Memorial University of Newfoundland

Faculty of Science

Department of Mathematics and Statistics

Multi-class Text Classification to predict the rating scores for TV-Series: A Comparison of Feature-based approach vs Deep Learning approach

By Inwook Back (201992624)

A PROJECT SUBMITTED TO THE DEPARTMENT OF MATHEMATICS AND STATISTICS, MEMORIAL UNIVERSITY OF NEWFOUNDLAND IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR THE DEGREE OF MASTER OF APPLIED STATISTICS (STATISTICS)

Aug 11, 20

ACKNOWLEDGEMENTS

I would first like to thank my supervisor, Dr. Zhaozhi Fan who gave me a great opportunity to study at MUN. You always supported and trusted me to keep studying with fun. I have learned so many things with this Master program at MUN. Also, I would like to thank the professors who taught me during the coursework. Many ideas and technical parts that I used in this study are based on what I learned from all courses.

Contents

List of Figures

List of Tables

Abstract

The main goal of this project is to figure out what really is important for multi-class text classification. Normally, there are many steps that are assumed to be important to solve this problem, such as text pre-processing, feature engineering, choosing proper algorithms, and hyper-parameters tuning. So, I have tried to make clear the importance of each step for the modeling performance. To achieve this goal, I collected the raw review data from the web, which contains 250K reviews from 5,000 TV-series, and sub-sampled 38,000 reviews. The tasks are mainly two parts. The first one is a feature-based approach that includes detailed feature engineering, such as extracting features from the texts. The other one is to simply run a latest deep learning model with basic pre-processing and the fine tuning of the algorithm. And I compared the performance of those two different ways to figure out a reliable method for text multi-classification. In addition, Bayesian Optimization was deployed for hyper-parameters tuning, which was meaningful for the Support Vector Machine. As a result, both approaches gave very similar results whose accuracies are all about 48%. But a noticeable result is that their predictions are very different from each other, and when ensembling these models, the accuracy increased from 48% to 53%. This implies that ensembling both ways can result in much better result than simply using one approach.

Introduction

There are some fancy neural network algorithms for text classification, such as Long short-term memory (LSTM) and Bidirectional Encoder Representations from Transformers (BERT). These actually bring us great results in many text classification problems, for example, the

accuracies of question-answering matching task or machine translation are very reliable. Even just putting raw text with basic pre-processing to the model gives surprisingly high accuracy.

In this situation, my concern is that what if the text has very delicate nuisance to classify the classes. Many previous sentiment classifications on movie review data just used high rating samples (8, 9, 10 out of 10) and low rating samples (1, 2, 3 out of 10), and to classify these two groups are relatively simple. And many other tasks, such as, question-answering matching have some clear patterns such as a bag of words to make the classification work well. So, I bring somewhat tricky problem to investigate the performance of these algorithms. The data is 250K raw reviews from 5000 TV-series which are not pre-processed at all, and I sub-sampled 38,000 reviews. The aim is to classify peoples' actual ratings in 1 to 4 scale, which means there are 4 different classes. Because the data is raw, a careful sampling strategy or pre-processing is necessary.

The main interest in this study is to find out what really is important for text multi-classification. There are many steps for this task, such as pre-processing of text, feature extraction, proper algorithms and hyper-parameters tuning. These are normally recommended steps, so, I investigated their impacts on the modeling result. To do this, I have taken much time to extract good features from the raw text. And I compared the accuracies of feature-based approach and that of deep learning approach. The best model was made from stacking both of the models.

The followings are goals of this study

1. To maximize the accuracy of multi-classification for rating scores
2. To extract features from texts

3. To compare feature-based approach and deep learning approach

1. Pre-settings

1.1 Target Variable

The target variable is rating score for TV-series. The original values are from 1 to 10 scale. But the aim of this study is multi-classification using texts, so I grouped them into 4 classes.

- 1 : 1 ~ 3 scores (very bad)
- 2 : 4 ~ 6 scores (bad)
- 3: 7 ~ 8 scores (good)
- 4: 9 ~ 10 scores (very good)

1.2 Sub-sampling

Because of the heavy computation, I reduced the sample size from 260K to 38,000 and from 5,000 TV-series to 216 TV-series. The followings are sub-sampling scheme.

1. Choose only the TV-series that has at least 30 samples for each target class
2. The maximum sample size for each target class is restricted to 50. So, a TV-series can have up to 50 x 4 = 200 samples total.

By doing this, we can obtain reasonably big “Bags of words”. The distribution of target classes is close to even, and samples will not be drawn from only a small number of TV-series. This

can be helpful to randomize confounding effects. In addition to this, I removed too short review and too long reviews. Reviews that have words from 15 to 400 were kept.

1.3 Text cleaning and pre-processing

Normally, raw text has many sources of noise. So, the goal of pre-processing is to alleviate these noises. For example, there are non-alphabetical characters, like " / ", "….", or many named entities. And the other main issue is stemming. Each word has an impact on the target variable, so it is very important to assure enough sample size to investigate the word effect. For example, "surprise" and "surprises" need to be merged into a single word to make the sample size larger to investigate the effect of the word "surprise".

The followings are pre-processing steps that I used.

1. Lower the case

   ex) Apple -> apple

2. Stemming

   ex) is, are, was -> be / amazed -> amaze

3. Cleaning all the abbreviations (Total 110 kinds of abbreviations were fixed)

   ex) should've -> should have / Theres - > There is

4. Remove the stop-words when doing feature extraction

   ex) The, a, on, … -> Remove all

5. Change the casting or characters' names into "pplnme"

ex) John Smith -> pplnme

1.4 Distributions of target variable and review texts

As I sub-sampled evenly for each target class, all 4 classes have the same sample sizes.

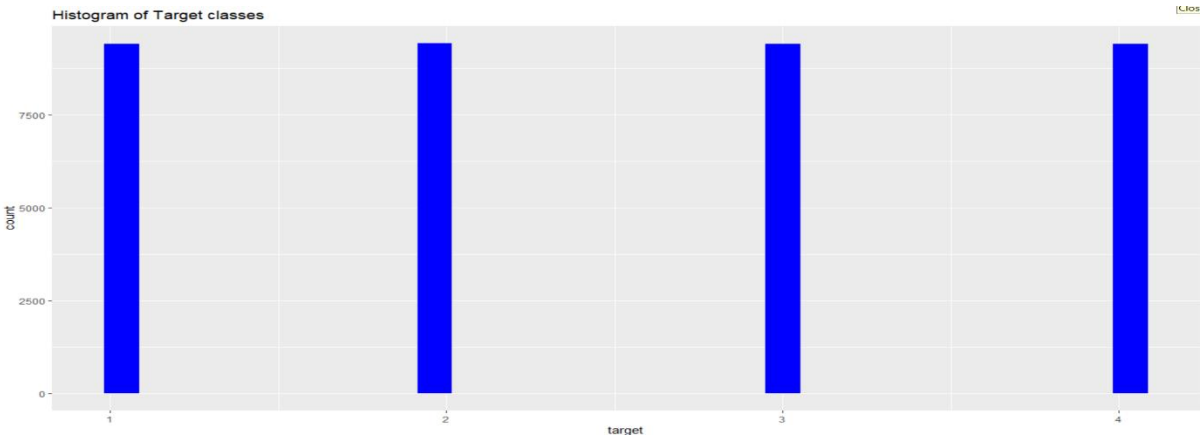

Figure 1. Distribution of Target Variable

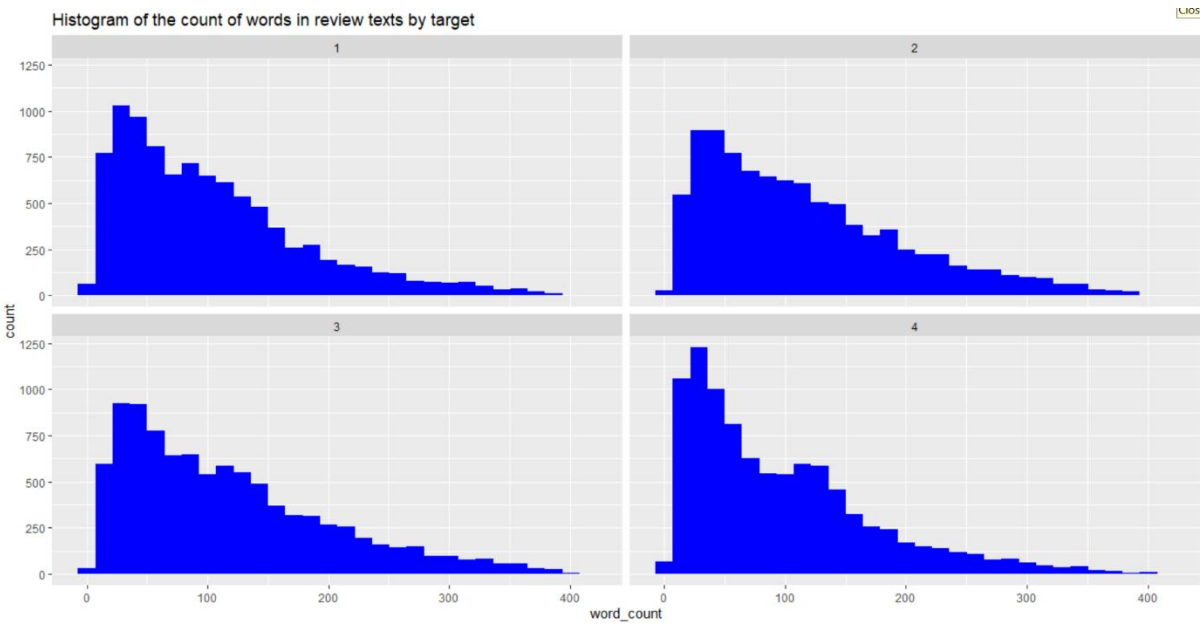

Figure 2. Distribution of word length by Target

From the Figure 2, number of words are skewed to the right, and their distributions are all similar. The mean and median values are 108 and 90 words, respectively.

2. Feature-based modeling

Except the deep learning models, many other ML algorithms require features rather than raw text. To deploy these models, feature extraction from the text is required. In this task, sentiment score can be a reliable measure to predict rating scores. There are some reference lists that have word and sentiment score columns, such as ‘bing’ and ‘afinn’. To calculate these scores, first I built a bag of words and only kept the words that are in the list of sentiment references. But this bag of words is not enough, which means many samples do not have the words in the bag at all. To supplement this, I used all the words list in the ‘bing’ and ‘afinn’ as a secondary sentiment score.

word	value
abandon	-2
abandoned	-2
abandons	-2
abducted	-2
abduction	-2
abductions	-2
abhor	-3
abhorred	-3
abhorrent	-3
abhors	-3
abilities	2
ability	2
aboard	1
absentee	-1

Figure 3 sentiment words and their values from 'Afinn' and 'Bing', 7,946 words

2.1 Bag of Words for each target class

Normally, a bag of words is collected keeping high frequency words in each target class or using TF-IDF scores. But neither of those can be used for this case, because all target classes share very similar high frequency words, such as ‘story’, ‘the show’, ‘time’ etc. That is, the bag of words for each target class is not distinct.

To solve this problem, I assumed that middle range of ratings(target = 2 or 3) can be expressed by weaker emotions than more extreme ratings(target = 1 or 4), or mixed expressions of those two extremes. So, there are three bags of words, which are ‘Good’, ‘Medium’, and ‘Bad’.

The following is the process to build the bag of words.

1. Collect the top 500 most frequent words by TV-series (Stop-words removed)
2. Good-bag : From target = 4 bag, remove words in target = 1 bag

   Bad-bag : From target = 1 bag, remove words in target = 4 bag

   Medium-bag : From target = 2 or 3 bag, remove words in target = 1 or 4 bag
3. Randomly choose 5 TV-series and their bags, and only collect the intersected words out of these 5 TV-series bags.
4. Repeat 1~3 steps 2000 times, and make a frequency table of all the intersected words.

fantastic	perfect	excellent	awesome	favorite	brilliant	recommend
171	164	155	149	125	107	104
wait	hook	surprise	binge	negative	highly	incredible
100	75	66	41	36	35	34
intrigue	amaze	enjoyable	job	realistic	twist	edge
25	24	22	22	22	22	21
glad	fresh	happy	opinion	performance	wonderful	superb
21	20	19	18	18	18	17
television	complex	fun	intense	masterpiece	unique	finally
17	14	14	14	13	13	12
role	beautifully	outstanding	perfectly	true	heart	strong
12	11	11	11	11	10	10
week	beautiful	develop	grip	grow	moment	relationship
10	9	9	9	9	9	9
simply	storyline	worth	blow	excite	forward	easy
9	9	9	8	8	8	7

Figure 4. Bag of Words (Good)

waste	awful	terrible	garbage	horrible	crap	poor
714	362	358	271	168	167	160
stupid	fail	ridiculous	dumb	attempt	lack	bother
137	130	65	55	51	48	46
complete	mess	badly	force	disappointment	predictable	suppose
39	34	32	32	30	30	30
ruin	decent	annoy	bunch	cheap	nonsense	poorly
29	26	24	24	20	20	20
barely	sense	disgust	dull	anymore	idea	minute
17	17	15	15	14	14	14
talk	appare	insult	basically	death	lame	money
14	13	12	11	11	11	11
suck	brain	cut	drag	call	half	shame
11	10	10	10	9	9	9
weak	wrong	actual	dialog	dialogue	fake	save
9	9	8	8	8	8	8

Figure 5. Bag of Words (Bad)

fairly	final	short	sound	bland	dimensional	due	element
26	8	8	8	7	7	7	7
major	crazy	distract	drop	flat	improve	agree	average
7	6	6	6	6	6	5	5
difference	difficult	frustrate	hold	solid	watchable	3rd	arc
5	5	5	5	5	5	4	4
carry	convince	credit	focus	introduce	single	spend	15
4	4	4	4	4	4	4	3
alive	atmosphere	background	body	check	choose	concern	conflict
3	3	3	3	3	3	3	3

Figure 6. Bag of Words (Medium)

From the bags of words, many words seem to be reasonable for each category. But still there are many words that would be unrelated to the target, which could make noise. So, only the words that are both in the bags and sentiment references (bing and afinn) were kept.

2.2 Sentiment score

The frequencies in the table could directly become weights to calculate sentiment scores, because the table indicates kind of importance for each target class. But the variation is very large, so I took log-transform to the frequency. The remaining job is to calculate a sentiment score for each row. If a row has those words in the lists, their weights are all summed.

One important problem is negation. If there is ‘not, no or never’ then, the meaning becomes opposite. To alleviate this problem, first I removed sentences that contain negation words and calculated sentiment scores. After that, for the row that returned ‘NA values’ due to the removal of negations sentences, which means no sentences are kept, I calculated sentiment scores again with negation sentences included. Negation can add noise, but it is better to have them than to keep many rows have ‘NA values’.

As a result, there are two kinds of sentiment scores. One is primary one from the bag of words, and the other is secondary one to supplement the primary one from general sentimental words.

About the medium bag of words, this does not seem to be useful when looking through the frequency table. And those words are not related to sentiment, so they cannot be combined with the sentiment reference. To check this usefulness, I tried the ‘Residual Deviance Lack of fit Test’ for each word with a baseline model. Out of 342 words, 118 words were significant. But the result was to discard them, because they did not improve the accuracy, which means medium bag of word would be just noise or a single word would not be enough to form a pattern.

2.3 Hyper-parameters tuning

In many ML algorithms, hyper-parameters affect the performance of the model much. The followings are ways for this.

1. Grid search
2. Random search
3. Bayesian optimization

Grid search takes too much time, and random search is not very reliable, so I chose the Bayesian optimization approach.

2.4 Theoretical background of Bayesian Optimization

There are two main concepts, one is prior distribution, and the other is acquisition function.

- X : Hyper-parameters space
- Y : f(X), error value given X input
- Prior: Gaussian Process, {Xn, Yn}, where $Yn \sim N(f(Xn), \nu)$
- a(X ; {Xn, Yn}, θ), acquisition function to determine next X input,

  GP Upper Confidence Bound

  $a(x ; \{Xn, Yn\}, \theta) = \mu(X ; \{Xn, Yn\}, \theta) - \kappa\, \sigma(X ; \{Xn, Yn\}, \theta)$
- $X_{next} = \text{argmax } a(X)$
- $X_{best} = \text{argmin } f(Xn)$

For the prior, Gaussian Process was assumed, because GP gives the closed form of marginal and conditional distributions. And cross-entropy function with 10-folds cross-validation was used for the error function f( ). After this setting, acquisition function searches the best hyper-parameters set, updating the best next guess, given the history of hyper-parameter sets and corresponding error values. The followings are modeling results of initial hyper-parameters setting and that of Bayesian optimization.

	Initial set	**Optimized set**
XGBOOST	48.5%	48.2%
SVM	42.3%	48.2%

Table 1. Accuracy by different Hyper-parameters set

From the result, there was no improvement for XGBOOST, but SVM showed a large difference by the different hyper-parameters sets. And from the fact that those algorithms and the result of deep learning all gave very similar results, the result of Bayesian optimization can be reasonable. So, when we do not have a reasonable guess about the hyper-parameters set, this optimization can be a strong alternative to tackle this problem.

2.5 Feature importance

The following are model features and their importance values.

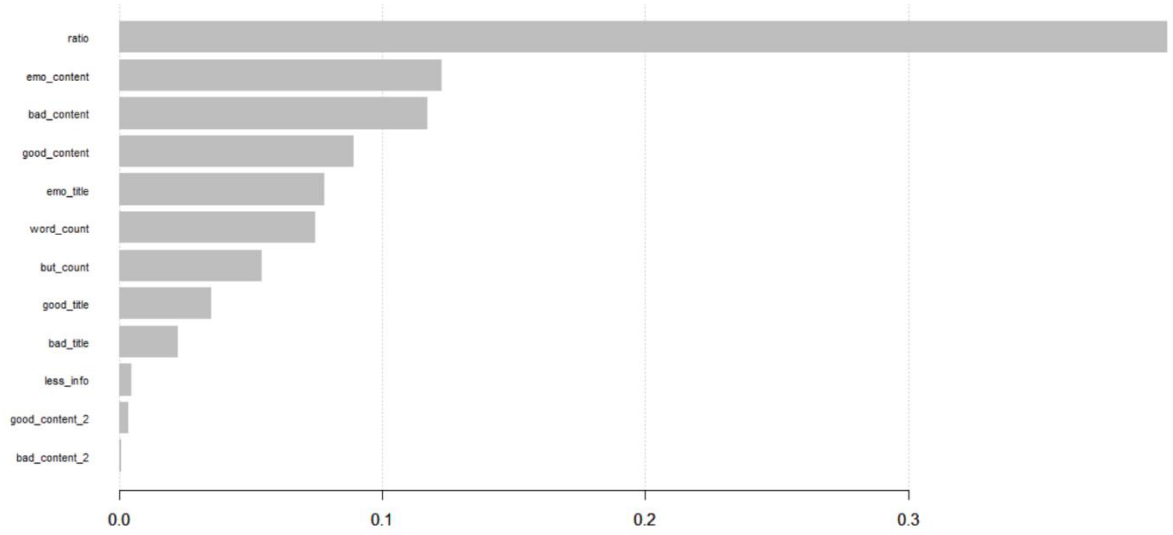

Figure 7. Feature Importance Plot

- Ratio : Represent the ratio of counts of the “good expressions” and “bad expressions”

- _content : Sentiment scores calculated from main text

- _title : Sentiment scores calculated from review title

- emo_ : Sentiment scores calculated under general sentiment reference 7946 words, not from the bags of words

- but_count : Number of “but, however” appeared

- less_info : 1: if there is no word to calculate sentiment scores.

- _2: Sentiment scores from bi-gram reference, for example, “Worth watching”

Many features are sentiment scores, so they are correlated each other. This is the reason why tree-based gradient boosting algorithms are good choices to model, considering their

robustness for multi-collinearity. And, the depth of tree would be deep in this case, because there seems to be many times of binary splitting.

3. Deep Learning based modeling

In many practical problems, deep learning algorithms have proved their excellence. Even just putting sentences into the machine without any detailed processing results in high accuracy. So, I tried to test its performance when a delicate understanding of context is needed to predict the target.

3.1 Build three different text inputs

Not all sentences in each row can be input to the machine, because it requires heavy calculation, and the length of words are very different over the samples. I cut the review text to 70 words. So, the other sentences will not be used. My concern is that if I cut to 70 words, then important sentences could be discarded, because there are many unimportant sentences. To alleviate this concern, I made three different inputs. The first one is 70 words from the beginning of the review, the second one is 70 words from the end, and the third one is re-constructed sentences by their importance.

The followings are my assumptions to reconstruct the sentences.

1. Sentences that include the bag of words, then they are important
2. Sentences that include ‘But’ or ‘However’, then they are important
3. Sentences that include sentiment words, then they are important

4. Sentences that include words which are related to ‘Main characters’, ‘Writer,’ ‘Story’, ‘Show’, ‘Acting’ are important

Sentences are sorted by the importance scores calculated by above rules, and only the first 70 words were selected for the input.

3.2 Pre-training

Pre-training is one of the highlights for text modeling. There are many corpuses from Wikipedia, News articles, Twitters and so on. So, we can leverage the dependence structure of words from these corpuses.

Probability and Ratio	$k = solid$	$k = gas$	$k = water$	$k = fashion$
$P(k\|ice)$	1.9×10^{-4}	6.6×10^{-5}	3.0×10^{-3}	1.7×10^{-5}
$P(k\|steam)$	2.2×10^{-5}	7.8×10^{-4}	2.2×10^{-3}	1.8×10^{-5}
$P(k\|ice)/P(k\|steam)$	8.9	8.5×10^{-2}	1.36	0.96

Figure 8. Probabilities about co-occurrence of words, From GloVe: Global Vectors for Word Representation https://nlp.stanford.edu/projects/glove/

By using this dependence structure, the dimension of word features can be reduced much. For example, when there is 100K words features and 10K samples, then the dimension of data set is 10,000 X 100,000 that is very high dimension. But, if we leverage the dependence structures from Wiki corpus and use its pre-trained embedding matrix, then we can reduce the dimension into 10,000 X 300. That is, 100K sparse factor variables are changed into 300-dim continuous features set. So, these continuous word vectors become new inputs for the neural network

architecture.

3.3 LSTM Algorithm

Basically, classification using neural network model is similar to a “Next word prediction task”.

I have read so many bad reviews for this “□□□□”

To predict the blank word, we sequentially input the words from “I” to “this”. If there are four candidates for the blank word, like {“Blue”, “Movie”, “Sea”, “Apple”}, then the algorithm learns from training set, and returns predicted probabilities for each word. In this case, ‘Movie’ would be the most probable. And my task is similar to this.

I was hooked from the start. rating : □

From this sentence the rating value 4 would be more probable than 1. The machine tries to learn every sequential pattern for the target value. And the followings are mechanisms that machine learns the patterns.

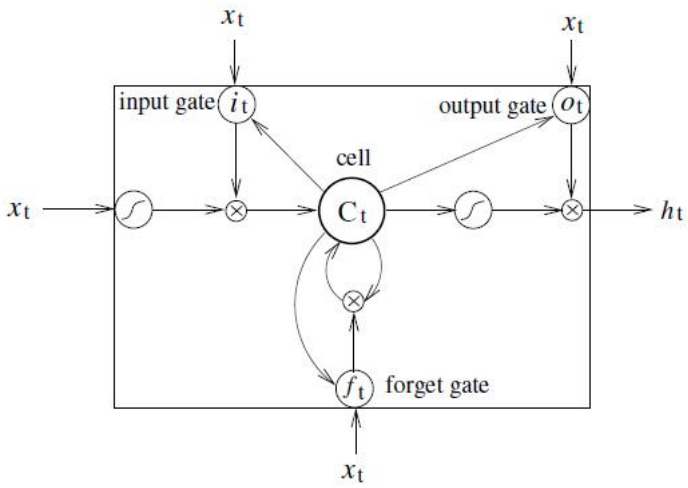

Figure 9. The process of LSTM algorithm, From Z. Huang, W. Xu, and K. Yu, (2015) Bidirectional LSTM-CRF Models for Sequence Tagging

The Figure 5 is the process how LSTM works.

- $i_t = \sigma (W_{xi} x_t + W_{hi} h_{t-1} + W_{cic} t_{-1} + b_i)$
- $f_t = \sigma (W_{xf} x_t + W_{hf} h_{t-1} + W_{cf} c_{t-1} + b_f)$
- $c_t = f_t c_{t-1} + i_t \tanh(W_{xc} x_t + W_{hc} h_{t-1} + b_c)$
- $o_t = \sigma (W_{xo} x_t + W_{ho} h_{t-1} + W_{co} c_t + b_o)$
- $h_t = o_t \tanh(c_t)$

where σ is the sigmoid function, and i, f, o and c are the input gate, forget gate, output gate and cell vectors. So, words are sequentially input to the machine, and only the important patterns are kept by passing through the forget gate. From the back-propagation, W (weights) and b (Bias) are adjusted referencing the loss. In this case, the loss function is the Categorical Cross Entropy function. The final output is calculated using final hidden state(h_t) with the soft-max function.

3.4 Model Architecture

1. There are two embedding layers to change sparse words matrix into dense continuous vectors. In this process, the pre-trained embedding weights are used for the embedding output. One is from Wiki corpus with 6-billion words, and the other is Twitter corpus with 27-billion words. Wiki corpus is formal text and Twitter corpus is more casual, so using both corpuses could give a balanced result. One important thing is that I fixed the embedding weights with pre-trained ones, because sample size is not enough to capture the whole dependence structure of the words in my data set. That is, the embedding weights are not updated during the back-propagate process.

2. For the drop-out ratio, I followed the suggestion in Srivastava et al. (2014), where the value is 0.5 for hidden layers. This will help to prevent the overfitting issue.

3. The outputs from two embedding layers are fed into Bi-directional LSTM layers independently.

4. The outputs from Bi-LSTM layers are concatenated, and then the concatenated one is fed into a final dense layer.

5. From the final dense layer, the probabilities of each target classes are calculated using soft-max function. So, the prediction is the one that has the largest predicted probability.

3.5 Modeling results

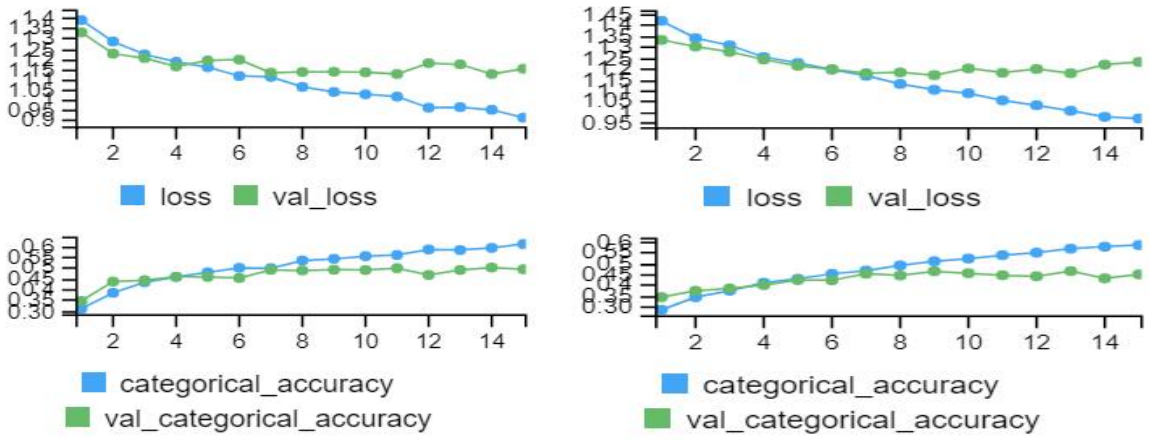

Figure 10. Loss and Accuracy plot for Bi-LSTM training process, the left one is 70 words from tail, and the right one is 70 words from head

There are three different fittings. The first one is to cut the sentences into 70 words from the start, the second one is 70 words from the end, and the last one is 70 words from the re-constructed sentences input. The followings are accuracies of these.

	70 words from beginning	70 words from end	Re-arrayed sentences
Accuracy	46.2%	48.9%	44.7%

Table 2. Accuracy by different input sentences

From the result, choosing 70 words from the end noticeably gives the best result. This means, people tend to write reviews saying their purpose at the end. And, different from my expectation, choosing important sentences does not work adequately either. This result may be because of the fact that the order of sentences is tortured during the re-arraying, and the algorithm could capture very long-term patterns over sentences. Still, the result shows the importance of choosing input sentences, because these differences are not trivial.

4. Comparison of Feature based vs Deep Learning

	Overall	Class 1	Class 2	Class 3	Class 4
Xgboost	48.5%	63.3%	33.7%	33.3%	64.0%
SVM	48.2%	61.0%	37.5%	33.0%	62.0%
Deep Learning	48.9%	61.0%	43.0%	44.1%	48.0%

Table 3. Accuracy by Model

There are three different algorithms, which are XGBOOST, SVM and Bi-LSTM. The winner was Bi-LSTM deep learning algorithm which has slightly better result than the others. One interesting thing is that the predictions from deep learning is very different from the others that used manual feature engineering.

From the accuracy table above, deep learning model gives much more even results through the 4 target classes. In contrast, feature based models give good prediction for extreme target classes (1 and 4) and poor result for middle range (2 and 3). This could be because feature engineering was mainly based on the degree of “Good” or “Bad” emotions rather than the one in the medium emotions. I briefly made a feature from the medium bag of words list, but it did not work well to increase the accuracy.

In contrast, deep learning algorithm would find hidden patterns for middle target range (2 or 3), even though it took the ratio from target 1 and 4. Nonetheless, three algorithms gave very similar accuracy within 1%p difference each other, so I could say it is very hard to predict people’s rating just using review text, because all accuracies were about 48%.

5. Model Stacking

From the accuracy results of three models, there could be an improvement when stacking these models. Because, their overall accuracies are very similar, but deep learning classified middle range well, and the others were good for extreme ranges. It means these classifiers are some independent each other. So, I made a new feature set from the predicted probabilities for each category. That is, 4 features from deep learning, 4 features from XGBOOST, and 1 feature from SVM. The test set was 11297 rows, so I split it 80% of training set and 20% of validation set, and ran a XGBOOST model again with these integrated features. There was a noticeable improvement, that the overall accuracy increased from 48% to 53%.

	Overall	Class 1	Class 2	Class 3	Class 4
Accuracy	53%	67%	40%	41%	64%

Table 4. Accuracy of Stacked model

Conclusion

I tried two different approaches to predict the rating value with review texts. One is feature-based approach that manually creates features from text, and the other relies on deep learning algorithms with simple pre-processing of texts. For feature-based algorithms, I used XGBOOST and Support Vector Machine, and for deep learning, Bidirectional-LSTM was used. As a result, all three gave very similar accuracies, which are 48.5%, 48.2% and 48.9%, respectively. The winner was deep learning algorithm even it needed just simple pre-processing compared to the other two models. So, when we use text data for modeling, considering a huge

amount of effort for detailed feature engineering, just using deep learning algorithm would be a good choice. One interesting thing is the prediction from deep learning is very different from feature-based models. It gave better accuracy for middle target range (2 and 3) with decreased accuracy from extreme range (1 and 4). This means deep learning algorithms captured very different patterns from my feature engineering process. And this was confirmed when stacking those models. The final stacked model increased the accuracy from 48% to 53%, which is great a performance, considering the difficulties of further feature extraction.

Limitation

1. There could be better feature extraction for middle range (target = 2 or 3)
2. Because of the heavy calculation of deep learning, I could not try various settings for deep learning, such as different input length or different model architectures.

References

1. U. Kamath, J. Liu and J. Whitaker (2019), Deep Learning for NLP and Speech Recognition, 1st Edition, Springer

2. T. Hastie, R. Tibshirani, J. Friedman (2008) The Elements of Statistical Learning, 2nd Edition, Springer

3. J. Devlin, M. Chang, K. Lee, and K. Toutanova (2019) BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding

4. J. Snoek, H. Larochelle, and R. Adams (2012) Practical Bayesian Optimization of Machine Learning Algorithms

5. S. Ghosh, O. Vinyals, and B. Strope (2016) Contextual LSTM (CLSTM) models for Large scale NLP tasks

6. Z. Huang, W. Xu, and K. Yu, (2015) Bidirectional LSTM-CRF Models for Sequence Tagging

7. N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov (2014) Dropout: A Simple Way to Prevent Neural Networks from Overfitting

APPENDIX: CODES

```
# Data manipulation
library('dplyr') # data manipulation
library('readr') # input/output
library('data.table') # data manipulation
library('tibble') # data wrangling
library('tidyr') # data wrangling
library('purrr')

# Treat text data
library('stringr') # string manipulation
library('tidytext')
library('textstem')

# Modeling
library('caret')
library('keras')
library('tensorflow')
library('text2vec')
library('xgboost')
library('mltools')
library('rBayesianOptimization')
library('MLmetrics')
library('e1071')

# general visualisation
library('ggplot2')



### Sub-sampling

# Remove duplicated id by TV-series
remove_duplicated <- function(df, title){

  df %>% filter(name == !!title) %>% distinct(id, .keep_all = T)
}

foo <- lapply(unique(foo$name), remove_duplicated, df = foo)
foo <- do.call(rbind, foo) # Duplicated user reviews are removed 235851 ->
228802


# This function is to keep TV series that have more than 30 observations
in the minimum target category.
search_suff_levels <- function(df, title){

  X <- df %>% filter(name == !!title)
```

```
  a <- min(table(X$target))

  return(ifelse(a >= 30, title, NA))
}

suff_titles <- lapply(unique(foo$name), search_suff_levels, df = foo)
suff_titles <- c(na.omit(unlist(suff_titles)))

# Sampling process. If a target category level has more than 50 rows, then
keeps 50 random samples.
sampling <- function(df, title){

  X <- df %>% filter(name == !!title)
  a <- min(table(X$target))
  if(a > 50){
    a <- 50
  }

  aa <- function(x){
    Z <- X %>% filter(target == x)
    Z <- Z %>% slice(sample(1:nrow(Z), a, replace = F))
  }

  Y <- do.call(rbind, lapply(unique(X$target), aa))
  return(Y)
}

foo <- lapply(suff_titles, sampling, df = foo)
foo <- do.call(rbind, foo)



### Code for text cleaning and pre-processing

# Lower the case
review <- review %>%
  mutate(content = tolower(content))

# Stemming
stemmed <- unlist(lapply(review$content, lemmatize_strings))
review <- review %>%
  mutate(content = stemmed)

# Fix abbreviations
review <- review %>%
  mutate(content = gsub( "aren't", "are not", content) ,
         content = gsub( "can't" , "cannot", content) ,
         content = gsub( "couldn't" , "could not", content) ,
         content = gsub( "didn't" , "did not", content) ,
         content = gsub( "doesn't" , "does not", content) ,
         content = gsub( "don't" , "do not", content) ,
```

```
        content = gsub( "hadn't" , "had not", content) ,
        content = gsub( "hasn't" , "has not", content) ,
        content = gsub( "haven't" , "have not", content) ,
        content = gsub( "he'd" , "he would", content) ,
        content = gsub( "he'll" , "he will", content) ,
        content = gsub( "he's" , "he is", content) ,
        content = gsub( "i'd" , "i would", content) ,
        content = gsub( "i'll" , "i will", content) ,
        content = gsub( "i'm" , "i am", content) ,
        content = gsub( "isn't" , "is not", content) ,
        content = gsub( "it's" , "it is", content) ,
        content = gsub( "it'll","it will", content) ,
        content = gsub( "i've" , "i have", content) ,
        content = gsub( "let's" , "let us", content) ,
        content = gsub( "mightn't" , "might not", content) ,
        content = gsub( "mustn't" , "must not", content) ,
        content = gsub( "shan't" , "shall not", content) ,
        content = gsub( "she'd" , "she would", content) ,
        content = gsub( "she'll" , "she will", content) ,
        content = gsub( "she's" , "she is", content) ,
        content = gsub( "shouldn't" , "should not", content) ,
        content = gsub( "that's" , "that is", content) ,
        content = gsub( "there's" , "there is", content) ,
        content = gsub( "they'd" , "they would", content) ,
        content = gsub( "they'll" , "they will", content) ,
        content = gsub( "they're" , "they are", content) ,
        content = gsub( "they've" , "they have", content) ,
        content = gsub( "we'd" , "we would", content) ,
        content = gsub( "we're" , "we are", content) ,
        content = gsub( "weren't" , "were not", content) ,
        content = gsub( "we've" , "we have", content) ,
        content = gsub( "what'll" , "what will", content) ,
        content = gsub( "what're" , "what are", content) ,
        content = gsub( "what's" , "what is", content) ,
        content = gsub( "what've" , "what have", content) ,
        content = gsub( "where's" , "where is", content) ,
        content = gsub( "who'd" , "who would", content) ,
        content = gsub( "who'll" , "who will", content) ,
        content = gsub( "who're" , "who are", content) ,
        content = gsub( "who's" , "who is", content) ,
        content = gsub( "who've" , "who have", content) ,
        content = gsub( "won't" , "will not", content) ,
        content = gsub( "wouldn't" , "would not", content) ,
        content = gsub( "you'd" , "you would", content) ,
        content = gsub( "you'll" , "you will", content) ,
        content = gsub( "you're" , "you are", content) ,
        content = gsub( "you've" , "you have", content) ,
        content = gsub( "'re", " are", content) ,
        content = gsub( "wasn't", "was not", content) ,
        content = gsub( "we'll"," will", content) ,
        content = gsub( "didn't", "did not", content) ,
```

```
        content = gsub( "tryin'","trying", content),
        content = gsub( "gey","gay", content),
        content = gsub( "fck","fuck", content),
        content = gsub( "gayyy","gay", content),
        content = gsub( "mem","mother", content),
        content = gsub( "dont","do not", content),
        content = gsub( "wont","would not", content),
        content = gsub( "cant","could not", content),
        content = gsub('shoudnt', 'should not', content),
        content = gsub("wouldnt", "would not", content),
        content = gsub('couldnt', 'could not', content),
        content = gsub("\\bdont\\b", 'do not', content),
        content = gsub("didnt", 'did not', content),
        content = gsub("\\bcant\\b", 'cannot', content),
        content = gsub("doesnt", 'does not', content),
        content = gsub("isnt", 'is not', content),
        content = gsub("arent", 'are not', content),
        content = gsub("\\bim\\b", 'i am', content),
        content = gsub("\\bhes\\b", 'he is', content),
        content = gsub("\\bshes\\b", 'she is', content),
        content = gsub("\\btheyre\\b", 'they are', content),
        content = gsub("\\btheres\\b", 'there is', content),
        content = gsub("\\bthats\\b", 'that is', content),
        content = gsub("\\bive\\b", 'i have', content),
        content = gsub("\\byouve\\b", 'you have', content),
        content = gsub("\\bweve\\b", 'we have', content),
        content = gsub("\\bshouldve\\b", 'should have', content),
        content = gsub("\\bwouldve\\b", 'would have', content),
        content = gsub("\\haven't\\b", 'have not', content),
        content = gsub("\\havent\\b", 'have not', content),
        content = gsub("\\hasn't\\b", 'has not', content),
        content = gsub("\\hasnt\\b", 'has not', content),
        content = gsub("\\hadn't\\b", 'had not', content),
        content = gsub("\\hadnt\\b", 'had not', content),

        content = gsub('colour', 'color', content),
        content = gsub('centre', 'center', content),
        content = gsub('favourite', 'favorite', content),
        content = gsub('theatre', 'theater', content),
        content = gsub('labour', 'labor', content),
        content = gsub('organisation', 'organization', content),
        content = gsub('citicise', 'criticize', content),
        content = gsub('howdo', 'how do', content),
        content = gsub('whatare', 'what are', content),
        content = gsub('howcan', 'how can', content),
        content = gsub('howmuch', 'how much', content),
        content = gsub('howmany', 'how many', content),
        content = gsub('whydo', 'why do', content),
        content = gsub('doI', 'do i', content),
        content = gsub('theBest', 'the best', content),
        content = gsub('howdoes', 'how does', content),
```

```
        content = gsub('exboyfriend', 'ex boyfriend', content),
        content = gsub("whst", 'what', content)
  )

# Change casting and character's name into "pplnme'
ner_change <- function(df, urls, i){

  info <- urls$url[i] %>%
    read_html() %>%
    html_nodes(xpath = "//td//a") %>%
    html_text()

  Sys.sleep(runif(1, 1, 3))

  a <- sort(c(seq(2, length(info), by = 4), seq(3, length(info), by = 4)))

  # Cleaning other info
  character_name <- gsub("\n", "", info[a])
  character_name <- ifelse(substr(character_name, 1, 1) == " ",
                           substr(character_name, 2, nchar(character_name)),
character_name)

  character_name <- tibble(chr_name = character_name) %>%
    unnest_tokens(word, chr_name) %>%
    mutate(flag = 1)

  bar <- df %>% filter(name == urls$name[i])

  foobar <- bar %>%
    unnest_tokens(word, content, strip_punct = FALSE) %>%
    left_join(character_name) %>%
    mutate(flag = ifelse(is.na(flag), 0, flag),
           word = ifelse(flag == 1, "pplnme", word))


  a <- foobar %>% group_by(id) %>%
    summarise(content = str_c(word, collapse = " "))


  a <- a %>% mutate(content = gsub("pplnme pplnme pplnme", 'pplnme',
content),
                    content = gsub("pplnme pplnme", 'pplnme', content),
                    content = gsub("pplnme pplnme pplnme pplnme", 'pplnme',
content),
                    content = gsub("pplnme pplnme pplnme pplnme pplnme",
'pplnme', content)) %>%
    rename(content_cld = content)

  result <- bar %>% left_join(a %>% select(id, content_cld))

  return(result)
```

```
}



### Build bag of words

get_freq_words <- function(df, col, ngram, k, series){

  order_words <- function(input, i){
    input %>% filter(target == i) %>%
      group_by(word) %>%
      summarise(n = n()) %>%
      arrange(desc(n)) %>%
      mutate(word = as.character(reorder(word, n)))
  }

  one <- df %>% filter(name == series) %>%
    unnest_tokens(word, !!sym(col), token = "words") %>%
    anti_join(stop_words)

  if(ngram == 1){
    input <- one
  }

  if(ngram == 2){
    foo <- one %>%
      group_by(id) %>%
      summarise(content_lem = str_c(word, collapse = " ")) %>%
      left_join(df %>% select(-!!sym(col)), by = "id")

    input <- foo %>%
      unnest_tokens(word, content_lem, token = "ngrams", n = 2)
  }

  if(ngram == 3){
    foo <- one %>%
      group_by(id) %>%
      summarise(content_lem = str_c(word, collapse = " ")) %>%
      left_join(df %>% select(-!!sym(col)), by = "id")

    input <- foo %>%
      unnest_tokens(word, content_lem, token = "ngrams", n = 3)
  }

  for (i in 1:n_distinct(df$target)) assign(str_c("a", i, collapse = ""),
order_words(input, i))

  a <- c(a1$word[1:k], a4$word[1:k])
  b <- c(a2$word[1:k], a3$word[1:k])

  bad <- a[!a %in% a4$word[1:k]]
```

```
  excellent <- a[!a %in% a1$word[1:k]]
  med <- setdiff(b, a)

  return(list(bad = bad, excellent = excellent, med = med))
}


intersect_words <- function(bar, col, num, ngram){

  s <- sample(1:length(unique(bar$name)), num, replace = F)
  foo <- lapply(unique(bar$name)[s], get_freq_words, df=bar, col=col,
ngram=ngram, k=500)
  roo <- list()
  for(i in 1:length(foo)){
    roo[[i]] <- foo[[i]]$bad
  }

  boo <- list()
  for(i in 1:length(foo)){
    boo[[i]] <- foo[[i]]$excellent
  }

  soo <- list()
  for(i in 1:length(foo)){
    soo[[i]] <- foo[[i]]$med
  }

  bad <- Reduce(intersect, roo)
  med <- Reduce(intersect, soo)
  excellent <- Reduce(intersect, boo)

  return(list(bad = bad, excellent = excellent, med = med))
}

bad <- list(); excellent <- list(); med <- list()
for(i in 1:2000){
  A <- intersect_words(train, col = 'stc_stem', num = 5, ngram = 1)
  bad[[i]] <- A$bad
  med[[i]] <- A$med
  excellent[[i]] <- A$excellent
}


# Make sentiment word references

afinn <- get_sentiments("afinn")
bing <- get_sentiments("bing") %>%
  rename(value = sentiment) %>%
  mutate(value = ifelse(value == 'negative', -1, 1))

sent <- rbind(afinn, bing) %>%
```

```
  filter(!duplicated(word))

# Good words reference
good_matrix <- tibble(word = unlist(excellent)) %>%
  group_by(word) %>%
  summarise(count = n()) %>%
  arrange(desc(count)) %>%
  left_join(sent) %>%
  mutate(score = log1p(count),
         score = score/mean(score, na.rm = T)) %>%
  filter(count >= 20 | (value > 0 & count >= 3)) %>%
  select(-value)

# Bi-gram good words reference
good_matrix_2 <- tibble(word = unlist(excellent_2)) %>%
  group_by(word) %>%
  summarise(count = n()) %>%
  arrange(desc(count)) %>%
  left_join(sent) %>%
  mutate(score = max(good_matrix$score)) %>%
  filter(count >= 13) %>%
  select(-value)

# Bad words reference
bad_matrix <- tibble(word = unlist(bad)) %>%
  group_by(word) %>%
  summarise(count = n()) %>%
  arrange(desc(count)) %>%
  left_join(sent) %>%
  mutate(score = log1p(count),
         score = score/mean(score, na.rm = T)) %>%
  filter(count >= 20 | (value > 0 & count >= 3)) %>%
  select(-value)

# Bi-gram bad words reference
bad_matrix_2 <- tibble(word = unlist(bad_2)) %>%
  group_by(word) %>%
  summarise(count = n()) %>%
  arrange(desc(count)) %>%
  left_join(sent) %>%
  mutate(score = max(bad_matrix$score)) %>%
  filter(count >= 7) %>%
  select(-value)

# Medium words reference
med_matrix <- tibble(word = unlist(med)) %>%
  group_by(word) %>%
  summarise(count = n()) %>%
  arrange(desc(count))
```

```r
# Function to calculate sentiment scores row by row
get_word_score <- function(sentence_matrix, reference_matrix, col, value,
negation, cname){

  token_stc <- sentence_matrix %>%
    unnest_tokens(sentence, !!sym(col), token = "regex", pattern = "\\.")

  if(negation == T){
    token_stc <- token_stc %>%
      ungroup() %>%
      mutate(stc_num = 1:nrow(token_stc)) %>%
      filter(!str_detect(sentence,
'\\bnot\\b|\\bno\\b|\\bnever\\b|\\bnothing\\b'))
  }else{
    token_stc <- token_stc %>%
      ungroup() %>%
      mutate(stc_num = 1:nrow(token_stc))
  }

  token_word <- token_stc %>%
    unnest_tokens(word, sentence, drop = F) %>%
    left_join(reference_matrix) %>%
    group_by(name, id) %>%
    summarise(score = sum(!!sym(value), na.rm = T))

  colnames(token_word)[3] <- cname
  return(token_word)
}

score_good <- get_word_score(stc_matrix, good_matrix, 'stc_stem', 'count',
negation = T, 'good_content')


### Code to re-construct the input sentences for Deep Learning

afinn <- get_sentiments("afinn")
afinn <- afinn[-which(afinn$word == "like"),]
bing <- get_sentiments("bing")
bing <- bing[-which(bing$word == "like"),]

token_word <- token_word %>% left_join(afinn)
token_word <- token_word %>% left_join(bing)

token_word <- token_word %>%
  mutate(value = ifelse(value > 0, 1, -1),
         value = ifelse(sentiment == "positive", 1, -1))

but_word <- c('but', 'however')
acting_word <- c('actor', 'actors', 'actress', 'acting', 'actings',
                 'cast', 'casting', 'castings', 'character', 'characters',
                 'pplnme')
```

```r
story_word <- c('story', 'storyline', 'story line', 'writer', 'plot',
'writers',
                'creator', 'creators', 'dialogue', 'script', 'dialogues',
'scripts')
series_word <- c('series', 'season', "show")

foo <- token_word %>%
  mutate(but = ifelse(word %in% but_word, 1, 0),
         acting = ifelse(word %in% acting_word, 1, 0),
         story = ifelse(word %in% story_word, 1, 0),
         series = ifelse(word %in% series_word, 1, 0),
         emos = ifelse(!(is.na(value)), 1, 0))

bar <- foo %>%
  group_by(name, id, stc) %>%
  summarise(but = max(but, na.rm = T)*1.5,
            acting = max(acting, na.rm = T)*1.5,
            story = max(story, na.rm = T)*2,
            series = max(series, na.rm = T)*2,
            emo_pos_sum = sum(value > 0, na.rm = T),
            emo_neg_sum = sum(value < 0, na.rm = T),
            emo_sum = sum(value, na.rm = T)*0.5,
            emo_times = sum(emos))

a <- rowSums(bar[, c(4,5,6,7,11)])
bar$flag <- a

foobar <- bar %>%
  filter(flag > 0) %>%
  group_by(name, id) %>%
  arrange(desc(flag), .by_group = T) %>%
  left_join((review %>% select(name, id, target)), by = c("name", "id"))

foobar <- foobar %>%
  mutate(negation = ifelse(str_detect(stc,
'\\bnot\\b|\\bno\\b|\\never\\b|\\bnothing\\b'), 1, 0))

No_words <- sapply(gregexpr("\\W+", foobar$stc), length)

foobar <- foobar %>% ungroup() %>%
  mutate(word_count = No_words) %>%
  group_by(name, id) %>%
  mutate(word_count_cum = cumsum(word_count))


### Make the input matrix

rm(foo, bar, bing, token_stc, token_word, No_words)

foobar <- foobar %>%
  mutate(from = ifelse(but > 0, "but", NA),
```

```
        from = ifelse(acting > 0, "acting", from),
        from = ifelse(story > 0, "story", from),
        from = ifelse(series > 0, "series", from),
        from = ifelse(is.na(from), "emo", from))

foobar <- foobar %>%
  spread(key = from, value = stc)

stc_matrix <- foobar %>%
  mutate(stc = paste0(acting, but, emo, series, story),
         stc = gsub('NA', "", stc),
         stc = gsub('\\s+', ' ', stc),
         stc_mark = paste(str_c("dorxm ", acting, " rmx"),
                          str_c("qjt ", but, " rmx"),
                          str_c("dlah ", emo, " rmx"),
                          str_c("tlflwm ", series, " rmx"),
                          str_c("tmxhfl ", story, "rmx"), sep = " "),
         stc_mark = gsub('NA', "", stc_mark),
         stc_mark = gsub('\\s+', ' ', stc_mark)
  )

stc_matrix <- stc_matrix %>%
  group_by(name, id) %>%
  summarise(stc = str_c(stc, collapse = ". "),
            stc_mark = str_c(stc_mark, collapse = ". "))



### Hyper-parameters tuning


# Error function for XGBOOST
error_cv <- function(max_depth, min_child_weight, colsample_bytree,
gamma){

  cv_xgb <- function(df, base, k_folds, seed, max_depth, min_child_weight,
colsample_bytree, gamma){

    prediction_1 <- list()
    new_data <- list()
    set.seed(seed)
    folds <- createFolds(df$target, k = k_folds, list = TRUE, returnTrain
= FALSE)

    for(j in 1:k_folds){

      dtrain <- df %>% dplyr::slice(-folds[[j]]) %>% select(one_of(base))
      dtest <- df %>% dplyr::slice(folds[[j]]) %>% select(one_of(base))
      new_data[[j]] <- dtest
```

```
    y <- createDataPartition(dtrain$target, p = 0.8)$Resample1
    dtrain <- dtrain[y, ]
    dvalid <- dtrain[-y, ]

    y_train <- dtrain$target
    y_valid <- dvalid$target
    y_test <- dtest$target

    dtrain <- dtrain %>% select(-target)
    dvalid <- dvalid %>% select(-target)
    dtest <- dtest %>% select(-target)

    dtrain <- as.matrix(dtrain)
    dvalid <- as.matrix(dvalid)
    dtest <- as.matrix(dtest)

    dtrain <- xgb.DMatrix(data = dtrain, label = y_train-1)
    dval <- xgb.DMatrix(data = dvalid, label = y_valid-1)
    cols <- colnames(dtrain)

    p <- list(objective = "multi:softprob",
              booster = "gbtree",
              eval_metric = "mlogloss",
              nthread = 4,
              eta = 0.001,
              max_depth = max_depth,
              min_child_weight = min_child_weight,
              gamma = gamma,
              subsample = 0.75,
              colsample_bytree = colsample_bytree,
              alpha = 0,
              lambda = 0,
              nrounds = 1000,
              num_class = 4)

    fit_xgb <- xgb.train(p, dtrain, p$nrounds, list(val = dval),
print_every_n = 30, early_stopping_rounds = 15)

    xgb.pred <- predict(fit_xgb, dtest, reshape = T)
    xgb.pred <- as.data.frame(xgb.pred)
    colnames(xgb.pred) <- 0:3
    #xgb.pred$prediction = apply(xgb.pred,1,function(x)
colnames(xgb.pred)[which.max(x)])
    prediction_1[[j]] <- xgb.pred

  }

  whole_pred <- do.call(rbind, prediction_1)
  new_data <- do.call(rbind, new_data)
  cv_error <- MultiLogLoss(whole_pred, new_data$target-1)
```

```
    new_data$target <- whole_pred

    return(list(new_data = new_data, cv_error = cv_error))
  }

  cv_final <- cv_xgb(X_final %>% select(-id, -name), base, 10, 20,
max_depth, min_child_weight,
                     colsample_bytree, gamma)$cv_error

  return(list(Score = -cv_final, Pred = 0))
}

# Baysian Opimization based on the reference error function above.
OPT_Res <- BayesianOptimization(error_cv,
                                bounds = list(max_depth = c(7L, 11L),
                                              min_child_weight = c(10L, 25L),
                                              colsample_bytree = c(0.6, 0.9),
                                              gamma = c(0, 2)),
                                init_grid_dt = NULL, init_points = 10, n_iter
= 30,
                                acq = "ucb", kappa = 2.576, eps = 0.0,
                                verbose = TRUE)


# Error function for Suppor Vector Machine
error_cv_svm <- function(gamma, cost){

  cv_xgb <- function(df, base, k_folds, seed, gamma, cost){

    prediction_1 <- list()
    new_data <- list()
    set.seed(seed)
    folds <- createFolds(df$target, k = k_folds, list = TRUE, returnTrain
= FALSE)

    for(j in 1:k_folds){

      dtrain <- df %>% dplyr::slice(-folds[[j]]) %>% select(one_of(base))
      dtest <- df %>% dplyr::slice(folds[[j]]) %>% select(one_of(base))
      new_data[[j]] <- dtest


      y <- createDataPartition(dtrain$target, p = 0.8)$Resample1
      X <- dtrain[y, ] %>% ungroup() %>% mutate(target = as.factor(target))
      Y <- dtrain[-y, ] %>% ungroup() %>% mutate(target =
as.factor(target))

      svm_fit <- svm(target ~ ., data = X,
                     method = "C-classification", kernel = kernel,
                     gamma = gamma, cost = cost)
```

```
      svm.pred <- predict(svm_fit, dtest)
      prediction_1[[j]] <- as.matrix(svm.pred)

    }

    whole_pred <- do.call(rbind, prediction_1)
    new_data <- do.call(rbind, new_data)
    xtab <- table(new_data$target, whole_pred)
    cv_error <- -sum(diag(xtab))/sum(xtab)
    new_data$target <- whole_pred

    return(list(new_data = new_data, cv_error = cv_error))
  }

  cv_final <- cv_xgb(df, base, 5, 20, gamma, cost)$cv_error

  return(list(Score = -cv_final, Pred = 0))
}


OPT_Res <- BayesianOptimization(error_cv_svm,
                                bounds = list(gamma = c(0.001, 0.05),
                                              cost = c(10, 100)),
                                init_grid_dt = NULL, init_points = 10, n_iter
= 15,
                                acq = "ucb", kappa = 2.576, eps = 0.0,
                                verbose = TRUE)




### Code for Bi-LSTM modeling

##### LSTM

set.seed(314)
y <- createDataPartition(stc_matrix_2$target, p = 0.7)
train <- stc_matrix_2[y$Resample1, ]
test <- stc_matrix_2[-y$Resample1, ]

max_words = 15000 # Maximum number of words to consider as features
maxlen = 70 # Text cutoff after n words


full <- rbind(train %>% select(stc_all_stem), test %>%
select(stc_all_stem))
texts <- full$stc_all_stem

tokenizer <- text_tokenizer(num_words = max_words) %>%
  fit_text_tokenizer(texts)
```

```
sequences <- texts_to_sequences(tokenizer, texts)
word_index <- tokenizer$word_index

data <- pad_sequences(sequences, maxlen = maxlen, truncating = 'post')

X_train <- data[1:nrow(train), ]
y_train <- train$target

set.seed(314)
s <- sample(1:nrow(X_train), 0.2*nrow(X_train), replace = F)

X_valid <- X_train[s, ]
y_valid <- as.matrix(y_train[s])

X_train <- X_train[-s, ]
y_train <- as.matrix(y_train[-s])

X_test <- data[(nrow(train)+1):nrow(data), ]
y_test <- as.matrix(test$target)

y_train <- y_train %>%  data.frame() %>%
  mutate(
    V1 = ifelse(y_train == 1, 1, 0),
    V2 = ifelse(y_train == 2, 1, 0),
    V3 = ifelse(y_train == 3, 1, 0),
    V4 = ifelse(y_train == 4, 1, 0)
  ) %>%
  select(
    V1,V2,V3,V4
  ) %>% as.matrix()

y_valid <- y_valid %>%  data.frame() %>%
  mutate(
    V1 = ifelse(y_valid == 1, 1, 0),
    V2 = ifelse(y_valid == 2, 1, 0),
    V3 = ifelse(y_valid == 3, 1, 0),
    V4 = ifelse(y_valid == 4, 1, 0)
  ) %>%
  select(
    V1,V2,V3,V4
  ) %>% as.matrix()

rm(data, full, sequences, tokenizer, y, train, test, texts)



### Getting the Embedding weights from difference corpus sources

# Glove Wiki ----------------------------------------------------------
------
```

```
lines = readLines("glove.6B.300d Wiki+Gigaword.txt")

wiki_embeddings_index = new.env(hash = TRUE, parent = emptyenv())

pb <- txtProgressBar(min = 0, max = length(lines), style = 3)
for (i in 1:length(lines)){
  line = lines[[i]]
  values = strsplit(line, " ")[[1]]
  word = values[[1]]
  wiki_embeddings_index[[word]] = as.double(values[-1])
  setTxtProgressBar(pb, i)
}


# Create our embedding matrix

wiki_embedding_dim = 300
wiki_embedding_matrix = array(0, c(max_words, wiki_embedding_dim))


for (word in names(word_index)){
  index = word_index[[word]]
  if (index < max_words){
    wiki_embedding_vector = wiki_embeddings_index[[word]]
    if (!is.null(wiki_embedding_vector))
      wiki_embedding_matrix[index+1,] = rnorm(300) # Words without an
embedding are all zeros
  }
}

saveRDS(wiki_embedding_matrix, "glove_wiki_300d_32.rds")

rm(pb, wiki_embedding_dim, wiki_embedding_matrix, wiki_embedding_vector,
wiki_embeddings_index,
   line, lines, i, index, word, values)


# Glove Twitter -------------------------------------------------------
---------

# Now we need to parse the GloVe embeddings

lines = readLines("glove.twitter.27B.200d.txt")

twitter_embeddings_index = new.env(hash = TRUE, parent = emptyenv())

pb <- txtProgressBar(min = 0, max = length(lines), style = 3)
for (i in 1:length(lines)){
  line = lines[[i]]
  values = strsplit(line, " ")[[1]]
  word = values[[1]]
```

```
  twitter_embeddings_index[[word]] = as.double(values[-1])
  setTxtProgressBar(pb, i)
}


# Create our embedding matrix

twitter_embedding_dim = 200
twitter_embedding_matrix = array(0, c(max_words, twitter_embedding_dim))

for (word in names(word_index)){
  index = word_index[[word]]
  if (index < max_words){
    twitter_embedding_vector = twitter_embeddings_index[[word]]
    if (!is.null(twitter_embedding_vector))
      twitter_embedding_matrix[index+1,] = rnorm(200) # Words without an
embedding are all zeros
  }
}

saveRDS(twitter_embedding_matrix, "glove_twitter_200d_32.rds")

rm(pb, twitter_embedding_dim, twitter_embedding_matrix,
twitter_embedding_vector, twitter_embeddings_index,
   line, lines, i, index, word, values)


# Embeddings ------------------------------------------------------------
------

# Dimensions

glove_wiki_embedding_dim = 300
glove_twitter_embedding_dim = 200

# Files (uploaded from local pc)
setwd("C:\\Users\\17096\\Desktop\\Data Science\\Word corpus for
embedding")

glove_wiki_weights = readRDS("glove_wiki_300d_32.rds")
glove_twitter_weights = readRDS("glove_twitter_200d_32.rds")


setwd("C:\\Users\\17096\\Desktop\\Data Science\\Practice")

# Model Architecture ----------------------------------------------------
--------------

# Setup input

input <- layer_input(
```

```
  shape = list(NULL),
  dtype = "int32",
  name = "input"
)


# Embedding layers

encoded_1 <- input %>%
  layer_embedding(input_dim = max_words, output_dim =
glove_wiki_embedding_dim, name = "embedding_1") %>%
  bidirectional(layer_lstm(units = maxlen,
                           dropout = 0.2,
                           recurrent_dropout = 0.5,
                           return_sequences = F))

encoded_2 <- input %>%
  layer_embedding(input_dim = max_words, output_dim =
glove_twitter_embedding_dim, name = "embedding_2") %>%
  bidirectional(layer_lstm(units = maxlen,
                           dropout = 0.2,
                           recurrent_dropout = 0.5,
                           return_sequences = F))


# Concatenate

concatenated <- layer_concatenate(list(encoded_1, encoded_2))

# Dense layers

dense <- concatenated %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 128, activation = "relu") %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 4, activation = 'softmax')


# Bring model together

model <- keras_model(input, dense)

# Freeze the embedding weights initially to prevent updates propgating
back through and ruining our embedding

get_layer(model, name = "embedding_1") %>%
  set_weights(list(glove_wiki_weights)) %>%
  freeze_weights()

get_layer(model, name = "embedding_2") %>%
  set_weights(list(glove_twitter_weights)) %>%
```

```r
  freeze_weights()


# Compile

model %>% compile(
  optimizer = optimizer_rmsprop(lr = 0.005),
  loss = "categorical_crossentropy",
  metrics = 'categorical_accuracy'
)

print(model)

# Model Training -----------------------------------------------------
----------

# Early stopping condition

callbacks_list <- list(
  callback_early_stopping(
    monitor = 'val_loss',
    patience = 10
  )
)

# Train model

history <- model %>% keras::fit(
  X_train,
  y_train,
  batch_size = 2048,
  validation_data = list(X_valid, y_valid),
  epochs = 15,
  callbacks =  callbacks_list
)
```