Evaluation of LSTM-cell based Recurrent Neural Network on Next Word Prediction

ChaoBang Huang
chbuus@bu.edu
Boston University
Boston, MA 02215

Yixiu Zhu
zhuyixiu@bu.edu
Boston University
Boston, MA 02215

Abstract

In this paper, we quickly compared different RNN cells learned from lectures such as the long short-term memory (LSTM) unit and the gradient recurrent unit (GRU) for building our next word prediction model. We then greatly focus on architecture design and feature engineering. Afterwards, we evaluated our results by letting the model make predictions of the next word based on a set of 5-word sentences from the text. Also, we conducted a closer analysis of architecture's performance and efficiency for our dataset and what the issues are with current implementation.

1 Introduction

As social media prevailed in the current world, people may find them typing on their devices every day. Next word prediction model presents people with a more efficient way of text communication and has already proved its prominence in the field of deep learning.

In this paper, we build our next word prediction model based on a recurrent neural network (RNN). We started with a set of 100,000-word text strings as our data source. We then briefly test the efficiency between LSTM and GRU units in our initial trial and make the decision of what cell to use. We also spend a tremendous amount of time doing feature engineering and choosing loss and metrics. During our training stage, we employed a host of regularization and optimization techniques and tuned our hyperparameters.Ultimately, we tested our model, letting it predict the top 5 most possible words after a given sentence.

Based on our simple experiment, we could conclude that our next word prediction model shows decent prediction accuracy for the original text file, which contains both training data and validation data. Although most of the sentences from the original text file have been seen during the training, we think it acceptable to do the testing on them on the grounds that the context of this application is for the model to predict the next word based on sentences users have typed before. Furthermore, granted that the performance would not be great on data it has not seen before, it is mainly because the data size is extremely small. We used the data set only because it is from the website Professor Chin provided, **20 Deep Learning Projects with Python**. It becomes the most severe limitation on our model's performance. If we were to choose the data set on our own, we would choose a way larger data set, and the issue of quick overfitting can be resolved. The generalized performance would also be tremendously improved.

1.1 Background

1.1.1 Why use RNN?

In real life, we type sentences consisting of multiple words. Each word is mutually connected and the sentences' meaning will distort or lose if we change any one of them. RNN is a class of

35th Conference on Neural Information Processing Systems (NeurIPS 2021), Sydney, Australia.

artificial neural networks where connections between nodes form a directed graph along a sequence. It allows sending all the old state information at all time steps. RNN enables our algorithm to establish connections not only between words nearby, but words associated within a long distance. In other words, order of input data would be taken into account, which is what CNN cannot achieve. Furthermore, the length of sentence to pass in is a hyper parameter to tune, RNN can naturally deal with sentences with different length.

1.1.2 Intro of LSTM and GRU

Initially, we were determining what RNN cells to use. Vanilla Recurrent Neural Network (VRNN) is the simplest form of RNN. However, it suffers from gradient vanishing. Also, as we train for long-term dependency tasks, this problem could lead to the decrease of training accuracy and cease the experiment due to gradient vanishing, which is even very hard to detect.

Therefore, LSTM and GRU were developed to tackle this problem suffered by traditional VRNN. They both implemented gated units to adjust the information flow within the network. Figure below shows the representation of RNN, LSTM, and GRU's sequence tagging model.

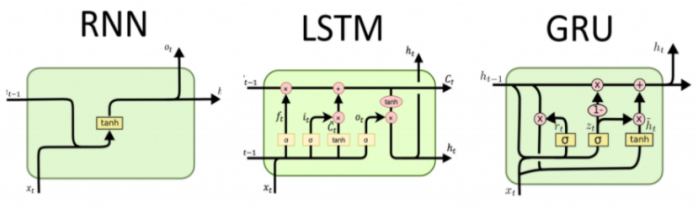


Figure 1: Representation of RNN, LSTM, and GRU's sequence model

LSTM networks are similar to VRNNs, but their hidden layer updates are replaced by memory cells. This makes LSTM better at exploring long-range dependencies in data which is imperative for sentence structures. To be specific, the LSTM network generally contains 3 gates: the input gate controls the information flow into the memory cell, the forget gate controls the information flow out of the system and the output gate controls how the information will be processed and output. The combination of three gates forms a complete path of remembering the long-term dependency of the system.

GRU was first introduced in 2015 (Bahdanau et al. (2015)). As a later successor, it competes with LSTM with a higher training efficiency due to its relatively simpler structure. This structure of RNN only contains two gates: the update gate controls the information that flows into the memory and the reset gate controls the information that flows out of memory.

Here we present a graph of these three RNN architectures for further comparison:

Table 1: Comparism of Vanilla, LSTM and GRU architectures

Vanilla	LSTM	GRU
Suffers from vanishing and exploding gradient issue, which leads to large errors and system instability	More accurate in longer sequence training	Faster in training
	More gates to control, though it can take more time to train	Efficient due to low memory usage and simpler structure

2 Implementation

2.1 Dataset and Data preprocessing

We performed our experiment based on our imported dataset. Our dataset contains a fiction (long string) named "The Adventures of Sherlock Holmes", which features around 100000 words. Note that this fiction is mainly written in Victorian English, which may show some of its preference in our later prediction results.

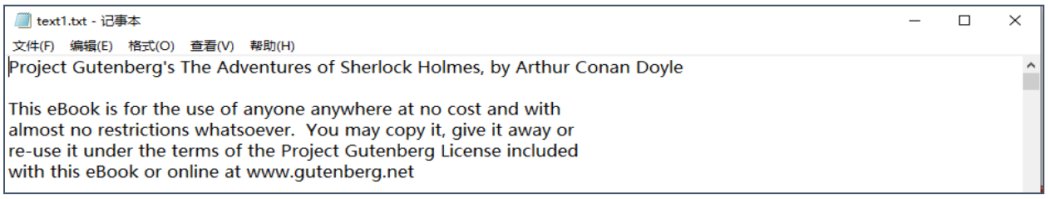


Figure 2: "The Adventures of Sherlock Holmes"

We managed to preprocess our training data with the following five steps. This process helped us to sort our data neatly and provided us a chance to closely visualize our data distribution.

Step 1: read the entire string
Step 2: get rid of all punctuation marks and split the entire string into list of words
Step 3: use np.unique to get unique occurrence of words and sort them
Step 4: remove training-irrelevant "noises"
Step 5: Split the data into features and labels

