Contents

1. Introduction

1.1. Overview

This project aims to solve the Microsoft Research Sentence Completion Challenge (Zweig and Burges, 2011) by using n-gram language models and distributional similarity method with Word2Vec. We propose a unigram and bigram model as a baseline and examine the performance of a trigram model and the distributional similarity method. Smoothing methods including absolute discounting and backoff have been implemented to the n-gram models. We try to find the effect of giving different hyper-parameter values and different numbers of training documents to the n-gram models. Perplexity has been used as a performance measure of the n-gram models and overall performance of all models including the distributional similarity model on the sentence completion challenge has been calculated by their accuracy of how many correct answers they give out of all questions.

1.2. Problem Outline

The Microsoft Research Sentence Completion Challenge (Zweig and Burges, 2011) presents the MSR Sentence Completion Challenge Data, which consists of 1,040 sentences. Each sentence has four impostor sentences, in which a single (fixed) word in the original sentence has been replaced by an impostor word with similar occurrence statistics. For each sentence, the task is then to determine which of the five choices for that word is the correct one. Seed sentences were selected from five of Sir Arthur Conan Doyle's Sherlock Holmes novels, and then imposter words were suggested with the aid of a language model trained on over 500 19th century novels. The language model was used to compute 30 alternative words for a given low frequency word in a sentence, and human judges then picked the 4 best impostor words, based on a set of provided guidelines.

2. Methods

2.1. N-gram Language Models

The n-gram models were trained with the Holmes Training Dataset provided by the original paper. The dataset consists of 522 documents; however, we did not use all of them for training due to limited computational resources. Up to 100 documents were used for each training and calculating perplexity according to the method given – we tried to measure how giving different numbers of documents during training and testing affect the model's perplexity. The detailed method is explained in *3.1.4. Hyper-parameter Settings & Size of Training Documents*.

Figure 1 illustrates how each n-gram model processed a sentence from training corpus and formed an n-gram dictionary. First, it tokenised the given sentence and added the "__START" and "__END" tokens at each end. After that, it counted the frequency of each word and created a count dictionary. It then added unknown tokens and applied absolute discounting for unseen words in testing corpus and converted the dictionary to probability distributions. An answer for a given question was chosen after looking at these probability distribution dictionaries and selecting one of the options that had the highest probability.

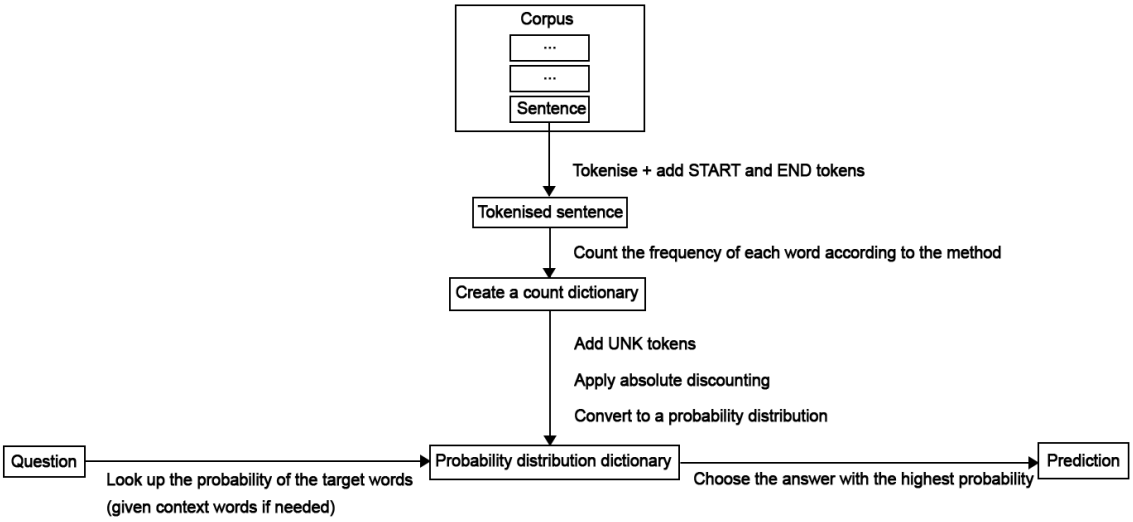

Figure 1. The Structure of the N-gram Models. It shows how the models form probability distribution dictionaries with the sentences from training corpus and choose an answer for given questions according to the dictionaries created.

Figure 2 shows how different n-gram models formed a count dictionary with a given sentence. The example sentence was randomly picked from the training corpus. The detailed explanation on process is in *3.1.1. Unigram and Bigram Models as Baselines* and *3.1.2. Trigram Model*.

