Math 156: Final Project Report
Generation of Autocomplete Text Using the Apriori Algorithm and Long Short-Term Memory Recurrent Neural Networks

Kalsuda Lapborisuth and Spenser Talkington
Department of Mathematics, University of California at Los Angeles,
520 Portola Plaza, Los Angeles, California 90095, USA
(Dated: July 28, 2020)

Autocomplete text is a time saving technology that proposes a completion to a given string of characters or words. In this paper, we detail implementations of autocomplete text using the Apriori Algorithm and Long Short-Term Memory Recurrent Neural Networks for the generation of potential next words in a given phrase. Additionally, we train these two methods to the Enron Email Dataset and provide a quantitative evaluation of the efficiency of these methods and and a qualitative comparison of the results generated by these methods. We find that in general, the quality of the suggestions by the LSTM method are superior to the suggestions generated by the Apriori algorithm, even when the two methods are trained for the same amount of time with the same text training data.

I. INTRODUCTION

Autocomplete is a feature that predicts the rest of a word, phrase, or sentence inputted by a user. The feature decreases the number of keystrokes needed in order to complete a sentence and assists users in writing difficult phrases. Since the prediction generated by autocomplete relies on a probabilistic association between words in the training data, autocomplete is especially effective in structured writing, such as programming languages and emails.

In this project, we implemented the Apriori Algorithm and Long Short-Term Memory Recurrent Neural Networks (LSTM) to suggest a possible next word to complete a phrase based on the context. We found that the Apriori algorithm tends to produce grammatical results half the time, while the LSTM method produces grammatical (if not telepathic) results nearly every time. Additionally, once the methods have been trained, the cost to produce a series of potential next words based on a given phrase is low, and so the methods are computationally feasible even on low-power devices such as phones.

We note that while the training process is both memory and processing intensive, the training can be conducted on a centralized cloud computer, freeing local resources with no expense to the predictive accuracy. This centralized processing likely explains the widespread adoptions of autocomplete technology in text messaging programs and within email clients.

In Section II we introduce the Enron Email Dataset. In Sections III and IV we detail the mathematical foundations of the Apriori Algorithm and LSTM. In Section V, we review our computational implementation of the two methods, and in Section VI, we detail the results.

II. DATASET

In the open discourse, the Enron Email Dataset is a touchstone for machine learning, since it is representative of the emails composed in a work setting, and is freely available (as opposed to your Google searches or emails which only Google has access to). Consequently we select this dataset to train the methods.

In 2001, it was discovered that Enron had committed insider trading, price fixing, and had systematically falsified its accounts to make it appear that its debts and liabilities were lower than they were, and their assets and profits were higher than they were. In the proceeding investigation and lawsuit, the Federal Energy Regulatory Commission secured about 500,000 email messages from a total of 150 users, who were in management positions at Enron Corporation. These emails comprise the Enron Email Dataset and are freely available online.

To process the email data, we concatenated all email data into one plain text file, and read the data into Python. We then scrubbed the data to replace capital letters with lowercase letters, and removed all special characters. Due to limited memory available for computation, we chose an arbitrary portion from the concatenated text with the length of 700000 characters to be our training data.

III. APRIORI ALGORITHM

Consider a set of words $W = \{w_1, w_2, \ldots, w_N\}$ and a set of phrases $P = \{p_1, p_2, \ldots, p_M\}$ (as in Section II). Let $U, V \subseteq W$. Define the support of a U as:

$$\text{supp}(U) = \frac{\text{number of phrases containing } U}{\text{total number of phrases}} \tag{1}$$

Define the confidence that V will be in a phrase given

U in in the phrase:

$$\text{conf}(U \to V) = \frac{\text{supp}(U \cup V)}{\text{supp}(U)} \tag{2}$$

Now, select a support threshold $s \in (0, 1]$ and a confidence threshold $c \in (0, 1]$. We are interested in creating rules $R_i = \{U_i, \{V_i\}\}$ such that $\text{supp}(U_i) \geq s$, $\text{supp}(V_i) \geq c$, and $\text{conf}(U_i \to V_i) \geq c$. A naïve implementation creates all possible phrases U_i and V_i and tests their support and confidence. This method is computationally intensive with many possible rules, i.e. partitioning words into three categories with repetition [1]:

$$|R| = 3^M - 2^{M+1} + 1 \tag{3}$$

This number of candidate rules is infeasible for a computational implementation, since M is the number of words, which is $M \gtrsim 10^4$.

