For this homework, make sure that you format your notbook nicely and cite all sources in the appropriate sections. Programmatically generate or embed any figures or graphs that you need.

Names: **Shaan Hossain, Jinesh Shailesh Mehta**

Step 1: Train your own word embeddings

(describe the provided dataset that you have chosen here)

From the given two choices, we have Spooky Authors Dataset.

Describe what data set you have chosen to compare and contrast with the your chosen provided dataset. Make sure to describe where it comes from and it's general properties.

NOTE: We'll refer to the Spooky Authors Dataset as the Spooky Dataset.

Description of our dataset which we'll refer to as the Covid Dataset.

- Our selected data: Coronavirus tweets NLP
- Description : The tweets have been pulled from Twitter and manual tagging has been done then. The names and usernames have been given codes to avoid any privacy concerns.
- Columns:
  - UserName
  - ScreenName
  - Location
  - TweetAt
  - OriginalTweet
  - Sentiment

Similarities with chosen provided dataset:

- There are long tweets in the Covid dataset that resemble the longer more coherent sentences in the Spooky dataset.
- Both datasets are in English and roughly follow the English rules of grammar.
- Every entry is associated with an author in both datasets whether it's a person tweeting or an author. You can also identify each entry with a unique ID.

Constrasts with chosen provided dataset:

- There are a variety of contrasts between the datasets, but one of the most striking differences is the increased vocabulary size in the covid dataset versus the spooky

datasets by a factor of 3 at 100% of each dataset. One reason this might happen is due to the medium of the Covid dataset of Twitter. There are quite a few random strings of jibberish that often occur only once in the Tweets when someone randomly smashes their keyboard.

- We noticed that the Covid dataset has a lot of short sentences. We're not sure how this impacts training, but it's important to note. If you have larger more complex sentences, it's possible that there is more context for the ngrams versus the short tweets in the Covid dataset.
- The sentences in the Spooky dataset are much more coherent and tend to have better grammar than the Covid dataset. With respect to this task, having formalized grammar, proper spelling, and more consistency could definitely impact the model during training.

In [ ]:

```
# import your libraries here
from typing import List, Dict
# libs used for preprocessing
from gensim.parsing.preprocessing import stem_text, remove_stopwords, strip_p
from nltk.tokenize import RegexpTokenizer
from nltk import ngrams
# libs used for file reading and parsing
from csv import reader
# libs used for plotting
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt
import matplotlib.cm as cm
%matplotlib inline
# libs used for generating word2vec models
from gensim.models import Word2Vec
# to compute the training using multiple threads
from multiprocessing import cpu_count
# other utilities
from pathlib import Path
from collections import Counter
from os import path
import random
# libs used for validating the word vectors and loading word vectors
from gensim.models import KeyedVectors
import numpy as np
```

In [ ]:

```
# Determines which dataset to create embeddings and
# train the neural network on
dataset_to_use = 'Spooky' # either 'Spooky' or 'Covid'
dataset_percentage = 100 # percentage range 1 to 100
```

In [ ]:
```
# variables initialization
training_file = ""
column_to_parse = None
if dataset_to_use == 'Spooky':
    training_file = "data/provided/train/train.csv"
    column_to_parse = 1
else:
    training_file = "data/custom/train/Corona_NLP_train.csv"
    column_to_parse = 4
```

a) Train embeddings on GIVEN dataset : Spooky Dataset

1. Function to parse data from training files

In [ ]:
```
# Read the file and prepare the training data
# so that it is in the following format

data = [['this', 'is', 'the', 'first', 'sentence', 'for', 'word2vec'],
                        ['this', 'is', 'the', 'second', 'sentence'],
                        ['yet', 'another', 'sentence'],
                        ['one', 'more', 'sentence'],
                        ['and', 'the', 'final', 'sentence']]

def parse_data(training_file_path: str, percentage: int, select_column:int) -
  """This function is used to parse input lines
  and returns a the provided percent of data.

  Args:
      lines (List[str]): list of lines
      percentage (int): percent of the dataset needed
      select_column (int): column to be selected from the dataset
  Returns:
      List[str]: lines (percentage of dataset)
  """
  sentences = []
  percentage_sentences = []
  with open(training_file_path, "r", encoding="utf8", errors="ignore") as csv
    csv_reader = reader(csvfile)
    #skipping header
    header = next(csv_reader)

    # line_length = len(list(csv_reader_copy))

    if header != None:
      for row in csv_reader:
        sentences.append(row[select_column])

    end_of_data = int(len(sentences) * percentage * .01)
    percentage_sentences = sentences[0:end_of_data]

  return percentage_sentences
```

2. Function to do preprocessing for the parsed data.

- Lower word case
- Remove stopwords
- Remove punctuations

In [ ]:
```
def preprocessing(running_lines: List[str]) -> List[List[str]]:
  """This function takes in the running test and return back the
  preprocessed text. Four tasks are done as part of this:
    1. lower word case
    2. remove stopwords
    3. remove punctuation
    4. Add - <s> and </s> for every sentence

  Args:
      running_lines (List[str]): list of lines

  Returns:
      List[List[str]]: list of sentences where each sentence is broken
                       into list of words.
  """
  preprocessed_lines = []
  tokenizer = RegexpTokenizer(r'\w+')
  for line in running_lines:
    lower_case_data = line.lower()
    data_without_stop_word = remove_stopwords(lower_case_data)
    data_without_punct = strip_punctuation(data_without_stop_word)
    processed_data = tokenizer.tokenize(data_without_punct)
    processed_data.insert(0,"<s>")
    processed_data.append("</s>")
    preprocessed_lines.append(processed_data)
  return preprocessed_lines
```

3. Computing the parsing and preprocessing for both the datasets.

In [ ]:
```
# parse and preprocess data
print(f'Tokenizing Provided Dataset: {dataset_to_use} Dataset\n')
sentences = preprocessing(parse_data(training_file, dataset_percentage, colum
# for token in sentences:
#   print(token)
```

```
Tokenizing Provided Dataset: Spooky Dataset
```

4. Provide functionality for generating word embeddings for a given model.

Note: For the Word2Vec model training, we are using Skip Gram model as shown with value sg in the below code. Also, we are keep generating the vocab in sorted order (descending).