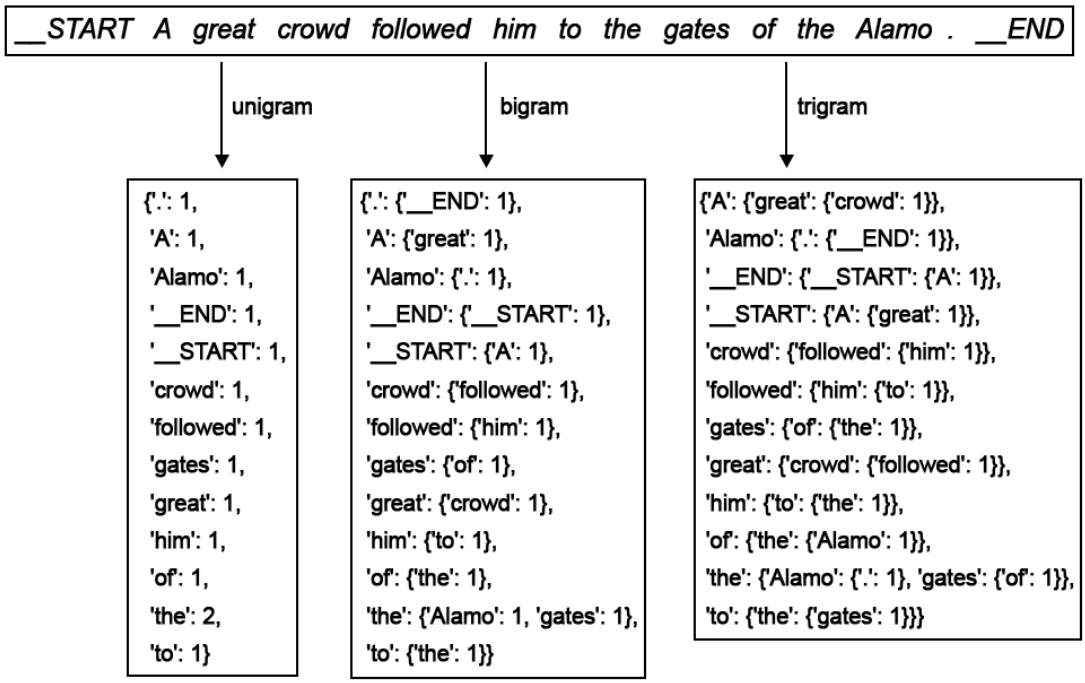

Figure 2. The N-gram Dictionaries of the Example Sentence. It shows how different n-gram language models process a sentence to create a count dictionary. The resulted dictionary is converted to a probability distribution later.

2.1.1. Unigram and Bigram Models as Baselines

Our baseline method uses a unigram and bigram language model. The unigram model calculates the probability of a word by counting how many times the word has been seen in the training corpus (Equation 1). It does not reflect distributional context and only counts the independent occurrence of the word. The bigram model calculates the probability of a word given its adjacent word on its left or right, i.e., it counts how many times the two words have been seen together in the corpus (Equation 2).

$$P(w_n) = C(w_n)$$

(Equation 1)

$$P(w_n|w_{n-1}) = \frac{P(w_{n-1}, w_n)}{P(w_{n-1})} = \frac{C(w_{n-1}, w_n)}{C(w_{n-1})}$$

(Equation 2)

After each model established their count dictionary, we replaced some words in the dictionary with 'UNK' tokens to catch probability of unseen words that would appear in the testing corpora and applied absolute discounting which subtracted a little from each count to save probability mass for the unseen words. The detailed method is explained in *3.1.3. Processing Unseen Words*. Finally, we converted the count dictionaries to probability distributions.

These models were used to choose the answer for each question in the sentence completion challenge. Among five choices for each question, the unigram model chose the ones that had the highest probability of independent occurrence in the training corpus (Equation 3).

$$pred_{unigram} = \underset{w_n \in choices}{\operatorname{argmax}} U(w_n) \ where \ U(w_n) = P(w_n)$$

(Equation 3)

Meanwhile, the bigram model calculated two separate probabilities for each choice word – the probability of the choice word given its previous word and the probability of the next word given the choice word. Thereafter, it multiplied both probabilities and chose one with the highest value as the answer (Equation 4). This allowed the bigram model to consider the surrounding words on both sides of the target word.

$$pred_{bigram} = \underset{w_n \in choices}{\operatorname{argmax}} B(w_n) \ where \ B(w_n) = P(w_n|w_{n-1}) * P(w_{n+1}|w_n)$$

(Equation 4)

2.1.2. Trigram Language Model

Similar to the baseline method using the bigram model, we created a trigram model that takes a look at two previous words of the target word. It established probability distributions for trigrams in the training corpus and went through the same process for unseen words as the baseline models – adding 'UNK' tokens and applying absolute discounting. After that, it calculated two probabilities – the probability of the choice word given its two previous words and the probability of the second next word given the choice word and its

next word – just like the bigram model. The answer for each question in the challenge was chosen by multiplying the two probabilities (Equation 5).

$$pred_{trigram} = \underset{w_n \in choices}{\operatorname{argmax}} T(w_n) \; where \; T(w_n) = P(w_n|w_{n-2}, w_{n-1}) * P(w_{n+2}|w_n, w_{n+1})$$

(Equation 5)

2.1.3. Processing Unseen Words

'UNK' tokens were added to the count dictionaries of the n-gram models in order to represent unseen words in the testing documents. The words that had been seen less than twice in the training documents were replaced with the tokens in the unigram dictionary and this edited dictionary was used by the bigram and trigram model. The bigram model checked if the second word of the given bigram had ever been seen by looking up the edited unigram dictionary and if it had not, the word got replaced with the 'UNK' token in the bigram dictionary. Similar to this, the trigram model checked if the second word and third word were present in the unigram dictionary and replaced them in the trigram dictionary if they were not.

2.1.4. Hyper-parameter Settings & Size of Training Data

The common discount factor which decided how much probability mass a model should give to unseen words was set to 0.3 and the threshold for words to be replaced with the 'UNK' token was set to 2, i.e., words were placed when they had been seen less than twice in the training documents. We tried using different numbers of documents to the models for training and different values of these parameters to the models to examine if they affect the performance and to see if we can ultimately improve our models by adjusting them. The only different parameter given to the models throughout the test was the size of the window used to decide how many context words they would consider calculating probability of a word. The value was 0 for the unigram, 1 for the bigram, and 2 for the trigram model.

