Next Word Predictor

By Daniel Hefel, Ian Huang, Luke Li

Professor Marc Verhagen

Fall 2019

Abstract/Introduction

This project aims to make a program that predicts the next word a user is going to write given the word or phrase that the user typed previously. We used 2 ways of implementing- nltk n-gram and RNN/LSTM methods to train the data. For the second method, we further broke this down into predicting the next word and integrating it into the suggestions along with n-grams, and then a second option of generating a text character-by-character using RNN trained on a text.

Motivation

After coming across the new gmail feature, which predicts what a user is going to type based on what the user has typed and user's tendency of typing through machine learning, we thought it would be pretty cool if we can use the tools that we learned in Ling 131, especially the N-gram models, to build a next word predictor. Other similar features that we are inspired by are iOS and Andriod's predictive text. The predictive ability would most likely not match up to the likes of which, but nonetheless it should provide an important learning experience.

Theory

1.1 N-Gram

This is a very simple and straightforward approach that we have done in class, as most of it is inherently done in nltk by default. But basically we take all word pairs and tally up how many times they appear, and do a search to find the most likely second pair to a certain word. The pairing would be as follows given a sequence of words:

A B C D E

(A,B), (B,C),(C,D),(D,E)

This will be done through the entire provided document, and there will be overlaps (say another (A,B)) and we will just add one to the count. The same fundamental idea holds for larger n-grams, however the probability of another overlap diminishes as we increase n, so a hexagram may have very few overlaps. Thus if we are given an input, the chances are great in which our input is not recognized in the model, and therefore no prediction can be made.

1.2 RNN and LSTM

Unlike normal neural networks, Recurrent Neural Network (RNN) retains the memory previously so it can use it again. The structure has loops and is more like the human brain in that it builds upon previous information. This is crucial as word prediction heavily depends on the words that come before it, and RNNs are much more suitable for handling such tasks.

Long Short Term memory (LSTM) is a special type of RNN where it retains only the recent information, as often times the information way back doesn't cause trouble. It also prevents the issue of long term dependencies, which is something that n-grams face given a large n. LSTMs are actually capable of learning these long term dependencies. LSTMs are known to be better for sequences and as such, very crucial to our task.

Methods

Due to the nature of this project being divided upon our group members, we had the idea of trying different methods and merging them into one. The most immediate implementation that came to mind is a bigram frequency distribution, in which we could take the last word and find the most common bigram that would occur after it. This method obviously does not account for anything besides the last word and the context of which depends entirely on the data we get the frequencies from. To help amend this issue, we decided to use the nltk web text and genesis text to start. First we get two most common bigrams, one from each text and return it. This way at least the topics are more varied, and it offered a humorous dichotomy of internet speech and occasionally morbid bigrams from the genesis text.

This was implemented in a rather simplistic manner, in which will be described.

```
from nltk import FreqDist
import nltk
nltk.download('webtext')
nltk.download('genesis')
STOPLIST = set(nltk.corpus.stopwords.words())
def is_content_word(word):
    return word.lower() not in STOPLIST and word[0].isalpha()
textweb = nltk.corpus.webtext.words()
textgod = nltk.corpus.genesis.words()
bigramsweb = [b for b in list(nltk.bigrams(textweb)) if is_content_word(b[1])]
bigramsgod = [b for b in list(nltk.bigrams(textgod)) if is_content_word(b[1])]
fd = nltk.ConditionalFreqDist(bigramsweb)
fd1 = nltk.ConditionalFreqDist(bigramsgod)
```

In the code above, we mostly used the nltk library and filtered the stopwords from the bigrams. The two bigram datasets we extracted them from is ‘webtext’ and ‘genesis’. We can then get our desired bigram half from using fd[‘text’] and fd1[‘text’] where text is the last word of the input sentence, cast it to a list and return the first element which is the most frequent.

