method=="bigram":
            return self.choosebigram(lm=lm)
        elif method=="trigram":
            return self.choosetrigram(lm=lm)
        elif method=="quadrigram":
          return self.choosequadrigram(lm=lm)
```

[ ]:
```
#Back Off smoothing bigram
SCC_test = scc_reader()
back_bi = []
for i in range(10):
  acc = SCC_test.predict_and_score(method="bigram",lm=lm)
  print(acc)
  back_bi.append(acc)


print("Score for quadrigram",sum(back_bi)/len(back_bi))
```

[ ]:
```
#Back Off smoothing trigram
SCC_test = scc_reader()
back_tri = []
for i in range(10):
  acc = SCC_test.predict_and_score(method="trigram",lm=lm)
  print(acc)
  back_tri.append(acc)


print("Score for quadrigram",sum(back_tri)/len(back_tri))
```

[ ]:
```
#Back Off smoothing quadrigram
SCC_test = scc_reader()
back_quad = []
for i in range(10):
  acc = SCC_test.predict_and_score(method="quadrigram",lm=lm)
  print(acc)
  back_quad.append(acc)


print("Score for quadrigram",sum(back_quad)/len(back_quad))
```

Both smoothing techniques with known=100 and lambda=0.2

[ ]:
```
class question(question):
    def chooseunigram(self,lm):
        choices=["a","b","c","d","e"]
        probs=[lm.get_prob(self.get_field(ch+")"),0,methodparams={"method":
↪"unigram"},smoothing="both",lmbda=0.2) for ch in choices]
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
        return np.random.choice(bestchoices)


    def choosebigram(self,lm,choices=[]):
        if choices==[]:
            choices=["a","b","c","d","e"]
        context=self.get_left_context(window=1)
        probs=[lm.get_prob(self.
↪get_field(ch+")"),context,methodparams={"method":
↪"bigram"},smoothing="both",lmbda=0.2) for ch in choices]
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
        return np.random.choice(bestchoices)

    def choosetrigram(self,lm,choices=[]):
        if choices==[]:
```

```
            choices=["a","b","c","d","e"]
        context=self.get_left_context(window=2)
        probs=[lm.get_prob(self.
↪get_field(ch+")"),context,methodparams={"method":
↪"trigram"},smoothing="both",lmbda=0.2) for ch in choices]
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
        return np.random.choice(bestchoices)

    def choosequadrigram(self,lm,choices=[]):
        if choices==[]:
            choices=["a","b","c","d","e"]
        context=self.get_left_context(window=3)
        probs=[lm.get_prob(self.
↪get_field(ch+")"),context,methodparams={"method":
↪"quadrigram"},smoothing="both",lmbda=0.2) for ch in choices]
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
        return np.random.choice(bestchoices)

    def predict(self,lm,method):
        if method=="chooseA":
            return self.chooseA()
        elif method=="random":
            return self.chooserandom()
        elif method=="unigram":
            return self.chooseunigram(lm=lm)
        elif method=="bigram":
            return self.choosebigram(lm=lm)
        elif method=="trigram":
            return self.choosetrigram(lm=lm)
        elif method=="quadrigram":
          return self.choosequadrigram(lm=lm)
```

[ ]:
```
SCC_test = scc_reader()
quadri_average_0_02 = []
for i in range(10):
  acc = SCC_test.predict_and_score(method="quadrigram",lm=lm)
  print(acc)
  quadri_average_0_02.append(acc)

print("Score for quadrigram",sum(quadri_average_0_02)/len(quadri_average_0_02))
```

Both smoothing techniques with known=100 and lambda=1

[ ]:
```
class question(question):
    def chooseunigram(self,lm):
```

```
        choices=["a","b","c","d","e"]
        probs=[lm.get_prob(self.get_field(ch+")"),0,methodparams={"method":
↪"unigram"},smoothing="both",lmbda=1) for ch in choices]
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
        return np.random.choice(bestchoices)


    def choosebigram(self,lm,choices=[]):
        if choices==[]:
            choices=["a","b","c","d","e"]
        context=self.get_left_context(window=1)
        probs=[lm.get_prob(self.
↪get_field(ch+")"),context,methodparams={"method":
↪"bigram"},smoothing="both",lmbda=1) for ch in choices]
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
        return np.random.choice(bestchoices)

    def choosetrigram(self,lm,choices=[]):
        if choices==[]:
            choices=["a","b","c","d","e"]
        context=self.get_left_context(window=2)
        probs=[lm.get_prob(self.
↪get_field(ch+")"),context,methodparams={"method":
↪"trigram"},smoothing="both",lmbda=1) for ch in choices]
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
        return np.random.choice(bestchoices)

    def choosequadrigram(self,lm,choices=[]):
        if choices==[]:
            choices=["a","b","c","d","e"]
        context=self.get_left_context(window=3)
        probs=[lm.get_prob(self.
↪get_field(ch+")"),context,methodparams={"method":
↪"quadrigram"},smoothing="both",lmbda=1) for ch in choices]
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
        return np.random.choice(bestchoices)

    def predict(self,lm,method):
        if method=="chooseA":
            return self.chooseA()
        elif method=="random":
            return self.chooserandom()
        elif method=="unigram":
```

```
            return self.chooseunigram(lm=lm)
        elif method=="bigram":
            return self.choosebigram(lm=lm)
        elif method=="trigram":
            return self.choosetrigram(lm=lm)
        elif method=="quadrigram":
          return self.choosequadrigram(lm=lm)
```

[ ]:
```
SCC_test = scc_reader()
quadri_average_0_1 = []
for i in range(10):
  acc = SCC_test.predict_and_score(method="quadrigram",lm=lm)
  print(acc)
  quadri_average_0_1.append(acc)


print("Score for quadrigram",sum(quadri_average_0_1)/len(quadri_average_0_1))
```

Both smoothing techniques with known=500 and lambda=0.2

[ ]:
```
class language_model(language_model):
  def _make_unknowns(self,known=500):
        unknown=0
        self.number_unknowns=0
        for (k,v) in list(self.unigram.items()):
            if v<known:
                del self.unigram[k]
                self.unigram["__UNK"]=self.unigram.get("__UNK",0)+v
                self.number_unknowns+=1

        for (k,adict) in list(self.bigram.items()):
            for (kk,v) in list(adict.items()):
                isknown=self.unigram.get(kk,0)
                if isknown==0 and not kk=="__DISCOUNT":
                    adict["__UNK"]=adict.get("__UNK",0)+v
                    del adict[kk]
            isknown=self.unigram.get(k,0)
            if isknown==0:
                del self.bigram[k]
                current=self.bigram.get("__UNK",{})
                current.update(adict)
                self.bigram["__UNK"]=current
            else:
                self.bigram[k]=adict

        for (k,adict) in list(self.trigram.items()):
            for (kk,v) in list(adict.items()):
```

