Advanced Natural Language Engineering

Coursework Report

Yiming SHEN
Candidate Number: 236789

Date: April 7, 2022

Abstract

With the development of artificial intelligence, in order to meet the growing demand for text, it is necessary to establish appropriate semantic models to promote the development of text processing. The Microsoft Research Sense Complete Challenge (Zweig and Burges, 2011) provides a public dataset of model performance for building and semantics. The usability of semantic models in the English context is studied by using this database, and the results are compared with several other common machine-learning methods. The results show that these algorithms have some advantages. The focus of this paper is to build and test semantic models and analysis and compare the performance of different semantic models.

Microsoft Research Sentence Completion Challenge

Few public data sets can generally measure the performance of semantic models. Experts at Microsoft Research Institute regard this challenge as an evaluation system. In this task, researchers try to find a general method that can simulate human thinking and understand the meaning of text to evaluate it. This challenge comes from Project Gutenberg data (Zweig and Burges, 2011), which comes from Arthur Conan doylesir's five original Sherlock Holmes novels, with a total of 1040 sentences. The low-frequency focus words in each sentence are used as options, and the four alternative words generated by the maximum entropy N-gram model are used as interference options. For each sentence, the N-gram model derived from 500 over 19 century novels was used to train 30 alternative words to select the best four pseudonym alternatives according to the criteria used by human language experts. The task of the semantic model is to determine which of the five choices of words is correct.

Introduction

There is a growing demand for automated processing, which is a good way to introduce text into appropriate semantic models. However, in practice, due to various reasons, the text is often ambiguous or inconsistent, which leads to the text classification results and the real value of a large difference, affecting the accuracy and stability of the system. Therefore, how to deal with the text and how to deal with the effect of the text becomes the primary problem to measure the semantic model performance.

Data from the experiment came from the Microsoft Research Sense Complete Challenge, a semantic model that will complete a 1,040-sentence title challenge. Using these datasets to assess the performance of the model, factors affecting the performance of the model can be clearly identified. At last, some conclusions which have some guiding significance for semantic modeling are summarized according to the experimental results, which can be used as reference

for further research. By changing the influencing factors, this paper tries to maximize the score of the model on SCC, adjust the setting of various hyperparameters in practice, and deepen the understanding of various semantic model processing texts.

Method

In a technical paper on the Microsoft Research Sentence Challenge, three fundamentally different approaches were proposed: human standards, potential semantic analysis, and n-gram variants (Zweig and Burges, 2011). Three methods were used to evaluate the answers and compare their test scores with traditional machine learning methods. The highest score of 91 per cent of human responses meant 91 per cent of predictions were reasonable. The n-gram model's baseline results were between 30% and 40% accurate. In potential semantic analysis, the score was 49 percent. In the underlying semantic analysis method, "Wordsmanship methods' 'requires words to be transformed into vector forms, with cosine values between different vectors being the semantic similarity of different words.

Two methods will be used and presented in the present report. The first is to use a simple N-gram model, including a 1-4g model. Then a new structural relationship is constructed by using hypergraph theory, combining the information of part of speech distribution and syntactic position, which is used as the basis for collocation analysis of words in sentences. The second method uses word embedding to calculate semantic similarity based on 2-3 gram semantic similarity model. By adjusting the different features of sentences, the effect of hyperparameter setting on the accuracy of sentence completion is studied.

N-Gram Language Model

The n-gram model (Cavnar, Trenkle, 1994) is a statistical model that uses conditional probabilities of how often words appear in context, before and after (Jurafsky, 1992). For example, "To be or not to be that is a question." In this sentence, calculate the probability of "question":

$$P\,(question \mid To\ be\ or\ not\ to\ be\ that\ is\ a) = \frac{C\,(To\ be\ or\ not\ to\ be\ that\ is\ a\ question)}{C\,(To\ be\ or\ not\ to\ be\ that\ is\ a)} \tag{1.1}$$

The method for estimating this probability is a relative frequency count, which also relies on a chain of probabilities. We need to take a very large corpus, count the number of times this sentence or word appears, and the number of times the next word appears.

When calculating the combined probability of the entire sequence, assuming that the random variable Xi takes the value "question" or P (Xi = "question"), the probability of the sequence containing the N-word is expressed as (w1, w2, w3...), and for probability, we have P (w1, w2, w3...), then:

$$P(w_1, w_2, ..., w_n) = P(w_1)P(w_2 \mid w_1)P(w_3 \mid w_2)...P(w_n \mid w_{n-1})$$

(1.2)

Since nothing specific may have happened, it is necessary to use the Markov hypothesis (Journel, 1999) and maximum probability to estimate probability (Candès E J, Sur P, 2020) by normalizing the n-gram count, assuming that the frequency of a word's occurrence depends entirely on the word appearing above. Experimental results show that N-Gram algorithm can reduce computational complexity and improve classification accuracy. The N-gram model performed well in many different computer-themed newsgroups (Cavnar, Trenkle, 1994), with a classification accuracy of 80%. The N-gram model has the advantages of small size, speed, and stability. All probabilities can be given in a given word sequence and are suitable as a way to complete a sentence challenge.

Word embedding

Potential semantic analysis (Landauer & Dumais, 1997) is a theory and method to extract and express contextual meaning through statistical calculation of large corpus (Landauer & Dumais, 1997). Potential semantic analysis combines natural language understanding with computer language processing and uses the similarity between words to understand and recognize sentences. It provides a set of corresponding constraints for all word contexts that occur or do not occur in a given word, which largely determines the similarity of meanings between words and word sets.

Word embedding involves modeling each word as a single word and looking for descriptions that capture semantic similarities between words (Levy, Goldberg, 2014). This method is based on block technology. For text, it has a higher accuracy and recall rate. But for speech recognition, there will be lower recognition rate and mismatch. In word2vec, for example, words and phrases are converted into vectors using neural networks as models. The similarity of words can be calculated effectively by using a low-dimensional matrix operation because it can be extended to larger corpus machine challenge tests. In this experiment, two word vectors were used:

1. Wiki Word Vector: Contains 1 million word vectors and trains in subword information on the 2017 Wikipedia, UMBC web corpus and statmt.org news dataset (16B token).

