Summary of changes made to the original report

Note: *The reviews of the instructor and peers were carefully reviewed and the following changes were made.*

- Changed the aspect ratio of images and plots used.
- Added captions and numbered each image and plot and referenced them while describing in the discussion.
- Added description of the dataset, preprocessing and other feature engineering processes.
- More details on the Markov model and GPT-2 have been added under the respective topics.
- More relevant examples and outputs have been added for clarity and for understanding of how the models performed and removed some jargons.
- A detailed analysis of perplexity and human level analysis and discussion has been added for all the models and the problems and solutions have been identified and discussed.
- Font mismatch has been adjusted and removed unnecessary sentences making the analysis precise.
- Huge sequence of sentences have been corrected and broken down into readable paragraphs.
- Formats have been corrected making all text and image data aligned and they all maintain a uniform flow.
- Added references towards the end of the report.

Final Report

Title

Generating Dwight Schrute Quotes using Markov model, LSTM and Attention based (GPT-2) Models

Members

Aldo Pioline Antony Charles, Gowtam Potluri

Introduction

In this project, we focus on building Machine learning and deep learning-based language models to perform text generation using concepts such as style transfer. Text generation inherently requires processes such as text abstraction, semantic analysis, etc for which we started with a simple model like Markov chain and then used powerful models like LSTM, GPT2 and analyzed the performance under each, thereby developing a text generation engine. More specifically, the project focuses on training a model on some text that shares common interests and using the trained features to create similar textual information and employing those features onto other text data using style transfer. With the rise of digital assistants, search engines, translation software such as google translate, the need for language models and algorithms that use such models to provide a solution is vital. Creating a language model is a difficult problem, but thanks to decades of research, we have many solid models and approaches in the field of Natural Language Processing and shared ideas from other fields of science, linguistics, we have a lot of ideas upon which powerful algorithms can be built. One such problem that we are focusing on for this project is the idea of analyzing and understanding a group of texts and generating new textual data based on the commonalities and understanding of the given(trained) text. So, for our development, we are focusing on one of the most culturally relevant sitcom character Dwight Schrute from the famous TV show, "*The Office (USA)",* where we have a list of all of Dwight's quotes and some metadata such as the emotion and using these to train a deep learning model and thereby generate new quotes that go along Dwight's style.

We first preprocess the textual data, perform exploratory analysis, and some feature engineering to make the dataset more relevant for our use case. Looking at the problem itself, there are multiple ways to create a solution(i.e.) We can use various Deep learning models such as LSTM, GPT2 etc. to understand the semantic relationship between words and to learn other features. Each of the models has its pros and pitfalls. So, the project also shows how each model performs for the given task. Then after choosing the right architecture, we focus on tweaking some of the layers and playing around to see the variation in result accuracy and performance. We also focus on improving the optimality of the algorithm itself because models like these take tremendous amounts of time and resources to train and work with, so an optimal algorithm is highly essential. Then we extend our findings to other styles of text data and we also focus on transferring one style of text into another text (similar to image style transfer) and using different models to contrast and compare the result accuracy and performance.

Dataset and Preprocessing

The dataset we have is a collection of 6900 rows of Dwight Schrute quotes collected from 9 seasons in a text file. The dataset also has annotations associated with the quotes to describe the context. To perform various pre-processing steps, we have converted the text file into a structured CSV file. We have removed all the whitespaces and forbidden characters indexing through all the quotes. Using the NLTK module, we have tokenized the sentences into words. We have converted all the words into lowercase and removed any alphanumeric characters. We have also removed the annotations present in each sentence to avoid model

learning unnecessary patterns. We constructed a Data frame with the number of words, number of unique words, number of stop words in each sentence.

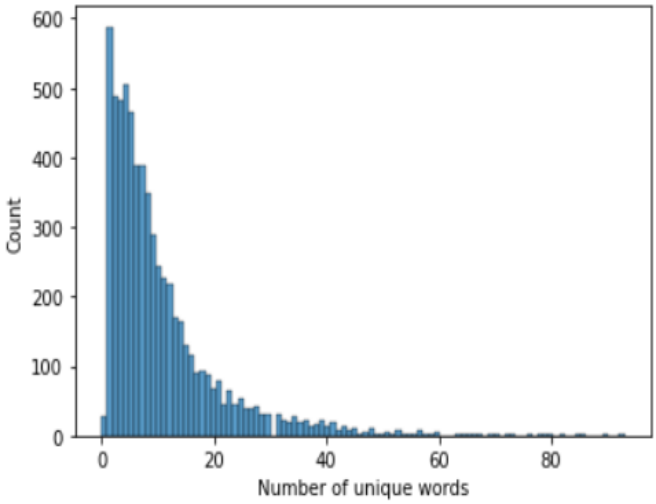


FIGURE 1: *Frequency of unique words present in each quote.*

From figure 1, we can see the histogram of the number of unique words. We can see that maximum sentences have close to 10 words in a sentence. This can be helpful to later compare the number of unique words in the sentences generated by the model. We have also created a word cloud to understand the most common words in the dataset.