Instead, we use the Apriori algorithm [2]. The Apriori algorithm proceeds by way of creating frequent phrases, and then creating rules.

Begin by creating the subset:

$$W^{(1)} = \{w_i \in W : \text{supp}(w_i) \geq s\} \tag{4}$$

Then create the further subset using the "join rule:"

$$\begin{aligned} W^{(2)} = \{&w_i^{(1)} \cup w_j^{(1)} : w_i^{(1)}, w_j^{(1)} \in W^{(1)} \wedge \\ &\text{supp}(w_i^{(1)} \cup w_j^{(1)}) \geq s \wedge \\ &|w_i^{(1)} \cup w_j^{(1)}| = 2\} \end{aligned} \tag{5}$$

And, similarly:

$$\begin{aligned} W^{(3)} = \{&w_i^{(2)} \cup w_j^{(2)} : w_i^{(2)}, w_j^{(2)} \in W^{(2)} \wedge \\ &\text{supp}(w_i^{(2)} \cup w_j^{(2)}) \geq s \wedge \\ &|w_i^{(2)} \cup w_j^{(2)}| = 3\} \end{aligned} \tag{6}$$

Continue until $W^{(n+1)} = \emptyset$.

Now, to create rules, for each $w_i^{(n)} \in W^{(n)}$, generate all partitions of $w_i^{(n)}$ into two sets, and select based on confidence. The set of rules is then:

$$\begin{aligned} R = \{\{u_i^{(n)}, \{v_i^{(n)}\}\} : &\text{conf}(u_i^{(n)} \to v_i^{(n)}) \geq c \wedge \\ &u_i^{(n)} \cap v_i^{(n)} = \emptyset \wedge \\ &u_i^{(n)} \cup v_i^{(n)} = w_i^{(n)} \wedge \\ &w_i^{(n)} \in W^{(n)}\} \end{aligned} \tag{7}$$

We note that $n \ll M$, and the number of candidate rules is the number of partitions into two non-empty sets, i.e. the number of binary strings of length n that are not all zero or all one [1]:

$$|R| = |W^{(n)}|\,(2^n - 2) \tag{8}$$

To select the next word from a given partial phrase ρ, if $|\rho| \leq n$, then create the set of potential next words (allowing duplication of words/elements in Ω, where r_j^L is the left part/hypothesis of the rule):

$$\Omega = \{w_i : w_i \in (r_j^L \setminus \rho) \ \wedge \ |r_j^L \cap \rho| = |\rho| \ \wedge \ r_j \in R\} \tag{9}$$

If $\Omega = \emptyset$, then relax the condition $|r_j^L \cap \rho| = |\rho|$ to $|r_j^L \cap \rho| = |\rho| - 1$, $|r_j^L \cap \rho| = |\rho| - 2$, etc. When $\Omega \neq \emptyset$, then select the next word in the partial phrase as the most frequent element in Ω.

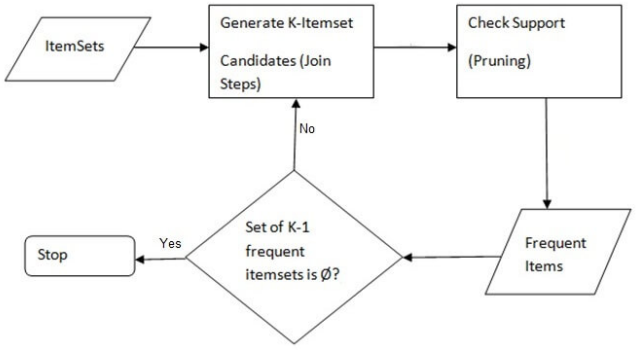

FIG. 1: Schematic representation of the generation of frequent phrases (itemsets) from all phrases (itemsets). Image from [1].