2. Google News Dataset: The model contains 300 dimensional vectors of 3 million words

and phrases. These phrases are obtained using the simple data-driven methods described in the usage.

Different word vectors have different semantic similarity to individual word modeling. In this paper, we will use these two different word vectors to compare, adjust hyperparameters and settings to observe the effect on score.

Solutions

This section details how the n-gram model and word embedding approach to sentence completion challenges. First, train 10 files randomly on the number of training files, all from the Gutenberg Program Text Set. The code and detailed comments will be shown in the appendix, as will links to the dataset on Gutenberg's project.

N-Gram Language Model

When importing text into n-gram training, there are three processes:

1. Split the text into tokens by word, store them in a list, and tag each paragraph with "_ START" and "_ END" tokens at the beginning and end.
2. Delete tokens with less than 2 occurrences and replace them with "_ UNK" tokens; Processing is done using Absolute count (Moore, Quirk, 2009), pruning a portion of each gram count to preserve the probabilistic quality of the event not seen.
3. Calculate the probability of each token in gram by counting.

We want the model to perform well on unseen text datasets, so we need to observe the inverse probability of the model, which is the perplexity. In this experiment, we used a random 1-5 text set from the Gutenberg Project (Zweig and Burges, 2011) to train and test for confusion. It is clear from Figure 1 that confusion increases with the number of training texts, and the higher the level of n in the n-gram, the lower the confusion, suggesting that the model fits the data better.

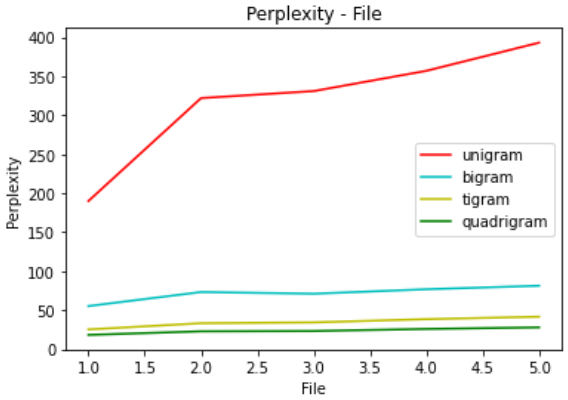


Figure 1: perplexity

For a sentence challenge using the n-gram model, in the case of trigram, select three words to the left of a sentence gap, enter five options in trigram, and select the most likely selection.

He held in his hand a _____ of blue paper

Figure 2: question sentence token selection

The n-gram model was trained five times in a random set of 10 files from the Gutenberg Project text set, with an average score of pairs as shown in figure 3. For reference, the five specific data and the total score are given in table 1 and the full data in the appendix.

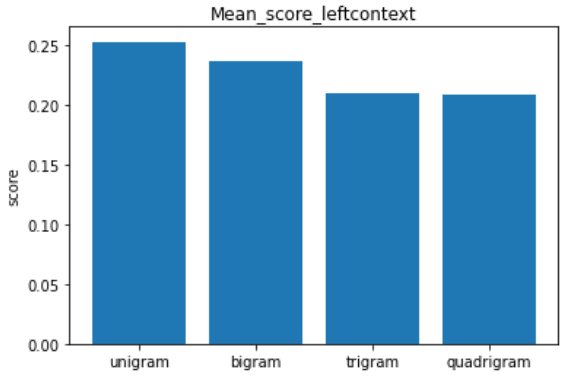


Figure 3: mean score of left contexts

File size: 10 Tests: 5

The n-gram model in figure 3 corresponds to an average score. We found that accuracy did not increase with the increase in n-gram size, but decreased. Selecting only the text on the left side of a sentence is so limiting that we can only get problem-solving information from a small number of sentences, and the more the n-gram size increases, the greater the difference. To illustrate this and improve the score, we need to select the right side of the question statement at the same time.

He held in his hand a _____ of blue paper

He held in his hand a _____ of blue paper

He held in his hand a _____ of blue paper

He held in his hand a _____ of blue paper

Figure 4: question sentence token selection

Each time, a phrase slice will be obtained, which will be turned into a token and searched in the gram dictionary, returning the most likely area selection and selecting the most likely option from the most likely area selection.

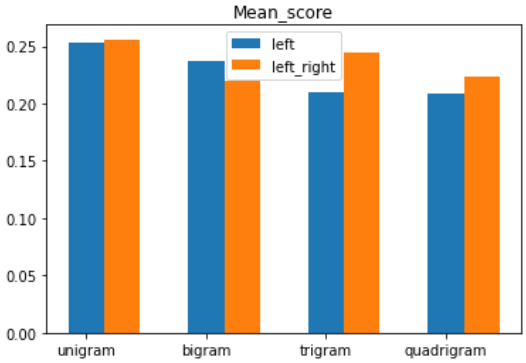


Figure 5: mean score of left contexts and right contexts

File size: 10 Tests: 5

As we can see, unigram's results were almost unchanged, with bigram scoring lower than the left-hand average, and trigram and quadrigram scoring a fraction higher than the left-hand average. This suggests that the composition of the context often requires a combination of text on the left and the right, extending to both sides of a sentence, and adding text on the right to sift through can also increase the model's score.

	left	left right
unigram	0.2530769230769231	0.25596153846153846
bigram	0.23692307692307696	0.21980769230769232
trigram	0.2103846153846154	0.24480769230769234
quadrigram	0.2092307692307692	0.2230769230769231

Table1: mean score of grams

n-gram word embedding

It is very convenient to use word vectors with a large corpus (Levy, Goldberg, 2014), for example, we are looking at similarities between "women" and "man," for example, using Google News-vectors, it is easy to come up with similarities of 0.76640123, which means there is a very high similarity between "woman" and "man."

Based on n-gram method, the word embedding method is improved by using this feature of word vector. Choose dictionaries that include simple bigram and trigram models to find answers with the highest similarity to words with the highest probability.

To unify the method, use the token acquisition method shown in Figure II to obtain the text content on the left for scoring training. When the model seeks the conditional

probability word corresponding to the text on the left, instead of selecting the word with the highest probability from the options, it uses a word vector to select the word with the highest probability similarity. However, there are a lot of non-critical words and punctuation, so we can't pass in word vectors for similarity calculation, we need to filter them first. The 10 most frequently occurring tokens are given in Figure 6.

