Microsoft Research Sentence Completion Challenge

Advanced Natural Language Engineering

Sumit Bajare

Word count:2980

Abstract

This report proposes an approach to the famous Natural Language Processing Challenge Microsoft Research Sentence Completion by examining diverse strategies for automatic sentence completion which consist of n-gram modelling and Continuous bag of words. The model is evaluated based on data taken from five Conan Doyle novels based on Sherlock Holmes, comprising of 1,040 sentences. This report confirms the legitimacy of the model on the information in Microsoft Sentence Completion Challenge. The trial results show that the proposed model, Continuous Bag of words outperforms N-gram model by 6%.

Keywords: N-gram, Continuous bag of words, Natural Language Processing.

Introduction

Sentence completion is a challenging semantic task which models the most suitable word to be chosen from a given set of data to finish the sentence. Microsoft Research has set a sentence completion challenge, which comprises of 1,040 sentences. The data was developed from Project Gutenberg, Seed sentences were chosen from Sherlock Holmes books and then imposter words were proposed with the aid of a language model trained on over 500 nineteenth century novels. Each one of the sentences in the data has four impostor sentences, in which a fixed word in the original sentence has been supplanted by an impostor word with comparable event statistics. Every sentence is accompanied by five candidate words. The objective of this challenge is to figure out which of the five decisions for that word is the most appropriate missing word. The words which make good probabilistic fit are selected. The difficulty in sentence completion is the final detail is to degree grammatical and semantic accuracy to pick the most appropriate sentence or phrase. Thus, in many ways, text completion epitomizes the goals of natural language understanding, as superficial encodings of meaning will be insufficient to determine which responses are accurate (Woods, 2016).

Fundamentally, text completion is a challenging semantic modelling problem, and solutions require models that can evaluate the global coherence of sentences. Notwithstanding its effortlessness, sentence completion can survey assorted capacities including linguistic proficiency, common knowledge, and logical reasoning at various levels. For each sentence the task is to conclude which of the options for

that express the proper one is. We will apply unique NLP techniques to do likewise. To date, several publications have evaluated reading comprehension models on sentence completion by using N-gram, Word similarity and recurrent neural network (RNN).

The main objective is to study, analyse and perform the task of sentence completion using text prediction. The aim is to investigated two language model on this challenge, N-gram model (Unigram, Bigram and Trigram) and Continuous bag of words (CBOW) and compare their accuracy and efficiency on the datasets. Accuracy is used as a performance metric because it was used by most of the authors that I have referred. The accuracy of these models when the optimal hyperparameter were found to achieve best performance are given below in the table 1.

Model	**Accuracy**
Unigram	24%
Bigram	22%
Trigram	26%
CBOW	34%

Table 1 Accuracy of each model

Initially, the report was planned to predict the words in the sentences using N-gram models and Latent Semantic Analysis Similarity (LSA). To improve on the n-gram model, absolute discount and Kneser-Ney smoothing techniques were used on the models. The research for Microsoft Research Sentence Completion Challenge suggested that among N-gram, Latent Semantic Analysis Similarity (LSA), Pointwise Mutual Information(PMI) and Recurrent Neural Network (RNN) models implementing Pointwise Mutual Information(PMI) will provide the highest accuracy. But there was no paper that implemented Continuous bag of words model to solve the problem, hence Continuous bag of words model was selected.

Related Work

Previous research put forward various architecture and techniques applied to Microsoft Research Sentence Completion Challenge (Burges, 2011), most common are N-gram, Latent Semantic Analysis Similarity (LSA), Pointwise Mutual Information (PMI) and Recurrent Neural Network (RNN). But there is less to non-implementation of the sentence completion using Continuous Bag of Words (CBOW). Recently BERT (Bidirectional Encoder Representations from Transformers) model researched by Google has become more popular. In the paper (Jacob Devlin Ming-Wei Chang Kenton Lee Kristina Toutanova, 2019) results show that a language model which is bidirectionally trained can have a deeper sense of language context and flow than single-direction language models. BERT's key technical innovation is applying the bidirectional training of Transformer, a popular attention model, to

language modelling. Whereases in other model a text sequence either from left to right or combined left-to-right and right-to-left training. Table 1 below show the models implemented in the researched paper.

Paper reference	Paper Name	Models & Accuracy
(Woods, 2016)	Exploiting Linguistic Features for Sentence Completion	N-gram (39%), LSA(49%),RNN(55%), skip-gram +RNN (59%), **PMI (61.44 %)**
(Tanushree C Hatmode,Prachi Y Duragkar, Radha H Khorgade, Ram B Jaiswal, Shubhashish S Charan, Rajesh Nasare, Hemant Turkar, 2021)	Sentence Completion Using NLP Techniques	RNN (43%), **LSA (54%)**
(Burges, 2011)	The Microsoft Research Sentence Completion Challenge	Generating Model (31%), Smoothed 3-gram (36%), Smoothed 4-gram (39%), Simple 4 gram (34%), **LSA (49%)**
(Geoffrey Zweig, John C. Platt Christopher Meek Christopher J.C. Burges, Ainur Yessenalina,Qiang Liu, 2012)	Computational Approaches to Sentence Completion	RNN (42), LSA(44), **LSA (53%)**

Methodology

N gram model

'Models that assign probabilities to sequences of words are called language models or LMs' (Daniel Jurafsky & James H. Martin, 2013). N-gram is simply a sequence of N words. For instance, San Francisco (is a 2-gram), The Three Musketeers (is a 3-gram) ,She stood up slowly (is a 4-gram) 'N-gram model predicts the occurrence of a word based on the occurrence of its N – 1 previous words' (Kumar, 2017). A bigram model (N = 2) predicts the occurrence of a word given only its previous word (N - 1= 1). Similarly, a trigram model (N = 3) predicts the occurrence of a word based on its previous two words (N – 1 = 2). This is the most basic and intuitive approach for text prediction.

To analyse the improvement in the system, smoothened n-gram model was applied using Kneser-Ney smoothing. Kneser-Ney advanced from absolute-discounting interpolation, which utilizes both higher-order and lower-order language models, redistributing some likelihood mass from 4-grams or 3-grams to less complex unigram models. The formulae for absolute-discounting smoothing as applied to a bigram language model is introduced underneath (Equation 1).

$$P_{abs}(w_i \mid w_{i-1}) = \frac{\max\left(c(w_{i-1}w_i) - \delta, 0\right)}{\sum_{w'} c(w_{i-1}w')} + \alpha\, P_{abs}(w_i)$$

Equation 1

In the above Equation 1 (Stanley F. Chen, Joshua Goodman, 1999), δ is the discount value, α is the normalizing constant.

