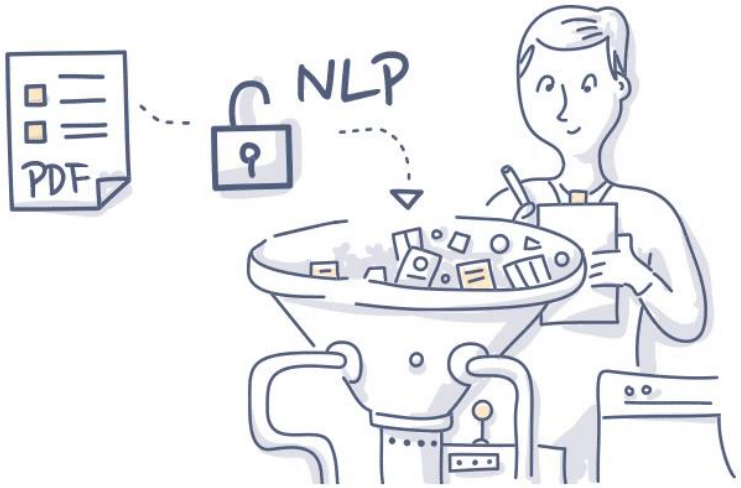

ADVANCED NATURAL LANGUAGE PROCESSING

- MEENAKSHI DATTU PATIL -

- MAY 2022 -

1 INTRODUCTION

This assignment uses the Microsoft Research Sentence Completion Challenge (MSRSCC) on which various models will be built and evaluated to estimate their performance. This paper will consist of a comparative analysis of various models and the ways in which they differ, and a performance evaluation will also be executed.

1.1 SENTENCE COMPLETION CHALLENGE

The challenge provided from the MSRSCC's authors (Geoffry Zweig and Christopher J.C. Burges), because there is a growing interest in semantic modelling for text, but there are very few publicly available large datasets using which results can be compared. The dataset for this assignment was constructed from 19th century novel data called "Project Gutenberg", it consists of 1,040 sentences seeded from five of the original Sherlock Holmes novels by Sir Arthur Conan Doyle - The Sign of the Four (1890), The Hound of the Baskervilles (1892), The Adventures of Sherlock Holmes (1892), The Memoirs of Sherlock Holmes (1894), and The Valley of Fear (1915). Each sentence has a low-frequency focus word, for which there will be four other alternative words that are generated using a maximum entropy n-gram model. The challenge is to select the correct "original" word, and not any of the other words. Most of the text is from the 19th novel, and the n-gram model is trained using 540 texts from the "Project Gutenberg" collection. What makes this dataset so unique and perfect to use is the fact that the quality of English is very high, there will be no copyright issues and since the novels are from a single author (Sir Arthur Conan Doyle), the writing style is very consistent. The four other words were decided by human judges, who hand-picked the words in such a way that the word will fit best without making the "original answer" less clear.

1.2 MOTIVATION

We can easily examine the theory underpinning the model's implementations and identify nuances in the dataset by evaluating their performance and implementing those models. The changes in the performance of the models can be documented and these observations can be used to maximise each model's score on the SCC. The effects of different hyper-parameter settings during the development process, as well as the effects of different features can be explored and hopefully an insight into the way different models process natural language can be gained.

2 METHODS

The MSR paper by Geoffry Zweig and Christopher J.C. Burges provides the details of six different approaches. Basically being 5 automated models and a human judge's method in which human judges were told to select the best fit from the available options.

Method	**% Correct** (N=1040)
Human	91
Generating Model	31
Smoothed 3-gram	36
Smoothed 4-gram	39
Simple 4-gram	34
Average LSA Similarity	49

The highest scorer was the human baseline with 90% of the answers being correct. The second highest scorer was the Average Latent Sematic Analysis (LSA) Similarity model by calculating the cosine of the angles between the vector forms of different words against the candidate words, providing 49% of the answers correctly. The other four models were variations of the n-gram model and provided between 31-39% of the answers correctly.

This paper will examine two different methods for sentence completion and test various features and parameter settings to see their effect on the accuracy. The first method is the n-gram model that make use of the unigram, bigram, trigram and quadgram(4-gram) statistics, while the other process makes use of the BERT model. These methods will show progression in the accuracy while applied on the test sentences and can also sometimes provide answers to each other's deficiencies. Additionally, this paper will also test the LSA model to test its performance as well.

2.1 N-GRAM LANGUAGE MODEL

N-gram is the simplest statistical model that assigns the probabilities to a sequence of words. The word n-gram can be defined as a sequence of N words which are set together. For example, if the sentence is, "The weather is lovely today". A unigram will break the sentence into single words, such as "The", "weather", "is", "lovely", and "today". Then there's the bigrams, trigrams and quadrigrams that split the sentence into two words, three words, and four words respectively. The N gram model uses chain rule of probabilities, Markov assumptions and the Maximum Likelihood of Estimations (MLE).