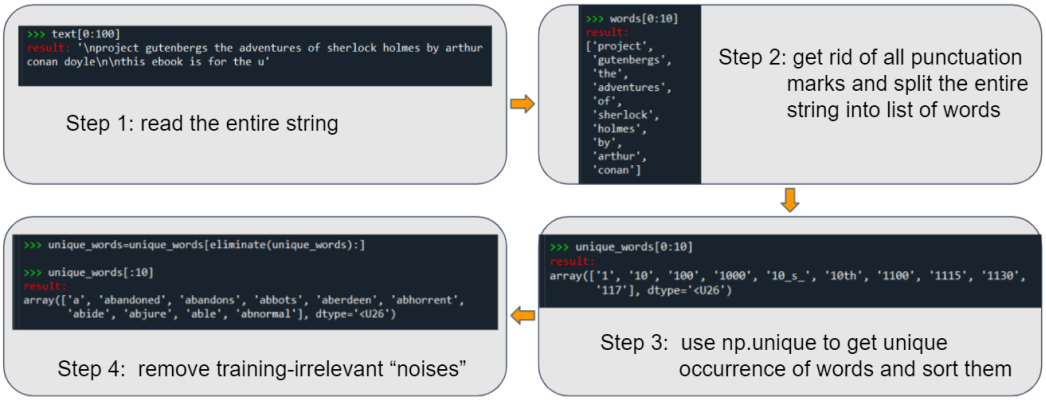


Figure 3: First four steps for data preprocessing

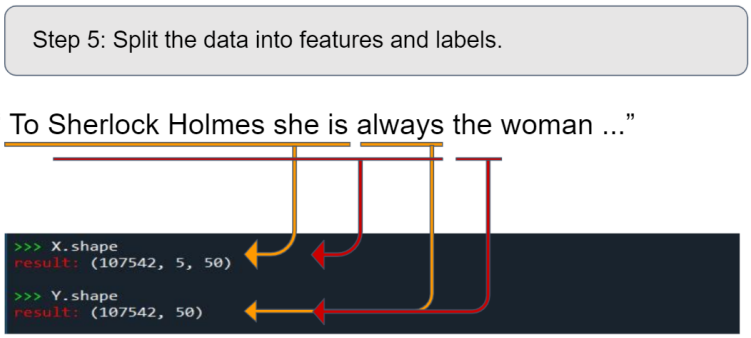


Figure 4: Fifth step of data preprocessing

2.2 Feature Engineering

The feature engineering technique is using the domain knowledge of the data to create features that can be used in training a machine learning algorithm.

2.2.1 First approach: One-Hot encoding

One hot encoding is a very naive but simple approach in natural language processing. It simply distinguishes each word in a vocabulary from every other word in the vocabulary. For One Hot encoding, a sparse N x 1 vector, where N is the number of unique classes, will be created for each unique word. A single 1 in a cell will be used to identify a unique word with 0s filled in all other cells. While One Hot encoding remains an easy way for implementation, it exposed its flaws during our testing experiment.

```
MemoryError: Unable to allocate 7.01 GiB for an array with shape
(107542, 8744) and data type float64
```

Figure 5: Proof of One Hot encoding uses excessive memory

According to the figure shown above, the One Hot encoding technique requests 7.01GiB for allocating memory, which is a huge waste of RAM and the problem will be worse if our data size increases. Furthermore, according to One Hot encoding’s logic, all words are linearly independent, which will generate extra difficulty for identifying the similar words. For example, “man” and “men” versus “men” and “pencil” would incur same loss, but it is not the thing we want to see.

Overall, One Hot encoding has proven to be unrealistic for the task. As we have identified over 8000 unique words (8000 classes) for our data source, One Hot encoding would be extremely inefficient and impossible to train.

2.2.2 New Approach: GloVe Vector

GloVe stands for global vectors for word representation. It is an unsupervised learning algorithm developed by Stanford University. It has already performed training on aggregated global word-word co-occurrence matrix from a corpus.

So for our case, we could use a dense 50x1 vector to represent each unique word in our data. Thus, similar words will be closer to each other in terms of Euclidean distance. The classification problem evolves to a kind of regression problem in this case. This trial also uses much less memory compared to One Hot encoding and allows us to arbitrarily add more unique words if needed. Below is the demonstration of getting a glove vector from the dictionary.

```
>>> glovelModel["person"]
result:
array([ 0.61734  ,  0.40035  ,  0.067786 , -0.34263  ,  2.0647   ,
        0.60844  ,  0.32558  ,  0.3869   ,  0.36906  ,  0.16553  ,
        0.0065053, -0.075674 ,  0.57099  ,  0.17314  ,  1.0142   ,
       -0.49581  , -0.38152  ,  0.49255  , -0.16737  , -0.33948  ,
       -0.44405  ,  0.77543  ,  0.20935  ,  0.6007   ,  0.86649  ,
       -1.8923   , -0.37901  , -0.28044  ,  0.64214  , -0.23549  ,
        2.9358   , -0.086004 , -0.14327  , -0.50161  ,  0.25291  ,
       -0.065446 ,  0.60768  ,  0.13984  ,  0.018135 , -0.34877  ,
        0.039985 ,  0.07943  ,  0.39318  ,  1.0562   , -0.23624  ,
       -0.4194   , -0.35332  , -0.15234  ,  0.62158  ,  0.79257  ])

>>> find_closest_embeddings(glovelModel["person"])[0:5]
result: ['person', 'someone', 'actually', 'every', 'knowing']
```

Figure 6: Demonstration of getting a glove vector from the dictionary

2.2.3 Design loss and metrics

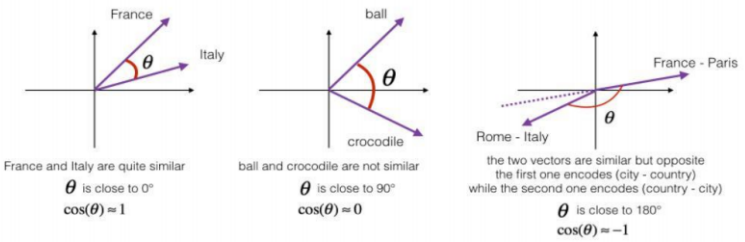
Since it is now a regression problem, using accuracy metrics does not make sense at all. As glove vectors were trained in terms of similarity between meanings of words, similar words would be closer to each other in terms of euclidean distance. We thereby chose Euclidean Distance as the loss function and cosine similarity as the metric. The cosine similarity is calculated by the dot product of two vectors divided by the product of norms of two vectors.

1 - Cosine similarity

To measure how similar two words are, we need a way to measure the degree of similarity between two embedding vectors for the two words. Given two vectors u and v, cosine similarity is defined as follows:

$$\text{CosineSimilarity(u, v)} = \frac{u.v}{||u||_2||v||_2} = cos(\theta) \tag{1}$$

where $u.v$ is the dot product (or inner product) of two vectors, $||u||_2$ is the norm (or length) of the vector u, and θ is the angle between u and v. This similarity depends on the angle between u and v. If u and v are very similar, their cosine similarity will be close to 1; if they are dissimilar, the cosine similarity will take a smaller value.



France and Italy are quite similar
θ is close to 0°
$\cos(\theta) \approx 1$

ball and crocodile are not similar
θ is close to 90°
$\cos(\theta) \approx 0$

the two vectors are similar but opposite
the first one encodes (city - country)
while the second one encodes (country - city)
θ is close to 180°
$\cos(\theta) \approx -1$

Figure 1: The cosine of the angle between two vectors is a measure of how similar they are

Figure 7: Demonstration of cosine similarity

3 Training Practices

3.1 Architectural design

We first constructed two LSTM layers with output of 128 x 1 vector. Since the second LSTM relies on the output of the first LSTM, we let the first LSTM layer return sequence. We also inserted a dropout layer into the two LSTM layers to prevent overfitting. Afterwards, since we want the output to have the same dimension as a glove vector, a fully connected layer with 50 neurons is followed at the end. One thing worth pointing out is that we do not use the softmax layer at the end. The reason

is that we have turned the prediction problem into the regression problem, and the sum of a glove vector's elements does not add up to 1 for sure.