```
                isknown=self.unigram.get(kk,0)
                if isknown==0 and not kk=="__DISCOUNT":
                    adict["__UNK"]=adict.get("__UNK",0)+v
                    del adict[kk]
            for word in k:
              isknown=self.unigram.get(word,0)
              if isknown==0:
                del self.trigram[k]
                k = list(k)
                k[k.index(word)] = "__UNK"
                k = tuple(k)
                current=self.trigram.get(k,{})
                current.update(adict)
                self.trigram[k]=current
              else:
                self.trigram[k]=adict

class question(question):
    def chooseunigram(self,lm):
        choices=["a","b","c","d","e"]
        probs=[lm.get_prob(self.get_field(ch+")"),0,methodparams={"method":
 ↪"unigram"},smoothing="both",lmbda=0.2) for ch in choices]
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
        return np.random.choice(bestchoices)


    def choosebigram(self,lm,choices=[]):
        if choices==[]:
            choices=["a","b","c","d","e"]
        context=self.get_left_context(window=1)
        probs=[lm.get_prob(self.
 ↪get_field(ch+")"),context,methodparams={"method":
 ↪"bigram"},smoothing="both",lmbda=0.2) for ch in choices]
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
        return np.random.choice(bestchoices)

    def choosetrigram(self,lm,choices=[]):
        if choices==[]:
            choices=["a","b","c","d","e"]
        context=self.get_left_context(window=2)
        probs=[lm.get_prob(self.
 ↪get_field(ch+")"),context,methodparams={"method":
 ↪"trigram"},smoothing="both",lmbda=0.2) for ch in choices]
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
```

```
        return np.random.choice(bestchoices)

    def choosequadrigram(self,lm,choices=[]):
        if choices==[]:
            choices=["a","b","c","d","e"]
        context=self.get_left_context(window=3)
        probs=[lm.get_prob(self.
 ↪get_field(ch+")"),context,methodparams={"method":
 ↪"quadrigram"},smoothing="both",lmbda=0.2) for ch in choices]
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
        return np.random.choice(bestchoices)

    def predict(self,lm,method):
        if method=="chooseA":
            return self.chooseA()
        elif method=="random":
            return self.chooserandom()
        elif method=="unigram":
            return self.chooseunigram(lm=lm)
        elif method=="bigram":
            return self.choosebigram(lm=lm)
        elif method=="trigram":
            return self.choosetrigram(lm=lm)
        elif method=="quadrigram":
          return self.choosequadrigram(lm=lm)
```

[ ]:
```
lm = language_model(trainingdir=trainingdir,files=training)
```

[ ]:
```
SCC_test = scc_reader()
quadri_average_200_02 = []
for i in range(10):
  acc = SCC_test.predict_and_score(method="quadrigram",lm=lm)
  print(acc)
  quadri_average_200_02.append(acc)


print("Score for quadrigram",sum(quadri_average_200_02)/
 ↪len(quadri_average_200_02))
```

Both smoothing techniques with known=500 and lambda=1

[ ]:
```
class question(question):
    def chooseunigram(self,lm):
        choices=["a","b","c","d","e"]
        probs=[lm.get_prob(self.get_field(ch+")"),0,methodparams={"method":
 ↪"unigram"},smoothing="both",lmbda=1) for ch in choices]
```

```
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
        return np.random.choice(bestchoices)


    def choosebigram(self,lm,choices=[]):
        if choices==[]:
            choices=["a","b","c","d","e"]
        context=self.get_left_context(window=1)
        probs=[lm.get_prob(self.
↪get_field(ch+")"),context,methodparams={"method":
↪"bigram"},smoothing="both",lmbda=1) for ch in choices]
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
        return np.random.choice(bestchoices)

    def choosetrigram(self,lm,choices=[]):
        if choices==[]:
            choices=["a","b","c","d","e"]
        context=self.get_left_context(window=2)
        probs=[lm.get_prob(self.
↪get_field(ch+")"),context,methodparams={"method":
↪"trigram"},smoothing="both",lmbda=1) for ch in choices]
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
        return np.random.choice(bestchoices)

    def choosequadrigram(self,lm,choices=[]):
        if choices==[]:
            choices=["a","b","c","d","e"]
        context=self.get_left_context(window=3)
        probs=[lm.get_prob(self.
↪get_field(ch+")"),context,methodparams={"method":
↪"quadrigram"},smoothing="both",lmbda=1) for ch in choices]
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
        return np.random.choice(bestchoices)

    def predict(self,lm,method):
        if method=="chooseA":
            return self.chooseA()
        elif method=="random":
            return self.chooserandom()
        elif method=="unigram":
            return self.chooseunigram(lm=lm)
        elif method=="bigram":
            return self.choosebigram(lm=lm)
```

```
        elif method=="trigram":
            return self.choosetrigram(lm=lm)
        elif method=="quadrigram":
          return self.choosequadrigram(lm=lm)
```

[ ]:
```
SCC_test = scc_reader()
quadri_average_500_1 = []
for i in range(10):
  acc = SCC_test.predict_and_score(method="quadrigram",lm=lm)
  print(acc)
  quadri_average_500_1.append(acc)

print("Score for quadrigram",sum(quadri_average_500_1)/
 ↪len(quadri_average_500_1))
```

Both smoothing techniques with known=1000 and lambda=0.2

[ ]:
```
class language_model(language_model):
  def _make_unknowns(self,known=1000):
        unknown=0
        self.number_unknowns=0
        for (k,v) in list(self.unigram.items()):
            if v<known:
                del self.unigram[k]
                self.unigram["__UNK"]=self.unigram.get("__UNK",0)+v
                self.number_unknowns+=1

        for (k,adict) in list(self.bigram.items()):
            for (kk,v) in list(adict.items()):
                isknown=self.unigram.get(kk,0)
                if isknown==0 and not kk=="__DISCOUNT":
                    adict["__UNK"]=adict.get("__UNK",0)+v
                    del adict[kk]
            isknown=self.unigram.get(k,0)
            if isknown==0:
                del self.bigram[k]
                current=self.bigram.get("__UNK",{})
                current.update(adict)
                self.bigram["__UNK"]=current
            else:
                self.bigram[k]=adict

        for (k,adict) in list(self.trigram.items()):
            for (kk,v) in list(adict.items()):
                isknown=self.unigram.get(kk,0)
                if isknown==0 and not kk=="__DISCOUNT":
                    adict["__UNK"]=adict.get("__UNK",0)+v
```