N-gram uses a clever way to estimate the probability of a word *w* given a history *h*, or the probability of an entire word sequence *W*. Here to represent the probability of a particular random variable X_i taking the value of "the", or $P(X_i = \text{"the"})$, we will use the simplification P(the). The sequence of N-words can be represented as $w_1, w_2 \ldots w_n$ or $w_{1:n}$. For the join probability, of each word in a sequence having value $P(X_1 = w_1, X_2 = w_2, X_3 = w_3,\ldots,X_n = w_n)$, we'll use $P(w_1, w_2,\ldots, w_n)$. We can then use the chain rule of probability to decompose this probability.

$$P(X_1 \ldots X_n) = P(X_1)P(X_2|X_1)P(X_3|X_{1:2}) \ldots P(X_n|X_{1:n-1}) = \prod_{k=1}^{n} P(X_k|X_{1:k-1})$$

And, after applying the chain rule to the words, we get:

$$P(w_{1:n}) = P(w_1)P(w_2|w_1)P(w_3|w_{1:2}) \ldots P(w_n|w_{1:n-1}) = \prod_{k=1}^{n} P(w_k|w_{1:k-1})$$

The main intuition behind the N-gram model is that instead of computing the probability of the word in its entire history of usage, we can approximate the history by just the last few words. Here, we come to the concept of Markov assumptions, that assumes that the class of a probabilistic model of a word depends on the previous word. Then the probabilities of the n-gram models is estimated using Maximum Likelihood Estimation (MLE). N-grams have performed well at speech recognition and machine translation tasks, and also in augmentative and alternative communication systems, which is very similar to this Sentence Completion. This is the reason why this model was determined to be suitable for this project.

2.2 BERT

BERT stands for Bidirectional Encoder Representation from Transformers, and it is the first unsupervised, deeply bidirectional system for pre-training natural language processing. The model is based on encoders that are made of multi-headed attention and a feed forward neural network. BERT pre-trains the deep bidirectional representations from unlabelled text by jointly conditioning on both left and right context in all layers. It is trained using a 16Gb text data file, on a Masked Language Modelling (MLM) task and a Next Sentence Prediction (NSP) task. The model can then be fine-tuned with one output layer to create models that can be used for a wide variety of tasks. BERT is conceptually simple while being empirically powerful.

Pre training the model is a onetime procedure that is an expensive task as it requires four days on 4 to 16 Cloud TPU's that were performed at Google. Fine tuning is relatively inexpensive as all the results in the paper can be replicated in a few hours on a GPU. Other important features of BERT include the ability of the model to a max length input of 512 tokens by padding and truncating the sequences to a max length of 512.

2.3 LSA

By understanding the context behind two words, we as humans can differentiate between them. But a machine will not be able to do so. Latent Sematic Analysis (LSA) tries to understand the concept around the words, to capture this hidden concept, that are called as topics. A word-vector is formed for each sentence by considering the sentence to be a document in sematic analysis. The similarity between two words is calculated as:

$$(q,w) = \frac{\sum_{i=1}^{N} w_{i,q} \times w_{i,j}}{\sqrt{\sum_{i=1}^{N} w_{i,q}^2} \times \sqrt{\sum_{i=1}^{N} w_{i,j}^2}}$$

The best part about LSA is that it can find hidden features in documents which is useful in extracting contextual usage meaning of words in documents.

LSI works in 3 steps that are mentioned below:

2.3.1 PRE-PROCESSING

This is the initial setup stage in which the input text is tokenized and stop words are removed from the document. A matrix is created where the rows show the unique terms, and columns denote the documents. The document matrix is modified using Term Frequency – Inverse Document Frequency (TF-IDF) method for providing weights to rare and frequent terms in the document.

$$DocumentTermWeight = f_{t,d} \times ln\left(\frac{N}{n_t}\right)$$

Where:

$f_{t,d}$: Count of term t in document d

N: The total count of documents

n_t : The count of documents having term t

2.3.2 SINGULAR VALUE DECOMPOSITION:

SVD is an important concept in linear algebra in which generalization of the eigen decomposition of a square normal matrix with an orthonormal eigen basis is performed. Here, SVD is used by LSA to generate vectors for a particular text. The term document matrix is used to calculate two matrices from which a left singular matrix and a right singular matrix.

$$X = LSR_T$$

Where:

L : Term - Concept weight matrix

R_T : Concept - Document weight matrix

S : Diagonal matrix representing concept weights

The matrix X is then approximated as, $X_K = L_K S_K R_T$.