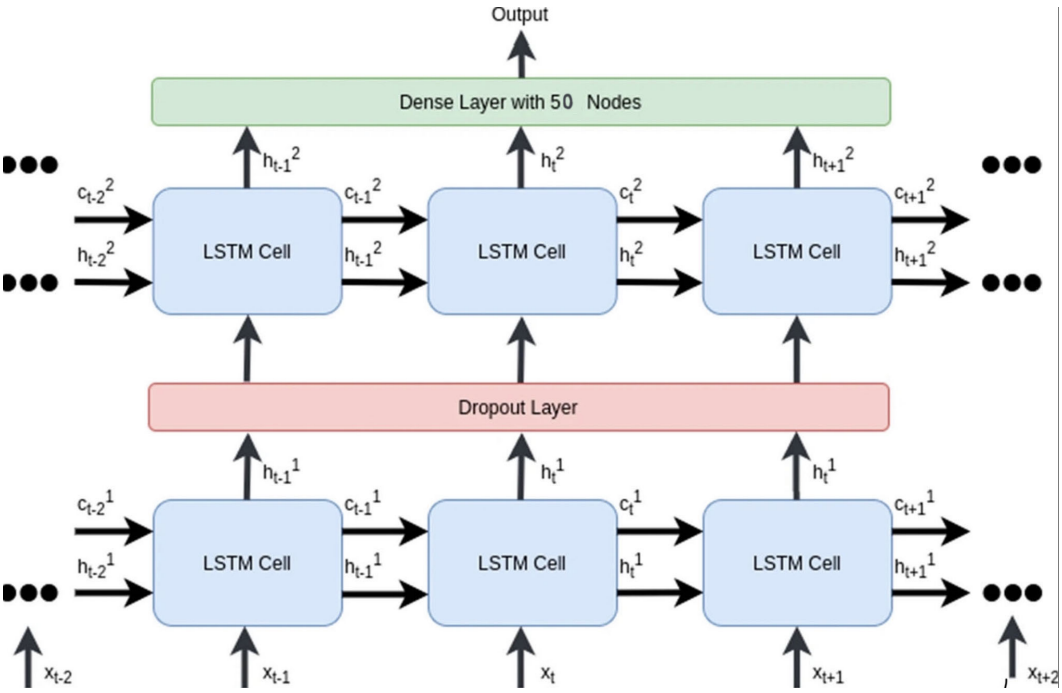


Figure 8: Visualization of our LSTM architectural design

3.2 Techniques we used

3.2.1 Regularizations

For preventing overfitting, we used a dropout layer and early stopping. The rate of dropping out is a hyperparameter we can tune, and we just set it to be 50%. Technically, the early stopping should monitor validation cosine similarity, but since the data set is too small and the interest of our application is for sentences the model has seen before (user have typed before), we just let early stopping monitor training cosine similarity. Patience, number of epochs to wait before the training is ended by early stopping, is set to be 20, which means that if over the next 20 epochs, the training cosine similarity is not improved, then the training will be ended mandatorily and weights for best training cosine similarity would be restored.

3.2.2 Optimizations

We first used Rmsprop to normalize the norm of gradient at current parameters. Gradient with momentum was also implemented to neutralize some noises of newly computed gradient vector. Finally, we utilized the learning rate exponentially decay for achieving the minimum more efficiently.

3.3 Training Results

It is clear from the images below that the training loss keeps going down and training accuracy keeps increasing. On the other hand, the validation loss decreases in the first about 8 epochs, and then it starts to increase and fluctuate. Similarly, the validation similarity keeps going up in the first about 8 epochs. After reaching the maximum, it starts to decrease and fluctuate. The model does suffer severe overfitting very quickly, but it is not due to the problem of architecture or feature engineering, but due to the small size of the data set. By the way, it is worth noting that the training similarity seems to be extremely high in the first epoch, but it is because of the value of learning rate. If the

learning rate was set to be 0.00001, then the training similarity in the first epoch is about 10%, though it increases extremely fast.

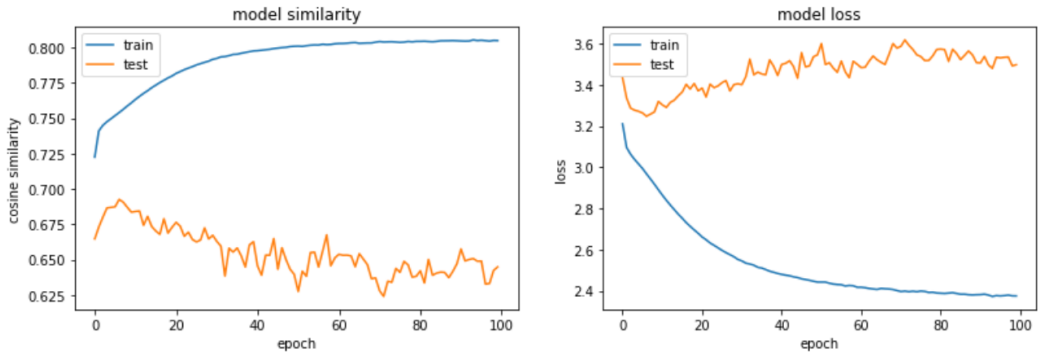


Figure 9: Model similarity and loss with the number of epochs we have trained

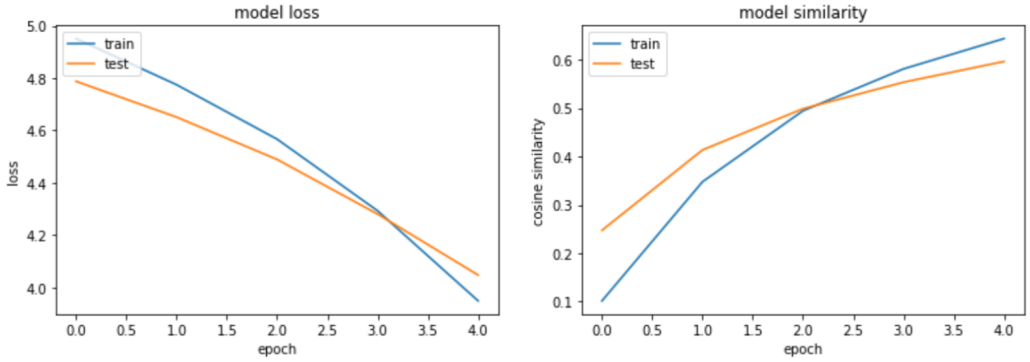


Figure 10: Model similarity and loss when we alter learning rate to 0.00001

4 Testing Results

In this section, we tested our model generalized performance in next word prediction tasks. We randomly indexed X to get one feature such as X[i], which is a 5-word sentence represented by 5 glove vectors. We call the output of our model y_hat, which is a 50 x 1 vector. We then pass y_hat in our helper function, closest_euclidean, which returns the list of words in order of euclidean distance from the output. At this point, we want the model to predict 5 words for users to choose, so we just return the first 5 words from the list. We compared it with the true label of the next word in the testing. If the list of 5 words includes the true label, then the prediction is considered as a success. As a result, we got 6 successful predictions out of 9 total sentences. Note that the context of this application is for words we have typed before, which means the words the model have not seen before, testing data, could be considered irrelevant in this case.