Although the results were rather hilarious and only accurate on the occasion, we decided that perhaps larger n-grams would provide better accuracy. So we added trigrams to our model, using the nltk corpus reuters for the source text. Obviously the same issues from bigrams carry over to trigrams, that being the context of the prediction being constricted to the source text the trigrams were extracted from. However, adding one extra word into consideration should improve our model by significant means. Once again, the implementation is very simple, as most of it was inherently done in nltk.

```
from nltk.corpus import reuters
from nltk import bigrams, trigrams
from collections import Counter, defaultdict
import nltk
nltk.download('reuters')

model = defaultdict(lambda: defaultdict(lambda: 0))

# Count frequencies
for sentence in reuters.sents():
  for w1, w2, w3 in trigrams(sentence, pad_right=True, pad_left=True):
     model[(w1, w2)][w3] += 1

# Transform the counts to probabilities
for w1_w2 in model:
  total_count = float(sum(model[w1_w2].values()))
  for w3 in model[w1_w2]:
     model[w1_w2][w3] /= total_count
```

Here, the model parameter is our result. The nested for-loop parses the sentences and extracts the trigrams, then tallies them up (counts the frequency) and puts it in the model. We then convert the counts into probabilities and our model is complete. To access the third word we simply case model to a dictionary, dict(model[‘word1’, ‘word2’]) where the keys are the first two words and the values are the probabilities. We can iterate over the keys and values and find the third term with the largest probability by using a loop.

```
# Return the trigram with largest frequency
def getTrigrams(x1, x2):
  probs = 0
  best = ''
  for x,y in dict(model[x1, x2]).items():
     if y>probs:
        probs = y
        best = x
  return best
```

Where x is the key, y is the value. The parameter probs is a placeholder for the max probability value, and the parameter best is the placeholder for the corresponding key which we will return.

Despite how successful or unsuccessful our bigram and trigram models can be, it simply is not enough for a valid predictor. There are plenty of scenarios where some words not found in the bigram/trigram dictionary would cause issues, we just had them return nothing in that case. It turns out that trigrams had more cases of null returns as more combinations would call for a larger dataset to pull trigrams from. After doing some research, it seemed like LSTMs were the way to go.

LSTM is a long short-term memory recurrent neural network, and it was in the well known keras library. In our implementation we used *The Adventures of Sherlock Holmes* as the dataset, which was 581887 words long. We decided to consider the last 5 words in the LSTM model. The preprocessing step is similar to a n-gram where we shift our window of n words and add them to a list. We then OneHotEncoded the data using boolean data types, and trained our model with a single layer of 128 neurons, another full layer and used the softmax function. The softmax function basically condenses the output from the previous layer and is commonly the last layer. We then used the RMSprop to optimize the training process, which is notoriously long and memory intensive in neural networks. The functions are pretty self explanatory, the sample function is one that chooses the optimal words from the predictions. However, just like the constraints of the n-grams, we are limited to 5 words before the prediction and all other words are ignored. However, we are certain that the performance would outperform a 5-gram greatly, as there would be not enough data to extrapolate enough permutations, even if there is it is very cost inefficient.

For the text generation via character prediction, we trained a model on grail.txt that would then generate a text of a length of the user's choosing that would be formed based on what the user entered. The model was created over a series of 50 epochs that gradually decreased loss and made the model more efficient. Originally, the goal was to train on text8, but unfortunately that was changed because creating the model alone would leave no time to see if it even worked. This led to next selecting Emma, which required around 2 hours per epoch to complete a model over it's ~800,000 words. Thus, grail.txt was chosen for its manageability, despite not being the ideal text to train on. In addition to these functions, we attempted to create something that would have opened a new window where the user could enter their text and see the prediction of the computer in real time for just the next "word" (here being anything separated by word boundaries, regardless if it is a real word), and then see the results. Unfortunately, we were unable to implement this, though we have included the code that would have been used for this in the widget.py file.