2.3.3 TESTING

After the weights are generated in the matrix using TF-IDF, a query matrix is generated. The matrix is then multiplied with L_K and S_K to generate new query vectors. From the history *h*, the probability of a word *w* is given by,

$$P_{LSA}(w|h) = \frac{(h,w) - m}{\sum_{q \in V}((h,q) - m)}$$

3 IMPLEMENTATION

This section will explain the implementation of the Microsoft Research Sentence Completion Challenge (MSRCC). Unigram and bigram models will only be documented as baseline models and no further work will be performed.

3.1 BASELINE

Two simple n-gram models were constructed to prepare the baseline for the project. A unigram model, which doesn't check for context and return only the common word. A bigram model, that checks the word to the left for context and generates the probability for each target word. From the dataset provided, which contains 522 of 540 sentences, the word *"the"* will have more than just two follow-up words.

Surprisingly, the unigram had an accuracy of 24.71%, which is better than a randomly generated baseline. This is impressive considering the fact that unigrams don't use context. n=0 is considered as a random system with 20% accuracy to start with as the unigram model (n=1) has an accuracy of 24%. The bigram model, on the other hand was slightly better, with an accuracy of 26.45%. Using a single word on the left for context provided an accuracy improvement.

3.2 N-GRAM MODEL

The baseline is set using the n-gram model, so it would make sense to try and generate more accurate solutions, in the form of trigram and quadrigram. In terms of building the model, they are not very different from the bigram model. For trigrams, nested dictionaries are created, in such a way that trigrams give another dictionary to search for a second context word. Using this dictionary, the target word's

probability is determined. The quadrigram model allows a 3rd context word to be added over the trigram model. This further narrows down the available options for the target word and should hence increase the accuracy of the model. But again, there is also the chance that some combinations of the target words and context phrases will not appear, because of which the accuracy will decrease.

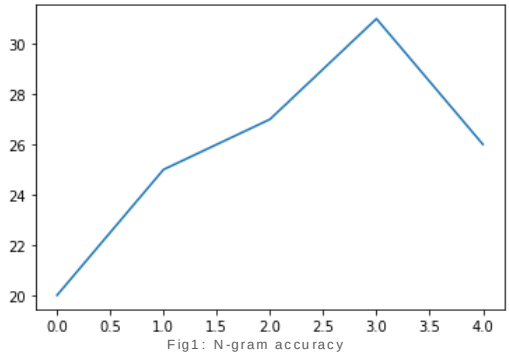

Fig1: N-gram accuracy

The accuracy of the n-grams was recorded and trends for the change of accuracy were noted down as *n* increased. The accuracy of the system is at the highest with the trigram model (where the peak is), with an accuracy of 31.58%, and then again decreasing , because of the reason mentioned above. From this, it is clear that a balance has to be maintained to get meaningful results, and not being too restrictive, as to lose combinations.

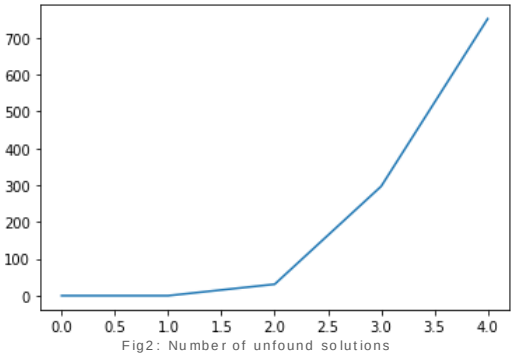

Fig2: Number of unfound solutions

The above graph shows the number of unfound solutions, possible reasons include the lack of context phrases or target words in the memory. In the bigram model, this issue occurred 31 times, mostly because the context phrase contained proper nouns or obscure words. It can clearly be observed that the error increases exponentially with the increase in *n*, and the last model, the quadrigram was unable to find the solution 752 times. To check the best fit model for the n-gram system for maximum *n*, the quadrigram model was rerun to find the accuracy with only the sentences it was able to find the solution for. Hence, 288 problems were fed into the model, and an accuracy of 44.65% was achieved with this setup, showing that there is a wide bottleneck in the system. N-gram models can be stacked to get rid of the downside of using large *n* models. The accuracy and number of unfound solutions are higher in smaller n-grams.

3.3 BERT

The BERT model is a pre trained language model that was developed by Google making it very easy to import, use and run quickly. The program imports BERT and passes the questions through its pre-trained masked language model. BERT cannot always find the correct answer from the provided five options and hence, an additional system needs to be implemented.