In [ ]:

```
# The dimension of word embedding.
# This variable will be used throughout the program
# you may vary this as you desire
EMBEDDINGS_SIZE = 200

# Train the Word2Vec model from Gensim.
# Below are the hyperparameters that are most relevant.
# But feel free to explore other
# options too:
sg = 1 # The training algorithm, either CBOW(0) or skip gram(1). The default
window = 5
size = EMBEDDINGS_SIZE
min_count = 1
workers = cpu_count()
sorted_vocab = 1 #1 for descending order

def generate_embeddings(model_name: str, sentences: List[List[str]]) -> (str,
    """This function is used to generate embeddings (model and word vectors)
    model name and provided sentences.

    Args:
        model_name (str): name of the model
        sentences (List[List[str]]): sentences to be used for model creation

    Returns:
        str, str : model path and word vector path
    """
    #generate word2vec model
    model = Word2Vec(
        sentences=sentences,
        vector_size=size,
        window=window,
        min_count=min_count,
        workers=workers,
        sg=sg,
        sorted_vocab=sorted_vocab)
    # save model
    model_file_name = f'word2vec.{model_name}.model'
    model_path = f'{model_name}/model'
    # make sure all the parent folders exists
    Path(model_path).mkdir(parents=True, exist_ok=True)
    model_file_path = f'{model_path}/{model_file_name}'
    model.save(model_file_path)
    # Store just the words + their trained embeddings.
    word_vectors_file_name = f'word2vec.{model_name}.wordvectors'
    word_vectors = model.wv
    word_vectors_path = f'{model_name}/wordvectors'
    # make sure all the parent folders exists
    Path(word_vectors_path).mkdir(parents=True, exist_ok=True)
    word_vectors_file_path = f'{word_vectors_path}/{word_vectors_file_name}'
    word_vectors.save(word_vectors_file_path)
    return model_file_path, word_vectors_file_path
```

5. Generate model and word vectors

In [ ]:
```
model_path, word_vectors_path = generate_embeddings(dataset_to_use, sentences
```

6. Display the vocabulary size

In [ ]:
```
# print out the vocabulary size for defined dataset
model = Word2Vec.load(model_path)
print('Vocab size: {} for {} Dataset.'.format(len(model.wv), dataset_to_use))
```

```
Vocab size: 25067 for Spooky Dataset.
```

7. Saving the word embeddings in a text file (load later if needed)

In [ ]:
```
# You can save file in txt format, then load later if you wish.
# make sure all the parent folders exists
embeddings_folder_path = f'{dataset_to_use}/embeddings'
Path(embeddings_folder_path).mkdir(parents=True, exist_ok=True)
model.wv.save_word2vec_format(f'{embeddings_folder_path}/{dataset_to_use}_emb
```

b) Train embedding on YOUR dataset

In [ ]:
```
# For getting word embeddings for covid dataset, just make 'dataset_to_use' d
# "Covid" and it will generate the respective embeddings in the 'model_name/e
# name 'model_name/embeddings.txt'
```

What text-normalization and pre-processing did you do and why?

- Text Normalization and Pre-processing steps done:
  - Lower word case
  - Remove stopwords
  - Remove punctuations
  - Tokenize line and add sentence seperator tokens
- Lower word case:
  - This normalization step reduces our vocabulary size and allows fewer embeddings, which decreased runtime and increased performance relative to accuracy.
- Remove stopwords:
  - Removing the stop words will reduce the size of the vocabulary, and these high frequency words can skew the predictions of the model.
  - Our vocabulary is again reduced, which will decrease training times.
- Remove punctations:
  - Same reason as lower word case.
- Tokenizing line and separatign by sentence tokens:

- This adds extra context onto the words and practically speaking had a drastic effect on accuracy. For predicting the next word, this extra context seemed to be useful. On the Spooky dataset it increased the accuracy from roughly .05% without it to roughly 9% with the sentence tokens

Step 2: Evaluate the differences between the word embeddings

(make sure to include graphs, figures, and paragraphs with full sentences)

Produce any graphs or figures that illustrate what you have found and write 2 - 3 paragraphs describing the differences you find between your two sets of embeddings and why you see them.

1. Defining functions to generate vocab and get top k frequent words from the parsed data

In [ ]:

```
def generate_vocab(sentences:List[List[str]])->Dict:
    """This function is used to generate vocabs from the given sentences.

    Args:
        sentences (List[str]): sentences

    Returns:
        dict: word with each having count
    """
    word_counts = Counter()
    for sentence in sentences:
        for i in sentence:
            word_counts[i] += 1
    return word_counts

def get_top_k_words(sentences:str, k:int)->Dict:
    """This function is used to get top k word from the given
    sentences.

    Args:
        sentences (str): list of all the sentences to be used for count
        k (int): top k words (max frequency)

    Returns:
        Dict[str, int]: k words with their count
    """
    word_counts = generate_vocab(sentences)
    word_counts["<s>"] = 0
    word_counts["</s>"] = 0
    k_most_common = []
    for i in word_counts.most_common(k):
        k_most_common.append(i[0])
    # print(word_counts.most_common(k)) #Seeing the corresponding counts
    return k_most_common
```

2. Generate keys

In [ ]:
```
TOP_N_KEYS = 10
wv = KeyedVectors.load(word_vectors_path, mmap='r')
keys = get_top_k_words(sentences, TOP_N_KEYS)
# print(keys)
# print(model.wv.most_similar(keys[0], topn=10))
```

3. Define embedding clusters and word clusters

In [ ]:
```
def generate_embeddings_word_clusters(model, keys:Dict):
    '''
    This function is used to generate embedding and word clusters
    for the given model and the respective keys.

    Args:
        model (word2vec): model to be used
        keys (Dict): keys which are most frequent in the model

    Returns:
        List, List: embedding and word cluster
    '''
    embedding_clusters = []
    word_clusters = []
    for word in keys:
        embeddings = []
        words = []
        # print(model.wv.most_similar(word, topn=10))
        for similar_word, _ in model.wv.most_similar(word, topn=10):
            words.append(similar_word)
            embeddings.append(model.wv[similar_word])
        embedding_clusters.append(embeddings)
        word_clusters.append(words)
    return embedding_clusters, word_clusters
```