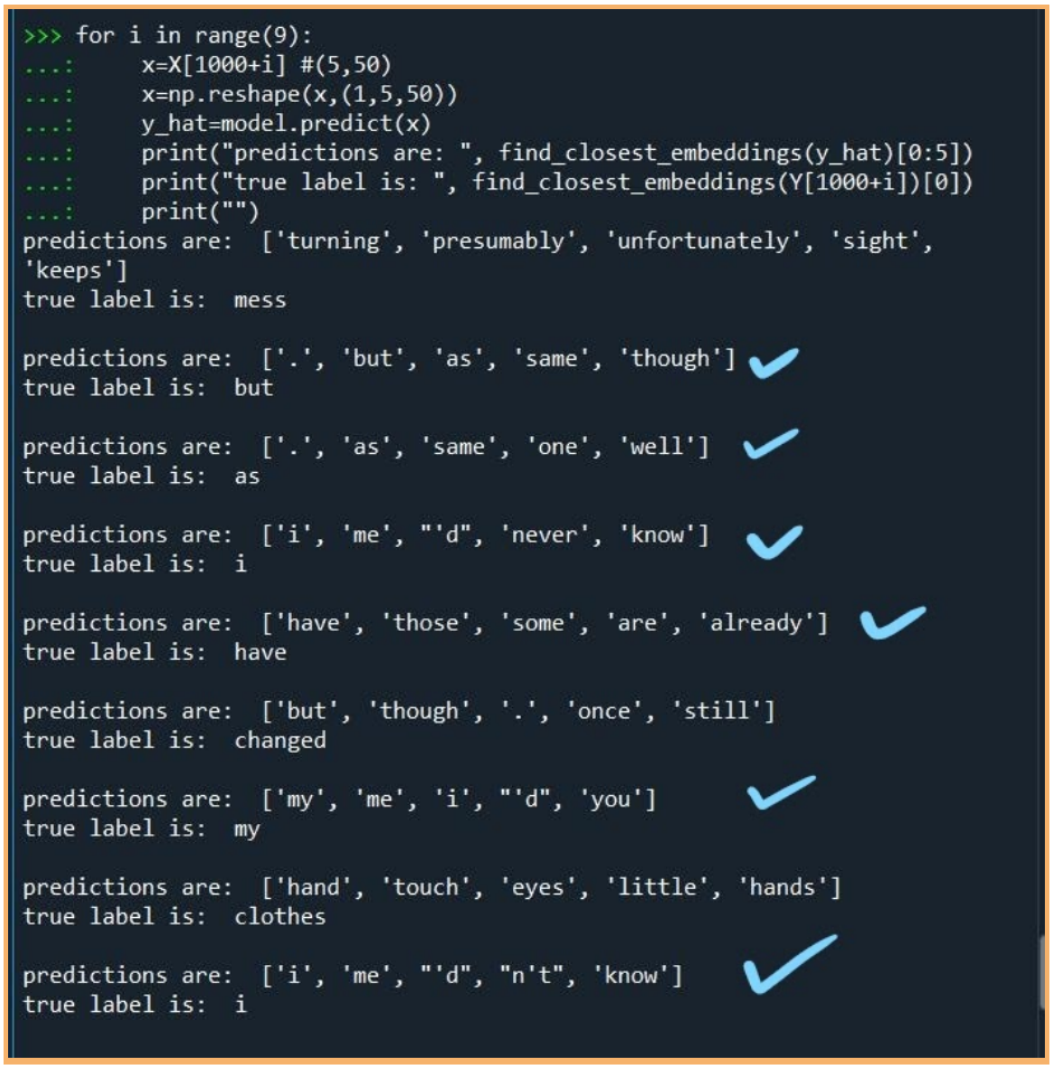


Figure 11: Correct testing predictions are checked for our 9 trials

5 Future Work

5.1 Imbalanced Data

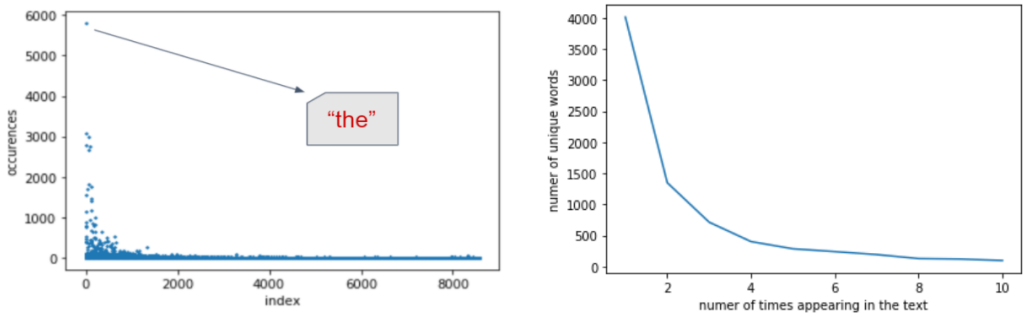


Figure 12: Two ways of visualization of imbalanced data

As we previously mentioned in the data processing section, we have sorted out 8576 unique words as our data labels. We can visualize its distribution from the figure above. We may notice some words appear much more frequently than others and some infrequently used words may even appear only once in our dataset. For example, the word "the" appears more than 5500 times and it may largely mislead our prediction model.

To address this issue, we could introduce larger sized data to mediate this problem. We could also use data augmentation and weighted loss to further our research on improving our model prediction accuracy.

5.2 Punctuations Removed

During our data preprocessing stage, we removed all the punctuations for a simpler data source. However, punctuation matters to the meaning of sentences. We would like to include punctuations into our set of classes and further improve our prediction accuracies.

5.3 Potential drawback of using pre-trained GloVe Vectors

GloVe vectors are pretrained vectors, so they may not be the best representation of words in our corpus. Ideally, it may be better to train vector representations of words via approaches such as the word embedding layer from Keras.

6 Conclusion

This research of building the next word prediction model could lead to the development of more user-friendly typing assistants. People could send more instant messaging and catch the critical time they may miss. In the future, we would like to integrate our designs into more advanced fields such as many to many translation (requires addition of decoder) and text auto-corrections (switch to character-level based predictions).

References

[1] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. ICLR, 2015.

[2]dprogrammer, Author. "RNN, LSTM & Gru." DProgrammer Lopez, 9 June 2020, dprogrammer.org/rnn-lstm-gru.

[3]"Fisseha Berhane, Phd." Operations on Word Vectors - v2, datascience-enthusiast.com/DL/Operations_on_word_vectors.html.

[4]Rabby, Md Fazle, et al. “Stacked LSTM Based Deep Recurrent Neural Network with Kalman Smoothing for Blood GLUCOSE PREDICTION.” BMC Medical Informatics and Decision Making, BioMed Central, 16 Mar. 2021, bmcmedinformdecismak.biomedcentral.com/articles/10.1186/s12911-021-01462-5.