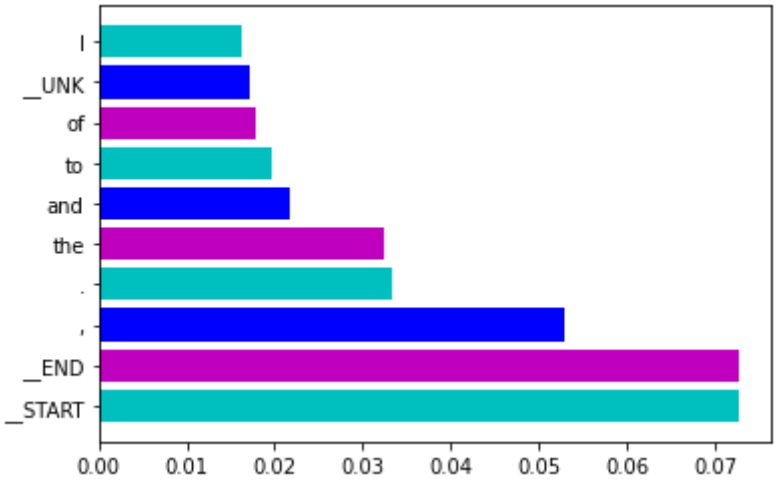


Figure 6: high frequency words

We found that many high-frequency words cannot be passed directly to word vectors and need to be filtered sequentially until a token that can be passed in is selected. Here are the results of experiments where n-gram and word embedding work together.

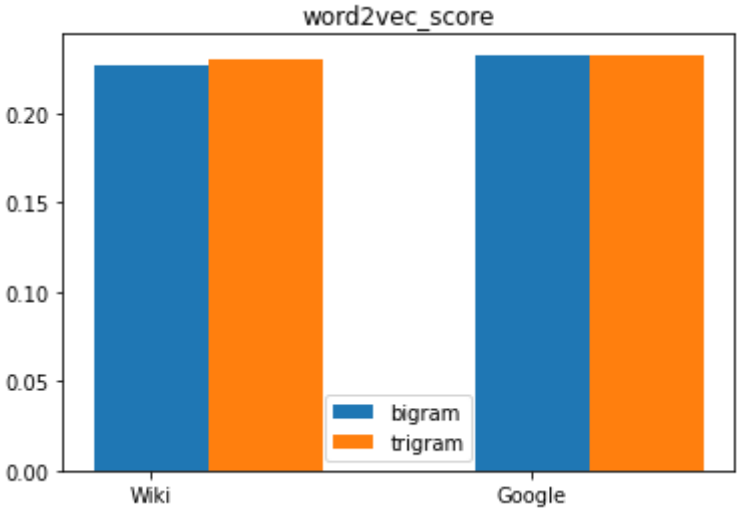


Figure 6: word vectors mean score

	bigram	trigram
fasttext-wiki-news	0.2273076923076923	0.2328846153846154
GoogleNews-vectors	0.2303846153846154	0.2325

Table 2: Average score in five Tests

Number of texts: 10 Number of tests: 5

The word-vector dataset used in this experiment is from Google and Wiki, and detailed links will be shown in the appendix. As it turned out, training the same 10 Gutenberg Project (Zweig and Burges, 2011) text datasets did not significantly improve scores compared to the simple n-gram model, and the results showed a completely different stable rotation. The training scores were close to 0.23 and did not differ significantly on either bigram or trigram, suggesting that the word-embedding method used cosine distances generated by word modeling as similarity, with a more stable output, while producing results that were not directly related to n-gram size. We can infer that word vector method is a stable output method that does not need to rely on training text especially because it is based on a large corpus.

Conclusions and Future Work

From the results of these several different approaches to the sentence challenge, the scores were not very high, mostly just above the randomly selected 0.2 and below the 0.25. N-gram has a simpler structure and is highly expansive, using the Kneser-Ney retreat in addition to the discount method. Of course, it is not enough to set the number of texts at 10. There is still a lot of text training to be done. In the course of the experiment, when the question was parsed into the quadrigram model, there were many sentences challenging title openings with less than three left or right text options that could not be used, greatly increasing errors and reducing training accuracy and scores. The effect of punctuation and other distractions on scores should also be taken into account when text processing is done. The text needs to be processed and then retrained, and then compared with the results of unprocessed text training. The code invocation, visualization method is more cumbersome, needs to further improve its own programming ability and academic training

References

Zweig G, Burges C J C. The microsoft research sentence completion challenge[R]. Microsoft Research Technical Report MSR-TR-2011-129, 2011.

Jurafsky D. An on-line computational model of human sentence interpretation[C]//Proceedings of the tenth national conference on Artificial intelligence. 1992: 302-308.

Cavnar W B, Trenkle J M. N-gram-based text categorization[C]//Proceedings of SDAIR-94, 3rd annual symposium on document analysis and information retrieval. 1994, 161175.

Journel A G. Markov models for cross-covariances[J]. Mathematical Geology, 1999, 31(8): 955-964.

Candès E J, Sur P. The phase transition for the existence of the maximum likelihood estimate in high-dimensional logistic regression[J]. The Annals of Statistics, 2020, 48(1): 27-42.

Dumais S T, Letsche T A, Littman M L, et al. Automatic cross-language retrieval using latent semantic indexing[C]//AAAI spring symposium on cross-language text and speech retrieval. 1997, 15: 21.
Levy O, Goldberg Y. Dependency-based word embeddings[C]//Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers). 2014: 302-308.

Moore R C, Quirk C. Improved smoothing for N-gram language models based on ordinary counts[C]//Proceedings of the ACL-IJCNLP 2009 Conference Short Papers. 2009: 349-352.