4. Define t-sne projection function

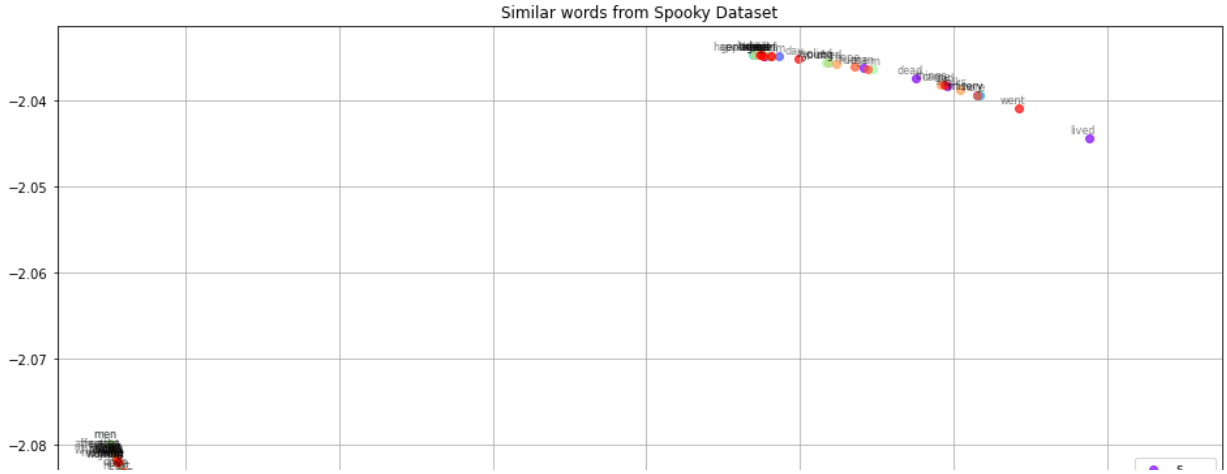
In [ ]:
```
def tsne_plot_similar_words(title:str, labels:List, embedding_clusters:List,
                            a:float, perplexity:int, filename:str=None):
    """This function is used to generate tsne plot projections for given para

    Args:
        title (str): plot title
        labels (List): list of labels
        embedding_clusters (List): list of embedding clusters
        word_clusters (List): list of word clusters
        a (float): alpha value for plot
        perplexity (int): perplexity value
        filename (str, optional): file name for the plot saving. Defaults to
    """
    embedding_clusters = np.array(embedding_clusters)
    n, m, k = embedding_clusters.shape
    tsne_model_en_2d = TSNE(perplexity=perplexity, n_components=2, init='pca'
    embeddings_en_2d = np.array(tsne_model_en_2d.fit_transform(embedding_clus

    plt.figure(figsize=(16, 9))
    colors = cm.rainbow(np.linspace(0, 1, len(labels)))
    for label, embeddings, words, color in zip(labels, embeddings_en_2d, word
        x = embeddings[:, 0]
        y = embeddings[:, 1]
        plt.scatter(x, y, c=color, alpha=a, label=label)
        for i, word in enumerate(words):
            plt.annotate(word, alpha=0.5, xy=(x[i], y[i]), xytext=(5, 2),
                         textcoords='offset points', ha='right', va='bottom',
    plt.legend(loc=4)
    plt.title(title)
    plt.grid(True)
    if filename:
        plt.savefig(filename, format='png', dpi=150, bbox_inches='tight')
    plt.show()
```

5. Generate embedding and word clusters. Also, generate 2d projections for spooky and covid dataset.

In [ ]:
```
PERPLEXITY = 100
ALPHA_VALUE = 0.7
embeddings_clusters, word_clusters = generate_embeddings_word_clusters(model,
# for i in range(1, 30, 3):
#     file_name = 'similar_words_perplexity_' + str(i) + '.png'
#     tsne_plot_similar_words('Similar words', keys, embedding_clusters, word
#     file_name)
images_folder_path = f'{dataset_to_use}/images'
Path(images_folder_path).mkdir(parents=True, exist_ok=True)
plot_file_path = f'{images_folder_path}/similar_words_perplexity_{PERPLEXITY}
tsne_plot_similar_words('Similar words from Spooky Dataset', keys, embeddings
    plot_file_path)
```

```
/usr/local/lib/python3.9/site-packages/sklearn/manifold/_t_sne.py:790: Future
Warning: The default learning rate in TSNE will change from 200.0 to 'auto' i
n 1.2.
  warnings.warn(
/usr/local/lib/python3.9/site-packages/sklearn/manifold/_t_sne.py:982: Future
Warning: The PCA initialization in TSNE will change to have the standard devi
```

```
ation of PC1 equal to 1e-4 in 1.2. This will ensure better convergence.
  warnings.warn(
*c* argument looks like a single numeric RGB or RGBA sequence, which should b
e avoided as value-mapping will have precedence in case its length matches wi
th *x* & *y*.  Please use the *color* keyword-argument or provide a 2-D array
with a single row if you intend to specify the same RGB or RGBA value for all
points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should b
e avoided as value-mapping will have precedence in case its length matches wi
th *x* & *y*.  Please use the *color* keyword-argument or provide a 2-D array
with a single row if you intend to specify the same RGB or RGBA value for all
points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should b
e avoided as value-mapping will have precedence in case its length matches wi
th *x* & *y*.  Please use the *color* keyword-argument or provide a 2-D array
with a single row if you intend to specify the same RGB or RGBA value for all
points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should b
e avoided as value-mapping will have precedence in case its length matches wi
th *x* & *y*.  Please use the *color* keyword-argument or provide a 2-D array
with a single row if you intend to specify the same RGB or RGBA value for all
points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should b
e avoided as value-mapping will have precedence in case its length matches wi
th *x* & *y*.  Please use the *color* keyword-argument or provide a 2-D array
with a single row if you intend to specify the same RGB or RGBA value for all
points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should b
e avoided as value-mapping will have precedence in case its length matches wi
th *x* & *y*.  Please use the *color* keyword-argument or provide a 2-D array
with a single row if you intend to specify the same RGB or RGBA value for all
points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should b
e avoided as value-mapping will have precedence in case its length matches wi
th *x* & *y*.  Please use the *color* keyword-argument or provide a 2-D array
with a single row if you intend to specify the same RGB or RGBA value for all
points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should b
e avoided as value-mapping will have precedence in case its length matches wi
th *x* & *y*.  Please use the *color* keyword-argument or provide a 2-D array
with a single row if you intend to specify the same RGB or RGBA value for all
points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should b
e avoided as value-mapping will have precedence in case its length matches wi
th *x* & *y*.  Please use the *color* keyword-argument or provide a 2-D array
with a single row if you intend to specify the same RGB or RGBA value for all
points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should b
e avoided as value-mapping will have precedence in case its length matches wi
th *x* & *y*.  Please use the *color* keyword-argument or provide a 2-D array
with a single row if you intend to specify the same RGB or RGBA value for all
points.
```



7. Produce any graphs or figures that illustrate what you have found and write 2 - 3 paragraphs describing the differences you find between your two sets of embeddings and why you see them.

NOTE: The graphs we're analyzing are called covid_perplexity_100.png and spooky_perplexity_100.png inside the analysis folder. Running our notebook will generate new images with perplexity 100 in the output cell as well as the images folder under each dataset.