The Kneser-Ney configuration holds the initial term of absolute discounting interpolation, yet changes the second term to exploit this relationship. Though absolute discounting interpolation in a bigram model would essentially default to a unigram model in the subsequent term, Kneser-Ney relies on the possibility of a continuation likelihood related with each unigram.

An example to display the adequacy of Kneser-Ney is the expression San Francisco, let's assume this expression is plentiful in a given preparation corpus. Then, at that point, the unigram likelihood of Francisco will likewise be high. Assuming we hastily use something like absolute discounting interpolation in a context where our bigram model is fragile, the unigram model portion might dominate and prompt a few strange outcomes. Kneser-Ney fixes this problem, unigram model simply provides how likely a word w_i is to appear, Kneser-Ney's second term determines how likely a word w_i is to appear in an unfamiliar bigram context. Kneser-Ney in whole follows (Equation 2) (Gauthier, 2014):

$$P_{KN}(w_i \mid w_{i-1}) = \frac{\max\left(c(w_{i-1}w_i) - \delta, 0\right)}{\sum_{w'} c(w_{i-1}w')} + \lambda \frac{|\{w_{i-1} : c(w_{i-1}, w_i) > 0\}|}{|\{w_{j-1} : c(w_{j-1}, w_i) > 0\}|}$$

Equation 2

Where λ is a normalizing constant.

Hyper-Parameters used for N-gram

Choosing the right hyperparameters is really important to build a very good model. For N-gram the hyperparameters investigated are in Table 2.

Hyper parameters	**Values**
number files	10,55,100
Known	2,3,4
Discount	0.75
MAX_FILES	2

Table 2 hyper-parameters investigated for N-gram model

The number of file count investigated were 10,55,100. The Out-Of-Vocabulary (OOV) words threshold were 2,3,4. The discount was kept at 0.75 which is its default value.

Word Embedding

Word embedding is an approach to addressing words as vectors. It is to convert the high dimensional feature space of words into low dimensional feature vectors by safeguarding the contextual similarity in the corpus. Word embedding model takes text corpus as input and produce the word vectors as output. Some of the famous pre trained models to create word embedding of a text are Word2vec (from Google), Fast text (from Facebook) and Glove (from Stanford).

Word2Vec model is utilized for Word representations in Vector Space which was founded by Tomas Mikolov and a group of the research teams from Google in 2013. It is a neural network model that attempts to explain the word embeddings based on a text corpus. Word2Vec model is not a single algorithm however is made out of the accompanying two pre-processing modules, Continuous bag of word (CBOW) and Skip gram. These two uses a distributed architecture which minimises the computation complexity.

Continuous bag of word (CBOW)

We can model this CBOW architecture as a deep learning classification model such that we take in the context words as our input and try to predict the target word. 'In fact building this architecture is simpler than the skip-gram model where we try to predict a whole bunch of context words from a source target word' (Sarkar, 2018)

The continuous bag of words model proposes a discriminative approach to predicting a masked word given its surrounding context. Unlike a language model that can only base its predictions on past words, as it is assessed based on its ability to predict each next word in the corpus, a model that only aims to produce accurate word embeddings is not subject to such restriction.

The CBOW architecture can be portrayed as a neural network with one hidden layer, an input layer of shape$(\text{vocab size} * \text{implanting size})$, a summation activity followed by a thick straight layer of shape $(\text{installing size} * \text{vocab size})$. The linear layer output has a Softmax applied to give a probability distribution over the vocabulary. This is then contrasted and a one-hot vector addressing the target word. The back propagation algorithm then updates the weights of the network. The word embeddings can essentially be removed from the weights of the first layer of this network. This was carried out through Pytorch.

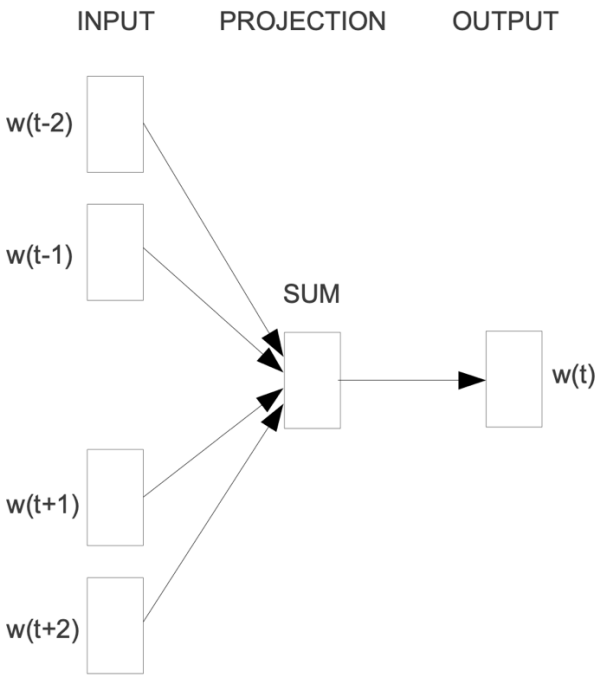

Figure 1 The CBOW architecture, It predicts the current word based on the context (Tomas Mikolov,Kai Chen,Greg Corrado,Jeffrey Dean, 2013).

The above Figure 1 illustrates the Continuous Bag Of Words Architecture (Tomas Mikolov,Kai Chen,Greg Corrado,Jeffrey Dean, 2013). The input layer of shape vocab size to embedding size, are the words at time step t $\pm$1,2, this window size can be varied. In the projection layer we are going to add all the words embeddings and use that to predict the current word . The output from this linear layer is a prediction over the entire vocabulary of words.

To generate context target words training models were produced from the training corpus by basically making one string for each text and making a sliding window, with the end goal that an target word was picked and the context was created from the immediate words around it. Which is $\pm\, context\ word\ size$ These were changed over completely to their vocabulary IDs prior to being taken care of to the model's embedding layer. Consequently the context was in the form $[w_{-i,\ldots\ldots,}w_i]$ for a window size of I, and targets were in the structure w_0. The quantity of context target matches can become very enormous and approximately identical to Equation 3 given below.

$$\left(\sum_{i=0}^{N} w_i\right) - 2 * k$$

Equation 3

Where k is the window size and N is the number of words.

CBOW and Skip-gram are just mirrored versions of each other. CBOW is trained to predict a single word from a fixed window size of context words, whereas Skip-gram does the opposite, and tries to predict several context words from a single input word. CBOW learn better syntactic relationships between words while Skip-gram is better in capturing better semantic relationships. Hence for the word 'cat', CBOW would retrieve as closest vectors morphologically similar words like plurals, i.e. 'cats' while Skip-gram would consider morphologically different words which are semantically relevant like 'dog' much closer to 'cat' in comparison.