Appendix

progress log

Date	Contents
20/03/2022	Complete the sentence challenge for n-gram models
23/03/2022	Complete the Word Embedding Method Sentence Challenge
27/03/2022	Optimize the model to add more comparison options
01/04/2022	Analysis, test, obtain experimental data, and begin drafting reports
05/04/2022	Inspection and Varsity

Project Goldberg Text Collection and Machine Challenge Links:
MSR Sentence Completion Challenge - Microsoft Research

English word vectors:
https://fasttext.cc/docs/en/english-vectors.html

Google Pre-trained word and phrase vectors:
https://code.google.com/archive/p/word2vec/

For space reasons, the complete contents of Tables 1 and 2 are shown in the appendix:

	1	2	3	4	5	Mean
left unigram score	0.2528846153846154	0.2567307692307692	0.2625	0.25384615384615383	0.23942307692307693	0.2530769230769231
left bigram score	0.2461538461538617	0.2394230769230769 3	0.2413461538461538 5	0.2394230769230769 3	0.2182692307692307 7	0.2369230769230769 6
left trigram score	0.2346153846153846 1	0.2221153846153846	0.1836538461538461 7	0.2105769230769230 8	0.2009615384615384 7	0.2103846153846154
left quadrigram score	0.2259615384615384 6	0.2125	0.1980769230769230 7	0.225	0.1846153846153846 3	0.2092307692307692
left&right	0.2759615384615385	0.2336538461538461 6	0.2461538461538461 7	0.2576923076923076 6	0.2663461538461538 4	0.2559615384615384 6

unigram score						
left&right bigram score	0.23461538461538461	0.22403846153846155	0.20576923076923076	0.21538461538461538	0.21923076923076923	0.21980769230769232
left&right trigram score	0.29615384615384616	0.21826923076923077	0.2375	0.24038461538461538	0.23173076923076924	0.24480769230769234
left&right quadrigram score	0.22115384615384615	0.23269230769230769	0.23461538461538461	0.21153846153846154	0.21538461538461538	0.22307692307692307

Table1.1: mean score of grams

	1	2	3	4	5	Mean
WikiNews-vectors bigram score	0.22403846153846155	0.23461538461538461	0.21538461538461538	0.21923076923076923	0.24326923076923077	0.22730769230769230
WikiNews-vectors trigram score	0.22596153846153846	0.22596153846153846	0.23557692307692307	0.23173076923076924	0.24519230769230768	0.23288461538461538
Google News-vectors bigram score	0.23846153846153847	0.23461538461538461	0.22403846153846155	0.23076923076923078	0.22403846153846155	0.23038461538461538
Google News-vectors trigram score	0.22692307692307692	0.225	0.23942307692307693	0.22019230769230769	0.25096153846153846	0.2325

Table 2.1: Average score in five Tests

Extra try:

Tried using neural network for sentence challenge, but even putting the model on cuda:0, the efficiency is still very slow, and after 10 trainings it still contains high loss value.

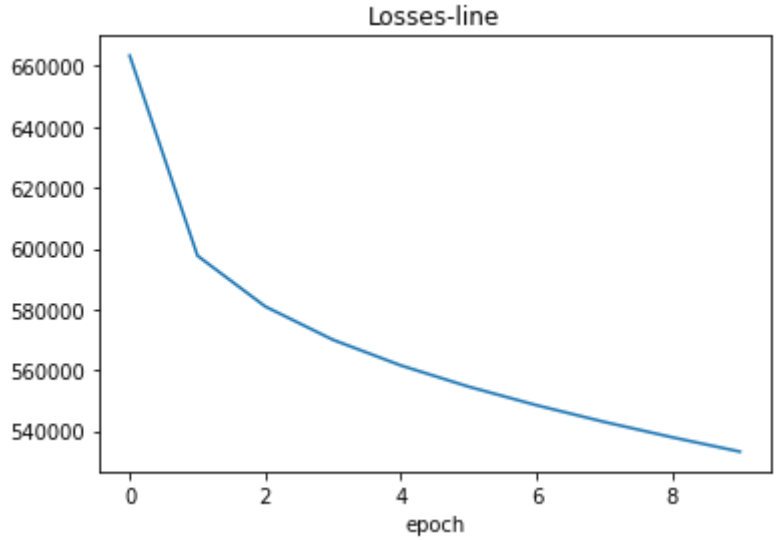
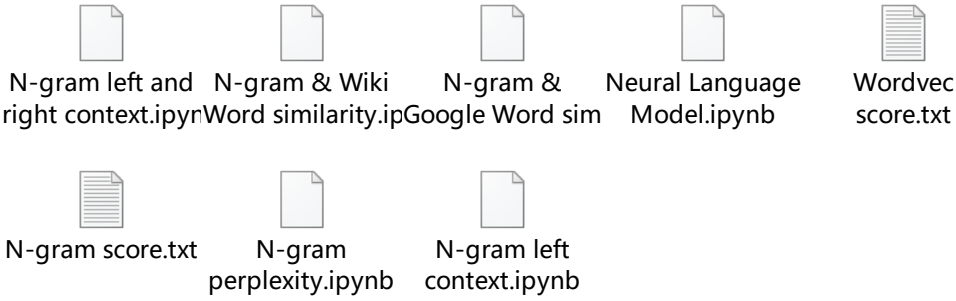