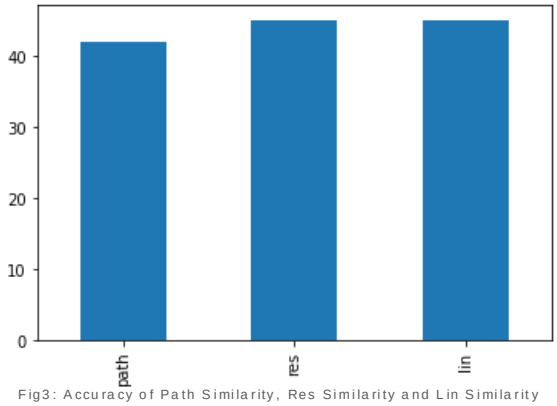

Fig3: Accuracy of Path Similarity, Res Similarity and Lin Similarity

From the graph above, it is evident that there is nearly 3% difference in the accuracy. Path similarity gives an accuracy of 42.69%, and lin similarity of 45.28%. From this implementation of BERT, it is clear that more investigation into the techniques for comparing predictions and options will give an overall better accuracy. One such technique is paraphrase identifier, in which a simple one or two layer neural network that can be used to check if two sentences are paraphrased by comparing them. This will provide a more accurate and efficient result than just checking the semantic similarity.

Fine tuning is also a method by which the accuracy can be improved. This can be performed by using an additional layer of transformers to optimise the model for a specific task. For this project, the code was taken from the public GitHub repository with some slight alterations to fit our training data. There was no significant change to the accuracy after fine tuning, mostly because BERT is pre trained as a masked language, and the fact that there are power and batch size limitations.

3.4 LSA

In the LSA model, bag of words is first implemented to the corpus followed by finding the Term Frequency – Inverse Document Frequency (TF-IDF) weights for the word vectors, after which Latent Sematic Indexing (LSI) is performed on the TFIDF vectors for 100 features. LSI helps in overcoming the issue of synonymy by increasing the recall. It is also independent of language, because of which it can perform cross-linguistic concept sharing and also example based categorization. After this the cosine similarity between the candidate response and question statement is calculated by normalizing the vectors and taking their dot product, with values ranging from 0.0 to 0.1. In the end, the average cosine similarity is found for all features and the option with the highest cosine similarity is returned. The LSA model is more sophisticated when it comes to semantic analysis. This model is an improvement from the baseline n-gram model, with an accuracy reading of 31.73%.

The LSA might find it difficult to interpret certain dimensions, that might not have any meaning in the natural language but produces results that can be justified on a mathematical level. This model is also not very good at capturing words of the same meaning, because each new occurrence is always treated as the word with the same meaning, as LSA treats the word as being represented like a single point in space. Since this model uses bag of words, there are certain inherited limitations, such as the unordered assortment of words.

4 FURTHER WORKS

The models that were used can be fine-tuned and executed to get better accuracy scores. The achieved scores from all the models are acceptable and have performed better than the baseline model. By striking a balance between dealing with the unknown words, and the various *n* gram models can be tuned to perform optimally. This will definitely improve the accuracy towards the challenge questions. Some additional work can be done on word embedding models that are tuned to the training set. The lack of constraints while dealing with less frequent words will mostly provide better performance of the model.

BERT has some limitations, such as the inability to handle long text sequences as it tends to ignore texts after 512 tokens, and hence, further experimentation on models such as XLNet, which is a BERT like model, with some modifications can help get better results. By expanding the term vocabulary from unigrams to bigrams or trigrams of words, we can improve the expressive power of the LSA model because more information is added in about word ordering.

5 CONCLUSIONS

The MSRCC is a difficult and insightful challenge for natural language processing and this paper investigates three of various methods to answer the sentence-completion questions. The words have to be picked using probabilistic models for which language models that go beyond generative probabilities and local contexts must be developed. One of the models used in this paper, *transformers*, has a large pre-training corpora and the ability to process varying sequences, outperformed other models. This challenge has made the concepts and methods of natural language processing clearer, and by executing various models and clearing errors, the methods used have been understood much clearer.

6 REFERENCES

[1] Geoffrey Zweig and Christopher J.C. Burges, "*The Microsoft Research Sentence Completion Challenge*" Technical Report MSR-TR-2011-129, Microsoft. (2011)

[2] G. Zweig, J. Platt, C. Meek, C. Burges, A. Yessenalina, and Q. Liu, "*Computational Approaches To Sentence Completion*" 50th Annual Meeting of the Association for Computational Linguistics, ACL 2012 - Proceedings of the Conference. (2012)

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova, "*BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*". (2019)

[4] Chirag Goyal, Analytics Vidhya, "*Step by Step Guide to Master NLP – Topic Modelling using LSA*", https://www.analyticsvidhya.com/blog/2021/06/part-16-step-by-step-guide-to-master-nlp-topic-modelling-using-lsa/. (2021)

[5] Daniel Jurafsky & James H. Martin, "*Speech and Language Processing*". (2021)

[6] Edward Ma, "*Why does XLNet outperform BERT?*", https://medium.com/dataseries/why-does-xlnet-outperform-bert-da98a8503d5b. (2019)