```
                    del adict[kk]
            for word in k:
              isknown=self.unigram.get(word,0)
              if isknown==0:
                del self.trigram[k]
                k = list(k)
                k[k.index(word)] = "__UNK"
                k = tuple(k)
                current=self.trigram.get(k,{})
                current.update(adict)
                self.trigram[k]=current
              else:
                self.trigram[k]=adict

class question(question):
    def chooseunigram(self,lm):
        choices=["a","b","c","d","e"]
        probs=[lm.get_prob(self.get_field(ch+")"),0,methodparams={"method":
↪"unigram"},smoothing="both",lmbda=0.2) for ch in choices]
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
        return np.random.choice(bestchoices)


    def choosebigram(self,lm,choices=[]):
        if choices==[]:
            choices=["a","b","c","d","e"]
        context=self.get_left_context(window=1)
        probs=[lm.get_prob(self.
↪get_field(ch+")"),context,methodparams={"method":
↪"bigram"},smoothing="both",lmbda=0.2) for ch in choices]
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
        return np.random.choice(bestchoices)

    def choosetrigram(self,lm,choices=[]):
        if choices==[]:
            choices=["a","b","c","d","e"]
        context=self.get_left_context(window=2)
        probs=[lm.get_prob(self.
↪get_field(ch+")"),context,methodparams={"method":
↪"trigram"},smoothing="both",lmbda=0.2) for ch in choices]
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
        return np.random.choice(bestchoices)

    def choosequadrigram(self,lm,choices=[]):
```

```
        if choices==[]:
            choices=["a","b","c","d","e"]
        context=self.get_left_context(window=3)
        probs=[lm.get_prob(self.
 ↪get_field(ch+")"),context,methodparams={"method":
 ↪"quadrigram"},smoothing="both",lmbda=0.2) for ch in choices]
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
        return np.random.choice(bestchoices)

    def predict(self,lm,method):
        if method=="chooseA":
            return self.chooseA()
        elif method=="random":
            return self.chooserandom()
        elif method=="unigram":
            return self.chooseunigram(lm=lm)
        elif method=="bigram":
            return self.choosebigram(lm=lm)
        elif method=="trigram":
            return self.choosetrigram(lm=lm)
        elif method=="quadrigram":
          return self.choosequadrigram(lm=lm)
```

[ ]:
```
lm = language_model(trainingdir=trainingdir,files=training)
```

[ ]:
```
SCC_test = scc_reader()
quadri_average_1000_02 = []
for i in range(10):
  acc = SCC_test.predict_and_score(method="quadrigram",lm=lm)
  print(acc)
  quadri_average_1000_02.append(acc)

print("Score for quadrigram",sum(quadri_average_1000_02)/
 ↪len(quadri_average_1000_02))
```

Both smoothing techniques with known=1000 and lambda=1

[ ]:
```
class question(question):
    def chooseunigram(self,lm):
        choices=["a","b","c","d","e"]
        probs=[lm.get_prob(self.get_field(ch+")"),0,methodparams={"method":
 ↪"unigram"},smoothing="both",lmbda=1) for ch in choices]
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
        return np.random.choice(bestchoices)
```

```
    def choosebigram(self,lm,choices=[]):
        if choices==[]:
            choices=["a","b","c","d","e"]
        context=self.get_left_context(window=1)
        probs=[lm.get_prob(self.
↪get_field(ch+")"),context,methodparams={"method":
↪"bigram"},smoothing="both",lmbda=1) for ch in choices]
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
        return np.random.choice(bestchoices)

    def choosetrigram(self,lm,choices=[]):
        if choices==[]:
            choices=["a","b","c","d","e"]
        context=self.get_left_context(window=2)
        probs=[lm.get_prob(self.
↪get_field(ch+")"),context,methodparams={"method":
↪"trigram"},smoothing="both",lmbda=1) for ch in choices]
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
        return np.random.choice(bestchoices)

    def choosequadrigram(self,lm,choices=[]):
        if choices==[]:
            choices=["a","b","c","d","e"]
        context=self.get_left_context(window=3)
        probs=[lm.get_prob(self.
↪get_field(ch+")"),context,methodparams={"method":
↪"quadrigram"},smoothing="both",lmbda=1) for ch in choices]
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
        return np.random.choice(bestchoices)

    def predict(self,lm,method):
        if method=="chooseA":
            return self.chooseA()
        elif method=="random":
            return self.chooserandom()
        elif method=="unigram":
            return self.chooseunigram(lm=lm)
        elif method=="bigram":
            return self.choosebigram(lm=lm)
        elif method=="trigram":
            return self.choosetrigram(lm=lm)
        elif method=="quadrigram":
          return self.choosequadrigram(lm=lm)
```

```
[ ]: SCC_test = scc_reader()
quadri_average_1000_1 = []
for i in range(10):
  acc = SCC_test.predict_and_score(method="quadrigram",lm=lm)
  print(acc)
  quadri_average_1000_1.append(acc)

print("Score for quadrigram",sum(quadri_average_1000_1)/
 ↪len(quadri_average_1000_1))
```

4.0.4 Recurrent Neural Network Language Model

Create quadrigram and word embeddings

```
[ ]: def create_quadrigram_set(files,training_dir):
  alltokens=["__END"]
  start=False
  for afile in files:
    print("Processing {}".format(afile))
    try:
      with open(os.path.join(training_dir,afile)) as instream:
        for line in instream:
          line=line.rstrip()
          if len(line)>0 and start:
            if "end of project gutenberg" not in line.lower() and "end of the␣
 ↪project gutenberg" not in line.lower():
              tokens=["__END","__START"]+tokenize(line)+["__END"]
              alltokens+=tokens
          if "*END*" in line:
            start = True
      start = False
    except UnicodeDecodeError:
      print("UnicodeDecodeError processing {}: ignoring file".format(afile))

  vocab={}
  threshold=20
  for token in alltokens:
    vocab[token]=vocab.get(token,0)+1
  unknowns=0
  for key,value in list(vocab.items()):
    if value < threshold:
      unknowns+=value
      vocab.pop(key,None)
  vocab["__UNK"]=unknowns

  word_to_ix = {word: i for i, word in enumerate(list(vocab.keys()))}
  ix_to_word = {i: word for i, word in enumerate(list(vocab.keys()))}
```

```
  filteredtokens=[]
  for token in alltokens:
    if token in vocab.keys():
      filteredtokens.append(token)
    else:
      filteredtokens.append("__UNK")
  quadrigrams = [([filteredtokens[i], filteredtokens[i + 1], filteredtokens[i +␣
 ↪2]], filteredtokens[i + 3])
        for i in range(len(filteredtokens) - 3)]
  return quadrigrams, vocab, word_to_ix, ix_to_word

QUADRIGRAM, VOCAB, WORD_TO_IX, IX_TO_WORD =␣
 ↪create_quadrigram_set(files=training,training_dir=trainingdir)
```

Creating Neural Language Model class

[ ]:
```
# torch.cuda.is_available() checks and returns a Boolean True if a GPU is␣
 ↪available, else it'll return False
is_cuda = torch.cuda.is_available()

# If we have a GPU available, we'll set our device to GPU. We'll use this␣
 ↪device variable later in our code.
if is_cuda:
    device = torch.device("cuda:0")
    print("GPU is available")
else:
    device = torch.device("cpu")
    print("GPU not available, CPU used")

CONTEXT_SIZE = 3
EMBEDDING_DIM = 1000

class NGramRecurrentLanguageModeler(nn.Module):
  def __init__(self, input_size, embedding_dim,hidden_size, output_size,␣
 ↪n_layers):
        super(NGramRecurrentLanguageModeler, self).__init__()

        # Defining some parameters
        self.embeddings = nn.Embedding(output_size, embedding_dim)
        self.hidden_size=hidden_size
        self.n_layers = n_layers

        #Defining the layers
        # RNN Layer
```

```
        self.rnn = nn.LSTM(input_size=input_size, hidden_size=self.hidden_size,␣
↪bidirectional=True, num_layers=n_layers, batch_first=True)   #,␣
↪bidirectional=True, num_layers=n_layers, batch_first=True
        # Fully connected layer
        self.fc = nn.Linear(self.hidden_size*2, output_size) #output_shape 3

 def forward(self, x):
        embeds = self.embeddings(x)
        batch_size = x.size(0)