Figure 7: loss of training text

Code files




```
# Import package
from nltk import word_tokenize as tokenize
import pandas as pd, csv
import operator
import os,random,math
import numpy as np

# Import file directory
path = ".../ lab2resources\sentence-completion\Holmes_Training_Data"

filenames = os.listdir(path)
n = len(filenames)
random.shuffle(filenames) # Randomly shuffle file names
trainingfiles = filenames[:int(n*0.5)] # Randomly select the name of training set file
heldoutfiles = filenames[int(n*0.5):] # Then grab the test set file name
```

```
class lanugage_model:

    def __init__(self,path,filesize,method):
        self.words = [] # An empty list of words used to store word tokens
        self.unigram = {} # An empty dictionary for storing unigram words
        self.bigram = {}
        self.trigram = {}
        self.quadrigram = {}

        self.gram = {} # The final gram is consistent with the method

        self.path = path # Get road strength
        self.filesize = filesize # Get file batch size
        self.method = method # Selection method

        self.get_words()
        self._processfiles()
        self._make_unknowns()
        self._discount()
        self._convert_to_probs()

    def get_words(self):
        # Get the word token, and add "_start" and "_end"
        # to mark the beginning and end of each sentence
        for file in trainingfiles[:self.filesize]:
            print(f"processing {file}.text ")
            try:
                with open(os.path.join(path,file)) as instream:
                    for line in instream:
                        line = line.rstrip()
                        if len(line)>0:
                            tokens = tokens=["__START"]+tokenize(line)+["__END"]
                            self.words.append(tokens)
            except UnicodeDecodeError:
                print("UnicodeDecodeError processing {}: ignoring rest of
file".format(file))


    def _processfiles(self):

        for i in self.words: # Get unigram model words and word count
            for j in i:
                self.unigram[j] = 0
```

```
        for i in self.words:
            for j in i:
                self.unigram[j] += 1

        if self.method == "bigram": # Get bigram model words and the number of words after each word
            for i in self.words:
                for j in range(len(i)-1):
                    self.bigram[i[j]] = {}
            for i in self.words:
                for j in range(len(i)-1):
                    self.bigram[i[j]][i[j+1]] = 0
            for i in self.words:
                for j in range(len(i)-1):
                    self.bigram[i[j]][i[j+1]] += 1

            self.gram = self.bigram # Finally, self The gram dictionary will store the same gram element as method.

        if self.method == "trigram": # Get trigram model words and the number of words after each word
            for i in self.words:
                for j in range(len(i)-2):
                    self.trigram[i[j]] = {}
            for i in self.words:
                for j in range(len(i)-2):
                    self.trigram[i[j]][i[j+1],i[j+2]] = 0
            for i in self.words:
                 for j in range(len(i)-2):
                    self.trigram[i[j]][i[j+1],i[j+2]] += 1

            self.gram = self.trigram

        if self.method == "quadrigram": # Get the word of the quadrigram model and the number of words after each word
            for i in self.words:
                for j in range(len(i)-3):
                    self.quadrigram[i[j]] = {}
            for i in self.words:
                for j in range(len(i)-3):
                    self.quadrigram[i[j]][i[j+1],i[j+2],i[j+3]] = 0
            for i in self.words:
                 for j in range(len(i)-3):
```

```
                    self.quadrigram[i[j]][i[j+1],i[j+2],i[j+3]] += 1

            self.gram = self.quadrigram


    def _convert_to_probs(self): # term frequency
        self.unigram = {k:v/sum(self.unigram.values()) for (k,v) in self.unigram.items()}

        # Calculate the word frequency of packaged bigram or trigram or quadrigram,
and return to gram dictionary
        self.gram = {key:{k:v/sum(adict.values()) for (k,v) in adict.items()} for (key,adict)
in self.gram.items()}

    def get_prob(self,token,context=""):
        # Be computed_ prob_ Line function call, which is used to count the probability
of obtaining words and calculate perflexity

        if self.method == "unigram":
            return self.unigram.get(token,self.unigram.get("__UNK",0))
        else:
            gram = self.gram.get(context[-1],self.gram.get("__UNK",{})) # Probability of
getting "__unk" token
            big_p = gram.get(token,gram.get("__UNK",0))

            lmbda = gram["__DISCOUNT"] # Get "_discount" token probability

            uni_p = self.unigram.get(token,self.unigram.get("__UNK",0))
            #print(big_p,lmbda,uni_p)
            p = big_p + lmbda * uni_p
            return p

    def nextlikely(self,current=""):    # Screen the words most likely to form a sentence
        blacklist=["__START","__DISCOUNT"]

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

        mostlikely=list(dist.items())
        # filter out any undesirable tokens
        filtered=[(w,p) for (w,p) in mostlikely if w not in blacklist]
        print(current,len(filtered))
        # choose one randomly from the top k
```

```
        words,probdist = zip(*filtered)
        res = random.choices(words,probdist)[0]
        return res

    def generate(self,end="__END",limit=20): # Generate a possible sentence
        current="__START"
        tokens=[]
        while    current != end and len(tokens) < limit:
            current=self.nextlikely(current=current)
            tokens.append(current)
        return " ".join(tokens[:-1])

    def compute_prob_line(self,line):
       # Be computed_ The probability function call is used to calculate the probability
        tokens=["__START"]+tokenize(line)+["__END"]
        acc=0
        for i,token in enumerate(tokens[1:]):
            acc += np.log(self.get_prob(token,tokens[:i+1]))
        return acc,len(tokens[1:])

    def compute_probability(self):
        # computes the probability (and length) of a corpus contained in filenames
        # Select the test set file name to calculate the property
        total_p=0
        total_N=0
        for i,afile in enumerate(heldoutfiles[:self.filesize]):
            print("Processing file {}:{}".format(i,afile))
            try:
                with open(os.path.join(path,afile)) as instream:
                    for line in instream:
                        line=line.rstrip()
                        if len(line)>0:
                            p,N = self.compute_prob_line(line)
                            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):

        #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()
        #print(p,N)
        pp = np.exp(-p/N)
        return pp

    def _make_unknowns(self,known=2): # Add unknown word token "_unk"
        unknown = 0
        if self.method == "unigram"or self.method == "bigram":
            for (k,v) in list(self.unigram.items()):
                if v < known: # If less than, the word is deleted from the dictionary and
replaced with a "_unk" token
                    del self.unigram[k]