[7] Jamie Bali – Github. https://github.com/JamieBali/MRSCC. (2022)

[8] Aparna – Github. https://github.com/Aparna2610/text-prediction. (2019)

APPENDIX:

```
# MICROSOFT RESEARCH SENTENCE COMPLETION CHALLENGE

# IMPORTS
get_ipython().system('pip install Transformers')
import csv
import sys
import time
import torch
import operator
import pandas as pd
from os import listdir
import os, random, nltk
start_time = time.time()
nltk.download("wordnet")
from tqdm.auto import tqdm
nltk.download("wordnet_ic")
nltk.download("lin_thesaurus")
from os.path import join, isfile
brown_ic = wn_ic.ic("ic-brown.dat")
from nltk.tokenize import word_tokenize
from transformers import BertTokenizer, BertModel, BertForMaskedLM, AdamW
from nltk.corpus import wordnet as wn, wordnet_ic as wn_ic, lin_thesaurus as lin
get_ipython().system('pip install gensim')
from gensim import corpora, models
from scipy import spatial

# N-GRAM MODEL
unigram = {}
bigram = {}
trigram = {}
quadrigram = {}

TRAINING_DIR = "path/Holmes_Training_Data"
filenames=os.listdir(TRAINING_DIR)
print(len(filenames))
index_of_found = []
for d in range(0, len(filenames)):
  try:
    doc = filenames[d]
    print(d)
    full_doc = ""
    with open(os.path.join(TRAINING_DIR, doc)) as instream:
      for line in instream:
        full_doc += line[:-1] + " "   #By removing the last 2 characters, we remove the \n
characters from the end of every line.
    full_doc = full_doc.split(" ")
    tokens = ["__END","__START"]
```

```
    #Here we tokenise the entire document so it can be easily converted into an n-gram.
    #This is not the most efficient system, but I don't know how to do regex in python.
    for token in full_doc:
      word = token
      #This ignores blank tokens as well as tokens that contain just speech marks
      if len(word) > 0 and not word == "\"":
        if word[-1] == "\"": #Removing punctuation individually
          word = word[:-1]
        if word[0] == "\"":
          word = word[1:]
        if word[-1] == "." or word[-1] == "!" or word[-1] == "?": #sentence enders need tokens
added to end and start new sentences
          tokens.append(word[:-1])
          tokens.append(word[-1])
          tokens.append("__END")
          tokens.append("__START")
        elif word[-1] == "," or word[-1] == ";":
          tokens.append(word[:-1])
          tokens.append(word[-1])
        elif not word == "":
          tokens.append(word)

    #Here we construct the n-grams, creating sums of occurrences for all documents
    for i in range(len(tokens)):
      unigram[tokens[i]] = unigram.get(tokens[i], 0) + 1
      if i < len(tokens) - 1:
        if not tokens[i] in bigram:
          bigram[tokens[i]] = {}
        bigram[tokens[i]][tokens[i+1]] = bigram[tokens[i]].get(tokens[i+1], 0) + 1
      if i < len(tokens) - 2:
        if not tokens[i] in trigram:
          trigram[tokens[i]] = {}
        if not tokens[i+1] in trigram[tokens[i]]:
          trigram[tokens[i]][tokens[i+1]] = {}
        trigram[tokens[i]][tokens[i+1]][tokens[i+2]] =
trigram[tokens[i]][tokens[i+1]].get(tokens[i+2], 0) + 1
      if i < len(tokens) - 3:
        if not tokens[i] in quadrigram:
          quadrigram[tokens[i]] = {}
        if not tokens[i+1] in quadrigram[tokens[i]]:
          quadrigram[tokens[i]][tokens[i+1]] = {}
        if not tokens[i+2] in quadrigram[tokens[i]][tokens[i+1]]:
          quadrigram[tokens[i]][tokens[i+1]][tokens[i+2]] = {}
        quadrigram[tokens[i]][tokens[i+1]][tokens[i+2]][tokens[i+3]] =


quadrigram[tokens[i]][tokens[i+1]][tokens[i+2]].get(tokens[i+3], 0) + 1
  except:
    print("error in file " + filenames[d])
sum = 0
for x in unigram:
```

```
  sum += unigram[x]
for x in unigram:
  unigram[x] /= sum

for x in bigram:
  sum = 0
  for y in bigram[x]:
    sum += bigram[x][y]
  for y in bigram[x]:
    bigram[x][y] = bigram[x][y] / sum

for x in trigram:
  for y in trigram[x]:
    sum = 0
    for z in trigram[x][y]:
      sum += trigram[x][y][z]
    for z in trigram[x][y]:
      trigram[x][y][z] /= sum

for x in quadrigram:
  for y in quadrigram[x]:
    for z in quadrigram[x][y]:
      sum = 0
      for t in quadrigram[x][y][z]:
        sum += quadrigram[x][y][z][t]
      for t in quadrigram[x][y][z]:
        quadrigram[x][y][z][t] /= sum

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

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

qs_df=pd.DataFrame(lines[1:],columns=lines[0])
qs_df.head()

def get_left_context(sentence):
  sent = ["_END", "_START"] + sentence.split(" ")
  last = "_START"
  twolast = "_END"
  for i in range(2, len(sent)):
    if sent[i] == "_____":
      return sent[i-3], sent[i-2], sent[i-1]
results = []
for num in range(0, len(qs_df["question"])):
  sentence = qs_df["question"][num]
  options = []
  options.append(qs_df["a)"][num])
  options.append(qs_df["b)"][num])
```

```
  options.append(qs_df["c)"][num])
  options.append(qs_df["d)"][num])
  options.append(qs_df["e)"][num])
  guesses = []
  c1, c2, c3 = get_left_context(sentence)
  for option in options:
    try:
      guesses.append(quadrigram[c1][c2][c3][option])
    except:
      guesses.append(0)
  greater = False
  highest = 0
  res = ""
  if guesses[0] > highest:
    greater = True
    res = "a"
    highest = guesses[0]
  if guesses[1] > highest:
    greater = True
    res = "b"
    highest = guesses[1]
  if guesses[2] > highest:
    greater = True
    res = "c"
    highest = guesses[2]
  if guesses[3] > highest:
    greater = True
    res = "d"
    highest = guesses[3]
  if guesses[4] > highest:
    greater = True
    res = "e"
    highest = guesses[4]
  if greater == True:
    results.append(res)
    index_of_found.append(num)
  else:
    print("UNFOUND")
    results.append(random.choice(["a","b","c","d","e"]))
with open(answers) as instream:
  csvreader=csv.reader(instream)
  lines=list(csvreader)
as_df=pd.DataFrame(lines[1:],columns=lines[0])
as_df.head()
answer_list = []
for x in as_df["answer"]:
  answer_list.append(x)
sum = 0
for x in index_of_found:
  if answer_list[x] == results[x]:
    sum += 1
```

```
print(sum / len(index_of_found))
df = pd.DataFrame([20,25,27,31,26], index=[0, 1, 2, 3, 4])
df.plot(legend=False)
df = pd.DataFrame([0, 0, 31, 297, 752])
df.plot(legend=False)
```