        # Initializing hidden state for first input using method defined below
        hidden = self.init_hidden(batch_size)
        c0 = self.init_hidden(batch_size)

        # Passing in the input and hidden state into the model and obtaining␣
↪outputs
        x = x.view(*x.shape[:1], -1, *x.shape[3:])
        embeds = embeds.view(*embeds.shape[:1], -1, *embeds.shape[3:])
        out,_= self.rnn(embeds.unsqueeze(1),(hidden,c0)) #,hidden   embeds.
↪view(len(x)



        # Reshaping the outputs such that it can be fit into the fully␣
↪connected layer
        out = out.reshape(out.shape[0],-1)  #out.contiguous().view(-1,␣
↪len(VOCAB))
        out = self.fc(out)

        return out  # , hidden

 def init_hidden(self, batch_size):
        # This method generates the first hidden state of zeros which we'll use␣
↪in the forward pass
        # We'll send the tensor holding the hidden state to the device we␣
↪specified earlier as well
        hidden = torch.zeros(self.n_layers*2, batch_size, self.hidden_size).
↪to(device)
        return hidden
```

Training

[ ]:
```
from torch._C import dtype
# Instantiate the model with hyperparameters
```

```
nLanguageModel =␣
 ↪NGramRecurrentLanguageModeler(input_size=CONTEXT_SIZE*EMBEDDING_DIM,␣
 ↪embedding_dim=EMBEDDING_DIM,hidden_size=128, output_size=len(VOCAB),␣
 ↪n_layers=1)
# We'll also set the model to the device that we defined earlier (default is␣
 ↪CPU)
nLanguageModel.to(device)

x_train = []
y_train = []

for context, target in QUADRIGRAM:

  x_train.append([WORD_TO_IX[w] for w in context])

  y_train.append([WORD_TO_IX[target]])


x_train = torch.tensor(x_train,dtype=torch.long).to(device)
y_train = torch.tensor(y_train,dtype=torch.long).to(device)



class WordToIxDataset(Dataset):
  def __init__(self, X, Y):
    self.X = X
    self.Y = Y
    if len(self.X) != len(self.Y):
      raise Exception("The length of X does not match the length of Y")

  def __len__(self):
    return len(self.X)

  def __getitem__(self, index):
    # note that this isn't randomly selecting. It's a simple get a single item␣
 ↪that represents an x and y
    _x = self.X[index]
    _y = self.Y[index]
    return _x, _y

loader = DataLoader(WordToIxDataset(x_train, y_train), batch_size=8192,␣
 ↪shuffle=True, num_workers=0)
print('Training set has {} instances'.format(len(loader)))
```

[ ]:
```
train_loader_iter = iter(loader)
imgs, labels = next(train_loader_iter)
print(imgs.shape)
```

```
print(labels.shape)
```

```
torch.Size([8192, 3])
torch.Size([8192, 1])
```

[ ]:
```
from torch._C import dtype
# Instantiate the model with hyperparameters
nLanguageModel =␣
 ↪NGramRecurrentLanguageModeler(input_size=CONTEXT_SIZE*EMBEDDING_DIM,␣
 ↪embedding_dim=EMBEDDING_DIM, hidden_size=128,output_size=len(VOCAB),␣
 ↪n_layers=1)
# We'll also set the model to the device that we defined earlier (default is␣
 ↪CPU)
nLanguageModel.to(device)

# Define hyperparameters
n_epochs = 25
lr=0.01

# Define Loss, Optimizer
loss_function = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(nLanguageModel.parameters(), lr=lr)

losses_001=[]
running_loss = 0.
last_loss = 0.

# Training Run
for epoch in range(1, n_epochs + 1):
  total_loss = 0
  for i, (context_x, target_y) in enumerate(loader):

    optimizer.zero_grad() # Clears existing gradients from previous epoch

    output = nLanguageModel(context_x.to(device)) #, hidden

    loss = loss_function(output, target_y.flatten().to(device))
    del output
    torch.cuda.empty_cache()

    loss.backward() # Does backpropagation and calculates gradients
    optimizer.step() # Updates the weights accordingly

    running_loss += loss.item()
    if i % 1000 == 999:
      last_loss = running_loss / 1000 # loss per batch
```

```
      print('  batch {} loss: {}'.format(i + 1, last_loss))
      tb_x = epoch * len(loader) + i + 1
      print('Loss/train', last_loss, tb_x)
      running_loss = 0.

    total_loss += loss.item()
  losses_001.append(total_loss)
  print('Epoch: {}/{}.............'.format(epoch, n_epochs), end=' ')
  print(losses_001)
```

[ ]:
```
from torch._C import dtype
# Instantiate the model with hyperparameters
nLanguageModel =␣
 ↪NGramRecurrentLanguageModeler(input_size=CONTEXT_SIZE*EMBEDDING_DIM,␣
 ↪embedding_dim=EMBEDDING_DIM, hidden_size=128,output_size=len(VOCAB),␣
 ↪n_layers=1)
# We'll also set the model to the device that we defined earlier (default is␣
 ↪CPU)
nLanguageModel.to(device)

# Define hyperparameters
n_epochs = 25
lr=0.001

# Define Loss, Optimizer
loss_function = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(nLanguageModel.parameters(), lr=lr)

losses_0001=[]
running_loss = 0.
last_loss = 0.

# Training Run
for epoch in range(1, n_epochs + 1):
  total_loss = 0
  for i, (context_x, target_y) in enumerate(loader):

    optimizer.zero_grad() # Clears existing gradients from previous epoch

    output = nLanguageModel(context_x.to(device)) #, hidden

    loss = loss_function(output, target_y.flatten().to(device))
    del output
    torch.cuda.empty_cache()