A. Example: Course Planning

Here we consider a concrete (small-scale) example of the Apriori algorithm. Suppose that Erkki the engineering counselor is trying to learn which classes first years typically take together so that he can inform the registrar. Erkki brings up the course schedules for six students:

$$\begin{aligned} p_1 &= \{\text{MATH 32A, MSE 104, ENGL 90, LING 1}\} \\ p_2 &= \{\text{MATH 31B, PHYS 1B, CHEME 45, HIST 1C}\} \\ p_3 &= \{\text{MATH 32A, PHYS 1B, HIST 1C, LING 1}\} \\ p_4 &= \{\text{MATH 32B, PHYS 1B, CHEME 45}\} \\ p_5 &= \{\text{MATH 32B, MSE 104, STATS 10, ENGL 90}\} \\ p_6 &= \{\text{MATH 31A, PHYS 1A, CEE M20, HIST 1C}\} \end{aligned} \tag{10}$$

All possible courses are:

$$\begin{aligned} W = \{&\text{MATH 31A, MATH 31B, MATH 32A, MATH 32B,} \\ &\text{MSE 104, CEE M20, ENGL 90, HIST 1C, LING 1,} \\ &\text{PHYS 1A, PHYS 1B, CHEME 45, STATS 10}\} \end{aligned} \tag{11}$$

Now, suppose that Erkki chooses a minimum support of $s = 0.33$. Then he finds:

$$\begin{aligned} W^{(1)} = \{&\text{MATH 32A, MATH 32B, MSE 104, ENGL 90,} \\ &\text{HIST 1C, LING 1, PHYS 1B, CHEME 45}\} \end{aligned} \tag{12}$$

He then creates all possible subsets of $W^{(1)}$ with 2 elements (viz. 8 choose 2 = 28), and selects those with support greater than or equal to s:

$$W^{(2)} = \{\{\text{MATH 32A, LING 1}\}, \{\text{MSE 104, ENGL 90}\} \{\text{HIST 1C, PHYS 1B}\}, \{\text{PHYS 1B, CHEME 45}\}\} \tag{13}$$

Now, the only candidate element for $W^{(3)}$ is:

$$\{\text{HIST 1C, PHYS 1B}\} \cup \{\text{PHYS 1B, CHEME 45}\} = \{\text{HIST 1C, PHYS 1B, CHEME 45}\} \tag{14}$$

This course combination only occurs in P_2, so its support is 1/6, and so:

$$W^{(3)} = \emptyset \tag{15}$$

Next, to form the rules, Erkki refers to $W^{(2)}$, in which there are four elements, and consequently eight possible rules. If Erkki selects a confidence level of 0.75, he will find the rules:

$$R = \{\{\text{MATH 32A, \{LING 1\}}\}, \{\text{LING 1, \{MATH 32A\}}\}, \{\text{MSE 104, \{ENGL 90\}}\}, \{\text{ENGL 90, \{MSE 104\}}\}, \{\text{CHEME 45, \{PHYS 1B\}}\}\} \tag{16}$$

Now, suppose that Erkki knows that a student is taking four classes and that three of their classes are:

$$\rho = \{\text{MATH 32A, PHYS 1A, HIST 1C}\} \tag{17}$$

Erkki then finds that:

$$\Omega = \{\text{LING 1}\} \tag{18}$$

And, so Erkki predicts that the student is taking:

$$\{\text{MATH 32A, PHYS 1A, HIST 1C, LING 1}\} \tag{19}$$

B. Limitations of the Apriori Algorithm

While the Apriori Algorithm can predict the next word in a phrase $\rho = \{\rho_1, \rho_2, \ldots, \rho_m\}$ for which some of the elements of $\rho_i \notin W$, the words it will predict ρ_{m+1} are limited to those that are present in the training data, i.e. $\rho_{m+1} \in W$. In particular, this suggests that for optimal predictive performance the training data should be sufficiently inclusive to encompass all words that would in fact come next.

Another limitation (and the source of its improvement over the naïve rule generation method) is that the Apriori Algorithm imposes the cutoffs for the support s and the confidence c, such that infrequent and improbable associations are suppressed. However, infrequent phrases do occur, and from an entropic standpoint carry the most information (unexpectedness). To improve this situation, one can lower c and s, but this increases the required computations, and biases results towards less frequent phrases. To remove this bias, the confidence in each rule can be used to assign a weight the entries in Ω, rather than a simple 1 or 0 counting mechanism.