2.1.5. Backoff

In addition to the smoothing methods mentioned in *3.1.3. Processing Unseen Words*, we also implemented backoff to the n-gram models which allows to use a lower-order n-gram model when there is zero frequency of a particular n-gram. When we tried to compute $P(w_n|w_{n-2}w_{n-1})$ but there was no example of the trigram $w_{n-2}w_{n-1}w$, we estimated its probability by using the bigram probability $P(w_n|w_{n-1})$. Likewise, when we did not have counts to compute $P(w_n|w_{n-1})$, we looked up the unigram $P(w_n)$. We applied this method to the bigram and the trigram model and measured the performance of these new models with backoff in addition to the pure models without backoff.

2.2. Distributional Similarity Using Word2Vec

We suggest a completely different method that uses distributional similarity between the choice word and the other words in the question sentence. The words were represented as vectors with 300 dimensions by using 'GoogleNews Vectors negative300' which is a pre-trained Word2Vec embedding.

Cosine similarity was used as the metric to calculate distributional similarity between word vectors. Each choice word was scored by this cosine similarity with other word vectors in the

question sentence (Equation 6, 7). The one that had the highest score was then chosen as the answer for the question (Equation 8).

$$similarity_{cos}(W_1, W_2) = \cos(\theta) = \frac{W_1 \cdot W_2}{\|W_1\|\|W_2\|}$$

(Equation 6)

$$score(W') = \sum_n (1 - similarity\_\cos(W', W_n \in question))$$

(Equation 7)

$$pred_{similarity} = \underset{W \in choices}{\operatorname{argmax}} (score(W))$$

(Equation 8)

We tried three different methods that decided how many words from the question sentence should be used to calculate cosine similarity. The first method was including only one word from the left and right, similar to the bigram model. The second was including two words like the trigram model and the last was including all the words in the question. Figure 3 illustrates how each method would compose question word vectors with the example sentence.

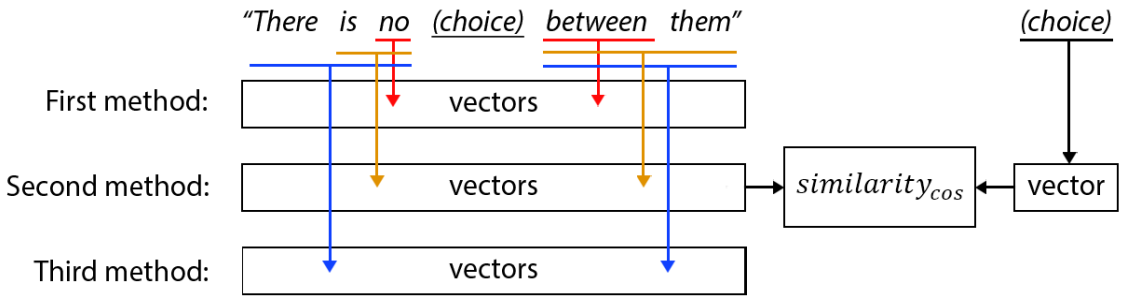

Figure 3. Three Methods of Composing Question Vectors.

2.3. Evaluation

2.3.1. Perplexity of N-gram Models

Beside measuring the performance on the real task, we used a specific evaluation method for the n-gram models to compare their performance to each other. Perplexity is a metric that quantifies how uncertain a model is about the predictions it makes. Lower perplexity means that the model is confident about its predictions, however, it does not necessarily mean that they are accurate. Although, it tends to correlate well with the performance on the real-world tasks.

Perplexity can be calculated using only the probability distribution that the model has learned from the training data. It is the inverse probability of training or testing data, normalised by the number of words (Equation 9), i.e., minimising perplexity is the same as maximising the probability.

$$PP(W) = P(w_1 w_2 \cdots w_n)^{-\frac{1}{n}}$$

(Equation 9)

We tried training the models with different numbers of documents and saw if it affected the perplexity. We gave 20, 50, and 100 training documents in each experiment and calculated perplexity on the same number of testing documents.

2.3.2. Performance on Sentence Completion

We used the trained n-gram models and the distributional similarity model to solve the questions provided in the challenge. First, we counted the number of questions that each model answered correctly. Next, we scored the models by giving one point to a correct answer and calculated accuracy by dividing the score by the number of the questions (Equation 10).

$$accuracy_{model} = \frac{number\ of\ correct\ answers}{number\ of\ total\ questions}$$

(Equation 10)

3. Results & Discussion

3.1. Perplexity of N-gram Models

Figure 4 shows the perplexity of each model on the training and testing data. The perplexity of the unigram model was the highest regardless of the number of training documents and the bigram and trigram model showed the similar perplexity throughout the experiment. However, there was difference that the bigram model tended to show higher perplexity when more training documents were given just like the unigram model, whereas the trigram model showed lower perplexity with more documents.