    loss.backward() # Does backpropagation and calculates gradients
    optimizer.step() # Updates the weights accordingly
```

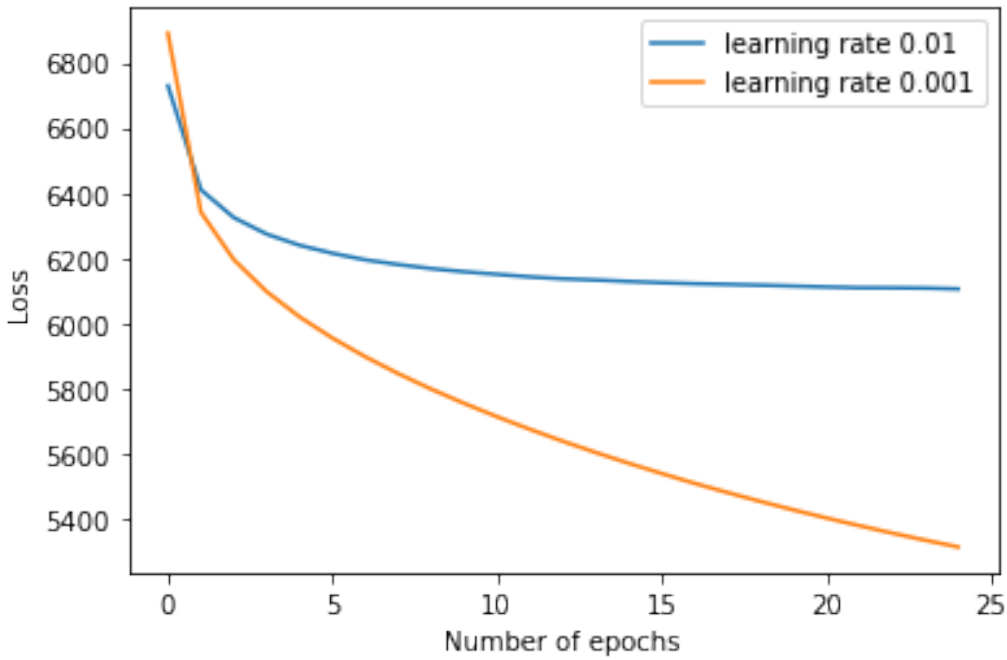
```
    running_loss += loss.item()
    if i % 1000 == 999:
      last_loss = running_loss / 1000 # loss per batch
      print('  batch {} loss: {}'.format(i + 1, last_loss))
      tb_x = epoch * len(loader) + i + 1
      print('Loss/train', last_loss, tb_x)
      running_loss = 0.

    total_loss += loss.item()
  losses_0001.append(total_loss)
  print('Epoch: {}/{}.............'.format(epoch, n_epochs), end=' ')
  print(losses_0001)
```

[ ]:
```
import matplotlib.pyplot as plt
plt.plot(losses_001, label='learning rate 0.01')
plt.plot(losses_0001, label='learning rate 0.001')

plt.xlabel("Number of epochs")
plt.ylabel("Loss")
plt.legend()

plt.show()
```



Testing

[ ]:
```
def get_logprob(context,target):
  #return the logprob of the target word given the context
  w_in_contxt=[]
  for w in context:
    if w in WORD_TO_IX.keys():
      w_in_contxt.append(WORD_TO_IX[w])
    else:
      w_in_contxt.append(WORD_TO_IX["__UNK"])
  context_idxs = torch.tensor(w_in_contxt, dtype=torch.long)
  context_idxs = context_idxs[None,:]
  log_probs= nLanguageModel.forward(context_idxs.to(device))
  if target not in WORD_TO_IX.keys():
    target_idx=torch.tensor(WORD_TO_IX["__UNK"])
  else:
    target_idx=torch.tensor(WORD_TO_IX[target])
  return log_probs.index_select(1,target_idx.to(device)).item()

class question(question):
    def choosenmodel(self):
        choices=["a","b","c","d","e"]
        context=self.get_left_context(window=3)
        probs=[get_logprob(context=context,target=self.get_field(ch+")")) for␣
 ↪ch in choices]
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
        return np.random.choice(bestchoices)

    def predict(self,lm,method):
        if method=="chooseA":
            return self.chooseA()
        elif method=="random":
            return self.chooserandom()
        elif method=="unigram":
            return self.chooseunigram(lm=lm)
        elif method=="bigram":
            return self.choosebigram(lm=lm)
        elif method=="trigram":
            return self.choosetrigram(lm=lm)
        elif method=="quadrigram":
          return self.choosequadrigram(lm=lm)
        elif method=="nmodel":
          return self.choosenmodel()
```

[ ]:
```
from torch._C import dtype
# Instantiate the model with hyperparameters
```

```
nLanguageModel =␣
 ↪NGramRecurrentLanguageModeler(input_size=CONTEXT_SIZE*EMBEDDING_DIM,␣
 ↪embedding_dim=EMBEDDING_DIM, hidden_size=32,output_size=len(VOCAB),␣
 ↪n_layers=1)
# We'll also set the model to the device that we defined earlier (default is␣
 ↪CPU)
nLanguageModel.to(device)

# Define hyperparameters
n_epochs = 25
lr=0.001

# Define Loss, Optimizer
loss_function = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(nLanguageModel.parameters(), lr=lr)

losses_0001=[]
running_loss = 0.
last_loss = 0.

# Training Run
for epoch in range(1, n_epochs + 1):
  total_loss = 0
  for i, (context_x, target_y) in enumerate(loader):

    optimizer.zero_grad() # Clears existing gradients from previous epoch

    output = nLanguageModel(context_x.to(device)) #, hidden

    loss = loss_function(output, target_y.flatten().to(device))
    del output
    torch.cuda.empty_cache()

    loss.backward() # Does backpropagation and calculates gradients
    optimizer.step() # Updates the weights accordingly

    running_loss += loss.item()
    if i % 1000 == 999:
      last_loss = running_loss / 1000 # loss per batch
      print('  batch {} loss: {}'.format(i + 1, last_loss))
      tb_x = epoch * len(loader) + i + 1
      print('Loss/train', last_loss, tb_x)
      running_loss = 0.

    total_loss += loss.item()
  losses_0001.append(total_loss)
  print('Epoch: {}/{}.............'.format(epoch, n_epochs), end=' ')
```

```
  print(losses_0001)
```

[ ]:
```
SCC_test = scc_reader()
lm=None

nn_0001_32 = []
for i in range(10):
  acc = SCC_test.predict_and_score(method="nmodel",lm=lm)
  print(acc)
  nn_0001_32.append(acc)


print("Score for quadrigram",sum(nn_0001_32)/len(nn_0001_32))
```

[ ]:
```
from torch._C import dtype
# Instantiate the model with hyperparameters
nLanguageModel =␣
 ↪NGramRecurrentLanguageModeler(input_size=CONTEXT_SIZE*EMBEDDING_DIM,␣
 ↪embedding_dim=EMBEDDING_DIM, hidden_size=32,output_size=len(VOCAB),␣
 ↪n_layers=1)
# We'll also set the model to the device that we defined earlier (default is␣
 ↪CPU)
nLanguageModel.to(device)

# Define hyperparameters
n_epochs = 25
lr=0.01

# Define Loss, Optimizer
loss_function = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(nLanguageModel.parameters(), lr=lr)

losses_0001=[]
running_loss = 0.
last_loss = 0.