FIGURE 2: *Word Cloud of the most used words in the dataset*

Using the VADER algorithm, we constructed a dictionary to map the emotions and their intensities. This intensity is captured using the compound score. A compound score of 0 indicates a neutral opinion while a compound score of greater than or equal to 0.05 indicates a positive opinion and the rest as negative.

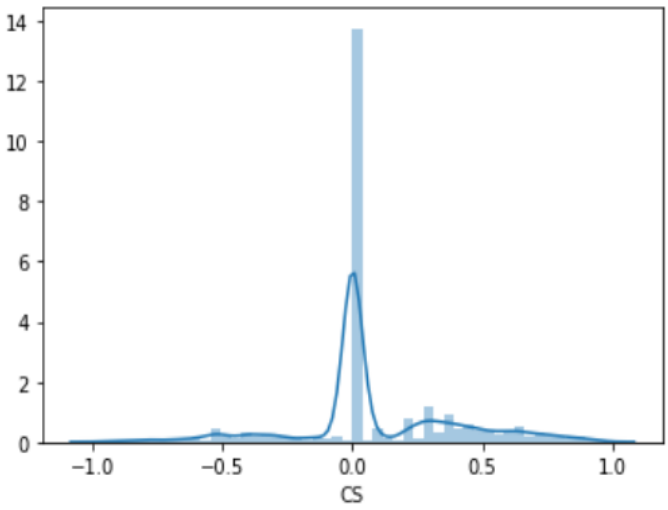


FIGURE 3: *VADER algorithm capturing the sentiment.*

As we can see from figure 3, most of the quotes are neutral. This can later be compared with the model generated quotes to see if the model properly captures the context and sentiment.

Methods

To solve the problem, we wanted to start with a simple Markov model which can be easily interpreted. A Markov chain is a stochastic process that models a finite set of states, with fixed conditional probabilities of jumping from a given state to another.

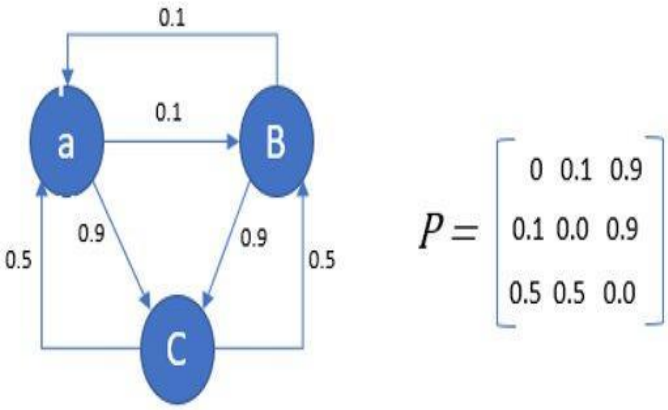


FIGURE 4: *Markov chain state transition and transition matrix*

- [“Once”, “upon”, “a”] -> [“time”]
- [“upon”, “a”, “time”] -> [“there”]

FIGURE 5: *Markov chain unigram prediction sample*

The Markov chain looks like a graph as shown in figure 4. Each node is a state and some probability assigned to the edge connecting two states. We can now express the probability of going from state a to state b as a matrix component as shown in Figure 5. To generate text with Markov Chains, we must be able to decide what our states are and how we can assign probabilities for the edges. In terms of python, we can think of this like creating a dictionary of every unique word in our corpus. The values would be the list of

words that appear after each of the unique word. The more often a word appears after another, the higher the probability that it will be selected in the text generator. We begin by generating n-grams of either 1,2 or 3 n-gram sequences and creating probability matrix values for each generated n-grams. We estimate the likelihood for each n-gram against the previous state value and generate the next state value. The problem with this is that Markov chain does not understand the context but rather only considers an n-gram at any point of time and generates the next subsequent text which is not ideal for high level text generation. We can also adjust the probability distribution temperature or diversity and corresponding text values are generated. Markov models are fast but to generate more complex texts, we need memory-based models which are described next.

Markov models are simple to implement so we split the data into our ngrams of interest and construct the transition matrix using Scipy's coo_matrix for each n gram probabilities. We then pick a random seed and sample the values from the transition matrix based on the higher values and generate the sentence. Since Markov model considers only the current ngram to generate the next sequence, the overall meaning of the sentence might be lost and that is a major flaw of this method (I.e.) The lack of memory problem.

After trying out the Markov model, we also wanted to consider a model which gives significant importance to the order of inputs. Unlike a feed-forward network which tries to optimize the input-output function without any concern to the order of inputs, we want to consider an architecture of Recurrent neural networks(RNN) called LSTM networks which would be ideal in the case of text processing where there is a significant amount of information embedded in the word order.