                    self.unigram["__UNK"] = self.unigram.get("__UNK",0) + v
            for (k,adict) in list(self.gram.items()):
                for (kk,v) in list(adict.items()):
                    isknown = self.unigram.get(kk,0)
                    if isknown == 0: # If equal to 0, the word is removed from the
dictionary and replaced with a "_unk" token
                        adict["__UNK"] = adict.get("__UNK",0) + v
                        del adict[kk]
                isknown = self.unigram.get(k,0)
                if isknown == 0:
                    del self.gram[k]
                    current = self.gram.get("__UNK",{})
                    current.update(adict)
                    self.gram["__UNK"] = current

                else:
                    self.gram[k] = adict

        if self.method == "trigram"or self.method == "quadrigram":
            for (k,v) in list(self.unigram.items()):
                if v < known:
                    del self.unigram[k]
                    self.unigram["__UNK"] = self.unigram.get("__UNK",0) + v

            #Trigram and quadrigram contain a word pair tuple, which needs to add a
layer of loop
            for (k,adict) in list(self.gram.items()):
                for (kk,v) in list(adict.items()):
                    for vv in kk:
                        isknown = self.unigram.get(vv,0)
```

```
                        dels = False
                        if isknown == 0:
                            adict["__UNK"] = adict.get("__UNK",0) + v
                            dels = True
                    if dels == True:
                        del adict[kk]
                isknown = self.unigram.get(k,0)
                if isknown == 0:
                    del self.gram[k]
                    current = self.gram.get("__UNK",{})
                    current.update(adict)
                    self.gram["__UNK"] = current

                else:
                    self.gram[k] = adict

    def _discount(self,discount=0.75):
        #discount each bigram count by a small fixed amount
        self.gram={k:{kk:value-discount for (kk,value) in adict.items()}for (k,adict) in
self.gram.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.gram.keys():
            lamb = len(self.gram[k])
            self.gram[k]["__DISCOUNT"] = lamb * discount

mylm = lanugage_model(path,10,"trigram")
mylm.compute_perplexity()

# Drawing
import matplotlib.pyplot as plt
import numpy as np

methods = ["unigram","bigram","trigram","quadrigram"]

unigram = []
bigram = []
trigram = []
quadrigram = []

a = np.arange(1,6)
```

```
for method in methods:
    for file in range(1,6):
            mylm = lanugage_model(path,file,method)
            perplexity = mylm.compute_perplexity()
            eval(method).append(perplexity)

fig = plt.figure(1)
ax = plt.subplot(111)
ax.plot(a, unigram, 'r',label='unigram')
ax.plot(a, bigram, 'c',label='bigram')
ax.plot(a, trigram,'y',label='tigram')
ax.plot(a, quadrigram,'g',label='quadrigram')
ax.legend()
plt.xlabel('File')
plt.ylabel('Perplexity')
plt.title('Perplexity - File')
plt.show()

parentdir =    "..\lab2resources\sentence-completion"

questions=os.path.join(parentdir,"testing_data.csv")
answers=os.path.join(parentdir,"test_answer.csv")

with open(questions) as instream:
    csvreader=csv.reader(instream)
    lines=list(csvreader)
qs_df=pd.DataFrame(lines[1:],columns=lines[0])

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 []

def get_right_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+1:i+window+1]
    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))# It needs to be converted to 3 when running the 4-ary model
qs_df['right_context']=qs_df['tokens'].map(lambda x: get_right_context(x,2))

qs_df.to_csv("…\sentence-completion\data.csv")#Restore the question set in the current directory, including the left or right text
questions=os.path.join(parentdir,"data.csv")

class question:

    def __init__(self,aline):
        self.fields=aline

    def get_field(self,field):
        return self.fields[question.colnames[field]]

    def get_context(self,field):
        left = eval(self.fields[question.colnames["left_context"]])
        option = self.fields[question.colnames[field]]
        left.append(option)
        right = eval(self.fields[question.colnames["right_context"]])

        return left + right

    def add_answer(self,fields):
        self.answer=fields[1]

    def chooseA(self):
        return("a")

    def chooserandom(self):
        choices=["a","b","c","d","e"]
        return np.random.choice(choices)

    def chooseunigram(self,lm):
```

```
        choices=["a","b","c","d","e"]
        probs=[lm.unigram.get(self.get_field(ch+")"),0) for ch in choices]
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
        #if len(bestchoices)>1:
        #       print("Randomly choosing from {}".format(len(bestchoices)))
        return np.random.choice(bestchoices)

    def choosebigram(self,lm):
        choices = ["a","b","c","d","e"]
        probs_total = []
        for ch in choices:
            context = self.get_context(ch+")")
            probs = 0
            for i in range(1,len(context)-2):
                prob = lm.gram.get(context[i],{}).get(context[i+1],0)
                probs += prob
            probs_total.append(probs)
        maxprob = max(probs_total)
        bestchoices=[ch for ch,prob in zip(choices,probs_total) if prob == maxprob]
        return np.random.choice(bestchoices)


    def choosetrigram(self,lm):
        choices = ["a","b","c","d","e"]
        probs_total = []
        for ch in choices:
            context = self.get_context(ch+")")
            probs = 0
            for i in range(len(context)-2):
                prob = lm.gram.get(context[i],{}).get((context[i+1],context[i+2]),0)
                probs += prob
            probs_total.append(probs)
        maxprob = max(probs_total)
        bestchoices=[ch for ch,prob in zip(choices,probs_total) if prob == maxprob]
        return np.random.choice(bestchoices)

    def choosequadrigram(self,lm):
        choices = ["a","b","c","d","e"]
        probs_total = []
        for ch in choices:
            context = self.get_context(ch+")")
            probs = 0
            for i in range(len(context)-3):
```

```
                    prob =
lm.gram.get(context[i],{}).get((context[i+1],context[i+2],context[i+3]),0)
                    probs += prob
                probs_total.append(probs)
            maxprob = max(probs_total)
            bestchoices=[ch for ch,prob in zip(choices,probs_total) if prob == maxprob]
            return np.random.choice(bestchoices)

        def predict(self,method="chooseA",lm=mylm):
            #eventually there will be lots of methods to choose from
            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)

        def predict_and_score(self,method="chooseA"):

            #compare prediction according to method with the correct answer
            #return 1 or 0 accordingly
            prediction=self.predict(method=method)
            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):
        return [q.get_field(field) for q in self.questions]

    def predict(self,method="chooseA"):
        return [q.predict(method=method) for q in self.questions]

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

N-gram Word similarity

```
from nltk import word_tokenize as tokenize
import pandas as pd, csv
import operator
import os,random,math
import numpy as np
from gensim.models import KeyedVectors,Word2Vec, FastText
from gensim.downloader import base_dir

filename="..GoogleNews-vectors-negative300.bin"    #fasttext-wiki-news-subwords-300
mymodel = KeyedVectors.load_word2vec_format(filename, binary=True)

# Import file directory
path = ".../ lab2resources\sentence-completion\Holmes_Training_Data"
```

```
filenames = os.listdir(path)
n = len(filenames)
random.shuffle(filenames) # Randomly shuffle file names
trainingfiles = filenames[:int(n*0.5)] # Randomly select the name of training set file
heldoutfiles = filenames[int(n*0.5):] # Then grab the test set file name

class lanugage_model:

    def __init__(self,path,filesize,method):
        self.words = [] # An empty list of words used to store word tokens
        self.unigram = {} # An empty dictionary for storing unigram words
        self.bigram = {}
        self.trigram = {}
        self.quadrigram = {}

        self.gram = {} # The final gram is consistent with the method

        self.path = path # Get road strength
        self.filesize = filesize # Get file batch size
        self.method = method # Selection method

        self.get_words()
        self._processfiles()
        self._make_unknowns()
        self._discount()
        self._convert_to_probs()

    def get_words(self):
        # Get the word token, and add "_start" and "_end"
        # to mark the beginning and end of each sentence
        for file in trainingfiles[:self.filesize]:
            print(f"processing {file}.text ")
            try:
                with open(os.path.join(path,file)) as instream:
                    for line in instream:
                        line = line.rstrip()
                        if len(line)>0:
                            tokens = tokens=["__START"]+tokenize(line)+["__END"]
                            self.words.append(tokens)
            except UnicodeDecodeError:
                print("UnicodeDecodeError processing {}: ignoring rest of
file".format(file))
```

```
    def _processfiles(self):

        for i in self.words: # Get unigram model words and word count
            for j in i:
                self.unigram[j] = 0

        for i in self.words:
            for j in i:
                self.unigram[j] += 1

        if self.method == "bigram": # Get bigram model words and the number of words after each word
            for i in self.words:
                for j in range(len(i)-1):
                    self.bigram[i[j]] = {}
            for i in self.words:
                for j in range(len(i)-1):
                    self.bigram[i[j]][i[j+1]] = 0
            for i in self.words:
                for j in range(len(i)-1):
                    self.bigram[i[j]][i[j+1]] += 1

            self.gram = self.bigram # Finally, self The gram dictionary will store the same gram element as method.

        if self.method == "trigram": # Get trigram model words and the number of words after each word
            for i in self.words:
                for j in range(len(i)-2):
                    self.trigram[i[j]] = {}
            for i in self.words:
                for j in range(len(i)-2):
                    self.trigram[i[j]][i[j+1],i[j+2]] = 0
            for i in self.words:
                  for j in range(len(i)-2):
                    self.trigram[i[j]][i[j+1],i[j+2]] += 1

            self.gram = self.trigram

        if self.method == "quadrigram": # Get the word of the quadrigram model and the number of words after each word
            for i in self.words:
                for j in range(len(i)-3):
```

```
                    self.quadrigram[i[j]] = {}
            for i in self.words:
                for j in range(len(i)-3):
                    self.quadrigram[i[j]][i[j+1],i[j+2],i[j+3]] = 0
            for i in self.words:
                for j in range(len(i)-3):
                    self.quadrigram[i[j]][i[j+1],i[j+2],i[j+3]] += 1

            self.gram = self.quadrigram


    def _convert_to_probs(self): # term frequency
        self.unigram = {k:v/sum(self.unigram.values()) for (k,v) in self.unigram.items()}

        # Calculate the word frequency of packaged bigram or trigram or quadrigram,
and return to gram dictionary
        self.gram = {key:{k:v/sum(adict.values()) for (k,v) in adict.items()} for (key,adict)
in self.gram.items()}

    def get_prob(self,token,context=""):
        # Be computed_ prob_ Line function call, which is used to count the probability
of obtaining words and calculate perflexity

        if self.method == "unigram":
            return self.unigram.get(token,self.unigram.get("__UNK",0))
        else:
            gram = self.gram.get(context[-1],self.gram.get("__UNK",{})) # Probability of
getting "__unk" token
            big_p = gram.get(token,gram.get("__UNK",0))

            lmbda = gram["__DISCOUNT"] # Get "_discount" token probability

            uni_p = self.unigram.get(token,self.unigram.get("__UNK",0))
            #print(big_p,lmbda,uni_p)
            p = big_p + lmbda * uni_p
            return p

    def nextlikely(self,current=""):    # Screen the words most likely to form a sentence
        blacklist=["__START","__DISCOUNT"]

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

```
        mostlikely=list(dist.items())
        # filter out any undesirable tokens
        filtered=[(w,p) for (w,p) in mostlikely if w not in blacklist]
        print(current,len(filtered))
        # choose one randomly from the top k
        words,probdist = zip(*filtered)
        res = random.choices(words,probdist)[0]
        return res

    def generate(self,end="__END",limit=20): # Generate a possible sentence
        current="__START"
        tokens=[]
        while    current != end and len(tokens) < limit:
            current=self.nextlikely(current=current)
            tokens.append(current)
        return " ".join(tokens[:-1])

    def compute_prob_line(self,line):
       # Be computed_ The probability function call is used to calculate the probability
        tokens=["__START"]+tokenize(line)+["__END"]
        acc=0
        for i,token in enumerate(tokens[1:]):
            acc += np.log(self.get_prob(token,tokens[:i+1]))
        return acc,len(tokens[1:])

    def compute_probability(self):
        # computes the probability (and length) of a corpus contained in filenames
        # Select the test set file name to calculate the property
        total_p=0
        total_N=0
        for i,afile in enumerate(heldoutfiles[:self.filesize]):
            print("Processing file {}:{}".format(i,afile))
            try:
                with open(os.path.join(path,afile)) as instream:
                    for line in instream:
                        line=line.rstrip()
                        if len(line)>0:
                            p,N = self.compute_prob_line(line)
                            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):

        #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()
        #print(p,N)
        pp = np.exp(-p/N)
        return pp

    def _make_unknowns(self,known=2): # Add unknown word token "_unk"
        unknown = 0
        if self.method == "unigram"or self.method == "bigram":
            for (k,v) in list(self.unigram.items()):
                if v < known: # If less than, the word is deleted from the dictionary and
replaced with a "_unk" token
                    del self.unigram[k]
                    self.unigram["__UNK"] = self.unigram.get("__UNK",0) + v
            for (k,adict) in list(self.gram.items()):
                for (kk,v) in list(adict.items()):
                    isknown = self.unigram.get(kk,0)
                    if isknown == 0: # If equal to 0, the word is removed from the
dictionary and replaced with a "_unk" token
                        adict["__UNK"] = adict.get("__UNK",0) + v
                        del adict[kk]
                isknown = self.unigram.get(k,0)
                if isknown == 0:
                    del self.gram[k]
                    current = self.gram.get("__UNK",{})
                    current.update(adict)
                    self.gram["__UNK"] = current

                else:
                    self.gram[k] = adict

        if self.method == "trigram"or self.method == "quadrigram":
            for (k,v) in list(self.unigram.items()):
                if v < known:
                    del self.unigram[k]
                    self.unigram["__UNK"] = self.unigram.get("__UNK",0) + v
```

```
                #Trigram and quadrigram contain a word pair tuple, which needs to add a
layer of loop
                for (k,adict) in list(self.gram.items()):
                    for (kk,v) in list(adict.items()):
                        for vv in kk:
                            isknown = self.unigram.get(vv,0)
                            dels = False
                            if isknown == 0:
                                adict["__UNK"] = adict.get("__UNK",0) + v
                                dels = True
                        if dels == True:
                            del adict[kk]
                    isknown = self.unigram.get(k,0)
                    if isknown == 0:
                        del self.gram[k]
                        current = self.gram.get("__UNK",{})
                        current.update(adict)
                        self.gram["__UNK"] = current

                    else:
                        self.gram[k] = adict

    def _discount(self,discount=0.75):
        #discount each bigram count by a small fixed amount
        self.gram={k:{kk:value-discount for (kk,value) in adict.items()}for (k,adict) in
self.gram.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.gram.keys():
            lamb = len(self.gram[k])
            self.gram[k]["__DISCOUNT"] = lamb * discount

mylm = lanugage_model(path,10,"trigram")
mylm.compute_perplexity()

parentdir =    "…sentence-completion"

questions=os.path.join(parentdir,"testing_data.csv")
answers=os.path.join(parentdir,"test_answer.csv")

with open(questions) as instream:
    csvreader=csv.reader(instream)
```

```
    lines=list(csvreader)
qs_df=pd.DataFrame(lines[1:],columns=lines[0])

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))
qs_df.to_csv("D:\Study\自然语言工程\高级
\week2\lab2resources\lab2resources\sentence-completion\data.csv")
questions=os.path.join(parentdir,"data.csv")
class question:

    def __init__(self,aline):
        self.fields=aline

    def get_field(self,field):
        return self.fields[question.colnames[field]]

    def get_context(self,field):
        left = eval(self.fields[question.colnames["left_context"]])
        option = self.fields[question.colnames[field]]
        left.append(option)
        return left

    def add_answer(self,fields):
        self.answer=fields[1]

    def chooseA(self):
        return("a")

    def chooserandom(self):
        choices=["a","b","c","d","e"]
        return np.random.choice(choices)

    def chooseunigram(self,lm):
```

```
        choices=["a","b","c","d","e"]
        probs=[lm.unigram.get(self.get_field(ch+")"),0) for ch in choices]
        maxprob=max(probs)
        bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
        #if len(bestchoices)>1:
        #       print("Randomly choosing from {}".format(len(bestchoices)))
        return np.random.choice(bestchoices)

    def choosebigram(self,lm):
        choices = ["a","b","c","d","e"]
        probs_total = []
        for ch in choices:
            context = self.get_context(ch+")")
            awnser_probs = lm.gram.get(context[1],{})#获取答案前一个词相关概率的所
有词
            vocab_bigram = sorted(awnser_probs.items(),key=lambda x:x[1],reverse
=True)#根据概率从大到小排列
            for i in vocab_bigram:
                try:
                    prob = mymodel.similarity(i[0],context[2])
                    probs_total.append(prob)
                    break

                except KeyError:
                    continue
        try:
            maxprob = max(probs_total)
            bestchoices=[ch for ch,prob in zip(choices,probs_total) if prob == maxprob]
            return np.random.choice(bestchoices)
        except ValueError:
            return np.random.choice(choices)

    def choosetrigram(self,lm):
        choices = ["a","b","c","d","e"]
        probs_total = []
        for ch in choices:
            context = self.get_context(ch+")")
            awnser_probs = lm.gram.get(context[0],{})#获取答案前一个词相关概率的所
有词
            vocab_bigram = sorted(awnser_probs.items(),key=lambda x:x[1],reverse
=True)#根据概率从大到小排列
            #遍历和前一个词相关的所有二元词组，如果
            for i in vocab_bigram:
                try:
```

```
                    if i[0][0] == context[1]:
                        prob = mymodel.similarity(i[0][1],context[2])
                        probs_total.append(prob)
                        break

                except KeyError:
                    continue
        try:
            maxprob = max(probs_total)
            bestchoices=[ch for ch,prob in zip(choices,probs_total) if prob == maxprob]
            return np.random.choice(bestchoices)
        except ValueError:
                return np.random.choice(choices)

    def choosequadrigram(self,lm):
        choices = ["a","b","c","d","e"]
        probs_total = []
        for ch in choices:
            context = self.get_context(ch+")")
            prob = lm.gram.get(context[0],{}).get((context[1],context[2],context[3]),0)
            probs_total.append(prob)
        maxprob = max(probs_total)
        bestchoices=[ch for ch,prob in zip(choices,probs_total) if prob == maxprob]
        return np.random.choice(bestchoices)

    def predict(self,method="chooseA",lm=mylm):
        #eventually there will be lots of methods to choose from
        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)

    def predict_and_score(self,method="chooseA"):

        #compare prediction according to method with the correct answer
        #return 1 or 0 accordingly
```

```
            prediction=self.predict(method=method)
            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):
            return [q.get_field(field) for q in self.questions]

    def predict(self,method="chooseA"):
            return [q.predict(method=method) for q in self.questions]

    def predict_and_score(self,method="chooseA"):
            scores=[q.predict_and_score(method=method) for q in self.questions]
```

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

SCC = scc_reader()

score_trigram = SCC.predict_and_score(method="trigram")
print(score_trigram)
```