1. Introductory for generation of graphs:
    - For the generation of the projection, we consider the top-k-words which are going to be most similar. For this homework, we assume the value to be 10. Next, we generated the embedding clusters and word clusters for those 10 words and created projection graphs. For the analysis part, we started looking into different range of perplexity for both the datasets starting from 0 to 10 and finally reached to the range close to 100. For the range nearer to 100, we got some converage for both the datasets, especially at the range from 95 to 103. Looking at these findings, we generate the graph at 100 perplexity and highlight some findings.
2. Describe Spooky
    - In this specific projection of our spooky dataset with perplexity 100, we've noticed that there are 4 groups. The top three groups appear to be tightly clustered, while the bottom group had a tight cluster to the middle center with scattered data points to the right. The scattered datapoints unravel slowly with 2 single outliers that are on top of each other. These are the points closest to the bottom right corner.
3. Describe Covid
    - There are 6 groups roughly speaking in the Covid dataset. The opposing clusters in the bottom left and top right corner appear to hold a large number of points each. The top left group is quite scattered. There are two outliers between the top and bottom left groups. There are 3 groups loosely arranged in the bottom right, which appear to be evenly distributed relative to each other.
4. Compare and contract the embeddings

- The Covid dataset appeared to have the two densest clusters at the opposing bottom left and top right corners while the Spooky dataset has data that's more tightly clustered overall. The Covid dataset appears to have more evenly distributed data in each of the two tightly clustered groups and the other 4 groups. From these comparisons of the two datasets, Spooky embeddings appear to be more easily separated into groups that are closely related, and the Covid embeddings suggest that the words themselves are less related. This would make sense given the higher vocabulary of the Covid dataset and the broader domain of the Covid dataset.

Cite your sources:

- Understanding Overview of the TSNE plot
- Sample example with digits

Step 3: Feedforward Neural Language Model

a) First, encode your text into integers

1. Define a function that would convert text to seqeunces

In [ ]:

```
def texts_to_sequences(tokenized_sentences:List[List[str]], vocab_index_mappi
    """This function is used to generate sequences from the given texts.

    Args:
        tokenized_sentences (List[List[str]]): list of list of tokens (all se
        vocab_index_mapping (dict): vocab mapped to index

    Returns:
        List[List[int]]: list of encoded sentences
    """
    encoded_sentences = []
    for sentence in tokenized_sentences:
        encoded_sentence = []
        for token in sentence:
            encoded_sentence.append(vocab_index_mapping[token])
        encoded_sentences.append(encoded_sentence)
    return encoded_sentences
```

2. Generate and encode sentences

In [ ]:

```
# Importing utility functions from Keras
from tensorflow.keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Input
from keras.losses import CategoricalCrossentropy
from tensorflow.keras.optimizers import SGD

# The size of the ngram language model you want to train
# change as needed for your experiments
NGRAM = 3

# It is used to vectorize a text corpus. Here, it just creates a mapping from
# word to a unique index. (Note: Indexing starts from 0)
# Example:
# tokenizer = Tokenizer()
# tokenizer.fit_on_texts(data)
# encoded = tokenizer.texts_to_sequences(data)

vocab = list(generate_vocab(sentences).keys())
# word -> index
vocab_index_mapping = {word:index for index, word in enumerate(vocab)}
encoded_sentences = texts_to_sequences(sentences, vocab_index_mapping)
# for sentence in encoded_sentences:
#     print(sentence)
```

b) Next, prepare your sequences from text

Fixed ngram based sequences

The training samples will be structured in the following format. Depending on which ngram model we choose, there will be (n-1) tokens in the input sequence (X) and we will need to predict the nth token (Y) X, y this, process however process, however afforded however, afforded me

3. Define function to generate ngrams from training samples

In [ ]:
```
def generate_ngram_training_samples(encoded_sentences: list) -> list:
    """Takes the encoded data (list of lists) and
    generates the training samples out of it.

    Args:
        encoded_sentences (list): encoded sentences

    Returns:
        list: generated ngrams from encoded sentences
        - list of lists in the format [[x1, x2, ... , x(n-1), y], ...]
    """
    generated_ngrams_list = []
    for sentence in encoded_sentences:
        generated_ngrams_list.append(list(ngrams(sentence, NGRAM + 1)))

    #enable to print generated ngrams
    # for generated_ngrams in generated_ngrams_list:
    #     for generated_ngram in generated_ngrams:
    #         print(generated_ngram)

    return generated_ngrams_list
```

4. Generating ngrams for encoded sentences.

In [ ]:
```
generated_ngrams = generate_ngram_training_samples(encoded_sentences)
```

c) Then, split the sequences into X and y and create a Data Generator

5. Defining function to split ngrams into X and Y.

In [ ]:
```
# Note here that the sequences were in the form:
# sequence = [x1, x2, ... , x(n-1), y]
# We still need to separate it into [[x1, x2, ... , x(n-1)], ...], [y1, y2, .

def split_ngram_to_training_sample(generated_ngrams: list):
    """This function is used to split the provided n grams into X and Y.

    Args:
        generated_ngrams (list): ngrams to be splitted

    Returns:
        X, Y: List X, List Y
    """
    generated_n_grams_copy = generated_ngrams.copy()
    X = []
    Y = []
    for ngrams in generated_n_grams_copy:
        for ngram in ngrams:
            ngram = list(ngram)
            Y.append(ngram.pop(len(ngram) - 1))
            X.append(ngram)
    return X, Y
```

6. Generating X and Y from generated ngrams

In [ ]:
```
X, Y = split_ngram_to_training_sample(generated_ngrams)
# for i in range(0, len(X)):
#     print("X: " + str(X[i]))
#     print("Y: " + str(Y[i]))

# print(X)
# print(Y)
```

7. Defining a function to convert X and Y from ngrams representation to embedddings representation

In [ ]:
```
def convert_ngrams_to_embeddings(X_ngrams :List, Y_grams:List, vocabulary_ind
    """This function is used to convert the provided ngrams into encoding rep
    and for that, use the already created vocabulary_index dict and word embe

    Args:
        X_ngrams (List): list of X ngrams
        Y_grams (List): list of Y ngrams
        vocabulary_index (Dict): vocab mapped to index
        word_embeddings (Dict): word mapped to embedding

    Returns:
        X, Y: X and Y (having embedding representation)
    """
    # you may find generating the following two dicts useful:
    # word to embedding : {'the':[0....], ...}
    # index to embedding : {1:[0....], ...}
    # use your tokenizer's word_index to find the index of
    # a given word

    #For X
    X_sentences_embeddings = []
    for items in X_ngrams:
        X_sentence_embeddings = []
        for index in items:
            word = vocabulary_index[index]
            embeddings = word_embeddings[word]
            X_sentence_embeddings.extend(embeddings)
        X_sentences_embeddings.append(X_sentence_embeddings)

    #For Y
    Y_total_embeddings = to_categorical(Y_grams)

    return X_sentences_embeddings, Y_total_embeddings
```