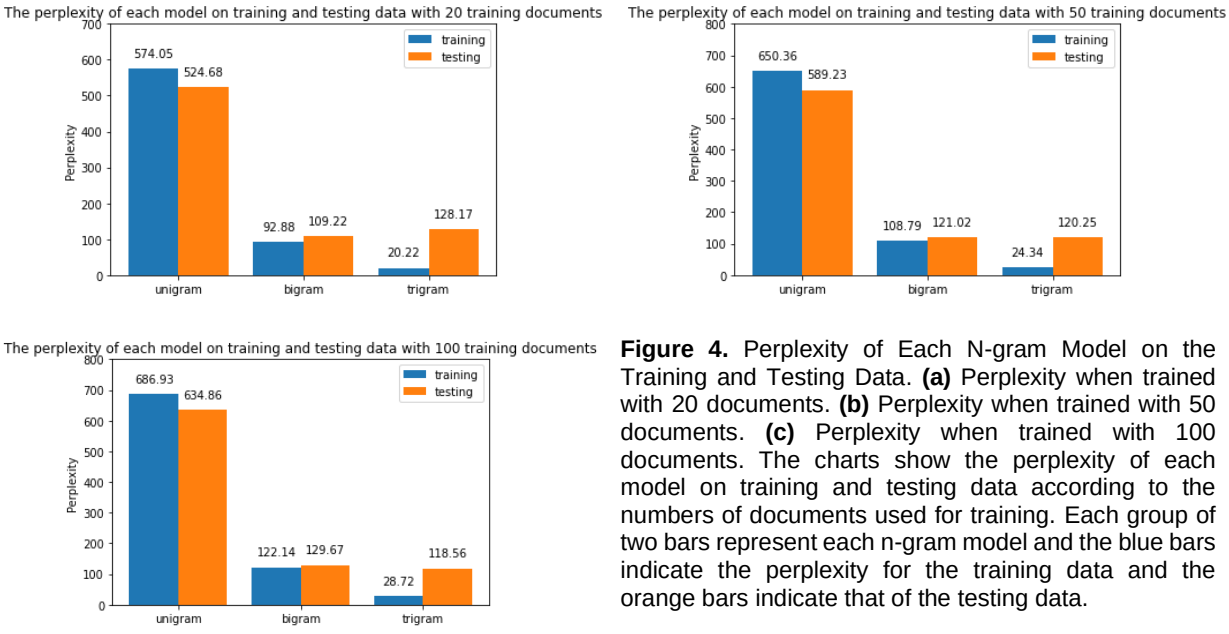

Figure 4. Perplexity of Each N-gram Model on the Training and Testing Data. **(a)** Perplexity when trained with 20 documents. **(b)** Perplexity when trained with 50 documents. **(c)** Perplexity when trained with 100 documents. The charts show the perplexity of each model on training and testing data according to the numbers of documents used for training. Each group of two bars represent each n-gram model and the blue bars indicate the perplexity for the training data and the orange bars indicate that of the testing data.

3.2. Performance on Sentence Completion

Table 1 presents the accuracy of the n-gram models with and without backoff on the sentence completion challenge. The trigram model showed about 30% of accuracy with and without backoff which was the highest among all the models. Furthermore, it brought higher accuracy as more training documents were given unlike the bigram and the unigram model which resulted in lower accuracy. This means that it has potential to perform as well as the distributional similarity method as presented in Figure 5 if more training data are given. The bigram model with 20 training documents showed the lowest accuracy of 20% even though it had lower perplexity than the unigram model.

	20	**50**	**100**
Trigram with backoff	26.923077	29.326923	30.192308
Trigram	26.538462	29.615385	30.192308
Bigram with backoff	20.000000	24.038462	22.500000
Bigram	20.096154	23.942308	22.500000
Unigram	25.288462	24.807692	24.615385

Table 1. The Accuracy of Each N-gram Model with Various Size of Training Documents. The columns indicate the number of documents given to the model for training and the rows indicate which n-gram model was used. The numeric values in the table indicate the accuracy of each model converted to percentage.

Figure 5 shows the accuracy of each method used in the distributional similarity method. The first method with which the model considered only one word from the left and right of the choice word, resulted in the lowest accuracy whereas the third method which considered all the words in the question sentence showed the highest accuracy. In other words, the performance improved as more words in a sentence were included in vectors.

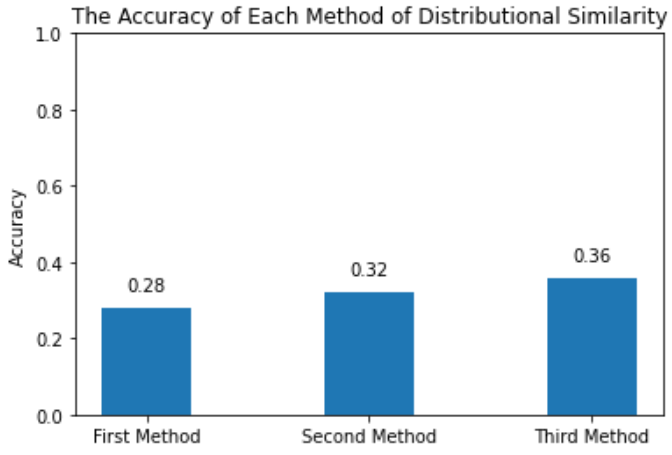

Figure 5. The Results of Each Method of Distributional Similarity. The first bar indicates the result of the first method which used only a single word from the left and right, the second bar indicates that of the second method which used two words, and the third bar indicates that of the third method which used all the words in a question.

We found that the distributional similarity model worked better than the n-gram models. We assume that this is because the Word2Vec embedding used in the model was pre-trained with way larger data, therefore, contained more vocabulary and their information. Since we saw that the performance of the trigram model improved with more training data, there is possibility

that it can work equally well as or even outperform the distributional similarity method if more training data are given. Future work can try exploiting all the documents in the Holmes Training Dataset, which we could not do due to limited resources. Furthermore, to improve the distributional similarity model, word embeddings using other methods such as GloVe, or BERT can be tried instead.