# Training Run
for epoch in range(1, n_epochs + 1):
  total_loss = 0
  for i, (context_x, target_y) in enumerate(loader):

    optimizer.zero_grad() # Clears existing gradients from previous epoch

    output = nLanguageModel(context_x.to(device)) #, hidden

    loss = loss_function(output, target_y.flatten().to(device))
    del output
```

```
    torch.cuda.empty_cache()

    loss.backward() # Does backpropagation and calculates gradients
    optimizer.step() # Updates the weights accordingly

    running_loss += loss.item()
    if i % 1000 == 999:
      last_loss = running_loss / 1000 # loss per batch
      print('  batch {} loss: {}'.format(i + 1, last_loss))
      tb_x = epoch * len(loader) + i + 1
      print('Loss/train', last_loss, tb_x)
      running_loss = 0.

    total_loss += loss.item()
  losses_0001.append(total_loss)
  print('Epoch: {}/{}.............'.format(epoch, n_epochs), end=' ')
  print(losses_0001)
```

[ ]:
```
SCC_test = scc_reader()
lm=None

nn_001_32 = []
for i in range(10):
  acc = SCC_test.predict_and_score(method="nmodel",lm=lm)
  print(acc)
  nn_001_32.append(acc)


print("Score for quadrigram",sum(nn_001_32)/len(nn_001_32))
```

[ ]:
```
from torch._C import dtype
# Instantiate the model with hyperparameters
nLanguageModel =␣
 ↪NGramRecurrentLanguageModeler(input_size=CONTEXT_SIZE*EMBEDDING_DIM,␣
 ↪embedding_dim=EMBEDDING_DIM, hidden_size=128,output_size=len(VOCAB),␣
 ↪n_layers=1)
# We'll also set the model to the device that we defined earlier (default is␣
 ↪CPU)
nLanguageModel.to(device)

# Define hyperparameters
n_epochs = 25
lr=0.001

# Define Loss, Optimizer
loss_function = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(nLanguageModel.parameters(), lr=lr)
```

```
losses_0001=[]
running_loss = 0.
last_loss = 0.

# Training Run
for epoch in range(1, n_epochs + 1):
  total_loss = 0
  for i, (context_x, target_y) in enumerate(loader):

    optimizer.zero_grad() # Clears existing gradients from previous epoch

    output = nLanguageModel(context_x.to(device)) #, hidden

    loss = loss_function(output, target_y.flatten().to(device))
    del output
    torch.cuda.empty_cache()

    loss.backward() # Does backpropagation and calculates gradients
    optimizer.step() # Updates the weights accordingly

    running_loss += loss.item()
    if i % 1000 == 999:
      last_loss = running_loss / 1000 # loss per batch
      print('  batch {} loss: {}'.format(i + 1, last_loss))
      tb_x = epoch * len(loader) + i + 1
      print('Loss/train', last_loss, tb_x)
      running_loss = 0.

    total_loss += loss.item()
  losses_0001.append(total_loss)
  print('Epoch: {}/{}.............'.format(epoch, n_epochs), end=' ')
  print(losses_0001)
```

[ ]:
```
SCC_test = scc_reader()
lm=None

nn_0001_128 = []
for i in range(10):
  acc = SCC_test.predict_and_score(method="nmodel",lm=lm)
  print(acc)
  nn_0001_128.append(acc)


print("Score for quadrigram",sum(nn_0001_128)/len(nn_0001_128))
```

```
[ ]: from torch._C import dtype
     # Instantiate the model with hyperparameters
     nLanguageModel =␣
      ↪NGramRecurrentLanguageModeler(input_size=CONTEXT_SIZE*EMBEDDING_DIM,␣
      ↪embedding_dim=EMBEDDING_DIM, hidden_size=128,output_size=len(VOCAB),␣
      ↪n_layers=1)
     # We'll also set the model to the device that we defined earlier (default is␣
      ↪CPU)
     nLanguageModel.to(device)

     # Define hyperparameters
     n_epochs = 25
     lr=0.01

     # Define Loss, Optimizer
     loss_function = nn.CrossEntropyLoss()
     optimizer = torch.optim.Adam(nLanguageModel.parameters(), lr=lr)

     losses_0001=[]
     running_loss = 0.
     last_loss = 0.

     # Training Run
     for epoch in range(1, n_epochs + 1):
       total_loss = 0
       for i, (context_x, target_y) in enumerate(loader):

         optimizer.zero_grad() # Clears existing gradients from previous epoch

         output = nLanguageModel(context_x.to(device)) #, hidden

         loss = loss_function(output, target_y.flatten().to(device))
         del output
         torch.cuda.empty_cache()

         loss.backward() # Does backpropagation and calculates gradients
         optimizer.step() # Updates the weights accordingly

         running_loss += loss.item()
         if i % 1000 == 999:
           last_loss = running_loss / 1000 # loss per batch
           print('  batch {} loss: {}'.format(i + 1, last_loss))
           tb_x = epoch * len(loader) + i + 1
           print('Loss/train', last_loss, tb_x)
           running_loss = 0.

         total_loss += loss.item()
```

```
  losses_0001.append(total_loss)
  print('Epoch: {}/{}.............'.format(epoch, n_epochs), end=' ')
  print(losses_0001)
```

[ ]:
```
SCC_test = scc_reader()
lm=None

nn_001_128 = []
for i in range(10):
  acc = SCC_test.predict_and_score(method="nmodel",lm=lm)
  print(acc)
  nn_001_128.append(acc)


print("Score for quadrigram",sum(nn_001_128)/len(nn_001_128))
```

[ ]:
```
from torch._C import dtype
# Instantiate the model with hyperparameters
nLanguageModel =␣
 ↪NGramRecurrentLanguageModeler(input_size=CONTEXT_SIZE*EMBEDDING_DIM,␣
 ↪embedding_dim=EMBEDDING_DIM, hidden_size=256,output_size=len(VOCAB),␣
 ↪n_layers=1)
# We'll also set the model to the device that we defined earlier (default is␣
 ↪CPU)
nLanguageModel.to(device)

# Define hyperparameters
n_epochs = 25
lr=0.001

# Define Loss, Optimizer
loss_function = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(nLanguageModel.parameters(), lr=lr)

losses_0001=[]
running_loss = 0.
last_loss = 0.

# Training Run
for epoch in range(1, n_epochs + 1):
  total_loss = 0
  for i, (context_x, target_y) in enumerate(loader):

    optimizer.zero_grad() # Clears existing gradients from previous epoch

    output = nLanguageModel(context_x.to(device)) #, hidden
```

```
    loss = loss_function(output, target_y.flatten().to(device))
    del output
    torch.cuda.empty_cache()

    loss.backward() # Does backpropagation and calculates gradients
    optimizer.step() # Updates the weights accordingly

    running_loss += loss.item()
    if i % 1000 == 999:
      last_loss = running_loss / 1000 # loss per batch
      print('  batch {} loss: {}'.format(i + 1, last_loss))
      tb_x = epoch * len(loader) + i + 1
      print('Loss/train', last_loss, tb_x)
      running_loss = 0.

    total_loss += loss.item()
  losses_0001.append(total_loss)
  print('Epoch: {}/{}.............'.format(epoch, n_epochs), end=' ')
  print(losses_0001)
```

[ ]:
```
SCC_test = scc_reader()
lm=None

nn_0001_256 = []
for i in range(10):
  acc = SCC_test.predict_and_score(method="nmodel",lm=lm)
  print(acc)
  nn_0001_256.append(acc)


print("Score for quadrigram",sum(nn_0001_256)/len(nn_0001_256))
```

[ ]:
```
from torch._C import dtype
# Instantiate the model with hyperparameters
nLanguageModel =␣
 ↪NGramRecurrentLanguageModeler(input_size=CONTEXT_SIZE*EMBEDDING_DIM,␣
 ↪embedding_dim=EMBEDDING_DIM, hidden_size=256,output_size=len(VOCAB),␣
 ↪n_layers=1)
# We'll also set the model to the device that we defined earlier (default is␣
 ↪CPU)
nLanguageModel.to(device)

# Define hyperparameters
n_epochs = 25
lr=0.01

# Define Loss, Optimizer
```

```
loss_function = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(nLanguageModel.parameters(), lr=lr)

losses_001=[]
running_loss = 0.
last_loss = 0.

# Training Run
for epoch in range(1, n_epochs + 1):
  total_loss = 0
  for i, (context_x, target_y) in enumerate(loader):

    optimizer.zero_grad() # Clears existing gradients from previous epoch

    output = nLanguageModel(context_x.to(device)) #, hidden

    loss = loss_function(output, target_y.flatten().to(device))
    del output
    torch.cuda.empty_cache()

    loss.backward() # Does backpropagation and calculates gradients
    optimizer.step() # Updates the weights accordingly

    running_loss += loss.item()
    if i % 1000 == 999:
      last_loss = running_loss / 1000 # loss per batch
      print('  batch {} loss: {}'.format(i + 1, last_loss))
      tb_x = epoch * len(loader) + i + 1
      print('Loss/train', last_loss, tb_x)
      running_loss = 0.

    total_loss += loss.item()
  losses_001.append(total_loss)
  print('Epoch: {}/{}.............'.format(epoch, n_epochs), end=' ')
  print(losses_001)
```

[ ]:
```
SCC_test = scc_reader()
lm=None

nn_001_256 = []
for i in range(10):
  acc = SCC_test.predict_and_score(method="nmodel",lm=lm)
  print(acc)
  nn_001_256.append(acc)


print("Score for quadrigram",sum(nn_001_256)/len(nn_001_256))
```

5 Failure cases

[ ]:
```
SCC = scc_reader()
SCC.predict_and_score(method="quadrigram",lm=lm)
```

[ ]:
```
nLanguageModel.to(device)
SCC_test = scc_reader()
lm=None

n0n_001_4 = []
for i in range(10):
  acc = SCC_test.predict_and_score(method="nmodel",lm=lm)
  print(acc)
  n0n_001_4.append(acc)


print("Score for quadrigram",sum(n0n_001_4)/len(n0n_001_4))
```

[ ]:
```
pd.options.display.max_colwidth = 500
questions=os.path.join(parentdir,"testing_data.csv")
answers=os.path.join(parentdir,"test_answer.csv")

#Visualise the sentences and their possibilities
with open(questions) as instream:
    csvreader=csv.reader(instream)
    lines=list(csvreader)
qs_model_df=pd.DataFrame(lines[1:],columns=lines[0])

#Visualise the answers to complete the sentence
with open(answers) as instream:
    csvreader=csv.reader(instream)
    lines=list(csvreader)


qs_model_df["answers"] = lines[1:]
qs_model_df["quadrigram_pred"]=SCC.predict(method="quadrigram",lm=lm)
qs_df["RNN_pred"]=SCC.predict(method="nmodel",lm=None)
qs_model_df
```

[ ]:
```
         id  \
0         1
1         2
2         3
3         4
4         5
…     …
1035  1036
```

```
1036  1037
1037  1038
1038  1039
1039  1040

                                                        question  \
0                                      I have it from the same source that you
are both an orphan and a bachelor and are _____ alone in London.
1                            It was furnished partly as a sitting and partly as a
bedroom , with flowers arranged _____ in every nook and corner.
2                                       As I descended , my old ally , the
_____ , came out of the room and closed the door tightly behind him.
3                                                          We got off , _____
our fare , and the trap rattled back on its way to Leatherhead.
4                                                           He held in his
hand a _____ of blue paper , scrawled over with notes and figures.
…
…
1035                           The bedrooms in this _____ are on the ground floor
, the sitting-rooms being in the central block of the buildings.
1036                                  Our visitor bore every mark of being an
average commonplace British tradesman , obese , _____ , and slow.
1037  The terror of his face lay in his eyes , however , steel gray , and
glistening coldly with a malignant , inexorable _____ in their depths.
1038             It is your commonplace , _____ crimes which are really puzzling
, just as a commonplace face is the most difficult to identify.
1039         On the last occasion he had _____ that if my friend would only come
with me he would be glad to extend his hospitality to him also.

              a)              b)            c)          d)             e)  \
0          crying  instantaneously      residing     matched        walking
1        daintily       privately  inadvertently   miserably    comfortably
2            gods            moon       panther       guard  country-dance
3         rubbing        doubling          paid      naming       carrying
4          supply          parcel          sign       sheet         chorus
…            …               …             …          …             …
1035         wing           coach       balcony     kingdom  neighbourhood
1036        blind       energetic      eloquent     pompous   sandy-haired
1037      cruelty       novitiate       justice      broker        success
1038   underlying     featureless   theological  flattering     inevitable
1039     believed          proved     discovered    remarked        dreamed

         answers quadrigram_pred
0         [1, c]              b
1         [2, a]              e
2         [3, d]              b
3         [4, c]              e
```

```
4         [5, d]                e
…          …                …
1035  [1036, a]                d
1036  [1037, d]                a
1037  [1038, a]                a
1038  [1039, b]                c
1039  [1040, d]                e

