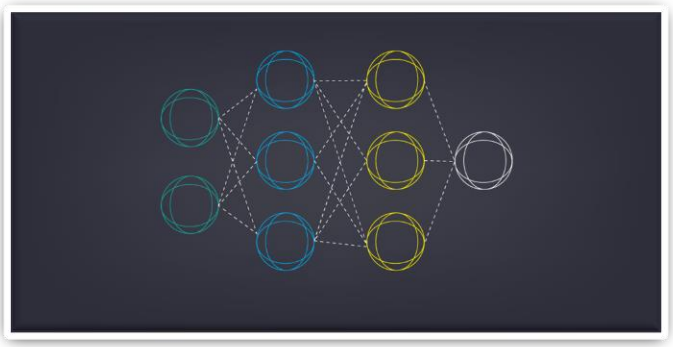

[redacted]
COMP316

LANGUAGE MODELLING

Using LSTM models for the process of Text Generation

[redacted] [redacted]

Contents


Introduction

The purpose of this project was to build a language model that would be able to develop fluent and coherent English sentences. Language modelling (LM) is the use of various statistical and probabilistic techniques to determine the probability of a given sequence of words occurring in a sentence. Language models analyse bodies of text data to provide a basis for their word predictions. An LSTM model is being used in this project.

Language Modelling is a task that is of paramount importance in modern NLP tasks. Frequent applications of language modelling include Speech Recognition, Machine Translation, Part Of Speech Tagging, Parsing, Sentiment Analysis, Optical Character Recognition, and Information Retrieval.

The project was broken up into three main process steps: pre-processing of the data in a way that would be optimal for the model to learn on, creating and training the model, and testing the model output.

Area of Application

Problem

Language Modelling is essential to a vast array of Natural Language Processing tasks that translate to real-world applications – from the many functions provided by Google to the voice assistants found in most modern phones, tablets, and computers, to the chatbots that are increasingly present in most websites one visits today. Without good language models, these tasks will fail.

The problem arises in the fact that natural languages are irregular, with many convoluted rules, which have many more exceptions. Language is fluid and ever-changing. As such, natural languages cannot easily be mapped into a formal language representation.

Solution and Goals

There are two main pathways to take when doing a language model: Statistical Language Modelling or Neural Language Modelling. Statistical is the more classical approach that makes use of techniques such as n-grams or parse trees. The neural modelling approach makes use of a neural network, which is based on a collection of connected units or nodes called artificial neurons, which loosely model the neurons in a biological brain. The solution that was chosen is a neural language model that makes use of a type of Recurrent Neural Network (RNN) architecture known as Long Short-Term Memory (LSTM). This was because neural networks have become the preferred approach for language modelling, and have been shown to achieve better results than classical methods both on standalone language models and when models are incorporated into larger models on challenging tasks like speech recognition and machine translation.

The end result of this project would be a language model that represents the English language well enough that the text sequences generated by the model would be of a good enough fluency so as to fool a human who is interpreting it at a glance.

Choice of NLP Techniques

- Regular Expressions – Regular expression patterns were used as a part of the data pre-processing phase to eliminate undesirable words and tokens. This would probably have been the best and most straightforward method to apply for this task. Alternatively, a pattern-matching library such as *MatchPy* could have been used, or the exception tokens could have been put into a map and removed using explicit hard-coding.
- POS Tagging – Used as a part of the pre-processing phase, to filter out undesirable tokens, and replace them with the same placeholder token. The averaged perceptron tagger was used for this purpose. Other options that could have achieved the same functionality are the use of regular expressions to match and filter out specific undesirable tokens, such as interjections (e.g. *"Oh!"*, *"Ouch!"*). Named-entity recognition could also have been used, especially to specifically identify things such as names of people, locations, et cetera. (As this is not available via *NLTK*, a wrapper class for *StandfordCoreNLP* could have been used). This was chosen as it was simple to use, and the tagger was easily accessed through *NLTK*.
- Neural Language Modelling – The solution of a neural language model was best suited to the task at hand, as it has been shown to give the best output. Other avenues that could have been explored include n-gram language modelling and the use of finite-state automata.

Experimental Setup

Choice of Language and Resources

Python was chosen as the language for this project. Other languages considered included Java, as it is the language with which the author had the most experience. Stanford CoreNLP would have been the language of choice had the decision been made to go with Java. However, further research indicated that Python is more suited to machine learning applications, especially as it allows for more flexibility in terms of development, dataset size, et cetera. Python 3.8.0 was specifically used, as support was not available for later versions in some of the libraries that were used.

The choice of IDE was split between Google Colab, Pycharm, and Visual Studio Code. Pycharm was preferred as it is a powerful IDE with many convenience features that surpass Colab and Visual Studio, such as its autocompletion and error-detection features that execute without the need to compile code. However, Pycharm projects occasionally have given issues when it comes to accessing and downloading libraries. It is for this reason that Visual Studio and Colab were used when smaller segments of code needed to be tested, but development was primarily done in Pycharm.

The libraries and API's that were used are Keras, TensorFlow, Natural Language Toolkit (NLTK), Pickle, Numpy, and Pyttsx3, as follows:

NLTK was the library of choice for NLP applications in Python. Its usefulness was particularly apparent when it came to the process of text pre-processing, where it was used for such tasks as sentence tokenisation.

Keras was used to build the model. This extended to such functions as creating vector representations of the dataset and loading and using the model.

Keras has the option of using two different backends: Theano and Tensorflow. As Theano support and development has ended, TensorFlow was chosen as the backend for Keras.

NumPy was used for its array data type, which is more memory- and time-efficient than standard python arrays or lists.

Pickle was used to save and load the model and tokenised data, as Pickle is used for serialising and de-serialising a Python object structure so that it can be saved on disk.

Other built-in python libraries that were used include the *collections* library, which was used as a part of the data pre-processing, the *random* library, which was used as part of testing, and the *re* library, which was used as part of the pre-processing phase.

Lastly, the Pyttsx3 library was used as a text-to-speech engine to form a part of the evaluation.

Dataset

As this is a task of language modelling, most English texts would be feasible.

The first dataset that was under consideration was Google's One Billion Word Corpus. This would have been advantageous as:

- Google has provided much data that would allow other researchers to replicate results that have been achieved by Google developers in various specific tasks.

- The corpus is of significant size (about 11 GB) which would have given the model much text to train on, which would increase its effectiveness as it would be able to recognise a wider variety of sequences.

However, the main disadvantage of this corpus was its significant size – it took very long to download, occupied much space in memory, and was too large for the data structures that the author was familiar enough with to use.

The corpus that was eventually decided to use was texts from the Project Gutenberg corpus. The main advantages of this are:

- It is available within NLTK, which was already decided to be used for the project
- It is of a feasible size.
- It is easier to process for the scale of this application.

The disadvantage here is that the data obtained may not be representative of all language and grammar constructs, especially as owing to copyright issues, Project Gutenberg does not have any relatively recent texts.

A data set that was considered but ignored was the Web and Chat Text. It was dismissed as much of it ended up being fairly nonsensical, filled with profanity, or in some other way not appropriate.

Implementation

Obtaining of Data and Data Pre-Processing

The first step was obtaining the dataset. As stated, this was available to download via NLTK (please note: *nltk.downlaod('gutenberg')* is commented as it only needs to be executed once in total, after which a local copy is available to the project). For the sake of ensuring consistency across things such as sentence structure, the decision was made to include only texts written by Jane Austen – specifically *Sense and Sensibility*, Emma and *Persuasion*.

The first line of each text, containing the title, author, and year of publication, was removed, and the text was appended to the variable *text*, which represents the corpus.

```
def load_text():
    # nltk.download('gutenberg')
    text = ""
    temp = nltk.corpus.gutenberg.raw('austen-sense.txt')
    temp = temp[temp.find("\n"):]
    text += temp

    temp = nltk.corpus.gutenberg.raw('austen-emma.txt')
    temp = temp[temp.find("\n"):]
    text += temp

    temp = nltk.corpus.gutenberg.raw('austen-persuasion.txt')
    temp = temp[temp.find("\n"):]
    text += temp

    return text
```

Some information on the text was then retrieved – namely, the average length of sentences in the text. This value was determined to be approximately 27 (see figure). Initially, only sentences within about 20 words in length to this average value were going to be part of the final, cleaned text. However, upon closer inspection, including sentences up to even as long as 100 words decreased the corpus size by over 80% - indicating that there were likely very many long and very many short sentences. The value was instead used solely to determine sequence length for input/output to the model.

```
def someInfo(text):
    from nltk.tokenize import sent_tokenize, word_tokenize
    # nltk.download('punkt')
    tmp1 = sent_tokenize(text)
    tmp2 = len(word_tokenize(text))
    print("Average sentence length :\n", tmp2 / len(tmp1))
```

```
Average sentence length :
 26.52304292929293
Total Sequences: 440176
```

The text then had to go through some cleaning. This was broken into two parts, viz more straightforward cleaning (that made use of primarily regular expressions to filter out such things as punctuation, extra whitespace, and chapter titles), and a slightly more involved cleaning process. Concise sentences (less than ten words) were also removed at this stage, prior to the text being stripped of punctuation.

```
def clean(text):
    #   return 0
    #  text = re.sub(r"\d", " ", text)
    #  text = re.sub(r'[\[\'\"\]\:{\}]', " ", text )
    text = text.lower()
    sents = [sen for sen in sent_tokenize(text) if 10 < len(sen)]

    text = re.sub(r"CHAPTER \d+", "", text)
    text = re.sub(r"[^A-za-z]+", " ", text)
    text = re.sub(r"/s+", " ", text)
    return text
```

The second part involves the removal of words that meet specific criteria. Firstly, the Part-Of-Speech tagger provided by *NLTK* (the *averaged perceptron tagger*) was used to assign part of speech tags to the words in the text. This was for the purpose of filtering out foreign words (*"FW"*), proper nouns (*"NNP"* and *"NNPS"*) and interjections(*"UH"*).

Proper nouns were written to a text file *"ProperNouns.txt"* for later use.

Lastly, words that occurred very infrequently in the text (less than five times) were removed. The counts of words were obtained using the *Counter* class made available by the *collections* module.

```
def remove_some_words(text):
    # nltk.download('averaged_perceptron_tagger')#nltk.download()
    tagged_sentence = nltk.tag.pos_tag(text.split())
    edited_sentence = []
    proper_nouns = []
    for word, tag in tagged_sentence:
        if tag != 'NNP' and tag != 'NNPS' and tag != 'FW' and tag != 'UH':
            edited_sentence.append(word)
        elif tag == 'NNP' and tag == 'NNPS':
            edited_sentence.append("PROPERNOUN")
            proper_nouns.append(word)

    write_doc(proper_nouns, "ProperNouns.txt")

    sen = " ".join(edited_sentence)

    lst = sen.split()
    counted = Counter(lst);
    sen = ""
    for el in lst:
        if counted[el] > 5:
            sen += ' ' + el

    return sen
```

In the last step, the cleaned text was broken into sequences of length 27 + 1 (27 input words in the sentence, plus one output word for the model to learn). This was done by converting the corpus into a list of words and then iteratively finding all consecutive sequences of words of the desired length.

These sequences were then saved to the textfile *"cleaned_sequences.txt"*.

```
# organize into sequences of tokens
length = 27 + 1
sequences = list()
tokens = text.split()
for i in range(length, len(tokens)):
    # select sequence of tokens
    seq = tokens[i - length:i]
    # convert into a line
    line = ' '.join(seq)
    # store
    sequences.append(line)
print('Total Sequences: %d' % len(sequences))
```

Optimisation and Training

The cleaned sequences were loaded from the textfile *"cleaned_sequences.txt"* into the list *lines*. The Tokenizer() class found in Keras is a class for vectorising texts, or/and turning texts into sequences. The new sequences consisting of the now-encoded words is stored in the variable *sequences*.

```
in_filename = 'cleaned_sequences.txt'
doc = read_file(in_filename)
lines = doc.split('\n')

# integer encode sequences of words
tokenizer = Tokenizer()
tokenizer.fit_on_texts(lines)
sequences = tokenizer.texts_to_sequences(lines)
# vocabulary size
vocab_size = len(tokenizer.word_index) + 1
```

The variable *sequences* were then converted into a NumPy array. From this, the x values (the first part of the sequences, without the next "output" word) and y values (the predicted words) were extracted. Y was then assigned the value returned by the *to_categorical()* method, which converts a class vector (integers) to a binary class matrix, e.g. for use with categorical cross-entropy, which is the type of loss that will be used.

```
# separate into input and output
sequences = array(sequences)
X, y = sequences[:, :-1], sequences[:, -1]
y = to_categorical(y, num_classes=vocab_size)
seq_length = X.shape[1]
```

The model was then built. It is an LSTM network consisting of an initial Embedding layer with a dimension of 200, LSTM layers with 100 units each, and a Dense layer with 100 units. The Embedding layer is the first layer as it provides a dense representation of words, which enables us to compress the input feature space. After some experimentation, an Embedding layer of dimension 200 gave the best results. As this is not a very complex project, it was decided only to use two LSTM layers.

The loss type of “categorical cross-entropy” was used as it is the one that is best suited to multi-class classification problems. Other loss options for multi-class classification are sparse-multiclass cross-entropy loss and Kullback Liebler Divergence loss. The optimiser that was chosen is the adam optimiser, as it is generally accepted as one of the best optimisers. Other optimiser options are SGD(Stochastic Gradient Descent), RMSProp, AdaDelta, AdaGrad, and Nesterov accelerated gradient.

A summary of the model is printed before it is trained.

```
model = Sequential()
model.add(Embedding(vocab_size, 200, input_length=seq_length))
model.add(LSTM(100, return_sequences=True))
model.add(LSTM(100))
model.add(Dense(100, activation='relu'))
model.add(Dense(vocab_size, activation='softmax'))
print(model.summary())
# compile model
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit model
model.fit(X, y, batch_size=128, epochs=100)
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 27, 200)           661600
_________________________________________________________________
lstm (LSTM)                  (None, 27, 100)           120400
_________________________________________________________________
lstm_1 (LSTM)                (None, 100)               80400
_________________________________________________________________
dense (Dense)                (None, 100)               10100
_________________________________________________________________
dense_1 (Dense)              (None, 3308)              334108
=================================================================
Total params: 1,206,608
Trainable params: 1,206,608
Non-trainable params: 0
_________________________________________________________________
None
```

After training the model, it was saved to the file *"model.h5"*. The tokeniser that was computed earlier is also saved, to a file *"tokeniser.pkl"* using the *dump()* function of *Pickle*. Both will be used in the evaluation. Another library that could have been used for this purpose is *Joblib*.

```
# save the model to file
model.save('model.h5')
# save the tokenizer
dump(tokenizer, open('tokenizer.pkl', 'wb'))
```

Evaluation

Overview

The main evaluation of the model was done as a Turing test. The volunteers were given some information as to the structure of the data and the testing session, and then repeatedly listened to a sentence fragment. The volunteer was tasked with indicating whether they thought the sentence fragment they had heard was generated by a machine or was a text written by a human being. Their answer was recorded, and they were fed the next sentence prompt until they terminated the session. Their results were then collated with prior results and interpreted.

Method

The following function is used to generate a sequence of words from the model. It uses the parameter *seed_text* as the initial seed text to predict the next word (copied to the variable *in_text*). *seed_text* is of length *n_words*. As the model predicts new words, it is appended to the variable *in_text*. It is also stored/ appended in the list variable *result*. Predicted words are appended to the

variable *in_text* so that the next predicted word will be based both off the seed text plus all previously generated words.

```
# generate a sequence from a language model
def generate_seq(model, tokenizer, seq_length, seed_text, n_words):
    result = list()
    in_text = seed_text
    # generate a fixed number of words
    for _ in range(n_words):
        # encode the text as integer
        encoded = tokenizer.texts_to_sequences([in_text])[0]
        # truncate sequences to a fixed length
        encoded = pad_sequences([encoded], maxlen=seq_length, truncating='pre')
        # predict probabilities for each word
        yhat = model.predict_classes(encoded, verbose=0)
        # map predicted word index to word
        out_word = ''
        for word, index in tokenizer.word_index.items():
            if index == yhat:
                out_word = word
                break
        # append to input
        in_text += ' ' + out_word
        result.append(out_word)
    return ' '.join(result)
```

The cleaned sequences that were saved in the first stage are loaded and stored in the variable *lines*. They will be used both to generate sequences by the model and as part of the testing.

```
# load cleaned text sequences
in_filename = 'cleaned_sequences.txt'
doc = read_file(in_filename)
lines = doc.split('\n')
seq_length = len(lines[0].split()) - 1
```

The model and tokeniser, that were saved in the previous phase, are loaded from their respective files

```
# load the model
model = load_model('model.h5')

# load the tokenizer
tokenizer = load(open('tokenizer.pkl', 'rb'))
```

The Pyttsx3 library is used to initialise a text to speech engine, called *engine*. The speaking rate of *engine* is set to 120 words per minute. This was obtained after some experimentation, as it is not too fast that it is impossible to get a sense of what is heard, but it is not so slow that it may be irritating after repeated testing. The default speech rate is 200 words per minute. Pyttsx3 was chosen as it is

easy to use, and can be used online – which is especially an advantage given the current pandemic. The other text-to-speech option that was considered is *gTTS*.

```
engine = pyttsx3.init()
# slow down speaking rate
engine.setProperty("rate", 120)
```

The variables that will be used to record the results of the human testers.

humanTrue represents the times the tester correctly guessed the text to be human-generated

humanFalse represents the times the tester incorrectly guessed the text to be human-generated

machineTrue is the number of times the human tester correctly guessed the text to be machine-generated

machineFalse is the number of times the human tester incorrectly guessed the text to be machine-generated

These values are stored/retrieved from the textfile *"Evaluation.txt"* for each run.

```
saved_Values = read_file("Evaluation.txt").split("\n")

humanTrue = int(saved_Values[0])
humanFalse = int(saved_Values[1])
machineTrue = int(saved_Values[2])
machineFalse = int(saved_Values[3])
```

The human tester is shown the following message, following which the string *text_to_speak* is fed to *engine* and is spoken. Depending on the human tester's input, the corresponding value is updated.

```
Hello there
A random sequence of words will be played for you.
Please indicated whether you think it could have been written by a human or a machine.
Please note: Punctuation, Proper Nouns, Interjections, etc. will not be present in either machine or human generated sequences.
Also note that these are merely fragments, not whole sentences
This may affect fluency, but not greatly.
Press H for human, M for machine, R to hear it again, and X to leave
h
```

The following loop is run until the tester terminates it. It randomly gets a line of human-generated text and generates a sequence of machine-generated words. It then randomly decided whether to use the machine-generated or human-generated text and feed it to the variable *engine*.

It then waits for input and displays a message or updates the relevant value accordingly. The loop is terminated whenever the tester enters the string "X".

```
while True:

    rand_test = randint(0, len(lines))
    seed_text = lines[rand_test]
    generated = generate_seq(model, tokenizer, seq_length, seed_text, 27)
    flag = rand_test % 2 == 0
    if (flag):
        text_to_speak = generated
    else:
        text_to_speak = seed_text
    text_to_speak = sub_noun(text_to_speak)
    engine.say(text_to_speak)
    engine.runAndWait()
    print("Press H for human, M for machine, R to hear it again, and X to leave")
    in_char = input().strip().upper()
    if in_char=="R":
        while in_char == "R":
            engine.say(text_to_speak)
            engine.runAndWait()
            in_char = input()

    if in_char=="H":
        if flag:
            humanFalse += 1

        else:
            humanTrue += 1
    elif in_char == "M":
        if flag:
            machineTrue += 1
        else:
            machineFalse += 1
    elif in_char == "X":
        break
    else:
        print("Sorry, you entered an incorrect key")
```

Sample output for the loop:

```
Press H for human, M for machine, R to hear it again, and X to leave
r
Press H for human, M for machine, R to hear it again, and X to leave
m
Press H for human, M for machine, R to hear it again, and X to leave
m
```

After the loop terminates, the *"Evaluation.txt"* file is updated.

To determine the result of this experiment, a confusion matrix is drawn up:

```
# confusion matrix
total = humanFalse+ humanTrue+machineTrue+machineFalse
accuracy = (humanTrue + machineTrue_) / total
humanTrue /= total
humanFalse /= total
machineTrue /= total
machineFalse /= total
print("\tH", "M", sep="\t\t")
print("H\t%.2f\t.%2f"%(_humanTrue, humanFalse))
print("M\t%.2f\t%.2f"%(machineFalse, machineTrue))
print("Accuracy = ", accuracy)
```

The confusion matrix at the latest run:

```
    H       M
H   0.23    .0.150641
M   0.27    0.35
Accuracy =  0.5769230769230769
```

This could have been done using a convenience method available via an API (e.g. using *sklearn.metrics.confusion_matrix()* from *SciKit Learn*). As the above implementation contains only four categories, the benefits of speed and not needing to download, load and initialise an additional library function outweighed the cost of hard-coding the confusion matrix.

Please note this measures the accuracy of the human tester, and not of the model. We can assume that if the human tester had a low accuracy, then the model was very good at creating a language model.

Strengths and Shortcomings

The model was fairly well able to predict and produce text sequences. The decision to use a text-to-speech engine for the human tester to evaluate the output was done based off the intuition that human beings generally are better at detecting poorly formed sentences verbally than they might be when faced with written text. It also relied upon the instantaneous decision of the person rather than presenting them with row upon row of written text, which may have faster led to fatigue. It had achieved a reasonably good in-training accuracy and was able to generate word sequences of a good enough quality that it was able to “fool” the human tester a fair number of times.

The model could have been improved with repeated tweaking and experimenting, such as with the dimension of layers, in particular the Embedding layer, in experimenting with the number of and type of layers, in particular exploring the worth of a Dropout layer for this specific application.

In terms of the evaluation, the sample size of test subjects was quite small, as, due to COVID restrictions, the process of teaching other potential testers how to install and run the things they would need in order to be able to run the test code was very difficult to impossible. Due to this reason, the author also had to be a participant, and this could have skewed the data, as the author was familiar enough by that point with both the training data and the quality of data a model would be able to produce that the author might have been subconsciously able to tell the difference between the two, in a way that most other individuals would not have, i.e. resulting in higher accuracy.

The reliability of the results is also decreased due to the time allotted for testing possibly being insufficient.

Bibliography

(n.d.). Retrieved from https://faroit.com/keras-docs/1.2.2/preprocessing/text/

(n.d.). Retrieved from https://keras.io/api/utils/python_utils/#to_categorical-function

28/09/2017. (2017, August 28). Retrieved from https://groups.google.com/forum/#!msg/theano-users/7Poq8BZutbY/rNCIfvAEAwAJ

Artificial neural network. (2020, July 31). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Artificial_neural_network

Basnet, B. (2016, November 18). *LSTM Optimizer Choice ?* Retrieved from DATA SCIENCE & DEEP LEARNING: https://deepdatascience.wordpress.com/2016/11/18/which-lstm-optimizer-to-use/

Bogan, E. (n.d.). *What is Pickle in python?* Retrieved from https://morioh.com/p/ca01f7fcf8bc

Brownlee, J. (2019, August 7). *Gentle Introduction to Statistical Language Modeling and Neural Language Models*. Retrieved from Machine Learning Mastery: https://machinelearningmastery.com/statistical-language-modeling-and-neural-language-models/

Brownlee, J. (2020, April 24). *How to Choose Loss Functions When Training Deep Learning Neural Networks*. Retrieved from Machine Learning Mastery: https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/

Rouse, M. (2020, March). *language modeling*. Retrieved from https://searchenterpriseai.techtarget.com/definition/language-modeling