Hyper-parameter tuning library Ray Tune library

Choosing the right hyperparameters is really important to build a very good model. But there will be a few situations where we need to employ some extra tools. Ray Tune is one such tool that we can use to find the best hyperparameters for our Continuous Bag of Words model in PyTorch. The Hyperparameter investigated for CBOW model is given below in Table 3.

Hyperparameter	***Values***
Adam Learning rate	2e-4, 2e-5, 2e-6
Embedding size	64, 128, 256
Stop word filter	True, False
window size	2, 3, 4, 5
epochs	20
Batch size	64
Adam Beta 1	0.9
Adam Beta 2	0.999
Adam Epsilon	1e-08

Table 3 Investigated hyperparameter for Continuous bag of words

After an underlying correlation between the entire training corpus and a subset, it was observed there was insignificant performance gain from utilizing the whole corpus with the generation context word method (Equation 3). Thusly the training corpus was confined to just incorporate 50 texts.

For training the model, the hyperparameter used are epochs, Adam Learning rate, Adam beta 1, Adam beta 2, Adam Epsilon, embedding size, stop word filter, window size and batch size (Table 3). Learning rate defines how much to change the model based on the estimated error when weights are updated. The learning rate tested were 2e-4, 2e-5, 2e-6. A model's optimizer is the algorithm that updates the weights of the model during training. Adaptive Moment Estimation (Adam) is the optimizer used in this model. Adam beta 1 is the exponential decay rate for the first moment estimates (0.9). Adam beta 2 is the exponential decay rate for the second-moment estimates (0.999). This value should be set close to 1.0 on problems with a sparse gradient. These are the default parameter for Adam optimization (Brownlee, 2017). The maximum number of epochs to train the model is 20, hence the scheduler can choose to stop any experiment that is within the maximum epoch. The batch size is the number of samples to work through before updating the internal model parameters. Batch size of 64 was used to perform this model because of

hardware limitation. The embedding size investigated were 64,128,256. There could be only two values (True or False) for Stop word filter since it's a Boolean.

Implementation and results

We apply our approach to the Microsoft Research Sentence Completion Challenge and show that it Continuous Bag of words (CBOW) outperforms N-gram model by 6 percentage points. The Generating N-gram model and Continuous Bag of Words model prediction was around 30 percentage, whereas the results after implementation of the model shows N gram (Trigram) has 26 percentage accuracy and Continuous Bag of Words had 32 percentage accuracy. CBOW performed better than N-Gram model. The predicted and actual accuracy of both the models implemented are shown in Table 4.

Model	Predicted Accuracy	Actual Accuracy
Unigram	24%	24.5%
Bigram	26%	22.3%
Trigram	30%	26.5%
CBOW	30%	33.59%

Table 4 predicted and actual accuracy of the model implemented

Figure 2, show the snapshot of the results of N-gram after using number of files (10,55,100), Out Of Vocabulary words (2,3,4) hyper-parameter tuning. The average unigram accuracy is 20.5 percentage points and the maximum accuracy of Unigram model is 24.5 percentage points. The average bigram accuracy is 19.5 percentage points and the highest Bigram accuracy obtained is 22.3 percentage points. The average Trigram accuracy calculated is 21.3 percentage points and the highest Trigram accuracy acquired is 26.5 percentage points.

	number of files	OOV threshold	unigram accuracy	bigram accuracy	trigram accuracy
0	10	2	0.225962	0.206731	0.213462
1	10	3	0.195192	0.190385	0.202885
2	10	4	0.189423	0.179808	0.211538
3	55	2	0.237500	0.223077	0.257692
4	55	3	0.240385	0.215385	0.242308
5	55	4	0.237500	0.213462	0.239423
6	100	2	0.240385	0.208654	0.265385
7	100	3	0.245192	0.210577	0.259615
8	100	4	0.244231	0.208654	0.240385

Figure 2 Results of N-gram after using Hyper-parameter tuning

Table 5, shows the results of Continuous bag of words (CBOW) after using Adam Learning rate (2e-04, 2e-05, 2e-06), embedding size (64,128,256), stop word filter

(True, False), window size (2,3,4,10) and epoch (3,5) (maximum epochs = 20) Hyper-parameters tuning.

Adam Learning rate	Embedding size	Stop word filter	Window Size	Epochs	Accuracy
2e-05	64	True	2	5	32.21
2e-05	64	True	3	3	30.51
2e-05	**256**	**True**	**4**	**5**	**33.59**
2e-06	64	True	3	3	20.30
2e-04	64	True	4	5	30.67
2e-04	128	True	3	3	30.00
2e-04	128	True	4	5	29.70
2e-04	256	True	10	3	24.70
2e-04	64	False	2	5	30.28
2e-04	64	False	4	5	30.67

Table 5 Results of continuous bag of words after hyper-parameter tuning

From the results, calculated average accuracy of Continuous bag of words is 29.37 percentage points and the maximum accuracy obtained is 33.59 percentage points.

Error Analysis

To improve the system performance the system's errors were examined. Few of the errors are the ones from the questions which test the dictionary definition of a word. And few of the error was due to the lack of general knowledge.

Conclusion

In this paper we have researched strategies for addressing sentence completion challenge questions. These questions are fascinating because they test the capacity to recognize semantically coherent sentences from incoherent ones but include no additional context from single sentence. To solve this sentence completion problem, we have implemented Continuous bag of words and N-Gram model. It was shown that Continuous bag of words model outperforms the traditional N-gram model by 6%.

Further studies

The drawback of this model is the computational requirement. For CBOW model by using Negative sampling techniques would improve performance on these models (Tomas Mikolov,Kai Chen,Greg Corrado,Jeffrey Dean, 2013). CBOW model can also be improved by better sub sampling. We are intrigued to observe the semantic between words, we can dispose them in this manner eliminating the commotion from

the information and it thusly gives us more noteworthy precision, quicker preparing, and better portrayals. Another technique that can improve the performance of CBOW model is by changing the model architecture since the current model cannot catch highlights from a high-volume data corpus. The new architecture is show in Figure 3 (kumar, 2020).

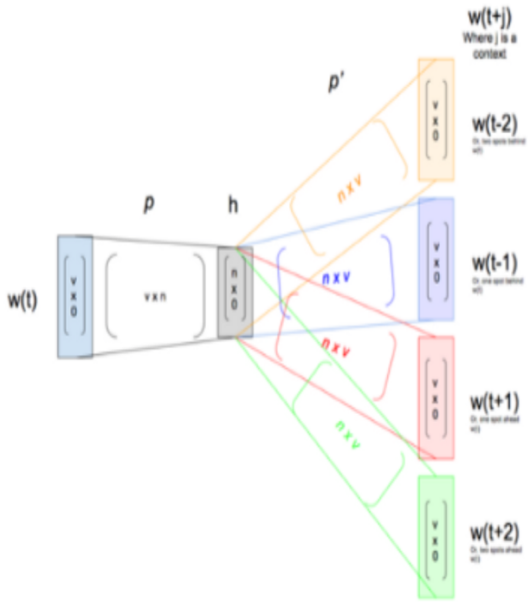

Figure 3 new CBOW architecture (kumar, 2020).

Works Cited

Woods, A. M., 2016. Exploiting Linguistic Features for Sentence Completion. *Association for Computational Linguistics,* August.p. 438–442.

Tomas Mikolov,Kai Chen,Greg Corrado,Jeffrey Dean, 2013. Efficient Estimation of Word Representations in Vector Space. *Proceedings of the International Conference on Learning Representations (ICLR 2013),* pp. 1-12.

Daniel Jurafsky & James H. Martin, 2013. *Speech and Language Processing.* 3rd ed. s.l.:Pearson.

Stanley F. Chen, Joshua Goodman, 1999. An empirical study of smoothing techniques for language modeling. *Computer Speech and Language,* Issue 13, pp. 359-394.

Burges, G. Z. a. C. J., 2011. The Microsoft Research Sentence Completion Challenge. *Microsoft Research Technical Report.*

Jacob Devlin Ming-Wei Chang Kenton Lee Kristina Toutanova, 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *Association for Computational Linguistics,* p. 4171–4186.

Tanushree C Hatmode,Prachi Y Duragkar, Radha H Khorgade, Ram B Jaiswal, Shubhashish S Charan, Rajesh Nasare, Hemant Turkar, 2021. Sentence Completion Using NLP Techniques. *IJCRT,* 9(6 June 2021).

Geoffrey Zweig, John C. Platt Christopher Meek Christopher J.C. Burges, Ainur Yessenalina,Qiang Liu, 2012. Computational Approaches to Sentence Completion. *Association for Computational Linguistics,* Volume 50, p. 601–610.
Brownlee, J., 2017. Gentle Introduction to the Adam Optimization Algorithm for Deep Learning. Machine Learning Mastery. URL https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/ (accessed 4.27.22).
Gauthier, J., 2014. Kneser-Ney smoothing explained [WWW Document]. foldl. URL http://www.foldl.me/2014/kneser-ney-smoothing/ (accessed 4.28.22).
kumar, B., 2020. Create your Mini-Word-Embedding from Scratch using Pytorch | DeepScopy [WWW Document]. URL https://deepscopy.com/Create_your_Mini_Word_Embedding_from_Scratch_using_Pytorch (accessed 4.28.22).
Kumar, P., 2017. An Introduction to N-grams: What Are They and Why Do We Need Them? [WWW Document]. XRDS. URL https://blog.xrds.acm.org/2017/10/introduction-n-grams-need/ (accessed 4.28.22).
Sarkar, D., n.d. Implementing Deep Learning Methods and Feature Engineering for Text Data: The Continuous Bag of Words (CBOW). KDnuggets. URL https://www.kdnuggets.com/implementing-deep-learning-methods-and-feature-engineering-for-text-data-the-continuous-bag-of-words-cbow.html/ (accessed 4.26.22).

Appendix

CBOW model Code with comments

```
# Importing all the necessary libraries for the code
import nltk
import os
import os.path
import random
from nltk import word_tokenize as tokenize
import operator
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from ray import tune
from ray.tune import CLIReporter
from ray.tune.schedulers import ASHAScheduler
from ray.tune.integration.pytorch_lightning import TuneReportCallback
import shutil
import tempfile
import pytorch_lightning as pl
from pytorch_lightning.loggers import TensorBoardLogger
from pytorch_lightning.utilities.cloud_io import load as pl_load
import re
from gutenberg.acquire import load_etext
from gutenberg.cleanup import strip_headers
nltk.download('punkt')
nltk.download('stopwords')
from nltk.corpus import stopwords
nltk.download('wordnet')
from nltk.corpus import wordnet as wn
from nltk import word_tokenize
import string
import pandas as pd, csv
import re
from sklearn.model_selection import train_test_split
import numpy as np
```

```
mrscc_dir = '/Users/sumitbajare/Documents/Sussex/Sem2/ANLP/lab2resources/sentence-completion' # parent directory
def get_train_val(training_dir=mrscc_dir,split=0.5):

    '''
        Getting the names of files in the training directory and
        split them into training and testing 50:50.
    :param: training_dir,split
    :return: filenames[:index],filenames[index:]
    '''

    filenames=os.listdir(training_dir)
    n=len(filenames)
    print(f"There are {n} files in the training directory: {training_dir}")
    random.seed(7) #if you want the same random split every time
    random.shuffle(filenames)
    index=int(n*split)
    return(filenames[:index],filenames[index:])

trainingdir=os.path.join(mrscc_dir,'Holmes_Training_Data/')
training,testing=get_train_val(trainingdir)

def processfiles(files, training_dir, filter="Conan Doyle"):

    '''
        Processing the file
    :param: files, training_dir, filter
    :return: texts
    '''

    texts = []
    for i, j in enumerate(files):
        text = ""
        try:
            with open(os.path.join(training_dir,j)) as instream:
                for line in instream:
                    text += line
                if re.search(filter, text, re.IGNORECASE) or i%15==0:
                    #returns a match object when the pattern is found and “null” if the pattern is not found
                    print("sherlock found at {}".format(i))
                    texts.append(strip_headers(text).strip())
        except UnicodeDecodeError:
            print(f"UnicodeDecodeError processing {j}: ignoring rest of file")
    return texts

def processfiles(all_texts, questions=questions, config={"stop":True, "window_size":4}):

    '''
        Generating a pairs of context and targets,
        For every target word n; n-2,n-1,n+1,n+2 context words will be generated
        creating a rolling window over the text
    :param: all_texts, questions=questions, config
    :return: train, vocab
    '''

    window = config['window_size']
    vocab = set()
    contexts,targets=[],[]
    stop = set(stopwords.words('english') + list(string.punctuation))
    for text in all_texts:
        if config['stop']:
            tokenized_text = [i for i in word_tokenize(text.lower()) if i not in stop]
        else:
            tokenized_text = [i for i in word_tokenize(text.lower())]
        vocab.update(tokenized_text)
        for i in range(window, len(tokenized_text) - window - 1):
            contexts.append(tokenized_text[i-window:i] + tokenized_text[i+1:i+window+1])
            targets.append(tokenized_text[i])
    train = pd.DataFrame()
    train['contexts']=contexts
    train['targets']=targets
    for i,row in questions.iterrows():
        stop = set(stopwords.words('english') + list(string.punctuation))
        if config['stop']==True:
            question_tokens = [i for i in word_tokenize(row.question.lower()) if i not in stop]
        else:
```

```
            question_tokens = [i for i in word_tokenize(row.question.lower())]
        vocab.update(question_tokens)
        vocab.update(list(row[choices]))
    return train, vocab

# PYthon Lightening DAtatset
class PLDataset(Dataset):

    def __init__(self, data: pd.DataFrame, vocab: dict):

        '''
            This is the constructor method
        :param: data, vocab
        '''
        self.data = data
        self.vocab = vocab

    def __len__(self):

        '''
            This is a built-in functions that gets the number of items in the container self.data
        :return data
        '''
        return len(self.data)

    def __getitem__(self, index: int):

        '''
           This is a built-in functions; Used for accessing list items, dictionary entries, array elements etc.
        :param: data, vocab
        :return: context_ids, target_id, dtype
        '''
        row = self.data.iloc[index]
        context = row.contexts
        target = row.targets
        return {'context_ids':torch.tensor([self.vocab[w] for w in context],
dtype=torch.long),'target_id':torch.tensor(self.vocab[target], dtype=torch.long)}


word_to_ix = {word: i for i, word in enumerate(vocab)}
test = PLDataset(train.head(), word_to_ix)
test.__getitem__(0)

# Testing Python Lightening DAtaset
class PLTestDataset(Dataset):

    def __init__(self, data: pd.DataFrame, vocab: dict, window: int=4):

        '''
            This is the constructor method
        :param: data, vocab, window
        '''

        self.data = data
        self.vocab = vocab
        self.window = window
        self.stop = set(stopwords.words('english') + list(string.punctuation))

    def __len__(self):

        '''
            This is a built-in functions that gets the number of items in the container self.data
        :return data
        '''

        return len(self.data)

    def __getitem__(self, index: int, target="_____"):

        '''
            This is a built-in functions; Used for accessing list items, dictionary entries, array elements etc.
        :param: index, target
        :return: context_ids, dtype, target_id
        '''
```

```
        row = self.data.iloc[index]
        question = row.question
        answer = row.answer.lower()
        question_tokens = [i for i in word_tokenize(question.lower()) if i not in self.stop]
        window_left,window_right = self.window,self.window
        for i,word in enumerate(question_tokens):
            if word == target:
                if i<window_left:
                    window_right = window_right+(window_left-1)
                if i>(len(question_tokens)-window_right):
                    window_left = window_left+(len(question_tokens)-i)
                context = question_tokens[i-window_left+1:i]+question_tokens[i+1:i+1+window_right]
                break
        return {'context_ids':torch.tensor([self.vocab[w] for w in context],
dtype=torch.long),'target_id':torch.tensor(self.vocab[answer], dtype=torch.long)}

test = PLTestDataset(questions.head(), word_to_ix)
test.__getitem__(1)

#Python Lightning DAtaModule
#A DataModule is simply a collection of a train_dataloader(s), val_dataloader(s), test_dataloader(s) and predict_dataloader(s)
#along with the matching transforms and data processing/downloads steps required.

class PLDataModule(pl.LightningDataModule):
    def __init__(self, train_data, test_data, batch_size=16, vocab=word_to_ix, window=4):

        '''
            This is the constructor method
        :param: train_data, test_data, batch_size, vocab, window
        '''

        super().__init__()
        print(len(train_data))
        self.train_data = train_data
        self.test_data = test_data
        self.batch_size = batch_size
        self.vocab = vocab
        self.window = window

    def setup(self):

        """
            Assign train & test datasets for use in dataloaders
        """

        self.train_dataset = PLDataset
        self.train_data
        self.vocab
        self.test_dataset = PLTestDataset
        self.test_data
        self.vocab
        self.window

    def train_dataloader(self):

        '''
            Generating the training dataloaders
        :return DataLoader
        '''

        return DataLoader

    def val_dataloader(self):

        '''
            Generate the validation dataloaders
        :return: Dataloader(self.test_dataset,batch_size=1,num_workers=2)
        '''

        return DataLoader(self.test_dataset,batch_size=1,num_workers=2)

    def test_dataloader(self):

        '''
             Generate the test dataloaders
```

```
        :return: DataLoader(self.test_dataset,batch_size=1,num_workers=2)
        '''

        return DataLoader(self.test_dataset,batch_size=1,num_workers=2)

# Creating the CBOW model
class CBOWModel(pl.LightningModule):