3.3. Errors

There was an error while running the code of the trigram model without backoff. When trying to get the discounted value of trigrams that had not been seen nor replaced with the 'UNK' tokens, the code raised the error since there was no value to look up. The models with backoff solved this simply by returning a lower-order n-gram, however, ones without backoff could not. We resolved this issue by just returning the default discount factor 0.3, as set in *3.1.4. Hyper-parameter Settings & Size of Training Documents*.

4. Conclusion

We solved the Microsoft Research Sentence Completion Challenge using n-gram language models and distributional similarity method with Word2Vec. The distributional similarity method which used a pre-trained Word2Vec embedding showed the best performance among all. However, the trigram model showed better performance as more training data were given, leaving potential that it could outperform the distributional similarity method with larger data. Furthermore, it brought lower perplexity with more training data – the perplexity of the bigram model was lower than that of the trigram with 50 training documents, however, the trigram model outperformed when 100 was given. This leaves future work to be done by making use of all the training documents in the Holmes Training Dataset, which was provided by the original challenge paper. Also, the distributional similarity method is worth exploring with other word embeddings such as GloVe and BERT.

5. Appendices

```
# Import libraries
import os,random,math,nltk,csv,re,string,gensim
from google.colab import drive
import pandas as pd, csv
from nltk import word_tokenize as tokenize
nltk.download('punkt')
from scipy import spatial
import numpy as np
import matplotlib.pyplot as plt

"""# Data Preparation"""

# Commented out IPython magic to ensure Python compatibility.
# Mount google drive
drive.mount('/content/drive')
# Assign the parent directory
parentdir = "/content/drive/My Drive/lab2resources/sentence-completion"
# Assign the training directory
```

```
trainingdir=os.path.join(parentdir,"Holmes_Training_Data")

# Function for splitting training and testing data
def get_training_testing(training_dir=trainingdir,split=0.5):
  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)
  return(filenames[:index],filenames[index:])

training,testing=get_training_testing(trainingdir)

# %config IPCompleter.greedy=True

import os
len(os.listdir(trainingdir))

# Read the question and answer files
questions=os.path.join(parentdir,"testing_data.csv")
answers=os.path.join(parentdir,"test_answer.csv")

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

# Visualize the answers
with open(answers) as instream:
    csvreader=csv.reader(instream)
    lines=list(csvreader)

# Store the answers as a list
answers_list=[item[1] for item in lines[1:]]
# Add to the question dataframe
qs_df['answer']=answers_list
qs_df.head()

"""# N-gram Language Model"""

class language_model():
```

```
# Train when initialised
def __init__(self,trainingdir=trainingdir,files=[],discount=0.75):
  self.discount=discount
  self.training_dir=trainingdir
  self.files=files
  self.train()

# Function for training
def train(self):

  # Dictionaries for each unigram, bigram, and trigram model
  self.unigram={}
  self.bigram={}
  self.trigram={}

  ''' Read the questions and create n-gram dictionaries, make 'unknown' tokens,
  apply absolute discounting and convert to probability. '''
  self._processfiles()
  self._make_unknowns()
  self._discount()
  self._convert_to_probs()

# Function for processing a line (question)
def _processline(self,line):

  # Tokenise the question and add 'start' and 'end' tokens at each end.
  tokens=["__START"]+tokenize(line)+["__END"]
  # Previous point for bigram and trigram (w_(n-1))
  previous="__END"
  # Second previous point for trigram (w_(n-2))
  previous2="__END"

  # Add counts for each word to the models
  for token in tokens:
    # Add the unigram count of the word
    self.unigram[token]=self.unigram.get(token,0)+1

    # Add the bigram count of the word given its previous word
    # {w1:(w2:n)}
    current=self.bigram.get(previous,{})
    current[token]=current.get(token,0)+1
    self.bigram[previous]=current

    # Add the trigram count of the word given its two previous words
    # An if statement that allows to start from the second word of the line.
```

```
      # {w1:(w2:(w3:n))}
      if previous != "__END":
        current=self.trigram.get(previous2,{})
        current2=current.get(previous,{})
        current2[token]=current2.get(token,0)+1
        self.trigram[previous2]=current
        self.trigram[previous2][previous]=current2

      # Move the window to the right
      previous2=previous
      previous=token

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

  # Function for converting the counts of words to probability distributions
  def _convert_to_probs(self):

    # Get the unigram probability -> count of the word/count of all words
    self.unigram={k:v/sum(self.unigram.values()) for (k,v) in self.unigram.items()}
    # Get the bigram probability -> count of the previous and current word seen together/count
of the previous word
    self.bigram={key:{k:v/sum(adict.values()) for (k,v) in adict.items()} for (key,adict) in
self.bigram.items()}
    self.trigram={key2:{key:{k:v/sum(adict.values()) for (k,v) in adict.items()} for
(key,adict) in dicts.items()} for (key2, dicts) in self.trigram.items()}

  # Function for getting the probability of words
  def get_prob(self,token,context="",method="unigram"):

    # Get unigram probability of the word, get one of the unknown token if not present.
    if method=="unigram":
      return self.unigram.get(token,self.unigram.get("__UNK",0))

    # Get probability of a word and its context word seen together
```

```
    elif method=="bigram":
      # Get the bigram probability of the word and context word, get unknown tokens if not
present.
      bigram=self.bigram.get(context[-1],self.bigram.get("__UNK",{}))
      big_p=bigram.get(token,bigram.get("__UNK",0))

      # Calculate the reserved probability mass according to the word's unigram probability.
      lmbda=bigram["__DISCOUNT"]
      uni_p=self.unigram.get(token,self.unigram.get("__UNK",0))

      # Calculate the final probability
      p=big_p+lmbda*uni_p
      return p

    # Get probability of a word and its past two words seen together
    elif method=='trigram':
      trigram=self.trigram.get(context[-2],self.trigram.get("__UNK",{}))
      trigram2=trigram.get(context[-1],trigram.get("__UNK",{}))
      tri_p=trigram2.get(token,trigram2.get("__UNK",0))
      # Calculate the reserved probability
      try:
        lmbda=trigram2["__DISCOUNT"]
      except:
        lmbda=self.discount
      uni_p=self.unigram.get(token,self.unigram.get("__UNK",0))
      p=tri_p+lmbda*uni_p
      return p
    return 'a'

  # Function for computing the probability of a line and the number of words in the line
  def compute_prob_line(self,line,method="unigram"):

    tokens=["__START"]+tokenize(line)+["__END"]
    acc=0

    # If trigram, starts from the third word of the line.
    if method == 'trigram':
      for i,token in enumerate(tokens[2:]):
        # Get the sum of trigram probabilities of the words in the line
        result=self.get_prob(token,tokens[i:i+2],method)
        # To prevent math domain error, add the value only if it's non-zero.
        if result != 0:
          acc+=math.log(result)
      return acc,len(tokens[2:])
    else:
```

```
      # If bigram, starts from the second word of the line.
      for i,token in enumerate(tokens[1:]):
        # Get the sum of bigram probabilities
        result=self.get_prob(token,tokens[i:i+1],method)
        if result != 0:
          acc+=math.log(result)
      return acc,len(tokens[1:])

  # Function for computing the probability and length of a corpus in a file
  def compute_probability(self,filenames=[],method="unigram"):
    if filenames==[]:
      filenames=self.files
    total_p=0
    total_N=0
    # Process the corpora in the files
    for i,afile in enumerate(filenames):
      print("Processing file {}:{}".format(i,afile))
      try:
        with open(os.path.join(self.training_dir,afile)) as instream:
          for line in instream:
            line=line.rstrip()
            if len(line)>0:
              p,N=self.compute_prob_line(line,method=method)
              total_p+=p
              total_N+=N
      except UnicodeDecodeError:
        print("UnicodeDecodeError processing file {}: ignoring rest of file".format(afile))
    return total_p,total_N

  # Function for calculating the perplexity of a model
  def compute_perplexity(self,filenames=[],method="unigram"):

    # Compute the proability and length of the corpus in the given file
    p,N=self.compute_probability(filenames=filenames,method=method)
    # Apply the formula
    pp=math.exp(-p/N)
    return pp

  # Function for replacing the words seen less than specific time with an unknown token.
  def _make_unknowns(self,known=2):

    # Create unknowns for unigram -> when the view count of a word is less than the given
parameter, replace it with unknown.
    # Increment the view count of unknown if already exists.
    for (k,v) in list(self.unigram.items()):
```

```
      if v<known:
        del self.unigram[k]
        self.unigram["__UNK"]=self.unigram.get("__UNK",0)+v

    # Create unknowns for bigram -> Look at each word in bigram and replace with unknown if
not present in the dict.
    isknown=0
    for (k,adict) in list(self.bigram.items()):
      for (kk,v) in list(adict.items()):
        isknown=self.unigram.get(kk,0)
        # If the second word of the bigram hasn't seen before, replace with unknown and delete
the original.
        if isknown==0:
          adict["__UNK"]=adict.get("__UNK",0)+v
          del adict[kk]
      isknown=self.unigram.get(k,0)
      # If the first word hasn't seen, replace it with unknown and assign the second word to
it.
      if isknown==0:
        del self.bigram[k]
        current=self.bigram.get("__UNK",{})
        current.update(adict)
        self.bigram["__UNK"]=current
      # If the word has seen, assign the second word to the word. (Do this in case the second
word was replaced to unknown)
      else:
        self.bigram[k]=adict

    # Create unknowns for trigram -> The same process as the bigram but also checks the third
word
    isknown=0
    for (k,dicts) in list(self.trigram.items()):
      for (kk,adict) in list(dicts.items()):
        for (kkk,v) in list(adict.items()):
          isknown=self.unigram.get(kkk,0)
          if isknown==0:
            adict["__UNK"]=adict.get("__UNK",0)+v
            del adict[kkk]

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