Installation with pip to run our project

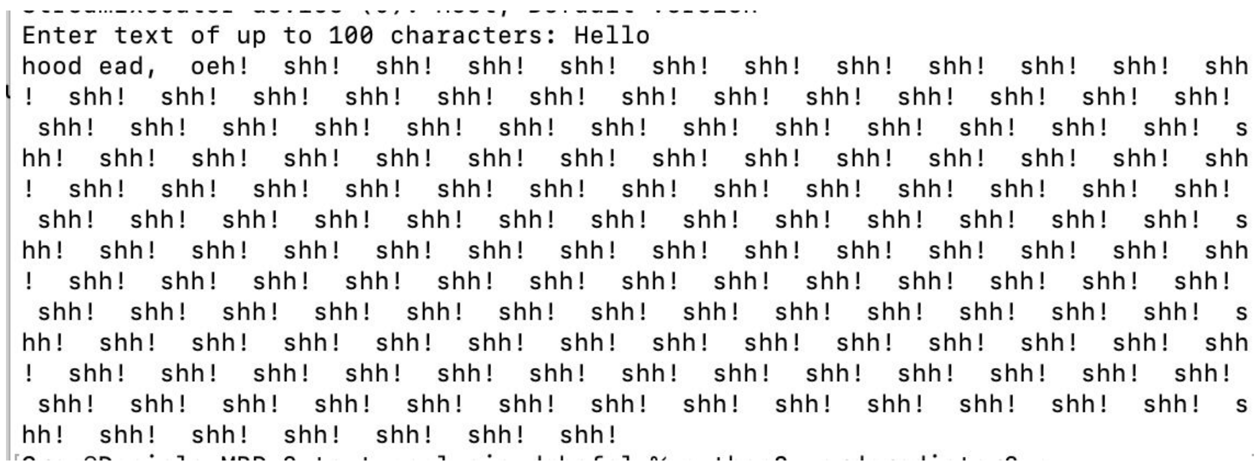
1. Tensorflow
2. Keras
3. Pickle
4. Heapq

5. Nltk
6. Numpy
7. Os

Results

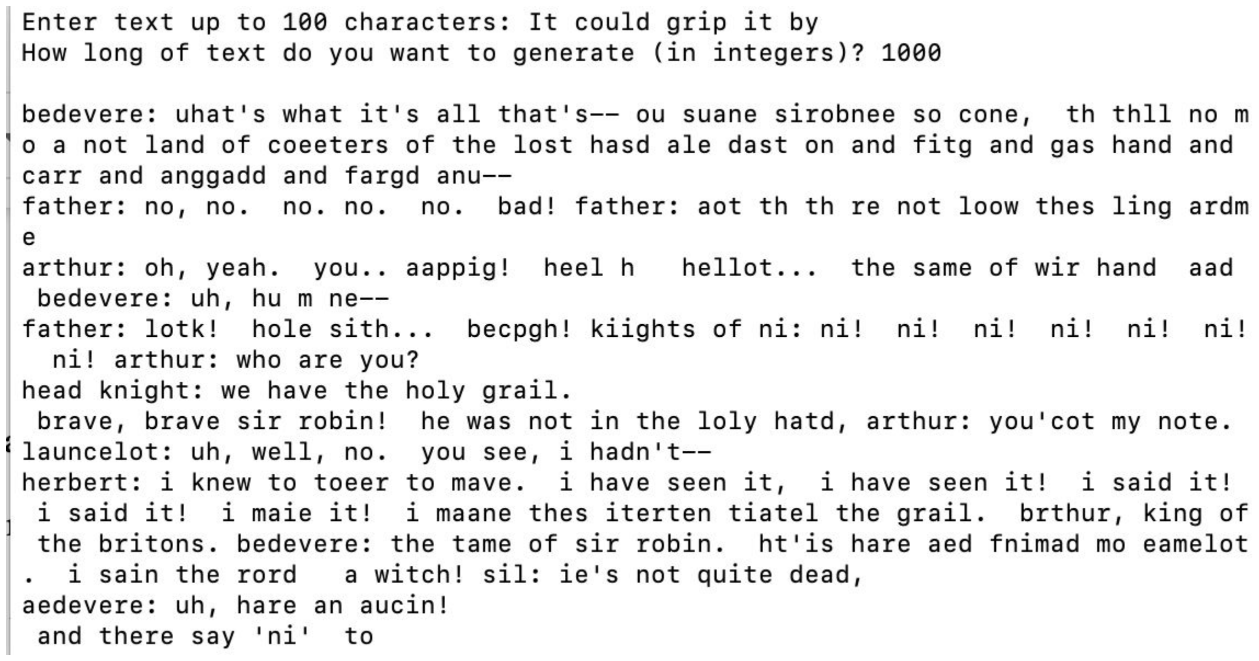
There are many ways in which this project can continue to develop and improve. Overall, our results are satisfactory given the limitations of our project, that mainly being time. There are major limitations and inconsiderations in our topic, as it can be very sophisticated. Some may include the ignorance of the context, sentiment, or even beyond the 5th to last word in a sentence. It did seem that increasing the number of words considered had a negative impact on the prediction as we tried increasing it to 9 previous words. This can be easily changed for curiosity, all that needs to be done is to modify the **wlength** variable in line 18. The bigram performance seemed to be the most lackluster in comparison, with trigrams being respectable when it recognized the inputs. We decided to leave it in since it provided some grab bag responses that were occasionally humorous. Some areas of improvement if given more time is perhaps using the entire english dictionary for our encoder, but this would involve batch processing, in which we did not have enough time or experience to implement. We could've also batch processed the text8 dataset, which is the Wikipedia database and about 11 gb in size. This would ensure more accuracy, but it would also take ages to complete since it seems like PyCharm doesn't natively support GPU acceleration. On top of that, in our research we found that Hidden Markov Models could be beneficiary in combination to our LSTM model, however we couldn't figure out the implementation in time. These areas would be the immediate improvements we would attempt. And although the end goal was drastically different from our initial plan, which was to just try a bunch of models and choose which word predictions overlapped, the results were still respectable and the implementation went through relatively smoothly.