RNN is a type of sequence-based neural network which has memory units to remember past information. Since the output of each cell is given back to the same cell as input it helps build memory about the positional information of the words in the given input sequence.

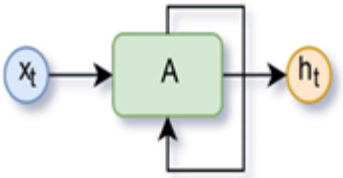


Figure 6: RNN Cell with a loop

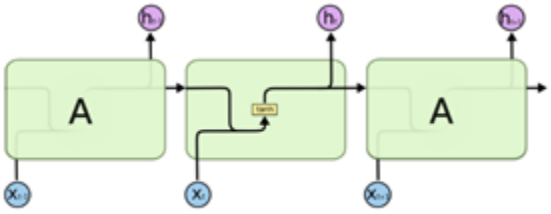


Figure 7 The repeating module in a standard RNN contains a single layer

However, RNN suffers from a vanishing gradient problem for long sequences of input and hence we considered LSTM. LSTM would retain long-term information by propagating only the relevant information back. LSTM has a modified hidden layer that contains the input gate, forget gate, and output gate. These help to determine which information from the past is relevant. Due to such a nature, LSTMs can preserve history for long input sequences.

The key to LSTMs is the cell state, which is represented by the horizontal line show in Figure 8.

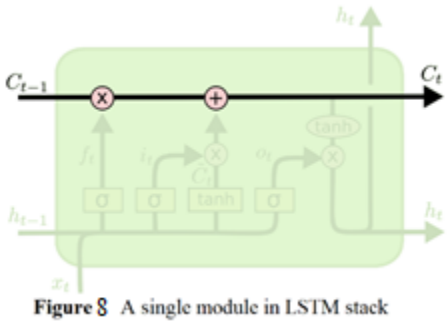
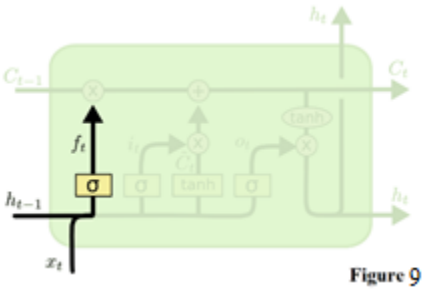


Figure 8 A single module in LSTM stack



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Figure 9 Forget gate layer

As we discussed earlier, the first step in LSTM is to decide what information is important to retain. This decision is made by a sigmoid layer called the "Forget gate Layer". Using the input h_{t-1} and x_t, the sigmoid function returns a number between 0 and 1. This helps us decide to either store or discard. The next step would be to decide what new information we must store in the cell state. This is divided into two parts, first, a sigmoid layer(input gate layer) which decides which values to be updated. Second, a tanh layer that creates a vector(C_t).

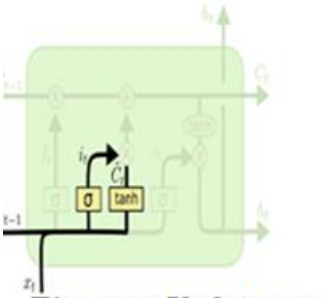
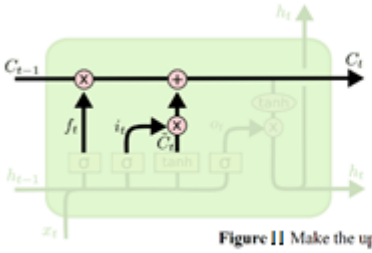


Figure 10 Update new information

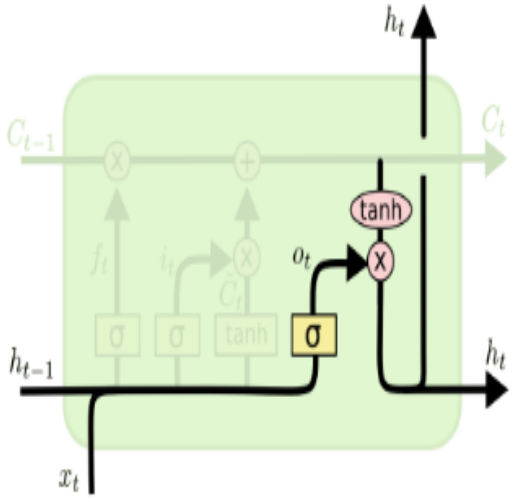


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Figure 11 Make the updates calculated in previous steps

Using these outputs, we can update the old cell state(C_{t-1}) by multiplying the old state by f_t and then adding $i_t * C_t$

Finally, we can decide what the output should be based on our cell state. First, we would run a sigmoid layer to decide what parts of the cell state we are going to output. The cell state would then be put through tang and then multiply by the output of the sigmoid gate.