```
        else:
          self.trigram[k][kk]=adict

      isknown=self.unigram.get(k,0)
      if isknown==0:
        del self.trigram[k]
        current=self.trigram.get("__UNK",{})
        current.update(dicts)
        self.trigram["__UNK"]=current
      else:
        self.trigram[k]=dicts

  # Function for applying absolute discount
  # Discount amount set to 0.75
  def _discount(self,discount=0.75):
    self.discount=discount
    #discount each bigram and trigram count by a small fixed amount
    self.bigram={k:{kk:value-discount for (kk,value) in adict.items()} for (k,adict) in
self.bigram.items()}
    self.trigram={k:{kk:{kkk:value-discount for (kkk,value) in adict.items()} for (kk,adict)
in dicts.items()} for (k,dicts) in self.trigram.items()}

    # For each word, store the total amount of the discount so that the total is the same.
    # For bigram, just reserve the probability mass according to the first word.
    for k in self.bigram.keys():
        lamb=len(self.bigram[k])
        self.bigram[k]["__DISCOUNT"]=lamb*discount

    # For trigram, reserve the probability mass according to the second word.
    for k in self.trigram.keys():
      adict = self.trigram[k]
      for kk in adict:
        lamb=len(adict[kk])
        self.trigram[k][kk]["__DISCOUNT"]=lamb*discount

# Create filesets with the training and testing data
MAX_FILES=100
filesets={"training":training[:MAX_FILES],"testing":testing[:MAX_FILES]}

# Train the language model with the three methods
mylm=language_model(files=filesets["training"])
methods=["unigram","bigram",'trigram']

# Compute perplexity of the training and testing data using each model and store the results
for plotting
```

```
perplexities={}
for f,names in list(filesets.items()):
  perplexities[f]={}
  for m in methods:
    p=mylm.compute_perplexity(filenames=names,method=m)
    print("Perplexity on {} with {} method is {}".format(f,m,p))
    perplexities[f][m]=p

# The function for plotting perplexity on a bar chart
def autolabel(rects):
  for rect in rects:
    height=rect.get_height()
    # Decide the location where the value for each bar is presented
    location=height
    # If the value is negative, place the value right above the x-axis
    if height<0:
      location=0
    # Annotate the value for each bar
    ax.annotate('{:.2f}'.format(height),
                xy=(rect.get_x() + rect.get_width() / 2, location),
                xytext=(0, 8),  # 3 points vertical offset
                textcoords="offset points",
                ha='center', va='bottom')

# Plot the results
labels=['1 month', '6 months', '1 year']

x = np.arange(len(labels))  # the label locations
width=0.4  # the width of the bars

fig, ax = plt.subplots()
rects1=ax.bar(x - width/2, perplexities['training'].values(), width, label='training')
rects2=ax.bar(x + width/2, perplexities['testing'].values(), width, label='testing')
ax.set_ylabel('Perplexity')
ax.set_title('The perplexity of each model on training and testing data with 100 training
documents')
ax.set_xticks(x)
ax.set_xticklabels(methods)
ax.set_ylim((0,800))
ax.legend()

autolabel(rects1)
autolabel(rects2)

plt.show()
```

```
# Class for word similarity method using word embeddings
class word_embedding:

  # Load pre-trained word2vec
  def __init__(self):
    embedding_file='/content/drive/MyDrive/GoogleNews-vectors-negative300.bin'
    self.embeddings=gensim.models.KeyedVectors.load_word2vec_format(embedding_file,
binary=True)

  def word2vec(self,tokens):
    # Define the dimensionality (300 dims)
    dim=self.embeddings['word'].size

    # Make a list of embeddings of each word
    word_vec=[]
    for word in tokens:
      if word in self.embeddings:
        word_vec.append(self.embeddings[word])
      # If the word is not present in the embedding, assign a random vector
      else:
        word_vec.append(np.random.uniform(-0.25,0.25,dim))
    return word_vec

  # Function for calculating the similarity of the question vec and target vec.
  def total_similarity(self,vec,ques_vec):
    score = 0
    # Compare all the word vectors in the question to the target vector in terms of cosine
similarity
    # Increase of decrease score according to the result
    # Lower cosine distance -> More similar -> Higher score
    for v in ques_vec:
      score += (1 - spatial.distance.cosine(vec, v))
    return score

# Create an embedding object
emb = word_embedding()

# Class for reading and processing a question
class question:

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

  # Get the given field from the question
```

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

# Get the tokenized question with 'start' and 'end' tokens at each end.
def get_tokens(self):
  return ["__START"]+tokenize(self.fields[question.colnames["question"]])+["__END"]

# Add the answer field.
def add_answer(self,fields):
    self.answer=fields[1]

# Get window amount of left context words of the blank.
def get_left_context(self,window=1,target="_____"):
  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-window:found]
  else:
    return []

# Get window amount of the right context words of the blank.
def get_right_context(self,window=1,target="_____"):
  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 []

# Make a prediction for the question using a given method
def predict(self,method='word_embedding_total',model=mylm):
  if method=="bigram_backoff":
      return self.choose_backoff(model,methods=["unigram","bigram"],i=1)
  elif method=='trigram_backoff':
    return self.choose_backoff(model,methods=['unigram','bigram','trigram'],i=2)
  elif method=='word_embedding_total':
    return self.embedding_prediction(emb=emb,method='total')
```

```
    elif method=='word_embedding_partial':
      return self.embedding_prediction(emb=emb,method='partial')
    else:
      return self.choose(model,method=method)

  # Function for predicting using word2vec similarity
  def embedding_prediction(self,emb,method):
    choices=["a","b","c","d","e"]
    # Tokenize the question (not use the class method since it does not need start and end
tokens)

    # Create a list of vectors consisting of the question's word vectors
    scores =[]
    candidates = [self.get_field(ch+")") for ch in choices]
    cand_vec = emb.word2vec(candidates)

    # If the method is 'total' which compares the choice word vector to all the words in the
question
    if method=='total':
      tokens = tokenize(self.fields[question.colnames["question"]])
      ques_vec = emb.word2vec(tokens=tokens)
    # If the method is not total, only compare the choice word to the context words returned
with the given window
    else:
      tokens = self.get_left_context(window=1) + self.get_right_context(window=1)
      ques_vec = emb.word2vec(tokens=tokens)

    # Calculate total similarity of all the choice words
    for word in cand_vec:
      s = emb.total_similarity(word, ques_vec)
      scores.append(s)
    # Return one with the highest score
    idx = scores.index(max(scores))
    ans = choices[idx]
    return ans

  # Make predictions and get scores by comparing to the correct answers
  def predict_and_score(self,method="word_embedding_total",model=mylm):
    prediction=self.predict(method=method,model=model)
    if prediction ==self.answer:
      return 1
    else:
      return 0
```

```
  # Calculate probabilities of choice words using the given method and return one with the
highest probability.
  def choose(self,lm,method="trigram",choices=[]):
    if choices==[]:
      choices=["a","b","c","d","e"]

    # For unigram, simply get the probability of the choice word.
    if method=='unigram':
      probs=[lm.unigram.get(self.get_field(ch+")"),0) for ch in choices]

    # For bigram, get a context word from the left and right each and calculate probabilities
of the word given each context word.
    # Then multiply both probabilties
    elif 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+")"),method)*
             lm.get_prob(self.get_field(ch+")"),lc,method) for ch in choices]

    # For trigram, get two context words from the left and right and calculate trigram
probabilities.
    else:
      rc=self.get_right_context(window=2)
      lc=self.get_left_context(window=2)
      probs=[lm.get_prob(rc[1],[self.get_field(ch+")"),rc[0]],method)*
             lm.get_prob(self.get_field(ch+")"),lc,method) for ch in choices]

    # Return one with the highest probability
    maxprob=max(probs)
    bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]

    # If there are multiple options left, choose one randomly.
    if len(bestchoices)>1:
       print("Randomly choosing from {}".format(len(bestchoices)))
    return np.random.choice(bestchoices)


  # Instead of choosing randomly when there are multiple options, back off to a lower order n-
gram.
  # Recursive and controlled with parameter 'i'
  # Starts with the method of the last element of the methods parameter, and back off to the
second last one and so on.
  # Higher order methods should be positioned later.
```

```
  def
choose_backoff(self,lm,methods=['unigram','bigram','trigram'],choices=["a","b","c","d","e"],i=
2):

    # Base case: return an answer according to the unigram probability if it couln't be sorted
out with bigram model.
    if methods[i] == 'unigram':
      return self.choose(lm,'unigram',choices)

    # Get bigram probabilities
    elif methods[i] == '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+")"),'bigram')*
             lm.get_prob(self.get_field(ch+")"),lc,'bigram') for ch in choices]

    # Get trigram probabilities
    elif methods[i] == 'trigram':
      rc=self.get_right_context(window=2)
      lc=self.get_left_context(window=2)
      probs=[lm.get_prob(rc[1],[self.get_field(ch+")"),rc[0]],'trigram')*
            lm.get_prob(self.get_field(ch+")"),lc,'trigram') for ch in choices]

    # Get the best choices and if there are multiple, call this function recursively with the
lower order model.
    maxprob=max(probs)
    bestchoices=[ch for ch,prob in zip(choices,probs) if prob == maxprob]
    if len(bestchoices)>1:
      print("Backing off on {} and window {}".format(len(bestchoices),i))
      self.choose_backoff(lm,methods[:i],bestchoices,i-1)

    # There will be only one element in bestchoices
    return bestchoices[0]

# Class for processing the questions in files.
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="word_embedding_total"):
    return [q.predict(method=method) for q in self.questions]

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

SCC = scc_reader()

# Train with different numbers of training documents
numbers=[20, 50, 100]
lms = {}
for n in numbers:
  mylm=language_model(trainingdir=trainingdir,files=training[:n],discount=0.3)
  lms[n]=mylm

results={}
methods=['trigram_backoff','trigram','bigram','bigram_backoff','unigram']
for (n, model) in lms.items():
  results[n]={}
  for method in methods:
    acc=SCC.predict_and_score(method,model)
    results[n][method]=acc
```

```
results_pd=pd.DataFrame(results)

# Plot the perplexity of each model
labels = ['unigram','bigram','trigram']

x = np.arange(len(labels))  # the label locations
width=0.4  # the width of the bars

fig, ax = plt.subplots()
rects1=ax.bar(x - width/2, perplexities['training'].values(), width, label='training')
rects2=ax.bar(x + width/2, perplexities['testing'].values(), width, label='testing')
ax.set_ylabel('Perplexity')
ax.set_title('The perplexity of each model on training and testing data with 100 training
documents')
ax.set_xticks(x)
ax.set_xticklabels(methods)
ax.set_ylim((0,800))
ax.legend()

autolabel(rects1)
autolabel(rects2)

plt.show()

# Calculate accuracy of all the models on the challenge
tri_backoff_sc = SCC.predict_and_score('trigram_backoff')
trigram_sc = SCC.predict_and_score('trigram')
bigram_sc = SCC.predict_and_score('bigram')
bi_backoff_sc=SCC.predict_and_score('bigram_backoff')
unigram_sc=SCC.predict_and_score('unigram')
embedding_total_sc=SCC.predict_and_score('word_embedding_total')
embedding_partial_sc=SCC.predict_and_score('word_embedding_partial')

embedding_partial_sc_1=SCC.predict_and_score('word_embedding_partial') # compare only a single
word from the left and right
print(embedding_partial_sc_1)

embedding_partial_sc_2=SCC.predict_and_score('word_embedding_partial') # compare two words
each form the left and right
print(embedding_partial_sc_2)

embedding_total_sc=SCC.predict_and_score('word_embedding_total') # compare all the words
print(embedding_total_sc)
```

```
embedding_results=[embedding_partial_sc_1,embedding_partial_sc_2,embedding_total_sc]

# Plot the accuracy of the distributional similarity method
labels=['First Method', 'Second Method', 'Third Method']

x = np.arange(len(labels))  # the label locations
width=0.4  # the width of the bars

fig, ax = plt.subplots()
rects1=ax.bar(x, embedding_results, width)
ax.set_ylabel('Accuracy')
ax.set_title('The Accuracy of Each Method of Distributional Similarity')
ax.set_xticks(x)
ax.set_xticklabels(labels)
ax.set_ylim((0,1))

autolabel(rects1)

plt.show()
```