One constraint on developing an RNN is time. To improve accuracy, more epochs should be undertaken during training utilizing a larger text/corpus. Currently for the character text generator, the model was trained over 50 epochs on a relatively short text, the script for Monty Python's Holy Grail that we have used multiple times in class. This text, with only around 65000 characters, still required upwards of 5 hours to train, and because of the formatting of the training data, it produces some results that can be odd, depending on what input the user gives. Whether that is because of the odd format of being a script, the fact that it is humorous and thus has many unexpected words, phrases, and other things that can affect prediction negatively, or simply because our text was too narrow in its topic remains unclear. A model that could give some humorous yet sensible results like this:



```
[illegible]
Enter text of up to 100 characters: Hello
hood ead,  oeh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh
!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!
 shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  s
hh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh
!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!
 shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  s
hh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh
!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!
 shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  s
hh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh
!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!
 shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  shh!  s
hh!  shh!  shh!  shh!  shh!  shh!  shh!
[illegible]
```

also yielded nonsensical babbling like this, even when the "prompt" given was a direct quote from the text itself:



```
Enter text up to 100 characters: It could grip it by
How long of text do you want to generate (in integers)? 1000

bedevere: uhat's what it's all that's-- ou suane sirobnee so cone,  th thll no m
o a not land of coeeters of the lost hasd ale dast on and fitg and gas hand and
carr and anggadd and fargd anu--
father: no, no.  no. no.  no.  bad! father: aot th th re not loow thes ling ardm
e
arthur: oh, yeah.  you.. aappig!  heel h   hellot...  the same of wir hand  aad
 bedevere: uh, hu m ne--
father: lotk!  hole sith...  becpgh! kiights of ni: ni!  ni!  ni!  ni!  ni!  ni!
  ni! arthur: who are you?
head knight: we have the holy grail.
 brave, brave sir robin!  he was not in the loly hatd, arthur: you'cot my note.
launcelot: uh, well, no.  you see, i hadn't--
herbert: i knew to toeer to mave.  i have seen it,  i have seen it!  i said it!
 i said it!  i maie it!  i maane thes iterten tiatel the grail.  brthur, king of
 the britons. bedevere: the tame of sir robin.  ht'is hare aed fnimad mo eamelot
.  i sain the rord   a witch! sil: ie's not quite dead,
aedevere: uh, hare an aucin!
 and there say 'ni'  to
```

It is important to note that even in this “babble”, there are still some well-formed words created. By retraining the model on a more representative corpus, the model should become more accurate and more likely to produce satisfactory results.