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

FIGURE 12: *Output the Information*

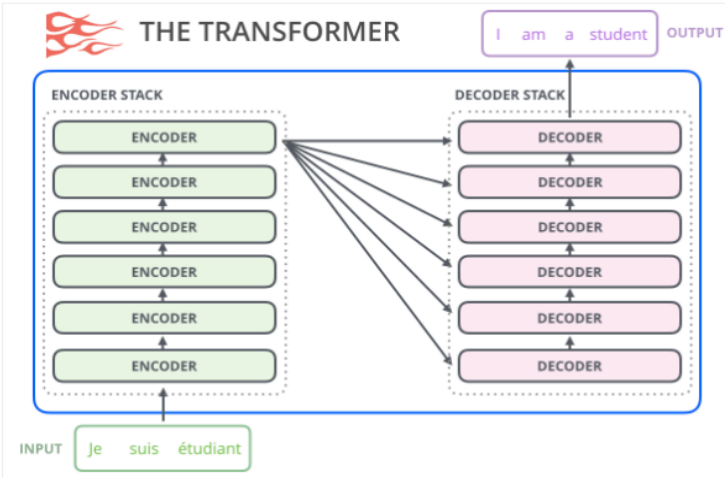
There are two types of text generators, character level text generator and word level text generator. We have chosen a character level text generator because the vocabulary of the model would be very small and hence would require less memory and can infer faster compared to word level text generator. The main aim of these models is to learn the dependencies between the characters and the conditional probabilities of characters in sequence so that we can generate new and original sequences of characters. To prepare the data for modeling, we have converted all the characters in the dataset to lowercase and applied set function to get the unique characters to reduce the vocabulary that network needs to learn and then vectorized the data by creating a map of each character to a unique integer creating a vocabulary size of 90. We have defined our training dataset by splitting the dataset into subsequences with a fixed length of 15 characters. Each training pattern of the network consists of 15-time steps of one character (X) followed by one character output(Y). On these created sequences, we slide along the whole book one character at a time, allowing each character a chance to be learned from the 15 characters that preceded it. Using this method, we have generated 826718 sequences. We have transformed these lists of input sequences into the form [*samples, time steps, features]* expected by an LSTM network. We have also re-scaled the integers between 0-1 to make the patterns easier to learn by the LSTM model which uses sigmoid activation function by default. After training the model, we have converted the output patterns into a one hot encoding. This is so that we can configure the network to predict the probability of each of the 90 different characters in our vocabulary rather than forcing it to precisely select the next character. Below we have discussed the architecture of our Keras model. Since, the problem is a single character classification problem with 90 classes, we have a softmax as our last layer and chose ADAM as out optimizer.

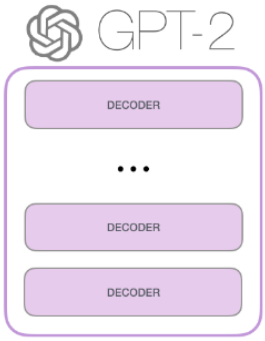
```
Build model...
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_sequences (InputLayer) [(None, 15, 90)]          0
_________________________________________________________________
bidirectional (Bidirectional (None, 15, 512)           710656
_________________________________________________________________
dropout_bidirectional_lstm ( (None, 15, 512)           0
_________________________________________________________________
lstm (LSTM)                  (None, 64)                147712
_________________________________________________________________
drop_out_lstm (Dropout)      (None, 64)                0
_________________________________________________________________
first_dense (Dense)          (None, 1350)              87750
_________________________________________________________________
drop_out_first_dense (Dropou (None, 1350)              0
_________________________________________________________________
second_dense (Dense)         (None, 450)               607950
_________________________________________________________________
drop_out_second_dense (Dropo (None, 450)               0
_________________________________________________________________
last_dense (Dense)           (None, 90)                40590
_________________________________________________________________
activation (Activation)      (None, 90)                0
=================================================================
Total params: 1,594,658
Trainable params: 1,594,658
Non-trainable params: 0
```

FIGURE 13: *Model Architecture*

Due to the sequential nature of the LSTMs, it would require more memory to train and takes a longer time to train. Hence, we also want to use Attention-based models like Transformer and compare the performance. The Transformer contains an Encoder and a decoder stack. Unlike LSTMs, Transformers take in the

complete input sequence at once. Stacking these Encoder and Decoder blocks, many architectures have raised one of them is GPT-2. GPT-2 is build using transformer decoder blocks. GPT-2 is a pre-trained language model that can be used for various NLP tasks like text generation, data summarization, and translation. The GPT-2 architecture is based on the Transformer which provides a mechanism based on encoder-decoder to detect input-output dependencies. At every stage, the model takes the previously generated data as additional input when generating the next output. OpenAI has 4 versions of GPT-2 based on the parameters to train. We have set up a Google Colab notebook to utilize the GPU processing power. To preprocess our dataset, download the model, and to finetune it, we have used a module called gpt-2-simple. Using the download_gpt2(), we have downloaded the 117M version. We have fine tuned our model by setting the hyperparameters steps=1000, save_every=200 in the finetune(). We have loaded the checkpoints and used the generate() to generate new quotes.