BERT

```
#Loading pre-trained models
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertForMaskedLM.from_pretrained('bert-base-uncased')
def make_segment_ids(list_of_tokens):
#This function assumes that up to and including the first '[SEP]' is the first segment, anything
afterwards is the second segment
  current_id=0
  segment_ids=[]
  for token in list_of_tokens:
    segment_ids.append(current_id)
  return torch.tensor([segment_ids])

def tokeniseText(question):
  masked_index = 0
  tokenized = ["[CLS]"] + tokenizer.tokenize(question)
  temp = []
  found = False
  for i, word in enumerate(tokenized):
    if word == "_" and found == False:
      if tokenized[i+4] == "_":
        masked_index = i
        temp.append("[MASK]")
        found = True
    #This fixes the issue of underscores appearing later in the word
    elif word == "_":
      pass
    else:
      temp.append(word)
  temp.append("[SEP]")

  segment_ids=make_segment_ids(temp)
  return torch.tensor([tokenizer.convert_tokens_to_ids(temp)]), masked_index, segment_ids
def predict(input_sentence):
  tokens_tensor, masked_index, segment_ids = tokeniseText(input_sentence)

  with torch.no_grad():
    outputs = model(tokens_tensor, token_type_ids=segment_ids)
    predictions = outputs[0]

#Find the token id which maximises the prediction for the masked token and then convert this back
to a word
  predicted_index = torch.argmax(predictions[0, masked_index]).item()
  predicted_token = tokenizer.convert_ids_to_tokens([predicted_index])[0]
```

```
  return predicted_token

def getMostLikelyFromPrediction(prediction, choices):
  best_word_path = ""
  best_sim_path = 0
  best_word_res = ""
  best_sim_res = 0
  best_word_lin = ""
  best_sim_lin = 0
  for x in choices:
    highest_sim_path = 0
    highest_sim_res = 0
    highest_sim_lin = 0
    for a in wn.synsets(prediction):
      for b in wn.synsets(x):
        temp_path = None
        temp_res = None
        temp_lin = None
        try:
          temp_path = wn.path_similarity(a, b)
          temp_res = wn.res_similarity(a, b, brown_ic)
          temp_lin = wn.lin_similarity(a, b, brown_ic)
        except:
          pass
        if not temp_path == None:
          if temp_path > highest_sim_path:
            highest_sim_path = temp_path

        if not temp_res == None:
          if temp_res > highest_sim_res:
            highest_sim_res = temp_res

        if not temp_lin == None:
          if temp_lin > highest_sim_lin:
            highest_sim_lin = temp_lin
    if highest_sim_path > best_sim_path:
      best_word_path = x
      best_sim_path = highest_sim_path

    if highest_sim_res > best_sim_res:
      best_word_res = x
      best_sim_res = highest_sim_res

    if highest_sim_lin > best_sim_lin:
      best_word_lin = x
      best_sim_lin = highest_sim_lin
  if best_word_path == "":
    n = random.randint(0,4)
  else:
    n = choices.index(best_word_path)
  if best_word_res == "":
```

```
    o = random.randint(0,4)
  else:
    o = choices.index(best_word_res)
  if best_word_lin == "":
    p = random.randint(0,4)
  else:
    p = choices.index(best_word_lin)
  letters = ["a","b","c","d","e"]
  return letters[n], letters[o], letters[p]

questions=os.path.join("path/testing_data.csv")
answers=os.path.join("path/test_answer.csv")
with open(questions) as instream:
  csvreader=csv.reader(instream)
  lines=list(csvreader)
qs_df=pd.DataFrame(lines[1:],columns=lines[0])
qs_df.head()
results_path = []
results_res = []
results_lin = []
for num in range(0, len(qs_df)):
  sentence = qs_df["question"][num]
  prediction = predict(sentence)
  print(prediction)
  letters = ["a)","b)","c)","d)","e)"]
  choices = []
  for x in letters:
    choices.append(qs_df[x][num])
  path, res, lin = getMostLikelyFromPrediction(prediction, choices)
  results_path.append(path)
  results_res.append(res)
  results_lin.append(lin)
with open(answers) as instream:
  csvreader=csv.reader(instream)
  lines=list(csvreader)
as_df=pd.DataFrame(lines[1:],columns=lines[0])
as_df.head()
answer_list = []
for x in as_df["answer"]:
  answer_list.append(x)
p_acc = 0
r_acc = 0
l_acc = 0
for i in range(0, len(qs_df)):
  if answer_list[i] == results_path[i]:
    p_acc += 1
  if answer_list[i] == results_res[i]:
    r_acc += 1
  if answer_list[i] == results_lin[i]:
    l_acc += 1
print("path accuracy = " + str(p_acc/len(answer_list)))
```

```
print("res accuracy  = " + str(r_acc/len(answer_list)))
print("lin accuracy  = " + str(l_acc/len(answer_list)))
df = pd.DataFrame([42,44,45], index=["path","res","lin"])
df.plot.bar(legend=False)
```