Perhaps the most severe limitation is that the Apriori Algorithm is irrespective of order. In particular, the word at the beginning of a phrase, ρ_1, carries the same weight as a word at the end of the phrase, ρ_m. While this is not a limitation when order is unimportant, it is a limitation in writing, since the order of words does matter.

IV. LONG SHORT-TERM NEURAL NETWORKS

As a result of the universal approximator theorem, "for a broad range of hidden unit activation functions and sufficiently many hidden units, a two-layer NN can uniformly approximate any continuous function on a compact input data to arbitrary accuracy" [3]. The class of all continuous functions is very large, and so neural networks are useful for a wide range of applications.

Here, we are interested in finding the next word in a phrase. Within the context of neural networks, this involves classifying an input phrase ρ to a category $w_i \in W$, where ρ and W are as in Section III.

In this section, we review neural networks, introduce recurrent neural networks, and detail Long Short-Term Memory Neural networks.

A. Neural Networks

Neural networks are functions of the form:

$$y(x) = f_N(W_N^T f_{N-1}(W_{N-1}^T \ldots f_2(W_2^T f_1(W_1^T x)))) \tag{20}$$

Where f_n are usually non-linear functions such as the sigmoid function, tanh function, or the relu function, and W_n are weights assigned to each layer (not sets of words). Here, we use a notation where it is understood that at each layer a bias term is included in the output vector from the previous layer.

With a set of phrases as training data, where the next word is separated from the phrase and used as the output:

$$P = \{(\rho^{(1)}, \rho^{(1)}_{|\rho^{(1)}|+1}),\ \ldots\ ,\ (\rho^{(M)}, \rho^{(M)}_{|\rho^{(M)}|+1})\} \tag{21}$$

it is then possible to assign a cost to the output y for an input $\rho^{(m)}$:

$$J_m = J(y(\rho^{(m)})|(\rho^{(m)}, \rho^{(m)}_{|\rho^{(m)}|+1})) \tag{22}$$

For regression, the cost function is typically of a least squares form, while for classification the cost function could take the form of the Fisher Criterion. Here we use the categorical cross-entropy:

$$J_m = -\rho^{(m)}_{|\rho^{(m)}|+1} \ln(y(\rho^{(m)})) \tag{23}$$

Within this framework, it is then possible to use gradient descent by way of evaluating the error using backpropagation. For more information on this, we refer the reader to Section 5.3 of Bishop [3].

While it is in principle possible to learn the associations between inputs and outputs, we again run into the issue of cardinality. In the naïve gradient descent neural network, the number of possible input phrases increases rapidly with $|\rho|$ (note $|\rho| \ll |W|$):

$$|\{\rho : |\rho| = t \ \wedge \ t \in \mathbb{N}\}| = \binom{|W|}{|\rho|} \tag{24}$$

$$= \frac{|W|!}{|\rho|!\,(|W| - |\rho|)!} \tag{25}$$

While this is in principle solvable, the solution demands more hidden units than can be modeled. In order to solve this problem, the dimension of the input must be reduced. To do so, feed the input vector one component (or a few) at a time into the network. If one component is fed, then the dimensionality is reduced to $|W|$. But then the relationship between the components of the input vector is lost. Recurrent neural networks solve this problem through a direct combination of the current data and previous output.

B. Recurrent Neural Networks

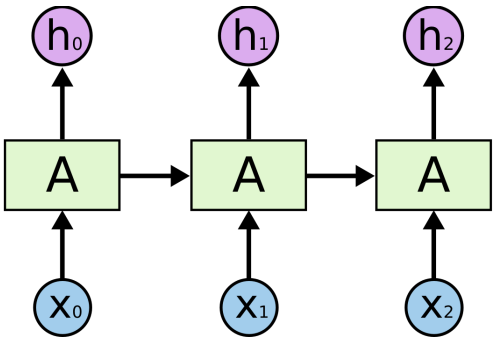

FIG. 2: Schematic representation of a Recurrent Neural Network with three data vectors x_0, x_1, and x_2. Image from [4].