Results

After performing the pre-processing steps discussed earlier, we have generated text using our Markov Model.

SEED: to sell
to sell more more more paper .

SEED: Welcome to
Welcome to the the Schrute the Schrute

SEED:she is ,
she is Caucasian Indian Caucasian brownish

SEED: crack dealing , gang
crack dealing , gang violence

SEED: Im (unigram)
Im gonna being good obviously glad always gonna not two gonna the Assistant just the assistant coming having not gonna somewhere relaxing the a suspicious gonna kidding officially on talking a no sorry like asking

SEED: Im gonna (Bigram)
Im gonna . live do need vomit have count do prevent randomly put intimidate take stay bring Oops find answer . need win have bring do prevent put have do do prove take randomly have count have live prevent do with tell tell . bring need set have live do need say

SEED: Im going to (Trigram)
Im going to engrave handle need do run ask be find pass handle use handle use leave need leave

FIGURE 14: *Markov model text generation samples*

From the above text we can see that Markov model has done a decent job capturing Dwight's personality. When we have given seed: To sell, the model accurately predicted the next words to be more paper. Similarly, we can see impressive results from the unigram, bigram and trigram models. However, the problem with Markov models is that it can't generate new words but can only choose words from the dataset that has high probability. Below are the results we have got when fine tuned the GPT-2 model on our dataset.

- **It is my job to see what is written and to look at it. If it is not corrected in a timely fashion, I will fire her.**
- **I do not respect anyone, including corporate, which I will mean it. If I were a corporation, I would treat everyone equally. I would tolerate one person being fired for one thing, and one person being fired for no reason at all.**
- **There is no culture here, there is hate. There is no way that I would ever be shocked or upset about that.**
- **Michael, I'm so sorry I'm late.**
- **Every day is awful. It's the worst thing that can happen to people's lives.**
- **Michael, come on. Allow me to get you, please. [Dwight is greasy and sore from the day's work]**
- **Oh, that's helpful. Case closed.**
- **Ever since Michael dumped DeAngelo, DeAngelo has been awful. I wish I could throw Michael under the bus and tell him what he's gotta do about his bad brain.**
- **Okay, okay. Good night, everyone!**
- **[throws Jim's cell phone out of the window] Wha-? [phone rings] Shut up!**

- **It appears to be a two bulletproof vest.**

The model seems to do a great job at understanding the context and generating new quotes just like we expect from Dwight. The model also seems to capture the appropriate length of the quotes.
For our main goal, we tried generating results using the Bi-directional LSTM model and it gave some interesting results. For example, out of the whole bunch of generated text, we have handpicked a few to understand how the model works and how effective and articulate the model was in generating new text.
Some of the good examples are:

1. **Ok ,[Michael! You have everyoned I five the facts**
2. **Genghis Khan Fosters in the baby!**
3. **That 's luka. Whacking moles, take in here Michael us got here.**
4. **is it? No, huh? [just the doors]**
5. **Okay, is what you think then did you know? who doing that**
6. **Wait , shorr] Angela, vied a plans right?**
7. **Ok ,[Michael! You're Good sing are compans heart of me must fire] I durn here.**
8. **Michael ,right that and with the happer idea.**
9. **Wrong . I have tractioning Michael.**
10. **Come on. [sighs for makes**

There were also badly generated sentences such as:

i. **To dobvare resk-un for hell more]**
ii. **Iâ€™ve seen dolves.**
iii. **It 'sBllall I've left the Scranitates**
iv. **I have know, that Jim, Jim, Jim.**
v. **Get lostsigge?**
vi. **Mm-hmm .l on was tranger acted.**

Almost all the good sentences that were generated go along with Dwight's unyielding, outspoken, decisive characteristics. Also, the sentences which have 'Michael' in them seem to have unique sentences like the original transcript dialogues that had "Michael" in them. In sentences (4) and (10) we could see the 'action' phrase such as just the door and sighs which go along well with the context of the sentence giving us an additional layer of information with respect to the emotions that the dialogue carries with it and LSTM seems to capture that relation well. Also, when a rarely used word such as 'Genghis Khan (2)' seems to be used, the model tries to predict the generation as relatable as possible. In sentences (3) and (4), the model has generated longer sentences, although do not have good grammar and semantics gives the readers an understanding of the sentence.

Even though the LSTM model has generated some decent to good sentences, they still lack the ability to correlate and connect the words over longer distances grammatically like GPT-2 does. This is an inability of LSTM networks to evaluate long term dependencies overtime. Also looking at the bad sentences that were generated, there were lots of jargon words that were inserted by the model mostly when it was unable to find a suitable word or characters that can occur next. Also, the model fails to sustain names and complex words such as (iii) where the trailing word must be 'Scranton' but it identifies it to be a meaningful English word and makes up one. Also, the model struggles to make sentences when the word 'Jim' appears, and it

starts repeating the name continuously also because the original dataset often has ‘Jim’ occurring continuously unlike ‘Michael’ which occur only once in most of the dialogues. Non-English characters were also introduced (i) which indicate the inability of the model to find an appropriate word to complete the sentence.

Evaluation

Testing Natural language generation models are complicated to analyze as there is no accurate metric till date to validate a generated text. The only gold standard for evaluating each generated text is only through a Human because there are lots of factors when it comes to saying if a sequence of text is right such as semantics, grammatical form, context maintenance throughout the text and various other factors that humans don’t explicitly check for but contribute towards the holistic evaluation.
Even though NLG models suffer from this problem, the research community have come up with some metric scales that can help push the model towards improving the quality of the generated text every time. These include BLEU, METEOR, Perplexity metric, Latent Semantic Analysis, etc.
Some of the above metrics have their drawbacks as they are good for evaluating Non-NLG text values because they consider different metrics. For example, BLEU takes one or many reference texts and the test text and compares the appearance of the tokens in reference text in the test text as its main criteria. This is not suitable for text generation evaluation because sometimes the generated text might hold the meaning of the original context sentence but might use a different word such as “nice” instead of “good.” METEOR, LSA also suffer from similar problems.
So, for our case, we used Perplexity analysis as an acceptable metric to evaluate the generated texts. Perplexity analysis is a measure of how well the probability distribution of the text space predicts the generated texts. (I.e.) It evaluates the cross entropy with respect to the probability distribution obtained by the original text space and the generated sample and gives a score.

$$PP(W) = 2^{H(W)} = 2^{-\frac{1}{N}\log_2 P(w_1, w_2, \ldots, w_N)}$$

The above is the formula for evaluating perplexity of a given sentence. It is the inverse probability of the test data which are normalized over all the words.
To define the perplexity metric, we used the MLE package from Nltk to calculate the perplexity for the Markov model and Keras’ categorical_cross_entropy method to calculate the perplexity for each seed and the respective prediction. The following were the observations.

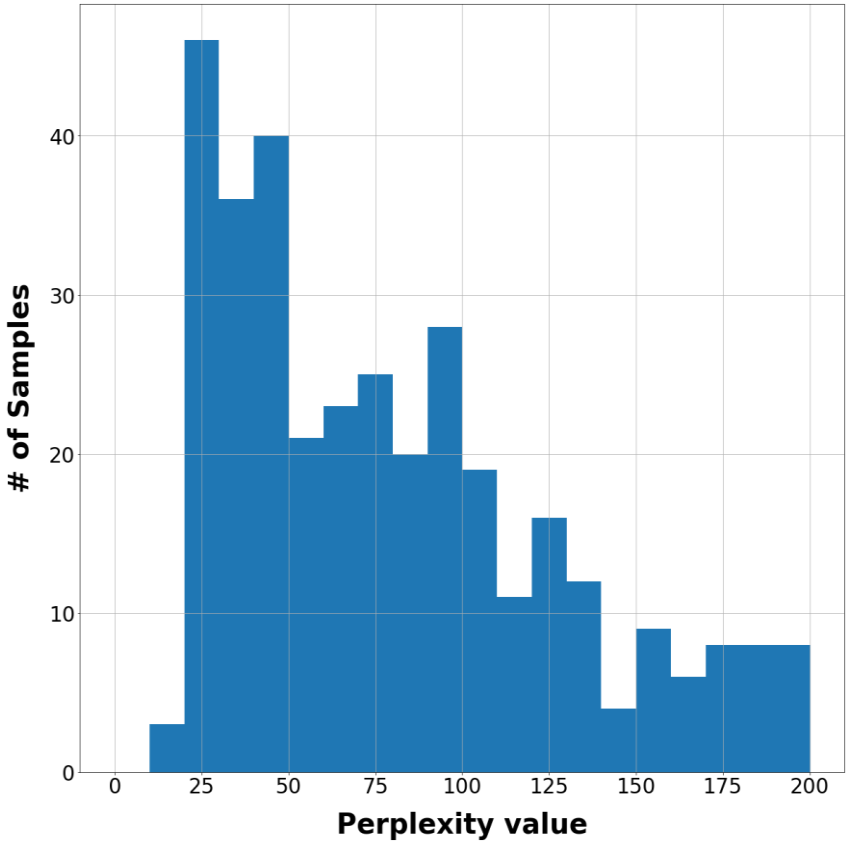


FIGURE 15: *Perplexity distribution histogram for a Unigram Markov Model samples*

As we can see from figure 15, the predicted texts have a wide range of perplexity values between the given range of 0 to 200 and although the number of samples with the perplexity range of 0 to 50 seem higher, the rest of the bars are almost half or equal to the samples with perplexity values less than 50. It means that the model is not clear to choose the best word for a given seed. So, the Markov model does not do a very good task in generating longer texts. The graph above is for a unigram model. For the current dataset, the bigram and trigram models resulted in very high perplexity scores because the dataset is made up of unique dialogues that have a different meaning to each other even if they use the same words and Markov models are not good enough to capture the underlying meaning for which we proceeded to use the Deep learning-based models.

We generated some text samples using the Bi-directional LSTM model and evaluated the perplexity over the sample set and we obtained different scores for each text as shown below.

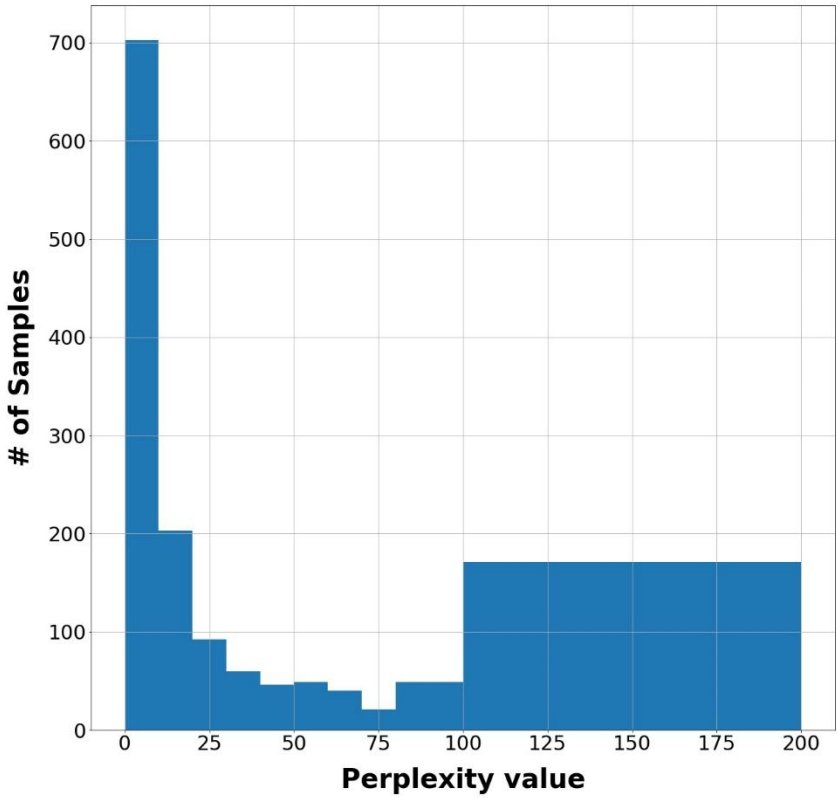


FIGURE 16: *Perplexity distribution histogram for LSTM samples*

From Figure 16 , you can see that there are lots of generated texts with a perplexity value in the range of less than 300 and very less samples in the other bins which signifies the fact that the model was able to produce text that were most probable with the probability distribution of the trained text most of the time. (I.e.) The generated text maintained the context and similarities to the seed text and original text space data. According to perplexity, the lower the value, the less confused the model is in determining the next word for generation. So based on the result, the model has done a good job generating texts. There seems to be equal number of samples with perplexity value greater than 100 which indicates complex and rarely occurring words and phrases where the model struggles to find the next occurring word which can be handled by adding more complex worded sentences to the dataset and retraining the model but in our case that would corrupt the ability of the model to maintain Dwight's style. Also, in comparison with the Markov model's perplexity histogram, there are a lot of samples with perplexity scores less than 25 from which the model can choose words from.

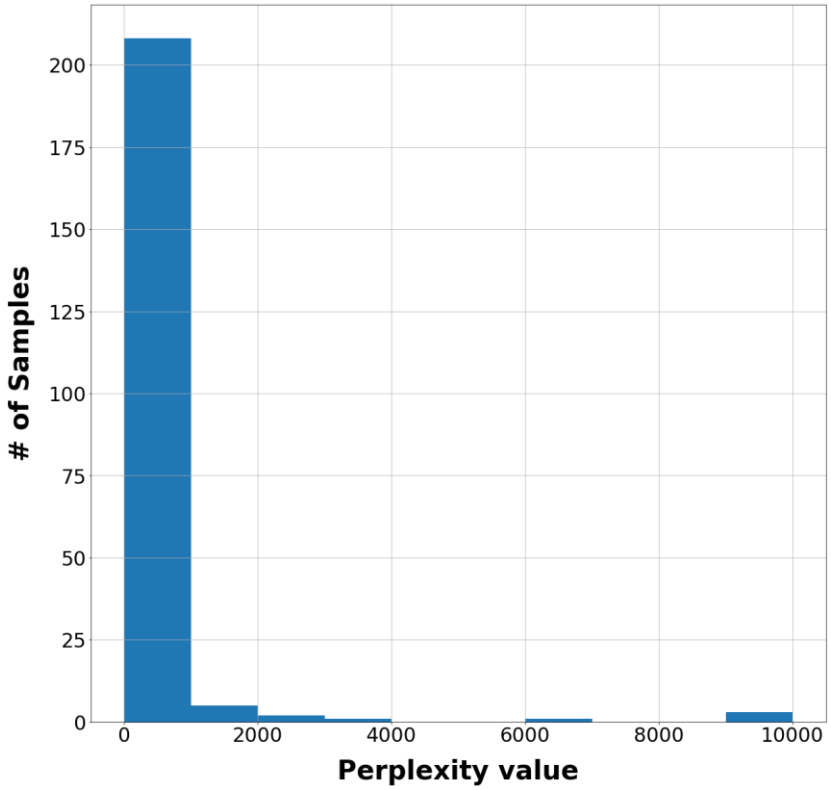


FIGURE 17: *Perplexity distribution histogram for GPT-2 samples*

From Figure 17, we can see that most of the samples have their perplexity scores in the lower value bins compared to that of the number of samples in the other bins. Compared with other models we have implemented; we can observe a very few samples are in the bins greater than 3000 suggesting that the GPT-2 model has done an excellent job at capturing the context and similarities to the seed text and the original text data. As we know from the perplexity scores, the GPT-2 model is less confused in determining the next word.

Discussion

As we have discussed earlier, creating a language model is a difficult problem. It was interesting to see how different models were able to capture different aspects of this text generation problem. Working with three different models, we have understood how the preprocessing of input sequences are different for each of the models. This is because each model has a different approach on how they process the input sequences to train. For all the models, the common step is to remove any of the unwanted characters from our corpus. For Markov model, we have created our corpus by adding all unique words to the set. We then had to create an input sequence which can be done by choosing a K value and then performing a sliding window technique to generate sequences. We have used this formatted text to train the model and later sample words to generate text. But the new word probability was not zeroed in our case and that could cause the inconsistencies in the generated text which needs to be explored for different n grams for a future scope.
For LSTMs, we have chosen a character level text generator. For the pre-processing step, we had to convert all the text to lower case and then construct a dictionary with all the unique characters mapping to a unique integer value. We have got a vocabulary size of 90. We must again generate sequences for the model to train on. We have chosen a length of 15 and generated 820,000 sequences for the model to train on. Later

we have generated onehotencoded vector on the output sequences to assign probabilities for each character in our vocabulary. From the above results we can see that LSTMs have been generating repetitive words in sequence. This might be because of our choice of the temperature value. The temperature is used to control the randomness of predictions. In our case, our model is just a classification problem with 90 classes to choose from. Hence, we have a softmax layer as our last layer. The logits would be divided by the temperature factor before passing it to the softmax layer. We have chosen a temperature value of 0.4, a smaller value of temperature produces more repetitive words. To continue our work, we would like to perform hyperparameter tuning to choose the best temperature value.

We would also want to explore how the results would change if we use Truncated backpropagation through time (TBPTT) replacing traditional BPTT. Overall, it's a very interesting project ideal for learning more about language models and inferencing useful information from just the raw data. For future evaluations, the metrics that NLP uses for evaluating textual data that are not generated are not highly ideal as evaluating generated texts.

We would need equally complex and sophisticated systems to evaluate the quality of the generated texts such as a high-level classifier that can distinguish between machine generated and human generated texts; Generative Adversarial Network based models that can generate and evaluate the texts, etc. Natural Language Generation Evaluation itself is a more complicated research area that is actively going on but metrics like perplexity can surely help models like LSTM learn the context of the given text data over training.

References:

- How to calculate perplexity score for GPT-2 model using GPT2- simple library https://github.com/gpt2ent/gpt-2-simple/blob/652fdab80131ce83f8f1b6fd00f597dd48ae2e36/gpt_2_simple/gpt_2.py#L552
- How to finetune GPT-2 model on custom dataset using GPT2-simple https://minimaxir.com/2019/09/howto-gpt2/
- LSTM character level generator model https://keras.io/examples/generative/lstm_character_level_text_generation/
- Calculating perplexity scores for LSTM model in Keras https://towardsdatascience.com/perplexity-in-language-models-87a196019a94
- How to use Markov Model to generate text http://www.wseas.us/e-library/conferences/skiathos2002/papers/447-308.pdf
- Vader algorithm for sentiment analysis https://github.com/cjhutto/vaderSentiment