LSA

```
#Loading dataset for LSA Model
books = []
mypath = "path/Holmes_Training_Data"
onlyfiles = [f for f in listdir(mypath) if isfile(join(mypath, f))]
headerSeparator = "*END*THE SMALL PRINT FOR PUBLIC DOMAIN ETEXTS*Ver.04.29.93*END*"
noOfFiles = 522
for fname in onlyfiles:
    fp = open(mypath + "/" + fname,"r",encoding='cp1252')
    bookContent = ""
    headerSeparatorDetected = False
    for line in fp.readlines():
        if not headerSeparatorDetected:
            if line.strip() == headerSeparator:
                headerSeparatorDetected = True
            continue
        line = line.strip()
        if not (len(line) == 0):
            bookContent = bookContent + " " + line.lower()
    books.append(bookContent)
    fp.close()
    noOfFiles = noOfFiles - 1
    if noOfFiles == 0:
        break
    print("No of sentences read",len(books))
stoplist = set(['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', 'your', '
       yours', 'yourself', 'yourselves', 'he', 'him', 'his', 'himself', 'she', 'her', 'hers',
       'herself', 'it', 'its', 'itself', 'they', 'them', 'their', 'theirs', 'themselves',
       'what', 'which', 'who', 'whom', 'this', 'that', 'these', 'those', 'am', 'is', 'are',
       'was', 'were', 'be', 'been', 'being', 'have', 'has', 'had', 'having', 'do', 'does',
       'did', 'doing', 'a', 'an', 'the', 'and', 'but', 'if', 'or', 'because', 'as', 'until',
       'while', 'of', 'at', 'by', 'for', 'with', 'about', 'against', 'between', 'into',
       'through', 'during', 'before', 'after', 'above', 'below', 'to', 'from', 'up', 'down',
       'in', 'out', 'on', 'off', 'over', 'under', 'again', 'further', 'then', 'once', 'here',
       'there', 'when', 'where', 'why', 'how', 'all', 'any', 'both', 'each', 'few', 'more',
       'most', 'other', 'some', 'such', 'no', 'nor', 'not', 'only', 'own', 'same', 'so',
       'than', 'too', 'very', 's', 't', 'can', 'will', 'just', 'don', 'should', 'now'])
texts = [[word for word in book.split() if word not in stoplist] for book in books]

#Create a dictionary of the text corpus
dictionary = corpora.Dictionary(texts)
dictionary.save('deerwester.dict')

# Bag of Words
corpus = [dictionary.doc2bow(text) for text in texts]
```

```
corpora.MmCorpus.serialize('deerwester.mm', corpus)
corpus = corpora.MmCorpus('deerwester.mm')
dictionary = corpora.Dictionary.load('deerwester.dict')

#Create TFIDF matrix
tfidf = models.TfidfModel(corpus)
corpus_tfidf = tfidf[corpus]

#Create LSI model of the TFIDF matrix
lsi = models.LsiModel(corpus_tfidf, id2word=dictionary, num_topics=20)
corpus_lsi = lsi[corpus_tfidf]
lsi.save('model.lsi')
lsi = models.LsiModel.load('model.lsi')
testing_data_FN = "path/testing_data.csv"
testing_data_answer_FN = "path/test_answer.csv"

class QuestionProcessor:
    def __init__(self):
        self.questionSet = {}
        self.loadQuestion()

    def loadQuestion(self):
        with open(testing_data_FN,'r') as csvfile:
            qReader = csv.reader(csvfile,delimiter=',',quotechar='"')
            isHeader = True
            for row in qReader:
                if isHeader:
                    isHeader = False
                    continue
                q = Question()
                q.qNo = int(row[0])
                q.question = row[1]
                q.options["a"] = row[2]
                q.options["b"] = row[3]
                q.options["c"] = row[4]
                q.options["d"] = row[5]
                q.options["e"] = row[6]
                self.questionSet[q.qNo] = q
        with open(testing_data_answer_FN,'r') as csvfile:
            aReader = csv.reader(csvfile,delimiter=',',quotechar='"')
            isHeader = True
            for row in aReader:
                if isHeader:
                    isHeader = False
                    continue
                qNo = int(row[0])
                answer = row[1]
                self.questionSet[qNo].answer = answer

class Question:
    def __init__(self):
```

```
        self.qNo = None
        self.question = ""
        self.options = {"a":None,"b":None,"c":None,"d":None,"e":None}
        self.answer = ""
    def __repr__(self):
        return self.question
qp = QuestionProcessor()

def getPositionOfBlank(question):
    qToken = word_tokenize(question.lower())
    index = 0
    for token in qToken:
        if token == "_____":
            return index
            break
        index += 1
    return -1

def getAnswer(q):
    blankPos = getPositionOfBlank(q.question)
    stoplist.add("_____")
    qsTokens = [word for word in q.question.split() if word not in stoplist]
    qs_vectors = []
    for tokens in qsTokens:
        qs_bow = dictionary.doc2bow(tokens.lower().split())
        qs_tfidf = tfidf[qs_bow]
        qs_lsi = lsi[qs_tfidf]
        qs_vectors.append(qs_lsi)
    optToInd = {'a':0, 'b':1, 'c':2, 'd':3, 'e':4}
    IndToOpt = {0:'a', 1:'b', 2:'c', 3:'d', 4:'e'}
    options_vectors = [[] for i in range(5)]

    for key,option in q.options.items():
        option_bow = dictionary.doc2bow(option.lower().split())
        option_tfidf = tfidf[option_bow]
        option_lsi = lsi[option_tfidf]
        options_vectors[optToInd[key]] =  option_lsi
    qs_features = []
    option_features = []
    qs_features = [[tup[1] for tup in qsVec] for qsVec in qs_vectors if len(qsVec) > 0]
    option_features = [[tup[1] for tup in opVec] for opVec in options_vectors if len(opVec) > 0]

    resultForQues = {}
    for i in range(len(option_features)):
        resultForOpt = 0
        for j in range(len(qs_features)):
            resultForOpt += 1 - spatial.distance.cosine(qs_features[j], option_features[i])
        resultForQues[IndToOpt[i]] = resultForOpt
    ans = max(resultForQues.items(), key=operator.itemgetter(1))[0]
    return ans
```

```
def runTest():
    correct = 0
    total = 0
    for qNo in range(1,1041):
        q = qp.questionSet[qNo]
        ans = getAnswer(q)
        if ans:
            total += 1
            if q.answer == ans:
                correct += 1
            print(qNo, ans, q.answer, correct,correct/total)
    print("Achieved Accuracy with LSA",correct*100/total)
if __name__ == '__main__':
    if 'runTest' in sys.argv:
        runTest()
    else:
        print("Usage: python3 model.py runTest")
runTest()
```