8. Converting X and Y from ngrams representation to embedddings representation from Spooky and Covid dataset

In [ ]:
```
#This step is computionally expensive. Set parameter in beginning to choose
#which embeddings to use
X_embeddings, Y_embeddings = convert_ngrams_to_embeddings(X, Y, vocab, wv)
```

9. Defining a data generator for given X, Y and batch size

In [ ]:

```
NUM_SEQUENCES_PER_BATCH = 1280 # this is the batch size

def data_generator(X_embeddings: List, y_embeddings: List, num_sequences_per_
    """Returns data generator to be used by feed_forward
    https://wiki.python.org/moin/Generators
    https://realpython.com/introduction-to-python-generators/

    Yields batches of embeddings and labels to go with them.
    Use one hot vectors to encode the labels

    Args:
        X_embeddings (List): embeddings of X
        y_embeddings (List): embeddings of y
        num_sequences_per_batch (int): number of seqences per batch

    Yields:
        X, y: X and Y arrays with size of the batch
    """
    steps_per_epochs = len(X_embeddings)//num_sequences_per_batch
    current_step = 0
    while True:
        start_cell = (int(current_step % steps_per_epochs)) * num_sequences_p
        end_cell = (int(current_step % steps_per_epochs) + 1) * num_sequences
        yield np.array(X_embeddings[start_cell:end_cell]),np.array(y_embeddin
        current_step += 1

def create_generator(X_embeddings, Y_embeddings):
    steps_per_epoch = len(X_embeddings)//NUM_SEQUENCES_PER_BATCH  # Number of
    ngram_generator = data_generator(X_embeddings, Y_embeddings ,NUM_SEQUENCE
    sample = next(ngram_generator)
    print(sample[0].shape)
    print(sample[1].shape)

    return ngram_generator, steps_per_epoch
```

10. Creating a data generator for given X, Y and batch size

In [ ]:

```
# Examples
# steps_per_epoch = len(spooky_X_embeddings)//num_sequences_per_batch  # Numb
# train_generator = data_generator(X, y, num_sequences_per_batch)

# sample=next(train_generator) # this is how you get data out of generators
# sample[0].shape # (batch_size, (n-1)*EMBEDDING_SIZE)  (128, 200)
# sample[1].shape   # (batch_size, |V|) to_categorical

# initialize data_generator
ngram_generator, steps_per_epoch = create_generator(X_embeddings, Y_embedding
# sample = next(ngram_generator)
# print(sample[0].shape)
# print(sample[1].shape)
```

```
(1280, 600)
(1280, 25067)
```

d) Train your models

1. Defining model for building

In [ ]:
```
# Define the model architecture using Keras Sequential API
def build_model(input_size:int, vocab_size:int):
    """This function is used to define architecture for the model

    Args:
        input_size (int): input size
        vocab_size (int): vocab size for output

    Returns:
        Sequential: keras model
    """
    # Define the model architecture using Keras Sequential API
    model = Sequential()
    model.add(Input((input_size, NGRAM*EMBEDDINGS_SIZE)))
    model.add(Dense(NGRAM*EMBEDDINGS_SIZE, activation='relu'))
    model.add(Dense(vocab_size, activation='softmax'))
    print(model.summary())
    return model
```

2. Generating a model

In [ ]:
```
# code to train a feedforward neural language model
# on a set of given word embeddings
# make sure not to just copy + paste to train your two models
model = build_model(len(X_embeddings), len(vocab))
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_4 (Dense)              (None, 220531, 600)       360600
_________________________________________________________________
dense_5 (Dense)              (None, 220531, 25067)     15065267
=================================================================
Total params: 15,425,867
Trainable params: 15,425,867
Non-trainable params: 0
_________________________________________________________________
None
```

3. Training the model

In [ ]:
```
# Start training the model
print("Model Compiling")
model.compile(loss=CategoricalCrossentropy(), optimizer=SGD(learning_rate=.1)
print("Model Training")
model.fit(x=ngram_generator, steps_per_epoch=steps_per_epoch, epochs=3)
```

```
Model Compiling
Model Training
Epoch 1/3
```

```
WARNING:tensorflow:Model was constructed with shape (None, 220531, 600) for i
nput KerasTensor(type_spec=TensorSpec(shape=(None, 220531, 600), dtype=tf.flo
at32, name='input_3'), name='input_3', description="created by layer 'input_3
'"), but it was called on an input with incompatible shape (None, None).
WARNING:tensorflow:Model was constructed with shape (None, 220531, 600) for i
nput KerasTensor(type_spec=TensorSpec(shape=(None, 220531, 600), dtype=tf.flo
at32, name='input_3'), name='input_3', description="created by layer 'input_3
'"), but it was called on an input with incompatible shape (None, None).
172/172 [==============================] - 86s 499ms/step - loss: 9.6415 - ac
curacy: 0.0868
Epoch 2/3
172/172 [==============================] - 78s 454ms/step - loss: 9.2620 - ac
curacy: 0.0884
Epoch 3/3
172/172 [==============================] - 84s 491ms/step - loss: 8.8883 - ac
curacy: 0.0884
```

Out[ ]: <keras.callbacks.History at 0x13f7fba00>

e) Generate Sentences

4. Defining functions for sentence generation and word generation

In [ ]:

```
def generate_seq(seed: list, n_words: int) -> str:
    """This function is used to generate a seq of words for given model and t

    Args:
        seed (list): seed for sequence
        n_words (int): number of words to be generated

    Returns:
        [str]: generated sentence
    """
    sequence = []
    for i in range(0, n_words):
        next_word = generate_next_word(seed)
        sequence.append(next_word)
        seed.append(next_word)
        seed.pop(0)
        # print(seed)
    return " ".join(sequence)

def generate_next_word(seed: list) -> str:
    """This function is used to generate next word for given seed

    Args:
        seed (list): list of words [w1, w2, .., (n-1)]

    Returns:
        str: generated word
    """
    sentence_embeddings = []
    array_embeddings = []

    #generate initial set of words
    for word in seed:
        word_index = vocab_index_mapping[word]
        word_embedding = wv[word_index]
        # print(word_embedding)
        sentence_embeddings.extend(word_embedding)
    # print(len(sentence_embeddings))
    array_embeddings.append(sentence_embeddings)
    # print(len(array_embeddings))
    array_embeddings = np.array(array_embeddings)
    # print(array_embeddings.shape)
    word_prob_matrix = model.predict(array_embeddings)
    # print(word_prob_matrix)
    next_word_index = np.random.choice(
                        np.arange(len(word_prob_matrix[0])), 1, p=word_prob_m
    # word_index = np.where(word_prob_matrix == (np.max(word_prob_matrix)))
    # print(word_index)
    # print(word_index[0][0])
    return vocab[next_word_index]
```