In Recurrent Neural Networks (RNN), the output from previous information is used to determine the behavior of the current layer [4]. We define the input vector to the nth layer as a combination of the activated output of the $n-1$th layer, including a bias term $h^{(0)}_{n-1} = 1$, and x_n is a data vector with a bias term $x^{(0)}_n = 1$:

$$z_n = \begin{pmatrix} x_n \\ h_{n-1} \end{pmatrix} \tag{26}$$

In this configuration, the activated output of the nth layer is given in terms of a nonlinear function f, frequently tanh, or the relu function, with hidden-layer weights W_h and input vector z_n. We take f to be the relu function, and:

$$h_n = f(W_h^T z_n) \tag{27}$$

However, RNN methods are often limited by a "vanishing gradient" problem. Because of the connection between the hidden layers, the input from many timesteps ago are multiplied several times by the constant weight vector, W_h, which result in outputs that collapse to zero or diverge towards infinity [5]. As a result, RNN are unable to model long-term dependencies.

In particular, a traditional RNN method for autocomplete will not be able to effectively use information from the start of a paragraph to predict the next word near the end of the paragraph.

C. The Long Short-Term Neural Network (LSTM)

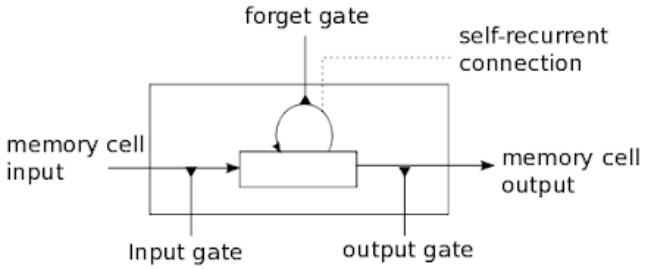

FIG. 3: Representation of a Memory cell in Long Short-Term Neural Network between its input state and output state [6].

To overcome this limitation, LSTM utilizes a function called a memory cell between its input layer and hidden layer, as depicted in FIG 3. The component of a state in LSTM is still the same as in RNN, however the memory cell decides how much of the input to consider, how much of the internal state to forget, and how much of the output to release [6].

Specifically, three new weight matrices are introduced to accomplish this task: W_i, W_f, and W_o. The weight matrices work in conjunction with sigmoid functions to create "valves" which scale quantities by a factor in $(0, 1)$:

$$h_n = \sigma(W_o z_n)\ f\left(W_h^T\left[\sigma(W_i z_n)\begin{pmatrix} x_n \\ h_{n-1} \end{pmatrix} + \sigma(W_f z_n)\begin{pmatrix} 0 \\ h_{n-1} \end{pmatrix}\right]\right) \tag{28}$$

The valves ensure that the outputs do not diverge, and the new weight vectors allow LSTM to memorize the parts of phrases that are important and to forget the parts that are not.

The weight vectors may be learned by stochastic gradient descent. However we use the Adam optimizer since we found that it gave results more quickly than stochastic gradient descent.

D. The limitations of LSTM

LSTM is memory instensive because it loads the entire training dataset into memory. While this is beneficial because long term associations can be determined, erroneous associations can be determined too. In particular, for the Enron Email Dataset, LSTM would look for associations between adjacent emails, even if their content was not related. As a result of this, we expect that the LSTM method would work best for one work by one author on one topic, such as Shakespeare's *Hamlet*.

Additionally, LSTM can generate associations that are technically correct, but that are not useful. For example, always suggesting the word "the." To address this problem, there is an emerging field of attention models which process the training data before giving it to LSTM to emphasize the "big picture," of the data and to direct what LSTM should focus on.

V. COMPUTATIONAL IMPLEMENTATION

For the computational implementation, we followed the descriptions of the algorithms provided in Sections III and IV. In order work on the project collaboratively, we used Python through Google Colab's implementation of iPython. We train our models the Google Colab normal instance without the Graphic Processing Unit (GPU) in order to fairly compare the speed of the two method. However, we should note that both methods can be trained more efficiently when optimized for parallel computing on the GPU.

A. Data pre-processing

We loaded and scrubbed the text with basic Python functions, removing all of the headers, email addresses, and special characters. After compiling all of the cleaned text together, we arbitrarily selected a largest portion possible from the text to train our model within the memory limit in Google Colab, its length being 700000 characters long.

Particularly for the Apriori algorithm, we then broke the data into phrases $P = \{p_m\}$ through the sentence ending characters `.`, `;` and `?`. If any sentences are longer than 7 words, we split them evenly into phrases of length 3-6 words. Each phrase was composed of one or more word $w_n \in W$ stored as a tuple.

However, for the LSTM algorithm, we generate the training input vectors based on the data as shown in FIG 4. The training inputs are two words appearing consecutively in the text and their corresponding target outputs are the word that appears right after those two input words. Using the natural language toolkit (`nltk`) package, we create a dictionary that maps each unique word to a number. We then use this dictionary to vectorize the training inputs and the target outputs.

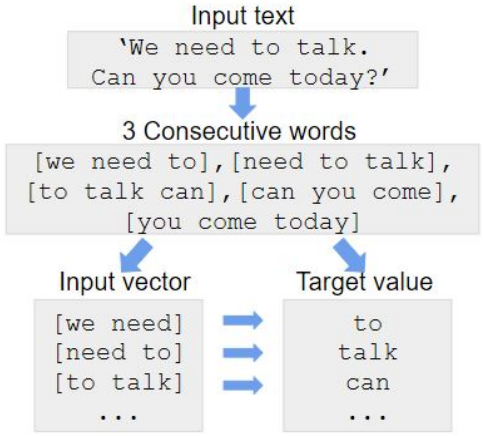

FIG. 4: An example of input vector and target value generation from input text for the LSTM model

We should note that the length of the training inputs should be adjusted to the nature of the text. The sentence structure in corporate email is often straightforward and concise. Consequently, the appropriate length of training inputs for this project is 2-3 words, as the next words are often heavily dependent only on the few words immediately preceding itself. If the text has a more complicated structure, several LSTM models trained on different input length can easily be used to better predict the next word for text with complicated structure.

B. Configuration of the methods

After pre-processing the data, we used the `keras` neural network library to implement the LSTM algorithm, and used the `efficient-apriori` library to implement the backend of the Apriori Algorithm. The LSTM network contains 5 layers: an input layer, two LSTM layers, and two fully connected layers, which uses relu and softmax as the activation function. We then wrapped both methods within functions that accepted a partial phrase and returned the three best next words.

For the Apriori algorithm, we trained two predictors with the parameters: first, $s = 0.0002$ and $c = 1$, and second, $s = 0.0003$ and $c = 1$. For the LSTM method, we also trained two predictors over 50 epochs and 150 epochs. (Note: an epoch is a sweep of descent with size equal to the number of parameters.)

VI. RESULTS

A. Quantitative

We compare the training time, training memory usage, suggestion speed, the size of the trained models generated by the Apriori Algorithm to those generated by the LSTM method.

Table 1: Quantitative comparisons of the models

	Training time	Training Memory	Prediction time	Trained model size
Apriori s = 0.0002	159 minutes	1.57 GB	0.5-3 seconds	1307 KB
Apriori s = 0.0003	77 minutes	1.87 GB	0.4 - 2.5 seconds	1108 KB
LSTM 50 epochs	55 minutes	4.89 GB	6 - 8 milliseconds	6629 KB
LSTM 150 epochs	178 minutes	5.17 GB	10 - 12 milliseconds	6620 KB

We find that both methods require at least an hour to train a model that is at least able to adequately predict the next word using the Enron Email Dataset.

In our implementation, the LSTM algorithm requires more memory to store the vectorized training inputs than the Apriori algorithm. For an input text with n words, the LSTM method needs $O(n^2)$ space to store the input. The Apriori method only requires $O(n)$ space to store phrase input. Conversely, the memory required by LSTM method during training is not dependent on the number of epochs, while the memory used by the Apriori method depends on the value of the minimum support, s.

We also find that both LSTM models are able to generate the best three predictions 250-500 times faster than the Apriori models (although even the Apriori algorithm still takes less than three seconds). Furthermore, we note that, despite the size of the LSTM models being 3-6 times larger than the size of Apriori models, the size of LSTM is not dependent on its training epochs while the size of Apriori models are dependent on the value of the minimum support s.

The constant model size, fast prediction time, and comparable training time makes LSTM preferable to the Apriori algorithm for a real world implementation of autocomplete text.

B. Qualitative

Additionally, we provide a qualitative analysis of the results produced by the Apriori Algorithm and LSTM. In particular, we provide three sets of three partial phrases with short $|\rho| = 2$, medium $|\rho| = 4$, and long $|\rho| = 6$ length. With each algorithm, we generate three possible words to follow the given assign phrase and rate these words as grammatically good, okay, or bad.

The partial phrases were selected to have clear following words or word types, express the diversity of the English language, and be representative of phrases commonly encountered in business emails.

Table 2.1: Qualitative comparison of the next word suggestions by Apriori (s = 0.0003) vs LSTM (50 epochs)

	Apriori Algorithm			LSTM Method		
Partial Phrase	1	2	3	1	2	3
let me	you	in	at	know	in	take
I would	you	in	at	like	be	appreciate
all supporting	you	in	at	the	of	to
after the presentation you	starting	these	at	will	are	have
when our next shipment	favorite	lite	miller	of	who	allow
I have completed the	for	up	starting	feedback	call	copy
after work I will go to the	attend	in	starting	success	company	research
if you have any questions about	for	up	signed	capacity	plant	this
that should be possible on Wednesday	you	in	at	several	the	of

Table 2.2: Qualitative comparison of the next word suggestions by Apriori 1 (s=0.0003) VS Apriori 2 (s= 0.0002)

	Apriori 1			Apriori 2		
Partial Phrase	1	2	3	1	2	3
let me	you	in	at	to	way	each
I would	you	in	at	be	to	way
all supporting	you	in	at	to	way	each
after the presentation you	starting	these	at	starting	these	at
when our next shipment	favorite	lite	miller	favorite	lite	miller
I have completed the	for	up	starting	for	signed	up
after work I will go to the	attend	in	starting	loss	please	your
if you have any questions about	for	up	signed	for	up	signed
that should be possible on Wednesday	you	in	at	the	big	to

Table 2.3: Qualitative comparison of the next word suggestions by LSTM (50 epochs) vs LSTM (150 epochs)

	LSTM 50			LSTM 150		
Partial Phrase	1	2	3	1	2	3
let me	know	in	take	take	know	to
I would	like	be	appreciate	be	like	suggest
all supporting	the	of	to	the	doing	gasoline
after the presentation you	will	are	have	would	will	are
when our next shipment	of	who	allow	of	meeting	set
I have completed the	feedback	call	copy	feedback	end	holidays
after work I will go to the	success	company	research	extent	group	official
if you have any questions about	capacity	plant	this	the	a	enron
that should be possible on Wednesday	several	the	of	and	of	to

For the partial phrases in Tables 2.1, 2.2, and 2.3, we see that the Apriori algorithm generates a good result in its three suggestions about half the time, while the LSTM method will usually generate at least one good result.

The low success rate of the Apriori algorithm is largely because if the combination of input words are not in the rules generated by the Apriori algorithm, the models fail to make the next best guess and simply returns a default set of words. See for example the partial phrase $\rho = \{\text{after, the, presentation, you}\}$. Furthermore, we note that neither model isable to handle a partial phrase that contains a word outside of the training data. For example, "supporting" is not in the training text, and so both methods perform poorly on the partial phrase $\rho = \{\text{all, supporting}\}$.

We find that in general, the quality of the suggestions by the LSTM method are superior to the suggestions generated by the Apriori algorithm, even when the two methods are trained for the same amount of time with the same text training data.

Interestingly, we find that any LSTM models that are trained longer than 1 hour are only able to predict the exact words from the training data for only 10-15% at a time, as exemplified in FIG 5 and FIG 6. We also note that the accuracy value that the model converged to varies when we train the model several time.

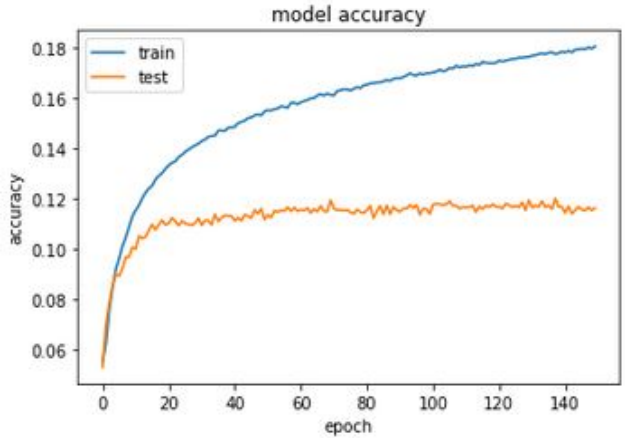

FIG 5: The accuracy of LSTM model tested against the training dataset when trained over 150 epochs.

