Classify the Sentiment of Sentences from the Rotten Tomatoes Dataset

Math 251 Final Project Paper

Zhixuan Qin and Yi Tang

Abstract

Sentiment classification is a challenging task that aims to identify the opinions and emotions in text and labeling them with appropriate sentiment labels such as positive, negative, or neutral. In this project, a variety of classifiers were fitted on the Rotten Tomatoes movie review dataset (containing 5 sentiment labels) from Kaggle under four different classification scenarios: 1 standard classification, 2 ordinal classifications based on marginal probability, and 1 ordinal classification based on cumulative probability. Our results indicated that ordinal classification based on cumulative probability generally produced better test accuracy compared with other classification scenarios since the sentiment labels are embedded with a natural order. Among all of the evaluated classifiers, ordinal random forest classifier with 400 trees (based on cumulative probability) yielded the highest accuracy of 0.6743. Using this method, our final submission of Kaggle competition achieved a final test accuracy of 0.65267 and ranked 116th on the leaderboard. Our results also revealed that the more complicated deep learning model such as recurrent neural network (RNN) and convolutional neural network (CNN) did not produce a high accuracy for this dataset, which may be related to the issue of overfitting and the difficulty to design and tune the models. Future research should look into different options in addressing the overfitting problem of RNN and CNN and continue explore various ways to design and tune these deep learning models.

1 Introduction

Sentiment is a thought, opinion, or idea based on a feeling about a situation, or a way of thinking about something. Sentiment classification is a challenging classification task, which uses natural language processing, text analysis, computational linguistics, and biometrics to identify opinions and emotions in text and assign proper sentimental labels (such as positive, negative, or neutral) to them. Sentiment classification has been widely used in business and product development settings to understand how customers feel about products, services, or brand. The objective of this project is to conduct sentiment classification on the Rotten Tomatoes dataset using a variety of classifiers and evaluate/compare the prediction results.

2 Dataset

The Rotten Tomatoes movie review dataset (accessible at: https://www.kaggle.com/c/sentiment-analysis-on-movie-reviews) contains a corpus of movie reviews used for sentiment analysis, which is originally collected by Pang and Lee (2006). Later, Socher et al. (2013) created fine-grained labels for all parsed phrases in the corpus using Amazon's Mechanical Turk. As a result, the text in Rotten Tomatoes dataset is not complete sentences but parsed short phrases. The dataset includes a total of 156,060 training data and 66,292 testing data. The training set has 4 columns: PhraseId, SentenceId, Phrase, and Sentiment, while the test set has the first three but no Sentiment. The first ten training samples are shown in Table 1. The length of each phrase varies, and some phrase may just contain one stop word (e.g., phrase 4 and 7) or one punctuation. However, the same stop word may have very different labels, leading to a certain challenge in this data set.

The training data are classified into 5 classes, which are 0-negative, 1-somewhat negative, 2-neutral, 3-somewhat positive, 4-positive. There is a natural order among the different classes. Such ordering information can be used during the classification task. Notably, neutral is the dominant class whose number of data points is over 10 times than the number of data points from the most minor class (negative; Figure 1).

Table 1. The first ten training samples with their PhraseId, SentenceId, Phrase, and Sentiment.

PhraseId	**SentenceId**	**Phrase**	**Sentiment**
1	1	A series of escapades demonstrating the adage ...	1
2	1	A series of escapades demonstrating the adage ...	2
3	1	A series	2
4	1	A	2
5	1	series	2
6	1	of escapades demonstrating the adage that what...	2
7	1	of	2
8	1	escapad es demonstrating the adage that what is...	2
9	1	escapad es	2
10	1	demonst rating the adage that what is good for ...	2

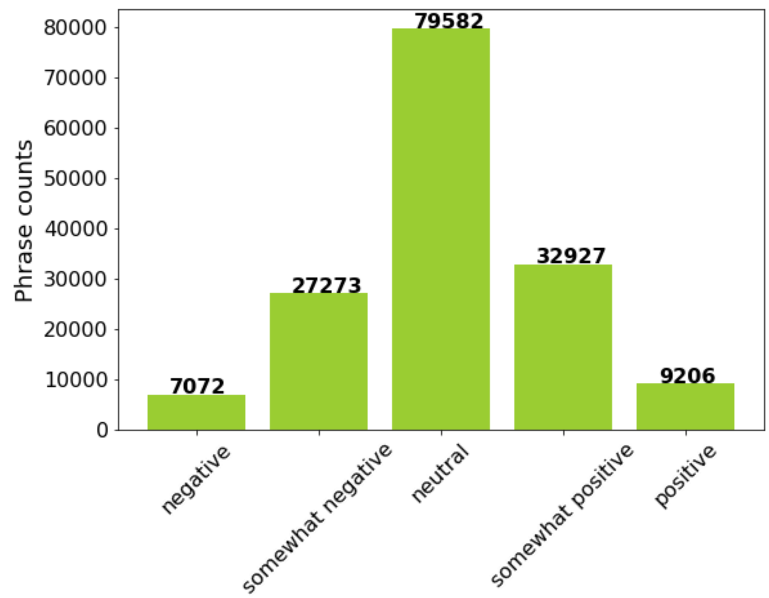

Figure 1. The distribution of the classes in the training data of the Rotten Tomatoes movie review dataset.

3 Methods

3.1 Sentence and text embedding

Recent studies have demonstrated strong transfer task performance using pre-trained sentence level embeddings (Conneau et al., 2017, Cer et al., 2018). The models take strings as input and produce an embedding representation of the string with a fixed dimension as output. The sentence level embeddings differ significantly from traditional bag of words approach in converting text information into numerical vectors to create features for machine learning classifiers. The bag of words approach simply counts how many times a word appears in a document and does not consider the relationship between different words within a sentence. However, sentence level embeddings such as Sentence Transformers make use of sophisticated Recurrent Neural Network (RNN) framework to learn text information at sentence level, which enables this approach to consider the text dependencies and connections. As a result, sentence level embeddings are normally preferred for complicated text classification such as sentiment classification.

The pre-trained embedding language models are publicly available in SentenceTransformers which is a Python framework for state-of-the-art sentence and text embeddings (www.SBERT.net). There are 26 models that were trained on SNLI and MultiNLI and then fine-tuned on the Semantic Textual Similarity (STS) benchmark train set. The 'roberta-large-nli-stsb-mean-tokens' model was used for this project as it has the highest STSb performance (86.39). We used GPU under Google Collaboratory to run the SentenceTransformers. The resulting training and test sets have dimension 156,060 × 1024 and 66292 × 1024, respectively.

3.2 **Training set splitting**

Because the test data set has no sentiment labels and in order to better guide the downstream classification task, we decided to only use the training set to perform the classification. Additionally, the training dataset with a total of 156,060 phrases is too big to train classifiers within a reasonable time frame especially when the classifiers need parameter-tuning. As a result, we decided to subsample training data from each sentiment class to further reduce the size of the training set and simultaneously address the issue of unbalanced classes. As shown in Figure 1, the training set is class-imbalanced with negative class to positive class ratio as 1: 3.9 : 11.3 : 4.7 : 1.3. For each of the class, we randomly split train and test with a prefixed ratio to manually balance the five classes for training. Specifically, the proportions of training size in the five classes from negative, somewhat negative, neutral, somewhat positive, to positive are 2/3, 1/3, 1/5, 1/4, and 2/3 respectively. Thus, the resulting training set has 4714, 9091, 15916, 8231, and 6137 phrase counts for their corresponding class respectively (Figure 2). The largest class ratio is 3.4 (neutral versus negative) which is lower than 5, thus we regard the new training set as relatively "balanced". For the test data set, we then have 2358, 18182, 63666, 24696, and 3069 phase counts for their corresponding class respectively. We then used the new training (44089×1024) and test sets (111971×1024) for the following classification task.