5. Generating sentences for a random seed sample

In [ ]:

```
# generation of sentences

# pick a random sample from the X as the initial seed
random_sample_index_list = X[random.randint(0, len(X)-1)]
seed = []
for index in random_sample_index_list:
    seed.append(vocab[index])
# print(seed)

for i in range(0, 50):
    generated_sentence = generate_seq(seed, 20)
    print(generated_sentence)
```

```
WARNING:tensorflow:Model was constructed with shape (None, 220531, 600) for i
nput KerasTensor(type_spec=TensorSpec(shape=(None, 220531, 600), dtype=tf.flo
at32, name='input_3'), name='input_3', description="created by layer 'input_3
'"), but it was called on an input with incompatible shape (None, 600).
scene packages infesting withaout stifle went reverend wicked wordy bearer fl
ames conceals cheap swarmed circulates unexplained lowness manly assenting op
portunely
spotted failure houri aesthetes glittering derivable majesty them insuperable
beholding undeniably concert defeated evoked slayin squeezed help conviction
pelief attempting
increased voluminously suns world past great </s> shoal rocks wearing tunic c
atch peculiarity deprecating attain clasped shouts </s> dwellings homewards
aroma less applauded mystifique warming deliberately advices communication ox
onian bras evinces requested group permeates vapory le form began like preten
ces
spiced babes spiral affright untoward rabbits wood possessed genealogical usu
al charities lettres selvatic hxwl misdeed ales pyramid elizabethan opens est
counterfeits enchained roseate fallacious kiao anon irksome sereno proposed c
onic gaily teios hostile generous scarabæus peevish relief physiology prim pr
ofitlessly
eugenie stephen apprised pleasure olden unearthed fibres somewhat satin conse
nt reproaching shew perds broomstick unremitting reduplication xpx disciples
thoroughness booked
saound keyhole deestrick ice pastime procured afterwards scarf unpossessed no
minate rotund tieck quest s pageantry temps dories yearned impact villainous
camorin animated myrmidons vibration prayer regimen heresy causes em acknowle
dgement nebber glimpsing retina extreme cities equations die causeless guttur
al veil
eton before cumaean originally resignation infallibly turned vigor follow gro
oming schaack accorded elbowing combining groping imperium rode mind inaction
cakes
appalling come su glacier worthy plastic contrive pounds me trimmed brake uni
tes crackle famous excessive emissary night launched paragrab raymond
unhappy jos adornments furtherance villainous cask treadin refusal firmanent
apparently merry haunt tapestries body t unwhisperable wrecked outraged windo
wed knew
wipin cornelius </s> jervas credentials taown solved unexpected nineteenth de
signated necessity mind transcendentals monstrously conceived unite astute co
lloquial relativity camps
strips omne first muttering copses xii man neber </s> face worship raiment di
ved and neber ripping whale anything disdain it
overlapping intruders letters spreading excellencies heart pensé tour sanguin
ary angel damnation unattractive acrost authoritative peine encumbered desola
te burrowers conversations chiefly
```

oblige gratitude retarded astrology cluster affrays indicates dread mecca pri
me nought extremest strange gruff </s> sullied discredit zay divinely effectu
al
certainty evidences manuscript maouths climbing debilitated horizontally sine
cures acumen palled deliberations headin donnelly warmly terminates twas comp
osed time hand partly
raymond came vd misarrangement blinded flagstones warring honeysuckle exhibit
desertion </s> isis flies conservatism crawford solicitude wretchedly progres
sion amazed but
bandaged dilapidation ravenna obstructed paternity peristyles enraptured temp
eramentally repairing cotopaxi zaffre peddler disposed extracts fanlight laug
hing biographer unnerved human examine
abashed sleeplessly mumbled whistled dwarf swine decorum auriclas construe gh
oulish orgies literary snorted someone sinew hire drastic acceptation inverts
minaret
affidavits palings net footstep eaves patient puncheon bag flying shorter per
meates justice notably concentrative prospector ran shewed boundless tolerabl
e surely
whilst slippered cordiality sake rife cards stream ownership chins torny pass
ion duplicate préparées nxbxdy constantinople obscenely diverted toute fetter
obstinate
menes searing missing billow gentlemen expresses maltese serving hesitated ap
proved bracknel thet remembrancers casket partition bundle resigning subcentu
rio blitzen bibulus
passive mystified alma unattractive provinces bringin hints viso surveyed ren
ts scymetars farming vaow springy beacon valentinianus consummation militia c
laimant appertaining
neatness sheer solace heeled tyrannical splash thing certainty severity disav
owed sensitive offcasts lied proceeded ushers bristly conversed ram envied un
heeding
legrasse desarving anemone island protects meed nose whateley bankrupt cthulh
u sidled glowing eyes piano supine technically domes knocked potion topsy
luminous terrible musée sure returned roamed transmitting javelin queerest en
graving world rivalry proceeds </s> andrée happy eyes lauding scoundrels orch
ids
singleness unextinguishable coarsest species son insinuating frankish games t
hat hovered regard place starry terrified captious institute emulate absent g
athering exit
secretive reflection philosophical crossed brilliant frontiers effacing matth
ew quart reliques pehabe decadent transversely resist sidewalk an occur stric
t conciliate air
juxtaposition directness tore taken gawd murderers future </s> stay street mi
ghtiest miss reflection opponents enter tribunal wicker traders devoutest cat
holic
clout bussy breathes likenesses springs raft streamlets compromised durable s
hut request hopeful blowed ligeia uncongenial canvassed things wholesome flig
ht awakeners
versailles undauntedly count successively doubts guffaw deputed lean grotesqu
e thet maturity common from once mossy benefit knew </s> inquisitive </s>
pesty paralysis huddling chords sulphur focus death half registering cloudine
ss shingled greasiness nourishment degrade holborn aussi surliness whining li
vadia question
afterward stepped frieze pearls piqued diurnal somnolence piute carpeted retu
rning privation vast parks thirty fatal tingle singly indifference ring mouth
ful
watchers overtaken deposits prefer chaldaea suddent markings reapproached bet
humbed patriarchal disengagement greed wake incinerated endearing basically d
elight hail sagaciously diabolical
practise tomb fades diddled rowena excitement electrical fhtagn exultingly pe

```
aaema mattered smothering </s> n shifts </s> looked liddeason mentioned proce
ssion
</s> compassionate wilbur occasioned cardiac air nature jealousy casually fra
mes collected best trepanning officious dios panoramic omnipotent shadow spec
ulum autumnal
extinguished appears conventional clutched entric </s> transmitter raphaelite
s neighbourhood </s> ruleth spellbound suspects opium unfinished eternal leng
th subordinate </s> remained
nova disdaining speaker provision lunch caprice throned pennant shoulder caus
ing vineyards mature instilled archytas drapery systematizing abide merchant
million laugh
examine volney perforations fleet reserving real husband it withholding sway
palpable strife wooden proper dollar speedy supposing expeditions woke isolat
ed
shrouded protectors alternations schoolmaster effulgence machine massive smal
lest plates unclean recognize blinds afore passengers haze chisel recognise b
urial convers unreal
sisters predominant nib aeronaut normal actions strengthen decomposed harkene
d manservant cherishing varieties humans phantasm establishment disappoint ci
tadel admirably vow inferiority
represents transportation counterparts secreting rumor gauze conspicuous untr
avelled brought closer chickens final absense steely greet spins bubastis met
aphysicians portholes deliberations
right theodatus minutiae speaks fire indemnity property layer reanimation beg
an canaries an raphaelites insulting daisies mix essayed chaldee solemnly bra
ided
previous fringed ambulance transference quid wainscots methodist parental val
iant exclusively visible still vain said sxrrxws england grated innumerable
</s> year
</s> invitations voice realm minuscules referrable molestation mocked greek i
dea london despite otterholm specific flights excelling eyebrows lid severed
friendly
basket deciphered static misuse diverted called dullard streaming ambush lade
n impend promises moccasined misfortune exonerated resonant himself devotee m
ovements wasn
branches agreed mann julie informed avenues l confronts patronized aaem brook
alleys apologize blade eye flanked forth left ionic revivification
harvest fetter shuddering mai facetious cent wyatt assaulted corpulence it bu
ckles hebrews generality beauclerk undisturbed missed distinct undersea romno
d courtesies
interrupt set remoteness snowless welcoming seemest terrifiedly seemed owd bu
ry telepathic raggiar drowsiness angular praying child crucible sounding </s>
```