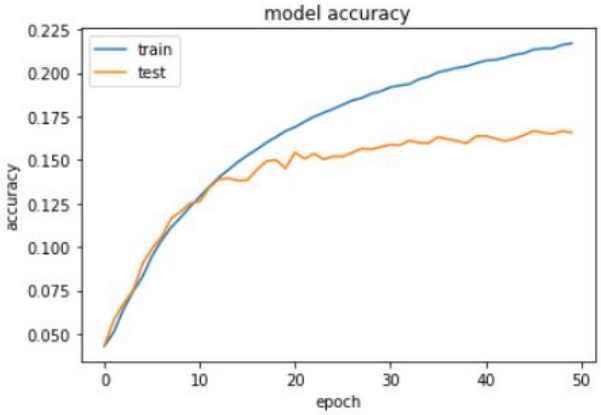

FIG 6: The accuracy of LSTM model tested against the training dataset when trained over 50 epochs.

This result is to be expected because various phrases, especially the more common ones, are often followed by several different words. For example, the next words that often follow 'I would' include 'be', 'like', 'suggest', and several other verbs. So the model might predict 'be' when the sentence in the training data is 'I would like', which would be considered to be wrong and inaccurate in this

case. However, this clearly does not mean that the suggestions made by the LSTM method is not grammatically or contextually viable for the next word. Therefore, we believe that this accuracy fails to capture the true potential of the LSTM method and, and to further understand the accuracy of any text autocomplete mode, it is important to test them on a dataset labeled by human in a similar manner to our qualitative comparison.

VII. CONCLUSION AND FURTHER DIRECTIONS

Autocomplete is an invaluable technology. The completion of phrases results in time savings in both professional and personal contexts which leads to increased productivity and wellbeing. In this report, we presented two common methods for generating autocomplete text: the Apriori Algorithm and Long Short-Term Memory Recurrent Neural Networks (LSTM). We find that both methods generally produce grammatical autocomplete text, and that LSTM usually produces qualitatively better autocomplete text than the Apriori algorithm.

Apriori is predicated on finding frequent phrases and relations among words, while LSTM is based on finding optimal combinations of weights for a partial phrase. We found that both the Apriori algorithm and LSTM demand substantial computational resources during training, but require minimal computational resources once trained. We believe that this is a large reason for the prevalence and impact of autocomplete. In particular, low-power devices like phones can easily implement autocomplete once training is finished.

Our computational implementations of autocomplete were limited by the computational resources available in Google CoLab to train the Apriori Algorithm and the LSTM method. In particular, for the Apriori Algorthm when we set the minimum support as $s = 0.0002$, the notebook took 159 minutes to run (the $s = 0.0001$ run timed out after 6 hours), and for the LSTM method the notebook consumed all 12 GB of memory available. We believe that by using massively parallized versions of these methods (on a GPU based architechture) would run more quickly and enable more accurate text prediction. Additionally, we believe that increasing the computer memory would enable more detailed models to generate text.

If we were to continue the project, we would have tested the methods on a different set of emails, or non-email text to test its transferability. However within the project time frame, we found that this was not feasible. An additional direction to take an extended project would be to extend LSTM to include methods from novel "attention based" neural networks.

[1] R. Jain, *A beginner's tutorial on the apriori algorithm* (2017).

[2] R. Agrawal, R. Srikant, *et al.*, in *Proc. 20th VLDB*, Vol. 1215 (1994) pp. 487–499.

[3] C. M. Bishop, *Pattern Recognition and Machine Learning* (Springer, 2006).

[4] Colah, *Understanding LSTM Networks* (2015).

[5] Y. Bengio, P. Simard, and P. Frasconi, IEEE Trans. **5**, 157 (1994).

[6] H. Wang and B. Raj, arXiv:1702.07800 (2017).