    def __init__(self, config, vocab):

        '''
            This is the constructor method
        :param: config, vocab
        '''

        super().__init__()
        self.config = config
        self.vocab = vocab
        self.embeddings = nn.Embedding(num_embeddings=config['vocab_size'],embedding_dim=config['embedding_dim'])
        self.linear = nn.Linear(in_features=config['embedding_dim'],out_features=config['vocab_size'])
        torch.nn.init.xavier_normal_(self.linear.weight)
        self.accuracy = pl.metrics.Accuracy()
        self.loss_function = nn.NLLLoss()

    def forward(self, inputs, target=None):
        embeds = torch.mean(self.embeddings(inputs), dim=1)
        logits = self.linear(embeds)
        out = F.log_softmax(logits, dim=1)
        loss = 0
        if target is not None:
            loss = self.loss_function(out, target)
        return loss, logits

    def training_step(self, batch, batch_index):

        '''
            Training the model
        :param: batch, batch_index
        : return: loss
        '''

        context_ids = batch['context_ids']
        target_id = batch['target_id']
        loss, outputs = self(context_ids, target_id)
        self.log("train loss", loss, prog_bar = True, logger=True)
        return {"loss":loss}

    def validation_step(self, batch, batch_index):

        '''
            Validating the model
        :param:batch, batch_index
        :return: val_loss, val_outputs
        '''

        context_ids = batch['context_ids']
        target_id = batch['target_id']
        loss, outputs = self(context_ids, target_id)
        self.log("validation loss ", loss, prog_bar = True, logger=True)
        return {"val_loss": loss, "val_outputs": outputs}

    def validation_epoch_end(self, outputs):

        '''
            validation loss and accuracy
        :param: outputs
        '''

        avg_loss = torch.stack([x["val_loss"] for x in outputs]).mean()
        all_outputs = [x["val_outputs"] for x in outputs]
        preds=[]
        for output in all_outputs:
            for i,row in questions.iterrows():
                choice_ids = [self.vocab[row[c]] for c in choices]
                choice_logits = [float(output[0, id]) for id in choice_ids]
```

```
            preds.append(np.argmax(np.array(choice_logits)))
        total,correct=0,0
        for answer,pred in zip(answers.answer, preds):
            total+=1
            if answer==choices[pred]:
                correct+=1
        print(f"test accuracy {correct/total}")
        self.log("ptl/val_loss", avg_loss)
        self.log("ptl/val_accuracy", correct/total)

    def configure_optimizers(self):

        '''
            Initializing the optimizer
        :return: optimizer
        '''

        optimizer = optim.AdamW(self.parameters(), lr=self.config['lr'])
        return optimizer


# Calling the model
config = {"lr": 2e-5,"batch_size": 128,"embedding_dim":256,"vocab_size":len(vocab),"n_epochs":6,"stop":False}
print("Training set size: {}".format(len(train)))
print("Vocab set size: {}".format(len(vocab)))
model = CBOWModel(config, vocab=word_to_ix)
data_module = PLDataModule(train, questions, batch_size=config['batch_size'],vocab=word_to_ix)
data_module.setup()

def train_tune(config, gpus=0):
    '''
        Hyperparamter tuning with ray
    :param: config, gpus
    '''
    train, vocab = processfiles(texts, config=config)
    word_to_ix = {word: i for i, word in enumerate(vocab)}
    print("Training set size: {}".format(len(train)))
    model = CBOWModel(config,vocab=word_to_ix)
    data_module = PLDataModule(train, questions, vocab=word_to_ix, batch_size=config['batch_size'])
    print("Steps per epoch {}".format(len(train)/config['batch_size']))
    data_module.setup()
    trainer =
pl.Trainer(max_epochs=5,gpus=config["n_gpus"],progress_bar_refresh_rate=1000,logger=TensorBoardLogger(save_dir=tune.
get_trial_dir(), name=" ", version="."),callbacks=[callback])
    trainer.fit(model, data_module)

def tune_cbow(config, num_samples=3, gpus_per_trial=0):
    '''
        Hyperparameter tuning CBOW wit ray
    :param:config, num_samples, gpus_per_trial
    '''
    scheduler = ASHAScheduler(metric='accuracy',mode='max',grace_period=3,reduction_factor=2)
    reporter = CLIReporter(parameter_columns=["lr", "batch_size", "embedding_dim", 'stop',
"window_size"],metric_columns=["loss","accuracy", "training_iteration"])
    trainable = tune.with_parameters(train_tune,gpus=config["n_gpus"])
    analysis = tune.run(trainable,resources_per_trial={"cpu": 1, "gpu":
config["n_gpus"]},config=config,scheduler=scheduler,progress_reporter=reporter,num_samples=num_samples,name="tune_cb
ow")

# hyperparameters
config = {"lr": tune.choice([2e-6,2e-5,2e-4]), "batch_size":
64,"embedding_dim":tune.choice([64,128,256]),"vocab_size":len(vocab), "n_epochs":20,"stop":tune.choice([True,
False]),"window_size":tune.choice([2,3,4,5,10]),"n_gpus":1}

analysis = tune_cbow(config, num_samples=10)

test = torch.tensor([ word_to_ix['went'], word_to_ix['city'], word_to_ix['walking'], word_to_ix['streets'], word_to_ix['capital'],
word_to_ix['building']])
test.shape

torch.argmax(log_probs)
ix_to_word = dict((v,k) for k,v in word_to_ix.items())
ix_to_word[int(torch.argmax(log_probs))]

# testing the data
```

```
class question:
    def __init__(self, aline, lm):

        '''
            This is the constructor method
        :param: aline, lm
        '''

        self.sentence=aline[1]
        self.choices = ["a", "b", "c", "d", "e"]
        self.word_choices = {index:word for index,word in zip(self.choices,aline[2:])}
        self.model = model

    def add_answer(self,fields):

        '''
            Adding answer field
        :param: fileds
        '''

        self.answer=fields[1]

    def get_window_context(self,sent_tokens,window_left, window_right,target="_____"):

        '''
           getting window context; changing the right and left window
        :param: sent_tokens,window_left, window_right,target
        return: tokens
        '''

        stop = set(stopwords.words('english') + list(string.punctuation))
        tokens = [i for i in word_tokenize(sent_tokens.lower()) if i not in stop]
        for i,token in enumerate(tokens):
            if token==target:
                if i<window_left:
                    window_right = window_right+(window_left-1)
                if i>(len(tokens)-window_right):
                    window_left = window_left+(len(tokens)-i)
            return tokens[i-window_left+1:i]+tokens[i:i+window_right]
        else:
            return []

    def predict(self, window=2):

        '''
            predict by getting the left words;getting rid of extra dimentions;
            converting words to ids and then turning into probabilities of the given model;
            picking the maximum predicted choice
        :param: window
        :return: prediction
        '''

        context = self.get_window_context(self.sentence, window, window)
        context = torch.tensor([word_to_ix[w] for w in context])
        _, log_probs = model(torch.unsqueeze(context, dim=0), target=None)
        log_probs = torch.squeeze(log_probs)
        choice_ids = {index:word_to_ix[word] for index,word in self.word_choices.items() if word in word_to_ix.keys()}
        choice_probs = {index:float(log_probs[id]) for index, id in choice_ids.items()}
        prediction = max(choice_probs, key=choice_probs.get)
        return prediction

    def predict_and_score(self):

        '''
            comparing the prediction with the correct answer
        ;return:1/0
        '''

        prediction=self.predict()
        if prediction == self.answer:
            return 1
        else:
            return 0

class mrscc_reader:
```

```
    def __init__(self, model, qs=questions, ans=answers):

        '''
            This is the constructor method
        :param: model, qs, ans
        '''

        self.qs=qs
        self.ans=ans
        self.model = model
        self.read_files()

    def read_files(self):

        '''
            Reading the files; adding answers to the question, to check the prediction
        '''

        self.questions=[question(questions.iloc[i], self.model) for i in range(len(questions))]

        for i,q in enumerate(self.questions):
            q.add_answer(answers.iloc[i])

    def get_field(self,field):
        '''
            retriving the questions field
        '''
        return [q.get_field(field) for q in self.questions]

    def predict(self):
        return [q.predict() for q in self.questions]

    def predict_and_score(self):
        scores=[q.predict_and_score() for q in self.questions]
        return sum(scores)/len(scores)
```

N-Gram Model code with comments

```
# importing the required libraries
import os,random,math
import collections
import nltk
from nltk import word_tokenize as tokenize
from nltk import sent_tokenize
import numpy as np
import operator
import pandas as pd, csv
import matplotlib.pyplot as plt

nltk.download('punkt')

# path where the files are
mrscc = '/Users/sumitbajare/Documents/Sussex/Sem2/ANLP/lab2resources/sentence-completion'
TRAINING_DIR = '/Users/sumitbajare/Documents/Sussex/Sem2/ANLP/lab2resources/sentence-
completion/Holmes_Training_Data'
questions_file = '/Users/sumitbajare/Documents/Sussex/Sem2/ANLP/lab2resources/sentence-completion/testing_data.csv'
answers_file = '/Users/sumitbajare/Documents/Sussex/Sem2/ANLP/lab2resources/sentence-completion/test_answer.csv'

results_directory = '/Users/sumitbajare/Documents/Sussex/Sem2/ANLP/lab2resources/sentence-completion'

def get_training_testing(training_dir,split=0.5):

    '''
        split the data between train and test
    :param: training_dir, split
    :return: trainingfiles, heldoutfiles
    '''

    filenames=os.listdir(training_dir)
    n=len(filenames)
    print("There are {} files in the training directory: {}".format(n,training_dir))
    random.seed(53)  #if you want the same random split every time
    random.shuffle(filenames)
```

```
    index=int(n*split)
    trainingfiles=filenames[:index]
    heldoutfiles=filenames[index:]
    return trainingfiles,heldoutfiles


class language_model():

    '''
        Language model to train a unigram, a bigram, and a trigram model;
        Using Kneser-ney smothing, absolute discount
    '''

    def __init__(self,known=2,discount=0.75,trainingdir=TRAINING_DIR,files=[]):

        '''
            This is the constructor method
        :param: known=2,discount,trainingdir, files
        :return:none
        '''

        self.training_dir = trainingdir
        self.files = files
        self.discount = discount
        self.known = known
        self.train()

    def train(self):

        '''
            Traning the models
        :param:none
        :return:none
        '''

        self.unigram = {}
        self.bigram = {}
        self.trigram = {}

        self.count_token = {}
        self.processfiles()
        self.make_unknowns()
        self.kneser_ney()
        self.convert_to_probs()


    def processline(self,line):

        '''
            Geting the ngramms for every line of the document
        :param: line
        :return:none
        '''

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

          # Counting the tokens
            self.count_token[token] = self.count_token.get(token,0) + 1

          # For getting bigrams
            current_big = self.bigram.get(previous,{})
            current_big[token] = current_big.get(token,0) + 1
            self.bigram[previous] = current_big

          # For getting trigrams
            pre_trigram[1], pre_trigram[0] = pre_trigram[0], pre_trigram[1]
            pre_trigram[1] = previous
            current_tri = self.trigram.get(tuple(pre_trigram), {})
            current_tri[token] = current_tri.get(token, 0) + 1
            self.trigram[tuple(pre_trigram)] = current_tri
```

```
        previous = token

    def processfiles(self):

        '''
            Processing the file
        :param: none
        :return:none
        '''

        for afile in self.files:
            try:
                with open(os.path.join(self.training_dir,afile)) as instream:
                    for line in instream:
                        line = line.rstrip()
                        if len(line) > 0:
                            self.processline(line)
            except UnicodeDecodeError:
                print("UnicodeDecodeError processing {}: ignoring rest of file".format(afile))


    def convert_to_probs(self):

        '''
            Converting ngram counts to probabilities
        :param: none
        :return:none
        '''

        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.kn = {k:v/sum(self.kn.values()) for (k,v) in self.kn.items()}
        self.kn_tri = {k:v/sum(self.kn_tri.values()) for (k,v) in self.kn_tri.items()}


    def get_prob(self,token,context="",methodparams={}):

        '''

            Getting the probability of a token
        :param: token,context,methodparams
        :return:p
        '''

        if methodparams.get("method","unigram") == "unigram":
            return self.unigram.get(token,self.unigram.get("__UNK",0))

        elif methodparams.get("method","bigram") == "bigram":
            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

        elif methodparams.get("method","trigram") == "trigram":
            if methodparams.get("smoothing","kneser-ney") == "kneser-ney":
                unidist = self.kn_tri
                unidist_bi = self.kn
            else:
                unidist_bi = self.unigram

            if len(context) < 2:
                context = ["__END", context[0]]
        trigram = self.trigram.get(tuple(context[-2:]),self.trigram.get("__UNK",{}))
        trig_p = trigram.get(token,trigram.get("__UNK",0))
        lmbda_tri = trigram["__DISCOUNT"]
```

```
        bigram = self.bigram.get(context[-1],self.bigram.get("__UNK",{}))
        big_p = bigram.get(token,bigram.get("__UNK",0))
        lmbda_bi = bigram["__DISCOUNT"]
        uni_p = unidist_bi.get(token,unidist_bi.get("__UNK",0))

        p = trig_p + (lmbda_tri * big_p) + (lmbda_bi * uni_p)
        return p

    def compute_prob_line(self,line,methodparams={}):

        '''
            Computing the probability of each line of the document
        :param: none
        :return:acc,len(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:]) # returns probability together with number of tokens

    def compute_probability(self,filenames=[],methodparams={}):

        '''
            Computing the probability of a corpus contained in filenames
        :param:filenames, methodparams
        :return:total_p,total_N
        '''

        if filenames == []:
            filenames = self.files

        total_p = 0
        total_N = 0
        for i,afile in enumerate(filenames):
            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"}):

        '''
            Computing perplexity of the data
        :param: filenames,methodparams
        :return:pp
        '''

        # 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):

        '''
            Making sure the unknown words given some threshold
        :param: none
        :return:none
        '''

        unknown = 0
        self.number_unknowns = 0
        for (k,v) in list(self.unigram.items()):
            if v < self.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]
            prev_1, prev_2 = k
            isknown_1, isknown_2 = self.unigram.get(prev_1,0), self.unigram.get(prev_2,0)
            if isknown_1 == 0 or isknown_2 == 0:
                del self.trigram[k]
                current = self.trigram.get("__UNK",{})
                current.update(adict)
                self.trigram["__UNK"] = current
            else:
                self.trigram[k] = adict


    def kneser_ney(self):

        '''
            Apply absolute discount and kneser-Ney smoothing on the models
        :param: none
        :return:none
        '''
        # Applying Discount on each bigram
        self.bigram = {k:{kk:value-self.discount for (kk,value) in adict.items()} for (k,adict) in self.bigram.items()}

        # Applying Discount on each bigram trigram
        self.trigram = {k:{kk:value-self.discount for (kk,value) in adict.items()} for (k,adict) in self.trigram.items()}

        # To reserve the probability mass store the total amount of the discount
        for k in self.bigram.keys():
            lamb = len(self.bigram[k])
            self.bigram[k]["__DISCOUNT"] = lamb * self.discount

        for k in self.trigram.keys():
            lamb = len(self.trigram[k])
            self.trigram[k]["__DISCOUNT"] = lamb * self.discount

        # kneser-ney unigram prob
        self.kn = {}
        for (k,adict) in self.bigram.items():
            for kk in adict.keys():
                self.kn[kk] = self.kn.get(kk,0) + 1

        self.kn_tri = {}
        for (k,adict) in self.trigram.items():
            for kk in adict.keys():
                self.kn_tri[kk] = self.kn_tri.get(kk,0) + 1


class question:

    def __init__(self,aline):

        '''
```

```
        This is the constructor method
    :param:aline
    :return:none
    '''

    self.fields=aline

def get_field(self,field):

    '''
        Getting the question field
    :param: field
    :return:fields
    '''

    return self.fields[question.colnames[field]]

def add_answer(self,fields):

    '''
        Getting the answer field
    :param: field
    :return:none
    '''

    self.answer=fields[1]

def get_tokens(self):

    '''
        Getting the tokens
    :param: none
    :return:tokenize
    '''

    return ["__START"]+tokenize(self.fields[question.colnames["question"]])+["__END"]

def get_left_context(self,window=1,target="_____"):

    '''
        Getting the left context
    :param: window, target
    :return:sent_tokens
    '''

    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 get_right_context(self,window=1,target="_____"):

    '''
        Getting the right context
    :param: window,target
    :return:sent_tokens
    '''

    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[found + 1:found + window + 1]
```

```
        else:
            return []

    def choose(self,lm,method="bigram",smoothing="Kneser-ney", choices=[]):

        '''

            Choose specific ngram;
            predicted answer for the sentence
        :param: lm,method,smoothing, choices
        :return:choice,probs
        '''

        if choices == []:
            choices = ["a","b","c","d","e"]

        if method == "bigram":
            rc = self.get_right_context(window=1)
            lc = self.get_left_context(window=1)
            probs = [lm.get_prob(rc[0],[self.get_field(ch+")")],methodparams={"method":method.split("_")[0],
"smoothing":smoothing}) * lm.get_prob(self.get_field(ch+")"),lc,methodparams={"method":method.split("_")[0],
"smoothing":smoothing}) for ch in choices]

        elif method == "trigram":
            rc = self.get_right_context(window=2)
            lc = self.get_left_context(window=2)
            probs = [lm.get_prob(self.get_field(ch+")"), lc, methodparams={"method":method.split("_")[0], "smoothing":smoothing})
                    * lm.get_prob(rc[0], [lc[-1]] + [self.get_field(ch+")")],methodparams={"method":method.split("_")[0],
"smoothing":smoothing})
                    * lm.get_prob(rc[1], [self.get_field(ch+")")] + [rc[0]],methodparams={"method":method.split("_")[0],
"smoothing":smoothing}) for ch in choices]

        else:
            context = self.get_left_context(window=1)
            probs = [lm.get_prob(self.get_field(ch+")"),context,methodparams={"method":method.split("_")[0]}) for ch in choices]

        maxprob = max(probs)
        bestchoices = [ch for ch,prob in zip(choices,probs) if prob == maxprob]

        return np.random.choice(bestchoices), probs

    def predict(self,lm,method="bigram", smoothing="kneser-ney"):

        '''
            Predict the answer given a language model;
            Applying Kneser-Ney smoothing
        :param: lm,method, smoothing
        :return:choose()
        '''

        return self.choose(lm,method=method,smoothing=smoothing,choices=[])

    def predict_and_score(self,lm,method="bigram", smoothing="kneser-ney"):

        '''
            Checking the prediction is equal to the answer
        :param: lm, method, smoothing
        :return:1/0
        '''

        prediction, probs = self.predict(lm,method=method,smoothing=smoothing)

        if prediction == self.answer:
            return 1, prediction, probs
        else:
            return 0, prediction, probs

class scc_reader:

    def __init__(self,qs,ans):

        '''
            This is the constructor method
        :param: qs, ans
        :return:none
```

```
        '''

        self.qs=qs
        self.ans=ans
        self.read_files()

    def read_files(self):

        '''
            Reading the question and answer file
        :param: none
        :return:none
        '''

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

        question.colnames={item:i for i,item in enumerate(qlines[0])}

        #creating a question instance for each line
        self.questions=[question(qline) for qline in qlines[1:]]

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

        #adding answers to questions
        for q,aline in zip(self.questions,alines[1:]):
            q.add_answer(aline)

    def get_field(self,field):

        '''
            getting the question field
        :param: field
        :return:get_field
        '''

        return [q.get_field(field) for q in self.questions]

    def predict(self,method="bigram"):

        '''
        :param: method
        :return:predict
        '''

        return [q.predict(method=method) for q in self.questions]

    def predict_and_score(self,lm,method="bigram",smoothing="kneser-ney"):

        '''
            Computing the accuracy;
            Calculating probability distribution of the options of each question
        :param: lm, method, smoothing
        :return:sum(scores)/len(scores), predictions, total_probs
        '''

        predictions = []
        scores = []
        total_probs = []
        for q in self.questions:
            score, pred, probs = q.predict_and_score(lm,method=method, smoothing=smoothing)
            scores.append(score)
            predictions.append(pred)
            total_probs.append(probs)

        return sum(scores)/len(scores), predictions, total_probs


    # Hyperparameters Tuning
fls = os.listdir(TRAINING_DIR)
```

```
number_files = np.linspace(10, 100, 3).astype(int)
known = [2, 3, 4]
discount = 0.75
smoothing = 'kneser-ney'
MAX_FILES = 2
results = []
predictions = []
total_probs = []
iter = 0
for n in number_files:
    for k in known:
        print('Processing {} documents...'.format(n))
        mylm = language_model(known=k, discount=discount, files=fls[:n])
        SCC = scc_reader(questions_file, answers_file)

        unigram_accuracy, unigram_predictions, unigram_probs =
SCC.predict_and_score(mylm,method="unigram",smoothing=smoothing)
        bigram_accuracy, bigram_predictions, bigram_probs =
SCC.predict_and_score(mylm,method="bigram",smoothing=smoothing)
        trigram_accuracy, trigram_predictions, trigram_probs =
SCC.predict_and_score(mylm,method="trigram",smoothing=smoothing)

        results.append((n, k, unigram_accuracy, bigram_accuracy, trigram_accuracy))
        predictions.append((unigram_predictions, bigram_predictions, trigram_predictions))
        total_probs.append((unigram_probs, bigram_probs, trigram_probs))

        iter += 1
```