f) Compare your generated sentences

Answer the question: Do your neural language models produce n-grams that were not observed during training? (1 paragraph, you may support this answer with code as desired).

- It can produce ngrams which are not observed during training. It is extremely unlikely to find a dataset which would cover all the possible n-grams that can be generated from the dataset. Lets say we consider n-gram "the cat was". For this, a possible ngram for these words could be "was the cat" which is something that may not be a part of the dataset. In that case, we will have a situation where we get a ngram that is not observed during training. If the words are close related to each other, in this instance, say "was", "the", "cat", then its possible that the model will generate a sequence with these words

which is unseen in the training dataset.
- Now, lets consider this scenario with numbers. Consider the sequence "1 2 3 4 3 2 1". Refering back to the naive bayes methodology of generating words in homework 2, given the initial character "4", the only number that could be generated from the matrix with probabilities would be "3". Now consider "4 3". After "3", the numbers "2" and "4" both appear after "3". Its possible to generate the sequence "4 3 4" given that "3" could appear after "4" and "4" could appear after "3". Relating this back to our word embeddings scenario, given a seed of "4 3", then model could predict the next value to be "2" or "4" because the embeddings should be similar in this scenario.
- If we extend this idea to a much larger vocab and a model trained on many more sequences could generate a character n after (n-1) that has never appeared two spots after (n-2). Here n denotes the position of the character.

Compare your generated sentences:

- This neural language model can consider context information in a more sophisticated manner, however, the sentences generated that we observed made less sense. We should note our model's accuracy was quite low at approximately 9%. Neither sentences had good grammar, but the words following each other in the HW2 language model made some level of sense while the words generated after a seed in our neural model oftentimes appeared to be highly unrelated and make very little sense.
- One example from the HW2 would be "- \~~- do they serve spicy \~~\". Compared to this example from our neural language model "ploughed sufficient sidled jewelled garrets enacted toils both o capacity longue knot circumference then harbour weird felt honeycombed new bathed." This could be attributed to the relatively simple architecture of our model that couldn't transform the data much before classifying it. This dataset also seemed to be relatively small, but we don't have enough context to make that judgement.
- While it's not clear from our data that neural language models have a better sense of context than the N_GRAM model, they can store much more information, which requires more data and a more thoughtful architecture to extract useful results.

Sentences generated from the neural language model using 100% of the spooky dataset:

- narrowed moat committee sister membrane look bitter parchment sun </s> distinguish crest felix jargon half endure simple like little </s>
- </s> intervention unknowing winked landsmen fettered </s> aout by prosecute cud excesses unatteneded apparently but pollutes pitchy </s> scored essentially
- air sympathised toll slayin egotism mace autobiography carting stormy tuberoses avert curator come communion purlieus sabbat </s> intruding toes right
- extinction lowness latitude reckoned creeping kinsfolk bags damnable resurrection chords </s> insufficient authors scene light know </s> raise fortifications like
- sensation effudit fatal inordinate africa way politician heard irish oldeb myself eyelid