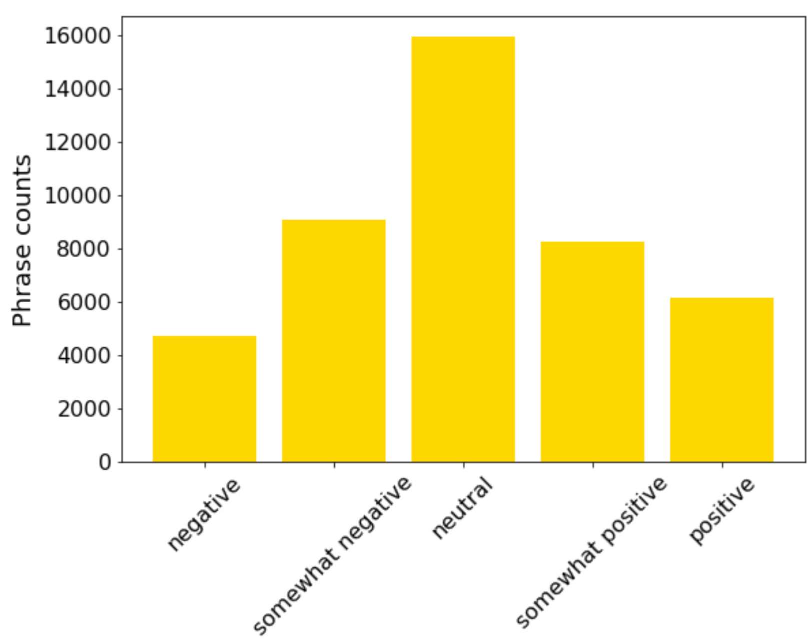

Figure 2. The new training set with "balance" classes (all ratios < 5).

3.3 **Ordinal classification**

Another import characteristic of our data set is the presence of a "natural" order among classes. Conventional classification methods for nominal classes could be applied to solve such ordinal problems, but the use of techniques designed specifically for ordered classes could potentially yield classification results with a better performance (Frank and Hall, 2001). Frank and Hall (2001) proposed a simple approach to conduct ordinal classification without any modification of the underlying learning algorithm. First the original training data needs to be transformed from a k-class ordinal problem to k-1 binary class problems, resulting in k-1 new binary data sets. Then classifiers which can produce class probability will be trained on each of the k-1 data sets. For class C_i (ordinal scenario 1),

$$P(C_1) = 1 - P(Y > C_1)$$

$$P(C_i) = P(Y > C_{i-1}) - P(Y > C_i), 1 < i < k$$

$$P(C_k) = P(Y > C_{k-1})$$

During prediction, a sample without class information is processed by each of the k-1 classifiers and the probability values of each of the k classes are calculated by the above approach. Probabilities of all classes are then compared and the class with the maximum probability is assigned to the test sample. It's worth noting that this method may lead to negative estimates of probability values for C_i when $1 < i < k$ at the prediction time (Frank and Hall, 2001). Cardoso and Pinto da Costa (2007) proposed to use the following formula (ordinal scenario 2):

$$P(C_i) = (1 - P(Y > C_i))P(Y > C_{i-1}), 1 < i < k$$

Alternatively, as we learned from the class, we can also predict the test label based on the cumulative probabilities according to the following objective formula (ordinal scenario 3):

$$\hat{y} = \min_i P(Y \leq C_i|x) > \frac{1}{2}, 1 \leq i \leq k$$

In this project, all three of the ordinal classification algorithms were implemented and the corresponding results were compared. The probability of testing data falling into each of the binary class is accessible through the corresponding python classifier using the predict_proba function, which will then be used to calculate the cumulative probability.

3.4 **Classification methods**

Multiple classification methods were applied, including Linear Discriminant Analysis (LDA), Quadratic Discriminant Analysis (QDA), Gaussian Naïve Bayes (GNB), Logistic Regression (LR), Support Vector Method (SVM), Random Forest (RF), and Neural Networks (NN), Convolution Neural Networks (CNN), and Recurrent Neural Networks (RNN). Two classification scenarios were processed, one is to treat the class attribute as a set of unordered values and apply each of the standard classification algorithms, second is to make use of the ordering information among classes by the technique described in section 3.3. For each of the classification scenario, we tuned the classifiers (if applicable) with a range of potential parameters (such as regularization, number of trees, number of neurons, etc.) and the best accuracy was reported in this report.

It is worth to mention that for CNN and RNN, there are other suggested embedding methods to convert text information to numerical vectors. Instead of using sentence transformer (as described in section 3.1), we used the default embedding method and word2vec for RNN and CNN respectively. We also padded the training data with 0s in the beginning to force all the data to have the same length of 50 and 100 for CNN and RNN respectively. We were not able to apply ordinal classification on CNN and RNN as the coded ordinal classification function was not compatible with the functions to conduct CNN and RNN. As a result, CNN and RNN were only trained under the standard classification scenario.

The overall architecture design of CNN and RNN is crucial for their overall performance, but the design and tuning of those deep learning models are very challenging. For RNN model, we designed the first layer as the embedding layer, the second layer as the long short-term memory (LSTM) layer (a special kind of RNN, capable of learning long-term dependencies) with 100 neurons, the third layer as the drop-out layer with a drop-out rate of 0.5 (to alleviate over-fitting), and the final dense layer that produces the probability distribution of the 5 classes using "softmax" activation. For CNN model, we designed the first layer as the embedding layer, the second layer as the filter layer containing 5 different filters (sizes range from 2-6) with each filter followed by a GlobalMaxPooling1D layer (to down-sample the input representation by taking the maximum value over the time dimension), the third layer as the concatenating layer to join the outputs from the 5 filters, the fourth layer as the drop-out layer with a drop-out rate of 0.1, the fifth layer as the dense layer containing 128 neurons with "relu" activation, the sixth layer as another drop-out layer with a drop-out rate of 0.2, and the final dense layer that produces the probability distribution of the 5 classes using "sigmoid" activation.

3.5 Programming language and libraries

Python 3 and Jupyter Notebook was used for this entire project. The scikit-learn library was used to implement LDA, QDA, GNB, LR, SVM, RF, and NN classifiers. The Keras library (now part of TensorFlow library) was used to implement RNN and CNN. The individual text embedding process for RNN and CNN (as described in section 3.1) was implemented using Keras and Gensim library, respectively.

4 Results and discussions

4.1 Overall accuracy by classifier

Our results indicated that when we only applied the standard classifiers and did not consider the order information from the label, GNB classifier yielded the lowest accuracy (0.5287). All other classifiers yielded accuracies above 0.62 (Table 2) with the highest test accuracy produced by regular 1-layer NN classifier (0.6615). The significant lower test accuracy of GNB compared with other classifiers might be related to its naive assumption of independent predictors. This assumption might not be met in the feature space extracted by sentence transformer.

When we used marginal probability (ordinal scenario 1 and 2) to conduct ordinal classification, the overall test accuracy slightly improved for LDA, LR and RF classifiers while the overall test accuracy slightly decreased for QDA, GNB and NN classifiers (Table 2). We also tried to apply ordinal classification for SVM and Adaboost, but the significant increase in the running time makes ordinal classification using SVM and Adaboost impractical for our data. Notably, ordinal classification (ordinal scenario 2) modified by Cardoso and Pinto da Costa (2007) slightly improved the test accuracy for all classifiers compared with the results from ordinal classification by simple marginal probability subtraction (ordinal scenario 1).

When we used cumulative probability (ordinal scenario 3) to conduct ordinal classification, the overall test accuracy further improved for almost all evaluated classifiers except QDA and NN (Table 2), suggesting that ordinal scenario 3 is the algorithm that is the most effective in using the natural order information from the class labels. Among all the evaluated classifiers and scenarios, RF trained under ordinal scenario 3 (400 trees) yielded the highest test accuracy of 0.6743.

Our results also indicated that the use of more sophisticated deep learning models (RNN and CNN) did not produce better overall accuracy compared with other simpler classifiers (Table 2). This may be related to the issue of overfitting (especially for CNN) since the current settings of RNN and CNN need to train 542,953 and 1,329,773 parameters respectively. Additionally, our design of the overall model architecture and the tuning of the parameters may not be the most powerful combination. It is also possible that the text embedding method adopted by RNN and CNN is not as effective as sentence transformer in preserving the connections and associations of the text information since the models developed for sentence transformer is built based on very sophisticated RNN and is pre-trained on very large datasets. As a result, the performance of RNN and CNN might be impacted by our current choice of text embedding methods.

Table 2. Summary of the accuracy results of the standard classifiers and their corresponding three ordinal classifiers.

Classifier	Standard	Ordinal1	Ordinal2	Ordinal3
LDA	0.6283	0.6337	0.6354	0.6379
QDA	0.6270	0.5993	0.5998	0.5987
GNB	0.5287	0.4624	0.4635	0.5293
LR	0.6429	0.6460	0.6576	0.6614
SVM	0.6436	*Na**	*Na**	*Na**
NN	0.6615	0.6379	0.6407	0.6439
RF	0.6505	0.6457	0.6639	0.6743
Adaboost	0.6545	0.6567	*Na**	*Na**
RNN	0.6197	*Na***	*Na***	*Na***
CNN	0.4445	*Na***	*Na***	*Na***

**There are no accuracy results for all three ordinal SVM and the last two ordinal Adaboost due to extremely long run time issue.*

***There are no accuracy results for RNN and CNN because the coded ordinal classification function is not compatible with the functions to conduct RNN and CNN.*

4.2 Accuracy for individual sentiment label by classifier

The standard and ordinal classification was further compared by investigating the changes in the accuracy for individual sentiment labels. Specifically, LR, NN, and RF were chosen as they produced the highest overall accuracy (Table 2).

For standard LR, the accuracy overall for each label were reasonably balanced with the lowest (0.4223) and highest (0.7592) accuracy for “somewhat positive” and “neutral”, respectively (Figure 3). After applying ordinal LR scenario 3, the accuracy for each label were not as balanced as those by the standard LR. Class “negative” had the lowest accuracy of 0.1124 while “neutral” had the highest accuracy of 0.8107.

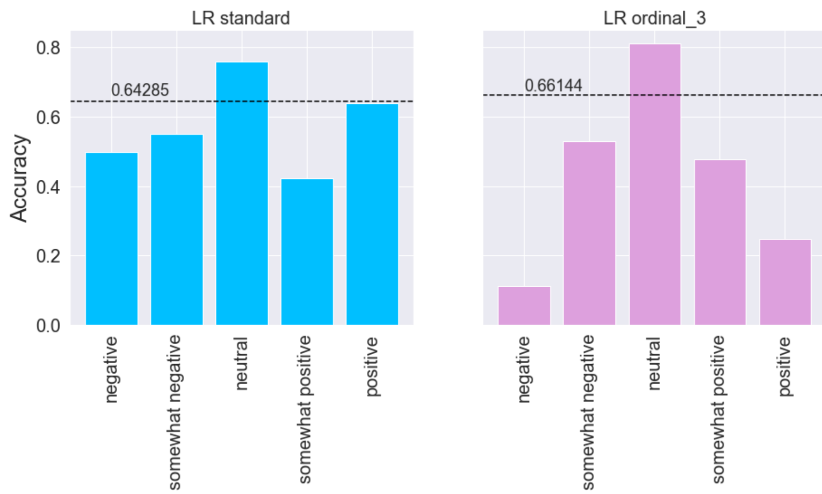

Figure 3. Accuracy for each sentiment label by applying standard logistic regression (left panel) and ordinal (scenario 3) logistic regression (right panel). The black dashed line in each plot indicates the overall accuracy for the corresponding classifier.

Standard NN yielded imbalanced accuracy for different labels (Figure 4). In particular, all the “negative” phrases in the test set were misclassified (accuracy = 0). On the other hand, the same classifier produced the highest accuracy (0.7506) for “neutral” phrases. The application of ordinal NN scenario 3 didn’t seem to improve the balance among the classes. In fact, the range of the highest (0.8459) and lowest (0.0951) accuracy is slightly larger than that of the standard NN. We also noticed that the accuracy for “negative” class was significantly increased by using ordinal NN. Such increase for “negative” class seemed to be a sacrifice for “somewhat negative” as its accuracy decreased from 0.6468 to 0.0951.

Accuracy for individual label did not change significant between standard and ordinal3 RF classifier. Both of the RF classification scenarios yielded slightly imbalanced accuracy for different labels (Figure 5). Specifically, “neutral” phrases had the highest accuracy (0.8083 and 0.8305 for standard and ordinal3 RF respectively), which is consistent with the fact that “neutral” label has the highest number of training data (Figure 2). In contrast, “somewhat positive” phrases had the lowest accuracy (0.3642 and 0.4482 for standard and ordinal3 RF respectively). Notably, switching to ordinal3 RF from standard RF slightly increased the accuracy for “neutral” and “somewhat positive” phrases but decreased the accuracy for “negative”, “somewhat negative”, and “positive” phrases.

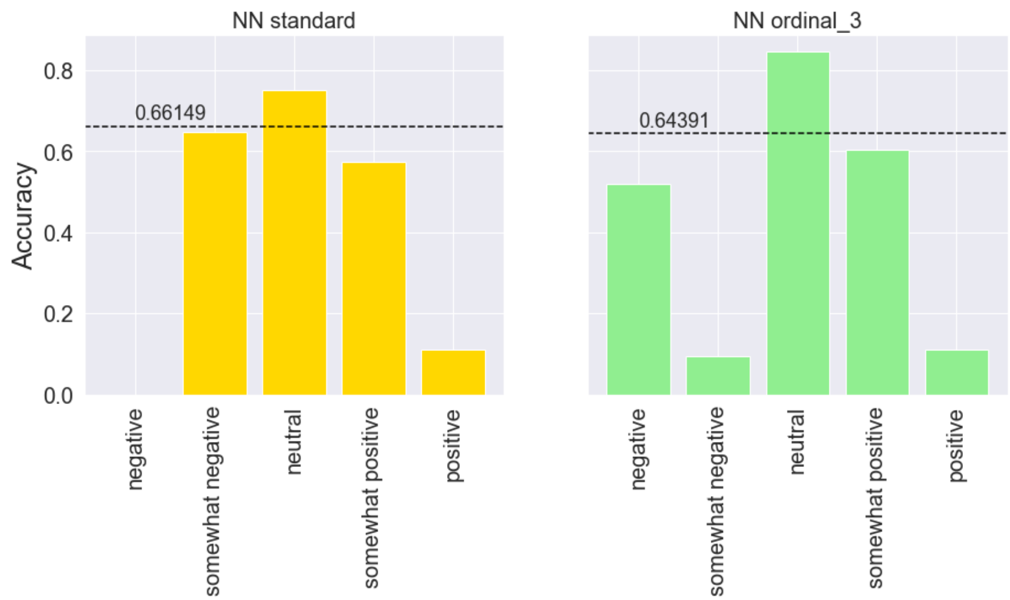

Figure 4. Accuracy for each sentiment label by applying standard neurol networks (left panel) and ordinal (scenario 3) neurol networks (right panel). The black dashed line in each plot indicates the overall accuracy for the corresponding classifier.

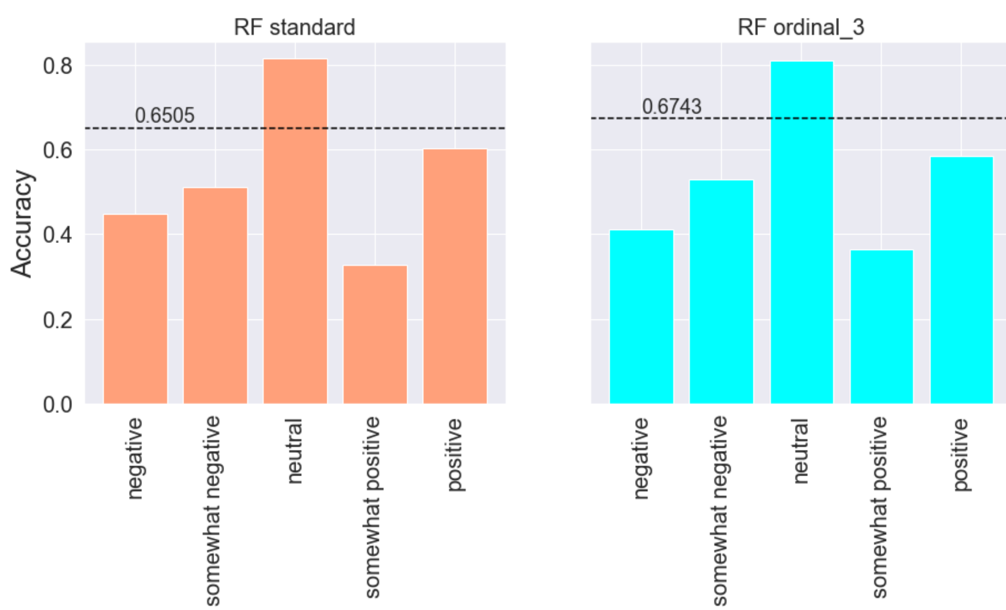

Figure 5. Accuracy for each sentiment label by applying standard random forest (left panel) and ordinal (scenario 3) random forest (right panel). The black dashed line in each plot indicates the overall accuracy for the corresponding classifier.

One potential issue with the set-up of ordinal classification is that the binary classification (one vs the rest) for the first and the last class is highly imbalanced, which potentially yield reduced accuracy for the first and the last class (as we see in LR and RF; Figure 3 and Figure 5). Additionally, it is likely that phrases from "negative" class are very similar to "somewhat negative" class, and phrases from "positive" class are very similar to "somewhat positive" class. As a result, we tried to merge the labels for those two pairs (new labels with "negative", "neutral", and "positive") and see whether the overall accuracy will be improved for the ordinal 3 LR classifiers. Our results indicated that ordinal3 LR classifier produced overall accuracy of 0.7161 for the merged dataset, which is higher than the accuracy produced from the un-merged dataset with 5 labels. To convert the classification problem back to 5-label classification, we then applied binary LR classifier to further divide the "negative" class into "negative" and "somewhat negative", and "positive" class into "positive" and "somewhat positive". In other words, 3 classifiers were trained on the training data, and the testing data were then fitted sequentially. Our final overall accuracy of LR classifier through this approach was 0.6377, which is lower than the reported accuracies in table 2.

4.3 **Final Kaggle Submission**

As mentioned in the Methods section 3.2, we used the original training set to perform all the classification work without using the original test set because it has no true labels. In the end of this project, we were able to obtain ordinal (scenario 3) random forest with its tuned parameters (tree = 400) as the best classifier. For the Kaggle submission, the original test dataset was converted to numerical vectors via the same sentence transformer process, the ordinal (scenario 3) random forest with 400 trees was used. The competition was successfully submitted, and we achieved a 0.65267 accuracy score which is ranked as 116th on the leaderboard.

5 Conclusion

Overall speaking, the Rotten Tomatoes dataset is very challenging to classify since the individual training data do not contain complete sentences but rather parsed phrases. Especially, we observed that many of the training data only include a few stop words or even punctuation, and this limited information can belong to different sentiment labels. Additionally, review comments normally involve sentence negation, sarcasm, terseness, language ambiguity, and more, which makes sentiment classification even more challenging. It is worth to mention that the current highest classification accuracy reported on Kaggle for this open competition is 0.76526.

After exploring different classification scenarios, we believe that for naturally ordered class labels (such as the sentimental label), ordinal classification based on cumulative probability potentially produce the best classification accuracy. Through this project, we also deeply feel the importance of parameter-tuning, which greatly impacts on the overall performance of the corresponding classifiers.

For future exploration, we plan to learn more about the more complicated deep learning models, CNN and RNN. We are aware that our current set up of the architecture of CNN and RNN may not be the most effective way to classify the Rotten Tomatoes dataset and there are many other potential parameters that can be tuned to further enhance the performance of the models. We will also look into different options to address the overfitting issue with our current set up of CNN and RNN.

References

Cardoso, J.S., and J.F. Pinto da Costa. 2007. Learning to Classify Ordinal Data: The Data Replication Method. Journal of Machine Learning Research 8 (2007) 1393 – 1429.

Cer, D., Y. Yang, S. Kong, N. Hua, N. Limtiaco, R.S. John, N. Constant, M. Guajardo-Cespedes, S. Yuan, C. Tar, Y.H. Sung, B. Strope, and R. Kurzweil. 2018. Universal Sentence Encoder. arXiv preprint arXiv:1803.11175.

Conneau, A., D. Kiela, H. Schwenk, L. Barrault, and A. Bordes. 2017. Supervised Learning of Universal Sentence Representations from Natural Language Inference Data. In Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, pages 670 – 680.

Frank E., and M. Hall. 2001. A simple Approach to Ordinal Classification. In Proceeding of the 12th European Conference on Machine Learning, Volume 1, 145 – 156.

Pang B., and L. Lee. 2005. Seeing Stars: Exploiting Class Relationships for Sentiment Categorization with Respect to Rating Scales. In ACL, 115 – 124.

Socher, R., A. Perelygin, J.Y. Wu, J. Chuang, C.D. Manning, A.Y. Ng, and C. Potts. 2013. Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank. EMNLP, 1631 – 1642.

The Sentence Transformer was run in Google Colab under GPU setting

In [ ]:
```
%%bash
pip install -q transformers
```

In [ ]:
```
!pip install -U sentence-transformers
```

In [ ]:
```
## load transformers
import numpy as np
import pandas as pd
from sklearn import preprocessing
from timeit import default_timer
import tensorflow as tf
import tensorflow.keras as keras
import io
import pickle
```

In [ ]:
```
## load training data set
from google.colab import files
uploaded_train = files.upload()
```

In [ ]:
```
## load testing data set
from google.colab import files
uploaded_test = files.upload()
```

In [ ]:
```
## read the datasets
train = pd.read_csv(io.BytesIO(uploaded_train['train.tsv']), sep='\t')
test = pd.read_csv(io.BytesIO(uploaded_test['test.tsv']), sep = '\t')
```

In [ ]:
```
from sentence_transformers import SentenceTransformer
#there are about 26 pretrained models
#roberta-large-nli-stsb-mean-tokens - returns 1024 dimentional vector
#distilbert-base-nli-stsb-mean-tokens - returns 768 dimentional vector

pretrained_model = 'roberta-large-nli-stsb-mean-tokens' #STSb performanc
e is highest
model = SentenceTransformer(pretrained_model)
```

In [ ]:
```
TRANSFORMER_BATCH=128

def count_embedd (df):
    idx_chunk=list(df.columns).index('Phrase')
    embedd_lst = []
    for index in range (0, df.shape[0], TRANSFORMER_BATCH):
        embedds = model.encode(df.iloc[index:index+TRANSFORMER_BATCH, id
x_chunk].values, show_progress_bar=False)
        embedd_lst.append(embedds)
    return np.concatenate(embedd_lst)
```

In [ ]:
```
# sentence embeddings for TRAIN dataset, 1024 dimentions each
start_time = default_timer()
train_embedd = count_embedd(train)
print("Train embeddings: {}: in: {:5.2f}s".format(train_embedd.shape, de
fault_timer() - start_time))
```

In [ ]:
```
# sentence embeddings for TEST dataset, 1024 dimentions each
start_time = default_timer()
test_embedd = count_embedd(test)
print("Test embeddings: {}: in: {:5.2f}s".format(test_embedd.shape, defa
ult_timer() - start_time))
```

In [ ]:
```
#save the train_embedd content into local
import pickle
with open('train_embedd.pickle', 'wb') as f:
    pickle.dump(train_embedd, f)
```

In [ ]:
```
#save the test_embedd content into local
import pickle
with open('test_embedd.pickle', 'wb') as f:
    pickle.dump(test_embedd, f)
```

The following work were all run in the local Jupter notebook

In [ ]:
```
import numpy as np
import pandas as pd
```

In [ ]:
```
train = pd.read_csv('train.tsv', sep='\t') # read the original train dat
aset
test = pd.read_csv('test.tsv', sep= '\t') # read the original test datas
et
```

In [ ]:
```
# load the sentence-transformed training dataset
import pickle
with open('train_embedd.pickle', 'rb') as f:
    train_embedd = pickle.load(f)
```

In [ ]:
```
# load the sentence-transformed testing dataset
import pickle
with open('test_embedd.pickle', 'rb') as f:
    test_embedd = pickle.load(f)
```

sentence-transformed training dataset split into "train" and "test" datasets to balance the number of phrases among classes

In [ ]:
```
Xtr = train_embedd
ytr = train['Sentiment']
c0 = Xtr[ytr == 0] # class 0
c1 = Xtr[ytr == 1] # class 1
c2 = Xtr[ytr == 2] # class 2
c3 = Xtr[ytr == 3] # class 3
c4 = Xtr[ytr ==4] # class 4
```

In [ ]:
```
from sklearn.model_selection import train_test_split
## train and test split according to the fix ratio for each class
Xtr_0, Xtst_0, ytr_0, ytst_0 = train_test_split(c0, ytr[ytr==0], test_si
ze = 1/3, random_state = 42)
Xtr_1, Xtst_1, ytr_1, ytst_1 = train_test_split(c1, ytr[ytr==1], test_si
ze = 2/3, random_state = 42)
Xtr_2, Xtst_2, ytr_2, ytst_2 = train_test_split(c2, ytr[ytr==2], test_si
ze = 4/5, random_state = 42)
Xtr_3, Xtst_3, ytr_3, ytst_3 = train_test_split(c3, ytr[ytr==3], test_si
ze = 3/4, random_state = 42)
Xtr_4, Xtst_4, ytr_4, ytst_4 = train_test_split(c4, ytr[ytr==4], test_si
ze = 1/3, random_state = 42)
```

In [ ]:
```
Xtr_new = np.concatenate((Xtr_0, Xtr_1, Xtr_2, Xtr_3, Xtr_4), axis = 0)
# new training
# Xtr_new.shape
ytr_new = np.concatenate((ytr_0, ytr_1, ytr_2, ytr_3, ytr_4), axis = 0)
# new training labels
# ytr_new.shape
```

In [ ]:
```
Xtst_new = np.concatenate((Xtst_0, Xtst_1, Xtst_2, Xtst_3, Xtst_4), axis
= 0) # new testing
ytst_new = np.concatenate((ytst_0, ytst_1, ytst_2, ytst_3, ytst_4), axis
= 0) # new testing labels
```

Implement Ordinal Classification Scenario 1

In [ ]:
```
class OrdinalClassifierS1():

    def __init__(self, clf):
        self.clf = clf
        self.clfs = {}

    def fit(self, X, y):
        self.unique_class = np.sort(np.unique(y))
        if self.unique_class.shape[0] > 2:
            for i in range(self.unique_class.shape[0]-1):
                # for each k - 1 ordinal value we fit a binary classific
ation problem
                binary_y = (y > self.unique_class[i]).astype(np.uint8)
                clf = clone(self.clf)
                clf.fit(X, binary_y)
                self.clfs[i] = clf

    def predict_proba(self, X):
        clfs_predict = {k:self.clfs[k].predict_proba(X) for k in self.cl
fs}
        predicted = []
        for i,y in enumerate(self.unique_class):
            if i == 0:
                # V1 = 1 - Pr(y > V1)
                predicted.append(1 - clfs_predict[y][:,1])
            elif y in clfs_predict:
                #Vi = Pr(y > Vi-1) - Pr(y > Vi)
                 predicted.append(clfs_predict[y-1][:,1] - clfs_predict[
y][:,1])
            else:
                # Vk = Pr(y > Vk-1)
                predicted.append(clfs_predict[y-1][:,1])
        return np.vstack(predicted).T

    def predict(self, X):
        return np.argmax(self.predict_proba(X), axis=1)
```

Implement Ordinal Classification Scenario 2

In [ ]:
```
class OrdinalClassifierS2():

    def __init__(self, clf):
        self.clf = clf
        self.clfs = {}

    def fit(self, X, y):
        self.unique_class = np.sort(np.unique(y))
        if self.unique_class.shape[0] > 2:
            for i in range(self.unique_class.shape[0]-1):
                # for each k - 1 ordinal value we fit a binary classific
ation problem
                binary_y = (y > self.unique_class[i]).astype(np.uint8)
                clf = clone(self.clf)
                clf.fit(X, binary_y)
                self.clfs[i] = clf

    def predict_proba(self, X):
        clfs_predict = {k:self.clfs[k].predict_proba(X) for k in self.cl
fs}
        predicted = []
        for i,y in enumerate(self.unique_class):
            if i == 0:
                # V1 = 1 - Pr(y > V1)
                predicted.append(1 - clfs_predict[y][:,1])
            elif y in clfs_predict:
#                 Vi = (1-Pr(y > Vi-1))*Pr(y > Vi-1)
                 predicted.append((1-clfs_predict[y][:,1])*clfs_predict[
y-1][:,1])
            else:
                # Vk = Pr(y > Vk-1)
                predicted.append(clfs_predict[y-1][:,1])
        return np.vstack(predicted).T

    def predict(self, X):
        return np.argmax(self.predict_proba(X), axis=1)
```

Implement Ordinal Classification Scenario 3

In [ ]:
```
class OrdinalClassifierS3():

    def __init__(self, clf):
        self.clf = clf
        self.clfs = {}

    def fit(self, X, y):
        self.unique_class = np.sort(np.unique(y))
        if self.unique_class.shape[0] > 2:
            for i in range(self.unique_class.shape[0]-1):
                # for each k - 1 ordinal value we fit a binary classific
ation problem
                binary_y = (y > self.unique_class[i]).astype(np.uint8)
                clf = clone(self.clf)
                clf.fit(X, binary_y)
                self.clfs[i] = clf

    def predict_proba(self, X):
        clfs_predict = {k:self.clfs[k].predict_proba(X) for k in self.cl
fs}
        predicted = []
        for y in self.unique_class:
            if y!=max(self.unique_class):
                predicted.append(clfs_predict[y][:, 0])
            else:
                predicted.append([1]*len(X))
        return np.vstack(predicted).T

    def predict(self, X):
        tmp = self.predict_proba(X)
        boo = tmp>=0.5
        return boo.argmax(axis = 1)
```

In [ ]:
```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.metrics import confusion_matrix
from timeit import default_timer
```

Run classfifiers LDA, QDA, GNB, Logistic Regression, Linear SVM, Random Forest, Adaboost, Neural Networks

- standard classification
- ordinal classification scenario 1, 2, 3
- Ordinal Linear SVM was too time-consuming and we only present standard Linear SVM
- Ordinal Adaboost was too time-consuming and we only present standard Adaboost and Ordinal 1 scenario

In [ ]:
```
#### LDA
clf_r = LinearDiscriminantAnalysis() # standard LDA
clf_o1 = OrdinalClassifierS1(clf_r) # ordinal 1 LDA
clf_o2 = OrdinalClassifierS2(clf_r) # ordinal 2 LDA
clf_o3 = OrdinalClassifierS3(clf_r) # ordinal 3 LDA

LDA = [clf_r, clf_o1, clf_o2, clf_o3]

time_lda = [] # run time
accuracy_lda = [] # accuracy

for clf in LDA:
    start_time = default_timer()
    clf.fit(Xtr_new, ytr_new)
    predict = clf.predict(Xtst_new)
    accuracy = np.sum(predict == ytst_new)/len(ytst_new)
    accuracy_lda.append(accuracy)
    time_lda.append(default_timer() - start_time)
    print(accuracy_lda)
    print(time_lda)
```

In [ ]:
```
#### QDA
clf_r = QuadraticDiscriminantAnalysis() # standard QDA
clf_o1 = OrdinalClassifierS1(clf_r) # ordinal 1 QDA
clf_o2 = OrdinalClassifierS2(clf_r) # ordinal 2 QDA
clf_o3 = OrdinalClassifierS3(clf_r) # ordinal 3 QDA

QDA = [clf_r, clf_o1, clf_o2, clf_o3]

time_qda = [] # run time
accuracy_qda = [] # accuracy

for clf in QDA:
    start_time = default_timer()
    clf.fit(Xtr_new, ytr_new)
    predict = clf.predict(Xtst_new)
    accuracy = np.sum(predict == ytst_new)/len(ytst_new)
    accuracy_qda.append(accuracy)
    time_qda.append(default_timer() - start_time)
    print(accuracy_qda)
    print(time_qda)
```

In [ ]:
```
#### Guassian Naive Bayes
clf_r = GaussianNB() # standard GNB
clf_o1 = OrdinalClassifierS1(clf_r) # ordinal 1 GNB
clf_o2 = OrdinalClassifierS2(clf_r) # ordinal 2 GNB
clf_o3 = OrdinalClassifierS3(clf_r) # ordinal 3 GNB

GNB = [clf_r, clf_o1, clf_o2, clf_o3]

time_gnb = [] # run time
accuracy_gnb = [] # accuracy

for clf in GNB:
    start_time = default_timer()
    clf.fit(Xtr_new, ytr_new)
    predict = clf.predict(Xtst_new)
    accuracy = np.sum(predict == ytst_new)/len(ytst_new)
    accuracy_gnb.append(accuracy)
    time_gnb.append(default_timer() - start_time)
    print(accuracy_gnb)
    print(time_gnb)
```

In [ ]:
```
#### Guassian Naive Bayes
clf_r = GaussianNB() # standard GNB
clf_o1 = OrdinalClassifierS1(clf_r) # ordinal 1 GNB
clf_o2 = OrdinalClassifierS2(clf_r) # ordinal 2 GNB
clf_o3 = OrdinalClassifierS3(clf_r) # ordinal 3 GNB

GNB = [clf_r, clf_o1, clf_o2, clf_o3]

time_gnb = [] # run time
accuracy_gnb = [] # accuracy

for clf in GNB:
    start_time = default_timer()
    clf.fit(Xtr_new, ytr_new)
    predict = clf.predict(Xtst_new)
    accuracy = np.sum(predict == ytst_new)/len(ytst_new)
    accuracy_gnb.append(accuracy)
    time_gnb.append(default_timer() - start_time)
    print(accuracy_gnb)
    print(time_gnb)
```

In [ ]:
```
#### Logistic Regression
clf_r = LogisticRegression(C = 2e-5, solver = 'lbfgs', max_iter=500) # s
tandard LR, C has been tuned and the optimal was used
clf_o1 = OrdinalClassifierS1(clf_r) # ordinal 1 LR
clf_o2 = OrdinalClassifierS2(clf_r) # ordinal 2 LR
clf_o3 = OrdinalClassifierS3(clf_r) # ordinal 3 LR

LR = [clf_r, clf_o1, clf_o2, clf_o3]

time_lr = [] # run time
accuracy_lr = [] # accuracy

for clf in LR:
    start_time = default_timer()
    clf.fit(Xtr_new, ytr_new)
    predict = clf.predict(Xtst_new)
    accuracy = np.sum(predict == ytst_new)/len(ytst_new)
    accuracy_lr.append(accuracy)
    time_lr.append(default_timer() - start_time)
    print(accuracy_lr)
    print(time_lr)
```

In [ ]:
```
#### Linear SVM, only standard, ordinal SVM was too time-consuming, we o
nly present the standard version
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.multiclass import OneVsOneClassifier
from sklearn.svm import LinearSVC
from sklearn.svm import SVC

C = 2*np.logspace(-8, 5, 14) # tune regulazation C
accuracy_svm= []
time_svm = []

for i in C:
    start_time = default_timer()
    clf = OneVsOneClassifier(LinearSVC(C = i))
    clf.fit(Xtr_new, ytr_new)
    predict = clf.predict(Xtst_new)
    accuracy_svm.append(np.sum(predict == ytst_new)/len(ytst_new))
    time_svm.append(default_timer() - start_time)
    print(accuracy_svm)
    print(time_svm)
```

In [ ]:
```
### Neural Networks
## regular NN found the highest accuracy = 0.6614927079333042,
## learning_rate_init=4.64158883e-02, and alpha = 1.66810054e-01

clf_r = MLPClassifier(hidden_layer_sizes=(100, ), activation='relu',
                       random_state=1, max_iter=1000,
                      learning_rate='constant',
                      learning_rate_init=4.64158883e-02,
                      alpha = 1.66810054e-01,
                      batch_size=200) # standard NN, parameters were tu
ned and the optimals were used
clf_o1 = OrdinalClassifierS1(clf_r) # ordinal 1 NN
clf_o2 = OrdinalClassifierS2(clf_r) # ordinal 2 NN
clf_o3 = OrdinalClassifierS3(clf_r) # ordinal 3 NN

NN = [clf_r, clf_o1, clf_o2, clf_o3]

time_nn = [] # run time
accuracy_nn = [] # accuracy

for clf in NN:
    start_time = default_timer()
    clf.fit(Xtr_new, ytr_new)
    predict = clf.predict(Xtst_new)
    accuracy = np.sum(predict == ytst_new)/len(ytst_new)
    accuracy_nn.append(accuracy)
    time_nn.append(default_timer() - start_time)
    print(accuracy_nn)
    print(time_nn)
```

In [ ]:
```
### Random Forest
##ordinal RF 3 with 400 number trees found the highest accuracy = 0.6743

#standard RF
estimators=[50, 100, 150, 200, 250, 300, 350, 400, 450, 500]
score_per_tree=[]
for n in estimators:
    rf = OrdinalClassifier(RandomForestClassifier(n_estimators=n))
    rf.fit(X_train, y_train)
    pred=rf.predict(X_test)
    score= np.sum(pred == y_test)/len(y_test)
    score_per_tree.append(score)
    print(score_per_tree)

confusion_matrix(y_test, pred)


#ordinal RF 1
score_per_tree=[]
for n in estimators:
    rf = OrdinalClassifierS1(RandomForestClassifier(n_estimators=n))
    rf.fit(X_train, y_train)
    pred=rf.predict(X_test)
    score= np.sum(pred == y_test)/len(y_test)
    score_per_tree.append(score)
    print(score_per_tree)

#ordinal RF 2
score_per_tree=[]
for n in estimators:
    rf = OrdinalClassifierS2(RandomForestClassifier(n_estimators=n))
    rf.fit(X_train, y_train)
    pred=rf.predict(X_test)
    score= np.sum(pred == y_test)/len(y_test)
    score_per_tree.append(score)
    print(score_per_tree)

#ordinal RF 3
score_per_tree=[]
for n in estimators:
    rf = OrdinalClassifierS3(RandomForestClassifier(n_estimators=n))
    rf.fit(X_train, y_train)
    pred=rf.predict(X_test)
    score= np.sum(pred == y_test)/len(y_test)
    score_per_tree.append(score)
    print(score_per_tree)

confusion_matrix(y_test, pred)
```

In [ ]:
```
#### Adaboost, only standard, and limited ordinal Adaboost
#### ordinal Adaboost was too time-consuming

#standard Adaboost
estimators=list(range(100, 501,50))
score_per_tree=[]
for n in estimators:
    ab = AdaBoostClassifier(base_estimator=DecisionTreeClassifier(min_samples_leaf=1, max_depth=10),
                                                  n_estimators=n)
    ab.fit(X_train, y_train)
    pred=ab.predict(X_test)
    score= np.sum(pred == y_test)/len(y_test)
    score_per_tree.append(score)
    print(score_per_tree)

#ordinal Adaboost 1
score_per_tree=[]
for n in estimators:
    ab = OrdinalClassifierS1(AdaBoostClassifier(base_estimator=DecisionTreeClassifier(min_samples_leaf=1, max_depth=10),
                                                  n_estimators=n))
    ab.fit(X_train, y_train)
    pred=ab.predict(X_test)
    score= np.sum(pred == y_test)/len(y_test)
    score_per_tree.append(score)
    print(score_per_tree)
```

Exploration: here we tried to merge the classes (merged 0 and 1, and 3 and 4) and do ordinal3 LR classification on training data with 3 labels. Then we apply binary LR to the 0 and 1 classes, and 3 and 4 classes to evaluate the overall accuracy

In [ ]:
```
#merge the negative and somewhat negative classes, and somewhat positive and positive classes
y_train2=np.array(y_train, copy=True)
y_test2=np.array(y_test, copy=True)
y_test3=np.array(y_test, copy=True)

for i in range(len(y_train2)):
    if y_train2[i]==1:
        y_train2[i]=0
    elif y_train2[i]==2:
        y_train2[i]=1
    elif y_train2[i]==3 or y_train2[i]==4:
        y_train2[i]=2



for i in range(len(y_test3)):
    if y_test3[i]==1:
        y_test3[i]=0
    elif y_test3[i]==2:
        y_test3[i]=1
    elif y_test3[i]==3 or y_test3[i]==4:
        y_test3[i]=2
```

In [ ]:
```
##train and fit the ordinal3 LR for merged 3 classes data
##we tuned the C and C=2e-4 gave us the best accuracy
lr = OrdinalClassifierS3(LogisticRegression(C= 2e-4, solver="lbfgs", max_iter=500))
lr.fit(X_train, y_train2)
pred=lr.predict(X_test)
score= np.sum(pred == y_test3)/len(y_test3)
print(score)  #0.7161140197789413, higher than ordinal3 LR on 5 labels
```

In [ ]:
```
##train the binary LR model for class 0 and 1
##again, C=2e-4 gave us the best accuracy
lr01=LogisticRegression(C= 2e-4, solver="lbfgs", max_iter=500)
index01=y_train2==0
lr01.fit(X_train[index01], y_train[index01])
```

In [ ]:
```
##train the binary LR model for class 3 and 4
##again, C=2e-4 gave us the best accuracy
lr34=LogisticRegression(C= 2e-4, solver="lbfgs", max_iter=500)
index34=y_train2==2
lr34.fit(X_train[index34], y_train[index34])
```

In [ ]:
```
#get the index for corresponding class predictions for test data
pred2_index=pred==1
pred01_index=pred==0
pred34_index=pred==2

#fit binary LR on test data
pred01=lr01.predict(X_test[pred01_index])
pred34=lr34.predict(X_test[pred34_index])

#create the final predicted label
y_test2[pred2_index]=2
y_test2[pred01_index]=pred01
y_test2[pred34_index]=pred34

#calculate the overall accuracy
score= np.sum(y_test2 == y_test)/len(y_test)
print(score) #0.6377142345728734, the accuracy is lower than standard LR
and all evaluated ordinal LR
```

RNN

In [ ]:
```
from keras.preprocessing.sequence import pad_sequences
from keras.preprocessing.text import text_to_word_sequence
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.models import Sequential
from keras.layers import Dense,LSTM
from keras.layers.embeddings import Embedding
from keras.layers import Embedding, LSTM, Dense, Dropout
```

In [ ]:
```
#load the data
test_directory = 'sentiment-analysis-on-movie-reviews/test.tsv/test.tsv'
train_directory='sentiment-analysis-on-movie-reviews/train.tsv'
test_raw= pd.read_csv(test_directory, sep='\t')
train_raw=pd.read_csv(train_directory, sep='\t')
train_label=train_raw["Sentiment"]
#drop unecessary columns
train_raw.drop(['PhraseId','SentenceId'],inplace = True,axis='columns')


#convert sentences to tokenized words
for i in range(len(train_raw['Phrase'])):
    train_raw['Phrase'][i] = text_to_word_sequence(train_raw['Phrase'][i
])


#convert tokenized words to numeric form required for model building
tokenizer = Tokenizer(num_words=5000)
tokenizer.fit_on_texts(train_raw['Phrase'])

train_raw['Phrase'] = tokenizer.texts_to_sequences(train_raw['Phrase'])

#convert each tokenized review into an input of the same length = 100 by
padding with 0s in the begining
max_length = 100
train_copy = train_raw['Phrase']
train_copy = pad_sequences(train_raw['Phrase'],maxlen=max_length)
vocab_size = len(tokenizer.word_index) + 1

X = train_copy
y =np.array(train_raw['Sentiment'])
```

In [ ]:
```
#resamplep to downsize the training data
index0=np.where(y==0)
y0=y[index0]
x0=X[index0]
X_train_0, X_test_0, y_train_0, y_test_0 = train_test_split(x0, y0, test
_size=0.33, random_state=42)

index1=np.where(y==1)
y1=y[index1]
x1=X[index1]
X_train_1, X_test_1, y_train_1, y_test_1 = train_test_split(x1, y1, test
_size=0.66, random_state=42)

index2=np.where(y==2)
y2=y[index2]
x2=X[index2]
X_train_2, X_test_2, y_train_2, y_test_2 = train_test_split(x2, y2, test
_size=0.8, random_state=42)

index3=np.where(y==3)
y3=y[index3]
x3=X[index3]
X_train_3, X_test_3, y_train_3, y_test_3 = train_test_split(x3, y3, test
_size=0.75, random_state=42)

index4=np.where(y==4)
y4=y[index4]
x4=X[index4]
X_train_4, X_test_4, y_train_4, y_test_4 = train_test_split(x4, y4, test
_size=0.33, random_state=42)

#concatenate the new training set labels
y_train=np.vstack((y_train_0.reshape((4738,1)),y_train_1.reshape((9272,1
))))
y_train=np.vstack((y_train,y_train_2.reshape((15916,1))))
y_train=np.vstack((y_train,y_train_3.reshape((8231,1))))
y_train=np.vstack((y_train,y_train_4.reshape(6168,1)))
y_train=y_train.flatten()

#concatenate the new setting set data
X_train=np.vstack((X_train_0,X_train_1))
X_train=np.vstack((X_train,X_train_2))
X_train=np.vstack((X_train,X_train_3))
X_train=np.vstack((X_train,X_train_4))

#concatenate the new testing set labels
y_test=np.vstack((y_test_0.reshape((2334,1)),y_test_1.reshape((18001,1
))))
y_test=np.vstack((y_test,y_test_2.reshape((63666,1))))
y_test=np.vstack((y_test,y_test_3.reshape((24696,1))))
y_test=np.vstack((y_test,y_test_4.reshape((3038,1))))
y_test=y_test.flatten()

#concatenate the new testing set data
X_test=np.vstack((X_test_0,X_test_1))
X_test=np.vstack((X_test,X_test_2))
```

```
X_test=np.vstack((X_test,X_test_3))
X_test=np.vstack((X_test,X_test_4))

#turn the label into one hot code
y_train_hot = np.zeros((y_train.size, y_train.max()+1))
y_train_hot[np.arange(y_train.size),y_train] = 1

y_test_hot = np.zeros((y_test.size, y_test.max()+1))
y_test_hot[np.arange(y_test.size),y_test] = 1
```

In [ ]:

```
#below model design produced the best accuracy
model2 = Sequential()
model2.add(Embedding(input_dim=vocab_size,
                    output_dim=embedding_vector_length,
                    input_length=max_length))
model2.add(LSTM(100))
model2.add(Dropout(0.5))
model2.add(Dense(5,activation = 'softmax'))
model2.compile(loss = 'categorical_crossentropy',
                        optimizer = 'adam',
                        metrics=['accuracy'])
model2.summary()

#fit and test the model
train_history=model2.fit(x=X_train,y=y_train_hot,batch_size=64,epochs=10
,
                         verbose=2,validation_data=(X_test,y_test_hot))
```

CNN

In [ ]:

```
import re
import string
import nltk
from nltk import word_tokenize
import gensim
from keras.layers import Dense, Dropout, Reshape, Flatten, concatenate,
Input, Conv1D, GlobalMaxPooling1D, Embedding
from keras.models import Model
```

In [ ]:
```
#load the data
test_directory = 'sentiment-analysis-on-movie-reviews/test.tsv/test.tsv'
train_directory='sentiment-analysis-on-movie-reviews/train.tsv'
test_raw= pd.read_csv(test_directory, sep='\t')
train_raw=pd.read_csv(train_directory, sep='\t')
train_label=train_raw["Sentiment"]

#remove punctuation
def remove_punct(text):
    text_nopunct = ''
    text_nopunct = re.sub('['+string.punctuation+']', '', text)
    return text_nopunct
train_raw['Text_Clean'] = train_raw['Phrase'].apply(lambda x: remove_pun
ct(x))

#Tokenize
#nltk.download('punkt')
tokens = [word_tokenize(sen) for sen in train_raw.Text_Clean]

#lower case all tokens
def lower_token(tokens):
    return [w.lower() for w in tokens]

lower_tokens = [lower_token(token) for token in tokens]

train_raw['Text_Final'] = [' '.join(sen) for sen in lower_tokens]
train_raw['tokens'] = lower_tokens
```

In [ ]:

```
#we add five one hot encoded columns to our data frame, corresponding to the 5 classes
neg=[]
som_neg=[]
neu=[]
som_pos=[]
pos = []

for l in train_raw.Sentiment:
    if l == 0:
        neg.append(1)
        som_neg.append(0)
        neu.append(0)
        som_pos.append(0)
        pos.append(0)
    elif l == 1:
        neg.append(0)
        som_neg.append(1)
        neu.append(0)
        som_pos.append(0)
        pos.append(0)
    elif l==2:
        neg.append(0)
        som_neg.append(0)
        neu.append(1)
        som_pos.append(0)
        pos.append(0)
    elif l==3:
        neg.append(0)
        som_neg.append(0)
        neu.append(0)
        som_pos.append(1)
        pos.append(0)
    elif l==4:
        neg.append(0)
        som_neg.append(0)
        neu.append(0)
        som_pos.append(0)
        pos.append(1)

train_raw['neg']= neg
train_raw['som_neg']= som_neg
train_raw['neu']=neu
train_raw['som_pos']= som_pos
train_raw['pos']= pos

train_raw = train_raw[['Text_Final', 'tokens', 'Sentiment', 'neg', 'som_neg','neu','som_pos','pos']]
train_raw.head()
```

In [ ]:
```
#resample to downsize the training data
df_train0, df_test0 = train_test_split(
  train_raw.loc[train_raw['Sentiment'] == 0],
  test_size=0.33,
  random_state=42
)

df_train1, df_test1 = train_test_split(
  train_raw.loc[train_raw['Sentiment'] == 1],
  test_size=0.66,
  random_state=42
)

df_train2, df_test2 = train_test_split(
  train_raw.loc[train_raw['Sentiment'] == 2],
  test_size=0.8,
  random_state=42
)

df_train3, df_test3 = train_test_split(
  train_raw.loc[train_raw['Sentiment'] == 3],
  test_size=0.75,
  random_state=42
)

df_train4, df_test4 = train_test_split(
  train_raw.loc[train_raw['Sentiment'] == 4],
  test_size=0.33,
  random_state=42
)

#concatenating
data_train=pd.concat([df_train0, df_train1,df_train2,df_train3,df_train4
])
data_test=pd.concat([df_test0, df_test1,df_test2,df_test3,df_test4])
```

In [ ]:
```
##get maximum training sentence length
all_training_words = [word for tokens in data_train["tokens"] for word i
n tokens]
training_sentence_lengths = [len(tokens) for tokens in data_train["token
s"]]
TRAINING_VOCAB = sorted(list(set(all_training_words)))
print("%s words total, with a vocabulary size of %s" % (len(all_training
_words), len(TRAINING_VOCAB)))
print("Max sentence length is %s" % max(training_sentence_lengths))

##get maximum testing sentence length
all_test_words = [word for tokens in data_test["tokens"] for word in tok
ens]
test_sentence_lengths = [len(tokens) for tokens in data_test["tokens"]]
TEST_VOCAB = sorted(list(set(all_test_words)))
print("%s words total, with a vocabulary size of %s" % (len(all_test_wor
ds), len(TEST_VOCAB)))
print("Max sentence length is %s" % max(test_sentence_lengths))
```

In [ ]:
```
##load word2vec
word2vec_path = 'GoogleNews-vectors-negative300.bin.gz'
word2vec = gensim.models.KeyedVectors.load_word2vec_format(word2vec_path
, binary=True)

def get_average_word2vec(tokens_list, vector, generate_missing=False, k=
300):
    if len(tokens_list)<1:
        return np.zeros(k)
    if generate_missing:
        vectorized = [vector[word] if word in vector else np.random.rand
(k) for word in tokens_list]
    else:
        vectorized = [vector[word] if word in vector else np.zeros(k) fo
r word in tokens_list]
    length = len(vectorized)
    summed = np.sum(vectorized, axis=0)
    averaged = np.divide(summed, length)
    return averaged

def get_word2vec_embeddings(vectors, clean_comments, generate_missing=Fa
lse):
    embeddings = clean_comments['tokens'].apply(lambda x: get_average_wo
rd2vec(x, vectors,

generate_missing=generate_missing))
    return list(embeddings)

#get embeddings
training_embeddings = get_word2vec_embeddings(word2vec, data_train, gene
rate_missing=True)
```

In [ ]:
```
#Tokenize and Pad sequences
MAX_SEQUENCE_LENGTH=50
tokenizer = Tokenizer(num_words=len(TRAINING_VOCAB), lower=True, char_le
vel=False)
tokenizer.fit_on_texts(data_train["Text_Final"].tolist())
training_sequences = tokenizer.texts_to_sequences(data_train["Text_Fina
l"].tolist())
train_word_index = tokenizer.word_index
print("Found %s unique tokens." % len(train_word_index))
train_cnn_data = pad_sequences(training_sequences,
                               maxlen=MAX_SEQUENCE_LENGTH)

#get the initial embedding weights
train_embedding_weights = np.zeros((len(train_word_index)+1, EMBEDDING_D
IM))
for word,index in train_word_index.items():
    train_embedding_weights[index,:] = word2vec[word] if word in word2ve
c else np.random.rand(EMBEDDING_DIM)
print(train_embedding_weights.shape)

#determine the running sequences
test_sequences = tokenizer.texts_to_sequences(data_test["Text_Final"].to
list())
test_cnn_data = pad_sequences(test_sequences, maxlen=MAX_SEQUENCE_LENGTH
)
```

In [ ]:
```
#Now we will get embeddings from Google News Word2Vec model and save the
m corresponding to the sequence number
#we assigned to each word. If we could not get embeddings we save a rand
om vector for that word.

EMBEDDING_DIM = 300
train_embedding_weights = np.zeros((len(train_word_index)+1, EMBEDDING_D
IM))
for word,index in train_word_index.items():
    train_embedding_weights[index,:] = word2vec[word] if word in word2ve
c else np.random.rand(EMBEDDING_DIM)
    print(train_embedding_weights.shape)
```

In [ ]:
```
#Text as a sequence is passed to a CNN. The embeddings matrix is passed
 to embedding_layer.
#Five different filter sizes are applied to each comment, and GlobalMaxP
ooling1D layers are applied to each layer.
#All the outputs are then concatenated. A Dropout layer then Dense then
 Dropout and then Final Dense layer is applied.
#model.summary() will print a brief summary of all the layers with there
output shapes.

def ConvNet(embeddings, max_sequence_length, num_words, embedding_dim, l
abels_index):

    embedding_layer = Embedding(num_words,
                                embedding_dim,
                                weights=[embeddings],
                                input_length=max_sequence_length,
                                trainable=False)

    sequence_input = Input(shape=(max_sequence_length,), dtype='int32')
    embedded_sequences = embedding_layer(sequence_input)
    convs = []
    filter_sizes = [2,3,4,5,6]
    for filter_size in filter_sizes:
        l_conv = Conv1D(filters=200,
                        kernel_size=filter_size,
                        activation='relu')(embedded_sequences)
        l_pool = GlobalMaxPooling1D()(l_conv)
        convs.append(l_pool)
    l_merge = concatenate(convs, axis=1)
    x = Dropout(0.1)(l_merge)
    x = Dense(128, activation='relu')(x)
    x = Dropout(0.2)(x)
    preds = Dense(labels_index, activation='sigmoid')(x)
    model = Model(sequence_input, preds)
    model.compile(loss='binary_crossentropy',
                  optimizer='adam',
                  metrics=['acc'])
    model.summary()
    return model

label_names = ['pos', 'som_pos','neu','som_neg','neg']
model = ConvNet(train_embedding_weights,
                MAX_SEQUENCE_LENGTH,
                len(train_word_index)+1,
                EMBEDDING_DIM,
                len(list(label_names)))
```

In [ ]:
```
x_train = train_cnn_data
y_train = data_train[label_names].values
y_tr=y_train

#train CNN
num_epochs = 3
batch_size = 32
hist = model.fit(x_train,
                 y_tr,
                 epochs=num_epochs,
                 validation_split=0.1,
                 shuffle=True,
                 batch_size=batch_size)

#test CNN
predictions = model.predict(test_cnn_data, batch_size=1024, verbose=1)
labels = [0,1,2,3,4]
prediction_labels=[]
for p in predictions:
    prediction_labels.append(labels[np.argmax(p)])

sum(data_test.Sentiment==prediction_labels)/len(prediction_labels)
```