[1040 rows x 9 columns]
```

[ ]:
```
SCC = scc_reader()
qs_model_df["RNN_pred"]=SCC.predict(method="nmodel",lm=None)
```

[ ]:
```
qs_model_df.loc[51:100]
```

[ ]:
```
      id  \
51    52
52    53
53    54
54    55
55    56
56    57
57    58
58    59
59    60
60    61
61    62
62    63
63    64
64    65
65    66
66    67
67    68
68    69
69    70
70    71
71    72
72    73
73    74
74    75
75    76
76    77
77    78
78    79
79    80
80    81
81    82
```

82 83
83 84
84 85
85 86
86 87
87 88
88 89
89 90
90 91
91 92
92 93
93 94
94 95
95 96
96 97
97 98
98 99
99 100
100 101

question \
51
It was one of the main arteries which _____ the traffic of the City to the north and west.
52 I stooped in some confusion and began to pick up the fruit , understanding for some reason my companion desired me to take the _____ upon myself.
53
He had a very dark , _____ face , and a gleam in his eyes that comes back to me in my dreams.
54
The fat man _____ his eyes round , and then up at the open skylight.
55 We passed up the stair , _____ the door , followed on down a passage , and found ourselves in front of the barricade which Miss Hunter had described.
56
Holmes was for the moment as _____ as I. His hand closed like a vice upon my wrist in his agitation.
57
The chimney is wide , but is _____ up by four large staples.
58 My companion sat in the front of the trap , his arms folded , his hat _____ down over his eyes , and his chin sunk upon his breast , buried in the deepest thought.
59
He used to make _____ over the cleverness of women , but I have not heard him do it of late.
60 I have _____ , therefore , to call upon

you and to consult you in reference to the very painful event which has occurred in connection with my wedding.
61 As I passed out through the wicket gate , however , I found my acquaintance of the morning waiting in the _____ upon the other side.
62
A low _____ had fallen upon our ears.
63
I realized it as I _____ back and noted how hill after hill showed traces of the ancient people.
64
You must find your own ink , pens , and blotting-paper , but we _____ this table and chair.
65
I hope to _____ that he has gone , for he has brought nothing but trouble here.
66 It threw a livid , unnatural circle upon the floor , while in the _____ beyond we saw the vague loom of two figures which crouched against the wall.
67
Perhaps our _____ now may do something to make it less obscure.
68
That was bad enough , for all that the _____ said.
69
He had heard nothing , and the _____ remained a complete mystery.
70
Then he _____ over the hill.
71
Not a _____ , not a rustle , rose now from the dark figure over which we stooped.
72 I mean to teach them in these parts that law is law , and that there is a man here who does not fear to _____ it.
73
I sat down upon a _____ in the corner and thought the whole matter carefully over.
74 The inspector hurried away on the instant to make _____ about the page , while Holmes and I returned to Baker Street for breakfast.
75 She could trust her own guardianship , but she could not tell what _____ or political influence might be brought to bear upon a business man.
76 When his body had been carried from the cellar we found ourselves still confronted with a problem which was almost as _____ as that with which we had started.
77
The back door was open , and as he came to the foot of the _____ he saw two men wrestling together outside.

78
I read nothing except the criminal _____ and the agony column.
79
I confess that they quite _____ my expectations , and that I am utterly unable to account for your result.
80
Was there a police-station _____ near.
81
I do not think that I have ever seen so _____ a man.
82 It was furred outside by a thick layer of dust , and damp and worms had eaten through the wood , so that a crop of livid fungi was _____ on the inside of it.
83
I don't think I ever _____ faster , but the others were there before us.
84
I have tried to _____ it from the measurements.
85
I am sorry to have _____ you.
86
At the end were the _____ of the high dignitaries who had signed it.
87
My companion noiselessly _____ the shutters , moved the lamp onto the table , and cast his eyes round the room.
88
The darkness was _____ , but much was still hidden by the shadows.
89
I was pained at the _____ , for I knew how keenly Holmes would feel any slip of the kind.
90 We had come out upon Oxford Street and I had ventured some remark as to this being a roundabout way to Kensington , when my words were _____ by the extraordinary conduct of my companion.
91 When I thought of the heavy _____ and looked at the gaping roof I understood how strong and immutable must be the purpose which had kept him in that inhospitable abode.
92 When leaving the house she was heard by the coachman to make some commonplace remark to her husband , and to _____ him that she would be back before very long.
93
Was she his _____ , his friend , or his mistress.
94 We had hardly reached the hall when we heard the _____ of a hound , and then a scream of agony , with a horrible worrying sound which it was dreadful to listen to.
95 A collection of my trifling achievements would certainly be _____ which contained no account of this very singular business.
96
His hair and whiskers were shot with gray , and his face was all crinkled and

_____ like a withered apple.
97 The point is a simple one , but the Inspector had _____ it because he had started with the supposition that these county magnates had had nothing to do with the matter.
98
He went over to the door , and turning the _____ he examined it in his methodical way.
99
My heart _____ within me as I saw it.
100 Another day two at the most and I have my case _____ , but until then guard your charge as closely as ever a fond mother watched her ailing child.

	a)	b)	c)	d)	e)	\
51	supported	surrounded	sheltered	conveyed	contained	
52	oath	message	blame	shirt	shadows	
53	rosy	childish	fearsome	colorless	yellow	
54	cast	folded	hastened	jingled	winked	
55	devoured	translated	unlocked	ascended	occupied	
56	startled	tired	thin	wise	clever	
57	torn	softened	higher	barred	constantly	
58	closed	slouched	wheeled	pulled	blown	
59	progress	matters	merry	advances	rules	
60	determined	related	overheard	absconded	noticed	
61	battle	chamber	incubator	shadow	spring	
62	courtship	struggle	blow	shrub	moan	
63	sailed	smiled	drove	shrunk	flowed	
64	behold	realised	provide	lose	owe	
65	heaven	explain	argue	inhabit	demand	
66	shadows	thicket	rushes	potage	field	
67	gate	weather	visit	limbs	skirts	
68	garret	saints	dog	coroner	birds	
69	affair	devil	snow	challenge	illusion	
70	vanished	bent	leant	wept	hovered	
71	scoundrel	muddle	soldier	whim	whisper	
72	pawn	contradict	invoke	store	subdue	
73	keg	sword	farm	fly	needle	
74	songs	pleasantries	mistakes	stories	inquiries	
75	sport	indirect	handsome	imprudence	comfort	
76	formidable	loud	quick	tall	invisible	
77	forest	parties	victory	stairs	mills	
78	courage	peace	news	nuts	plans	
79	visited	enjoyed	lightened	emptied	surpass	
80	lying	rolling	gathered	anywhere	comin'	
81	accurate	numerous	thin	provocative	marked	
82	resounding	sleeping	beheaded	running	growing	
83	choked	suspected	drove	remembered	conceived	

84	recommend	fling	accomplish	reconstruct	carry
85	killed	convinced	practised	expected	interrupted
86	signatures	relics	hearts	anxieties	portraits
87	toed	awaited	grasped	closed	mounted
88	rising	healed	ponderous	neglected	attractive
89	mistake	porch	fireplace	ceiling	pump
90	arrested	startled	enriched	perplexed	benumbed
91	step	sandwiches	breathing	rains	boxes
92	assure	detach	teach	dismiss	reproach
93	discomfiture	client	choice	musings	opportunity
94	image	clatter	baying	tinkle	click
95	incomplete	considered	audible	discovered	disembowelled
96	chattering	picturesque	hopeful	puckered	glistening
97	detected	baked	touched	rendered	overlooked
98	lamp	fish	lock	bucket	boulder
99	cruised	lies	lingered	leaped	struggled
100	moaning	arose	complete	tonight	mechanically

	answers	quadrigram_pred	RNN_pred
51	[52, d]	e	b
52	[53, c]	b	e
53	[54, c]	d	e
54	[55, a]	e	c
55	[56, c]	b	a
56	[57, a]	d	c
57	[58, d]	a	a
58	[59, d]	d	e
59	[60, c]	c	e
60	[61, a]	e	d
61	[62, d]	b	a
62	[63, e]	e	b
63	[64, c]	c	b
64	[65, c]	b	e
65	[66, a]	a	a
66	[67, a]	e	c
67	[68, c]	a	d
68	[69, d]	e	c
69	[70, a]	c	b
70	[71, a]	c	a
71	[72, e]	c	a
72	[73, c]	b	d
73	[74, a]	c	e
74	[75, e]	a	b
75	[76, b]	c	e
76	[77, a]	c	c
77	[78, d]	d	e
78	[79, c]	d	b

79	[80, e]	a	e
80	[81, d]	d	a
81	[82, c]	e	c
82	[83, e]	c	d
83	[84, c]	d	d
84	[85, d]	e	e
85	[86, e]	e	d
86	[87, a]	c	e
87	[88, d]	d	d
88	[89, a]	e	c
89	[90, a]	b	a
90	[91, a]	d	b
91	[92, d]	d	e
92	[93, a]	c	e
93	[94, b]	c	e
94	[95, c]	b	a
95	[96, a]	d	d
96	[97, d]	c	d
97	[98, e]	b	e
98	[99, c]	c	d
99	[100, d]	e	b
100	[101, c]	d	c