corner wide land god </s> seaport c away

- depart s </s> alive awful </s> </s> </s> </s> overwhelmed cheek una scene fact </s> apartment hope men scapegrace there
- mechanically wanton laboriously loft outlawed indifference hill recovering untraceable communication showed rowboat paper beau men orders apace resurrection ushers nigger
- riddles all raymond instigated scandal anomalousness more villany </s> typifies lottery crammed froze stairs despair unfrequently despicable engendered inroads spirit
- unknown equally lord armands thickened observer appeared walnut funerals caravan conquers stature elapsing piously ian unclothed whereto babel sharing misfortune
- strange sun multitudinous account </s> trot drum joyful retired death knees one vast twelve buoyantly print groped park inquietude places
- billows financial void pitchers carcass antiphonal zest inconvenient </s> earnest lathered however </s> rehoboth </s> </s> exergues it shifts op
- foremost refresh subtle decorated frail islands conciliatory resurgence described patriotic aeronaut exploring avenger chemist slips presence transmitting shuns semi excessive
- ship convicted senatorial obliterating eend vaow caked divest dark welcome alkaloid noxious loftily prophecies small observation partly morbid steeds seizure
- turns monstrosity slovenly senct beguiled favourites gates guinea bristol unflinching spiritual puppy annihilate use multiplicity glaciers frank displace remained subject
- liveliest exhalations unspeakable infection gossiped constellations species arabesque till grounds simple </s> </s> came imminence unsought steeply topical headquarters gloss
- rally dispatched bon aëroplane cautioned greedy small mr arduous syllable adams masculine soften bartholinus representatives infidels eluding gilman degenerating heather
- towering alexandre wan mistiness civilisation bestowed athenians foiled farther disillusion marie rummer chuckle drains scholar execrate pleasureable denied queerness strides
- elephant foresight recuperating tray caude refining vengeful bellows pictures subserved ruins green shock unreserved me death </s> hands said ketch
- wheezed mercantile hideous elihu course antennæ absolutely </s> i different wig </s> thing country </s> head strictness huddled main embrasure
- years meanest hollands poorhouse disjointedly es flavius maurice ignominious angelic water preignac cloud story eccentrick </s> animadversions pea cyrus swammerdamm
- stick denominated dents invulnerable risers cultivate bluddennuff delighted massacring smug answer interred calmed alcyone conversation overturning use stealing contradictions irrefutable
- trammpled grief sowing obstrusive polite jostled puff ongas shelterless triumphs meaning unfastened drawer heated holmes venetian indulges retarded o bafflingly
- lowness factitious mankind economy untamed de elagabalus nerve animating dyin volcanoes carries grouping soared idlers unmixed purse obadiah engine overran

- aid mithridates tempered rubbish darkness untimely helvia curiosity crisp violate and bruise outstretched disputing offend quieted opponents proffit irrelevant affair
- directing </s> directionless right grew assisting glorified murmuring distinctly bribing sister blunting decomposition happiness dignity </s> countenance years dwelled introduce
- adversities tastes afflatus educations hallowmass longer veil substantive dully hanging fig succourless rest estranged rest disputed merciful murmur countenance hope
- </s> me wonderful white topical vernunft exterminating energetically old </s> sea speak </s> forward furnaces dread whitewash physiognomy apothecary argument
- ideal countenance knows intelligence oaken lobe draughts merrival dieu famed esteban ramble satisfactorily existence analyzed lobster humans fanciful frayed informe
- gemmary shepherd exceedingly projected dawn xura worthy rhine magnificence sky furtively artistic west seemed other approached souvenir misery sartain kingdom
- </s> love hardly throws like abhorrence understood target hardens wants disfigured lithe reverent emphatically empires cameleopards orontes horses class daddyship
- delved eventful terming weird cough ignorance travels malignancy tool persuasions lh languidly composed manners gnorri repute amorphous society berlifitzing morning
- says conquering </s> evenly second attention fishermen way horror exhibit attention met </s> person exclude relish plumped merits visibility america
- desires republican know lapse untrue encampment induce ut light gone wiry servitude critter stretch computed unus dross clearly air looking
- muttering been evidence water room </s> harangue dangers turk wall directions thing open extravagances necessarily telescopes profuseness possible evidently smote
- devotions scourged tortures longer signs distributing nill animation apotheosis trotter pesty italian peg dissimulation unmingled natural brimmed me bargeman runnin
- decreases unconsciousness daedalus thy innovation fortified hogsheads slovenly rate discussions run acted abstemious peradventure exuberant frivolous tampering relaxation mythology worthlessness
- sidereal brusquerie proved harmonized paralyzing forth stranger believe </s> arkham scholarly seamen innumerable preferably immolation gesticulations striven flux robber post
- firmness thar men procure masterminds irregularity </s> introduction </s> darkness it unhuman air stupendous damsel polynices culled notoriously cenis resolve
- shunned an flight artemis maillardet moody knew fiend theorize halo comfortless trotter interferin antagonist lustre robust leads unused romance vascones
- </s> energy torrent fungous which power spirit venny it recognition tressel overnourished dark gay ascension decision soother habitués stupid enterred
- pumps flos reproduce cannibal flame green diavolo spot before sime sublunary cicale tantalized binding administration retreating </s> sufficient hatred dream
- man intrigue maps looked bristol inheritance wall instil wines unhappily fervour alarmedly doubles anecdotes embellished haggardness alloy stopping height springs
- ploughed sufficient sidled jewelled garrets enacted toils both o capacity longue knot

circumference then harbour weird felt honeycombed new bathed
- slaying diagnosed proposed lonely untenanted deficient daemonologist domestic grey donnelly opiates care diet illogically voice petto tantalisingly crucifix turk government
- fences essayed palliative ruts threw correcting converse attention summoning rebuke imagining ridicule zaiat imprisonment coast account parthenon inattentive words verbatim
- befitting like cureless overheard younguns started dwelling like me infernal faded object horrific remoter swept misapplied collected things knowed seaweed
- area nova protection focussing lost même sixteenth portent naivete luxuriant muddy stupid altering disinterested abaft maple nourished waiting dulls mad
- trabes placing learn nature sorrow civilian maintained </s> better monster z </s> stare combined sentiments delight hands twilights alas flight
- expectant </s> </s> embrace r identified staggering cincinnatus tones place crying associations selfishness half polynesian reasons hxmx submarine occupying suffered
- empire </s> let prosy lived clear square strange austrians plotted lower superintended prayers ver solely night harpies felt greedy superadded

Sentences generated with Jinesh Language Model from HW2

Model: bigram, laplace smoothed Sentences:

(Removing sentence start and end tokens because they are causing display issues in HTML file)

- ten and twenty dollars
- any polish food
- it
- it on martin luther king avenue
- do they serve spicy
- forget it doesn't matter
- around icsi
- it should be sunday
- five dollars
- just two miles give me more than a few minute
- i would like to spend less than a mile
- about christopher's cafe
- what is the list please
- okay
- not say about restoran-rasa-sayang
- now i'd like to eat some more about monday
- please give me about tuesday
- i would be any like to icsi
- i wanna go out to icsi
- i wanna burrito

- i see the a uh which of hong-kong villa international house
- please give me information about great china
- give me a soup do you have any day
- the distance
- i want to eat indian restaurants that question
- where can i want to spend fifteen dollars
- i don't care how about guerrero's
- i'm not say uh to a car so it should be inexpensive
- okay now
- i'm looking for a thai barbecue flint's barbecue the service at pasand
- uh thai food i'd like to eat sushi
- chicken
- uh find me about restaurants
- start over
- i like to find a sunday
- i want to eat breakfast
- i can you get dinner
- i want to eat uh i'm looking for dessert
- dinner thursday
- what i would a cheap cafe
- not on shattuck
- start over
- i have additional information about any day
- so it can be reasonably cheap
- start over
- next thursday or however you have some indian food in asian

Sources Cited

In [ ]: