Fake news detection through Long Short-Term Memory

Abstract:
The object of this project is for detecting fake news by reading its titles. To detect it, Long Short-Term Memory model was used. Text data was preprocessed by using word2vec and tokenizer. The accuracy of the final model was higher than 0.8. It detected the real news well, but it did not detect the fake news well due to the data imbalance. Therefore, there seems to be a need for finding other ways to compensate for the problem.

Name: SOHYUN LEE
Student ID: 25637733

Colab Link:
https://colab.research.google.com/drive/1ciTwnzOkk2sFwDzADa486KwknZ_KkkW_?usp=sharing

Ⅰ. Introduction

The spread of false information is getting worse. This phenomenon can cause confusion in society and cause a negative frame to certain groups. It is important to analyze the content of news directly and identify yourself whether it is false information, but this is difficult and takes amount of time. My goal is to make a deep learning model which detects fake news through the title of it.

Ⅱ. Method

1. Data collection

I used 'Fake News' dataset from Kaggle for model implementation. It consists of 23196 rows and 5 columns, and each column consists of 'title', 'news url', 'source domain', 'tweet num', and 'real'. 'real' is a label column, where 1 is real and 0 is fake(Golovin, 2022.).

2. Data preprocessing

The dataset had 1472 duplicate titles, so I removed them. After removing them, the dataset had 327 missing values in 'news url' and 'source domain' columns. Since the number of missing values is only part of the entire dataset, I also removed them from the dataset. Thus, 21397 rows were left.

The dataset has both real and fake news data, but the ratio of the two is imbalanced.

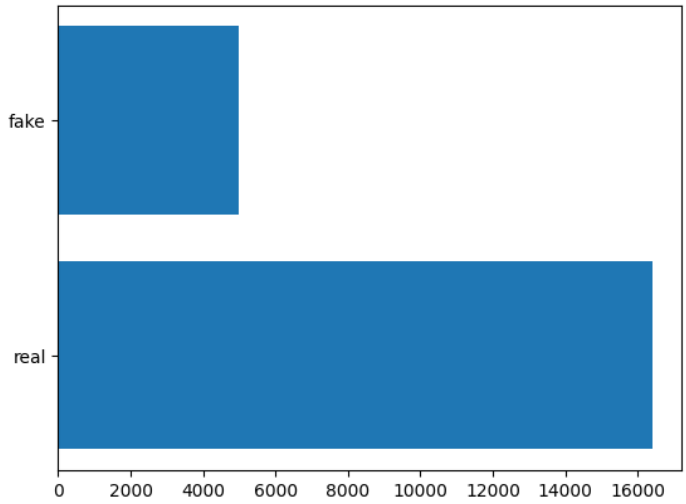

Figure 1: Data imbalance

The imbalance can cause model can't learn enough about 'fake' labels(Google Developers, 2024). To overcome this problem, I decided to use inverse-Frequency class weights for each class. The method to calculate weights is as below,

$$w_i = \frac{N}{K \sum_{n=i}^{N} t_{ni}}$$

where N is the number of samples, K is the number of classes, w_i is the weight of the class i, and t_{ni} is the indicator that nth sample belongs to the ith class(MathWorks, 2024b). According to the method, the calculated weight of

class 0 was about 2.13 and that of class 1 was about 0.65.

To preprocess title, that is, text data, I removed non-alphabet symbols like '!\"#$%&'()*+,-./:;<=>?@[\]^_`{|}~'. Also, I changed all capital letters of the alphabet to lowercase and lemmatized each word. Finally, I removed stop words from each title. In the case of source domain, I only left the main name of each site.

Since machine learning model cannot read text data directly, those data should be encoded into numerical data. In the title, I decided to use word2vec for vectorizing each word. Also, I used tf.keras.preprocessing.text.Tokenizer for indexing. The tokenizer converts text to numeric sequences by allocating a unique index to each word and the word2vec transforms those sequence indexes into embedding vectors. Therefore, the model can reflect not only the order of words but also the semantic relationship between words. To unify the length of each sequence generated by indexing, I extended the length of all sequences to the length of the longest sequence through zero padding. In the source domain, I used OneHotEncoder which encodes categorical features as a one-hot numeric array.

3. Model Implementation

The model I decided to use was LSTM(Long Short-Term Memory), which is a type of recurrent neural network. It excels in sequence prediction tasks, capturing long-term dependencies.

A. Title

First, I only used title data for training. I used sequential API to build a model. The model I built consists of an Embedding layer, a LSTM layer, and a Dense layer. The Embedding layer is for converting each word to a fixed size vector. Vectorization is based on the pre-trained word2vec model. The LSTM layer is for processing such vectorized data, and it has 32 units. The Dense layer is for output, and I used sigmoid as activation function since the task of the model is binary classification.

Model: "sequential"

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 35, 100)	1,829,600
lstm (LSTM)	(None, 32)	17,024
dense (Dense)	(None, 1)	33

Figure 2: Model structure 1

The number of parameters in the Embedding layer is

18296($the\ number\ of\ word$) x 100=1829600

The number of parameters in the LSTM layer is

(100 x 32($the\ number\ of\ LSTM\ units$)+32 x 32+32)

x 4=17024

The number of parameters in the Dense layer is

32($the\ number\ of\ LSTM\ units$) x 1($the\ number\ of\ Dense\ units$)+1=33

I used Adam optimizer and binary cross entropy loss to compile the model. Adam optimization is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments(TensorFlow, 2024). Binary cross entropy loss is generally used in binary classification. It tracks incorrect labeling of the data class by a model and penalizes the model if deviations in probability occur into classifying the labels(Roberts, 2023). I set the batch size to 32 and set the model to train for 20 epochs. I also set EarlyStopping with patience as 5 so that the model can stop training if the validation loss does not improve for 5 continuous epochs.

B. Title + Source

Next, I also included source data for training to increase the performance of the model. Since two data should be input, I decided to use functional API instead of sequential API. The functional API can handle models with multiple inputs and outputs. I set two types of inputs. One is title, and the other is source. In the case of title, I used an Embedding layer and a LSTM layer so that those titles can be processed. In the case of source, I used a Dense layer, and the activation function was ReLU. The layer has 16 units. Then, I concatenated two outputs derived from each input and used them as an input of a final Dense layer. The activation function of the Dense layer was sigmoid.

Model: "functional_1"

Layer (type)	Output Shape	Param #	Connected to
title_input (InputLayer)	(None, 35)	0	-
embedding_1 (Embedding)	(None, 35, 100)	1,829,600	title_input[0][0]
source_input (InputLayer)	(None, 1866)	0	-
lstm_1 (LSTM)	(None, 32)	17,024	embedding_1[0][0]
dense_1 (Dense)	(None, 16)	29,872	source_input[0][0]
concatenate (Concatenate)	(None, 48)	0	lstm_1[0][0], dense_1[0][0]
dense_2 (Dense)	(None, 1)	49	concatenate[0][0]

Figure 3: Model structure 2

The numbers of parameters in the Embedding layer and the LSTM layer are the same as the previous model. The number of parameters in the Dense layer for the source data is

1866($source\ dimension$) x 16($the\ number\ of\ Dense\ units$)+16=29872

The dimension of concatenated vector is

32($the\ number\ of\ LSTM\ units$)+16($the\ number\ of\ Dense\ units$)=48
Therefore, the number of parameters in the final Dense layer is
48($Concatenate\ output\ dimension$) x
1($the\ number\ of\ Dense\ units$)+1=49

Like the previous model, I used Adam optimizer and binary cross entropy loss for compilation. Likewise, I set the batch size to 32 and the epochs was 20. EarlyStopping with 5 patience was also set.

Ⅲ. Result

In the first model, the history of train dataset's loss and validation dataset's loss is as below.

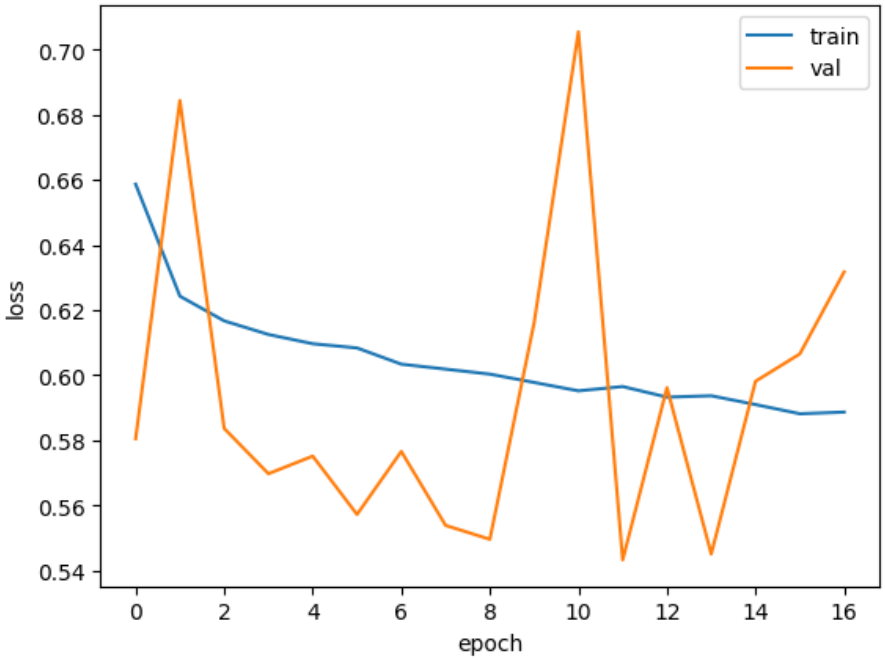

Figure 4: Loss history 1

The tendency of train loss was getting lower per epoch, but the loss of validation was variable. To evaluate the model, I compared the test data labels with the model's predictions on the test data. Since the model only predicts probabilities, I set the threshold which is 0.5 so that the prediction can be either 0 or 1 depending on whether it exceeds the threshold. The classification report of the model's prediction is as below.

	precision	recall	f1-score	support
0	0.48	0.55	0.52	983
1	0.86	0.82	0.84	3297
accuracy			0.76	4280
macro avg	0.67	0.69	0.68	4280
weighted avg	0.77	0.76	0.77	4280

Figure 5: Classification report 1

According to the classification report, the accuracy of the model is 0.76. It is not a low level but looking at the part where the label is 0, it can be guessed that this model did not classify properly. Its recall and f1-score are around 0.5 and precision is even lower than 0.5. It means that the model didn't learn about enough 0 class even though I used inverse class frequency method.

The history of train dataset's loss and validation dataset's loss in the second model is as below.

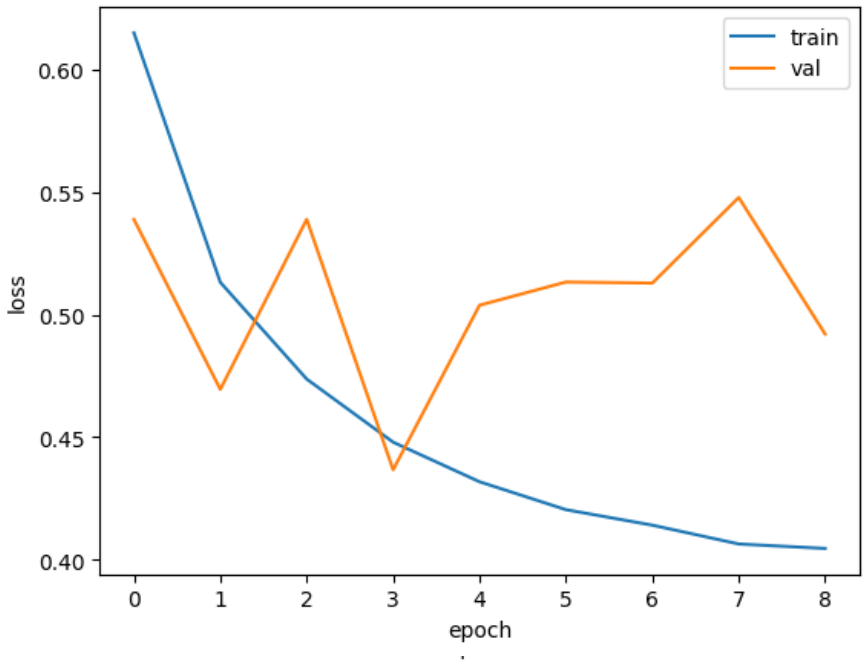

Figure 6: Loss history 2

Like the previous model, the tendency of train loss was getting lower, and the validation loss fluctuated. The classification report of the model's prediction is as below.

	precision	recall	f1-score	support
0	0.64	0.63	0.63	983
1	0.89	0.89	0.89	3297
accuracy			0.83	4280
macro avg	0.76	0.76	0.76	4280
weighted avg	0.83	0.83	0.83	4280

Figure 7: Classification report 2

The accuracy of the model is 0.83, which is higher than that of the previous model. Also, the precision, recall, and f1-score about 0 class are higher than those of the previous model. All of them are not lower than 0.6. Therefore, it is shown that the shortcomings of the previous model were improved by adding source data.

Ⅳ. Conclusion

Although the accuracy of the final model is higher than 0.8, there is a limitation. As I mentioned before, the model can detect real news, but it does not detect fake news well. The main reason for this problem is data imbalance. Since the number of real news is much higher than the number of fake news, the model did not learn fake news data properly. I already used inverse frequency weighting method to overcome it, but the method did not resolve the cause completely. Therefore, to improve the model's performance, other methods like undersampling or oversampling would be used.

References

Golovin, A. (2022). *Fake News*. Kaggle. https://www.kaggle.com/datasets/algord/fake-news

Google. (2024. August 13). *Datasets: Imbalanced datasets*. Google Developers. https://developers.google.com/machine-learning/crash-course/overfitting/imbalanced-datasets

MathWorks. (2024b). *Train sequence classification using inverse frequency class weights*. MathWorks. https://kr.mathworks.com/help/deeplearning/ug/sequence-classification-using-inverse-frequency-class-weights.html

Roberts, A. (2023, January 1). *Binary cross-entropy: Where To Use Log Loss In Model Monitoring*. Arize AI. https://arize.com/blog-course/binary-cross-entropy-log-loss/

TensorFlow. (2024 June 7). *tf.keras.optimizers.Adam*. TensorFlow v2.16.1. https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam