Warsaw University of Technology

Faculty of Electronics and Information Technology

Dimitrios Georgousis

Andrea Amato

Group 12

Introduction to Artificial Intelligence

Project 7: Sentiment analysis

Final report

Description

Sentiment analysis (or opinion mining) involves identifying and categorizing opinions expressed in a text to determine whether the attitude is positive, negative, or neutral. The task will involve classifying customer reviews from the Amazon Reviews dataset available at Kaggle, so we consider it to be a classification task and, more specifically, since our dataset does not include neutral reviews, it reduces to a binary classification task. This project aims to develop, compare and improve multiple machine learning models for sentiment analysis and determine the best-performing approach.

Dataset Description

The dataset is of the form: `__label__<X> <Title>: <Text>` and this pattern repeats. Labels can only be `__label__1` (negative) and `__label__2` (positive). Most of the reviews are in English, but there are a few in other languages, like Spanish.

Dataset for Final submission

We test various algorithms and preprocessing techniques and to do that in reasonable time, we use a small slice of the complete dataset (20.000 reviews) for Naive Bayes, LSTM, and DistilBERT training. This dataset is typically split in 80/20 fashion (80% for training + validation if needed and 20% for testing).

Label Balancing

We take reviews such that exactly half have positive and exactly half have negative sentiment.

Data Preprocessing

The initial data preprocessing is the same for all models/implementations:

- Keep only English reviews
- Replace emojis with descriptive words
- Remove HTML tags
- Remove URLs
- Remove hashtags
- Remove special characters and digits
- Covert the text to lowercase
- Remove stop-words
- Final DataFrame has 2 fields: `label` (sentiment value) and `text` (title + review content text combined)

Further preprocessing before feeding into models:

- Naive Bayes Models
    - 1: Count vectorizer applied to `text` inputs for different ngram ranges.
    - 2: TF-IDF vectorizer applied to `text` inputs
    - 3: The same initial embeddings of out `text` inputs as in the LSTM model are used

  In each of these 3 cases we train a Multinomial Naive Bayes and look at accuracy, classification report and confusion matrix of the final model's predictions on the test set.
- LSTM
    - We use a tokenizer (from the tensorflow.keras Python library) to turn our texts into token sequences
    - We pad these sequences to a fixed length
    - We create an embeddings weight matrix (a matrix which has precomputed embeddings for each word of our vocabulary, where vocabulary is set of words present in reviews). For LSTM we chose the GloVe embeddings instead of the BERT ones, as these are the embeddings usually used with LSTM models as our online searches seem to indicate. Also, there are some library/implementation differences between the BERT embeddings and the input accepted by the LSTM layers that would require further processing of the embeddings hindering performance and perhaps the accuracy of the resulting weight matrix
    - Finally, this embeddings weight matrix is given to the model, while the input to the LSTM model remains sequences of tokens
- DistilBERT
    - Tokenization is performed using the DistilBertTokenizer from the Hugging Face library, which converts the cleaned text data into token sequences.
    - These token sequences are then padded to a fixed length of 512 tokens to ensure uniform input size for the DistilBERT model.
    - The tokenized data is split into training and testing sets, with 80% of the data used for training and 20% for testing.
    - The tokenized inputs include input_ids and attention_mask which are used by the DistilBERT model to focus on the relevant parts of the text during training and evaluation.

Performance Evaluation

Metrics

The task is binary classification and appropriate evaluation methods will be used. Metrics such as:

- Accuracy: describing the number of correct predictions over all predictions
- Precision: measure of how many of the positive predictions are true positives
- Recall: measure of how many of the positive cases the classifier predicted correctly
- F1-Score: harmonic mean of precision and recall

The confusion matrix supplies useful information to us too.

Also, for models trained in epochs we can look at the information about their training the way their loss and accuracy developed through different epochs.

Comments on Results

Classification with Naive Bayes

As was established in our earlier efforts/deliverables (midterm etc.) the multinomial Naive Bayes model seems to be best fit for the sentiment analysis task (at least as long as Naive Bayes models are concerned).

Count Vectorizer: ngrams is a tuple of two values that shows us what combinations of our vocabulary are made by the vectorizer. We test for ngram values of:

- (1,1): use all possible words
- (2,2): use all possible pairs of words
- (3,3): use all possible triples of words

We show the according results:

```
Multinomial Naive Bayes (MNB) with ngram_range=(1, 1)
Accuracy: 0.8385964912280702


              precision    recall  f1-score   support

           0       0.83      0.84      0.84      1945
           1       0.85      0.83      0.84      2045

    accuracy                           0.84      3990
   macro avg       0.84      0.84      0.84      3990
weighted avg       0.84      0.84      0.84      3990

[[1641  304]
 [ 340 1705]]
```

```
Multinomial Naive Bayes (MNB) with ngram_range=(2, 2)
Accuracy: 0.8095238095238095


              precision    recall  f1-score   support

           0       0.84      0.75      0.79      1945
           1       0.78      0.87      0.82      2045

    accuracy                           0.81      3990
   macro avg       0.81      0.81      0.81      3990
weighted avg       0.81      0.81      0.81      3990

[[1460  485]
 [ 275 1770]]
```

```
Multinomial Naive Bayes (MNB) with ngram_range=(3, 3)
Accuracy: 0.6265664160401002


              precision    recall  f1-score   support

           0       0.78      0.32      0.46      1945
           1       0.59      0.92      0.72      2045

    accuracy                           0.63      3990
   macro avg       0.69      0.62      0.59      3990
weighted avg       0.68      0.63      0.59      3990

[[ 627 1318]
 [ 172 1873]]
```

TF-IDF vectorizer:

```
Multinomial Naive Bayes (MNB) with TfidfVectorizer
Accuracy: 0.849874686716792


              precision    recall  f1-score   support

           0       0.83      0.87      0.85      1945
           1       0.87      0.83      0.85      2045

    accuracy                           0.85      3990
   macro avg       0.85      0.85      0.85      3990
weighted avg       0.85      0.85      0.85      3990

[[1696  249]
 [ 350 1695]]
```

Input as embeddings generated from the LSTM embeddings (made with GloVe):

```
Multinomial Naive Bayes (MNB) with embeddings
Accuracy: 0.5959899749373434


              precision    recall  f1-score   support

           0       0.59      0.58      0.58      1945
           1       0.61      0.61      0.61      2045

    accuracy                           0.60      3990
   macro avg       0.60      0.60      0.60      3990
weighted avg       0.60      0.60      0.60      3990

[[1131  814]
 [ 798 1247]]
```

Count Vectorizer

It seems that as we go from unigrams to bigrams and trigrams the performance of our model is seriously impacted. We can attribute this result to the fact that unigrams manage to capture the sparsity of data better (since they are individual words) and are less prone to overfitting. The other two approaches use more than one words combined together which makes then prone to 'picking up' noise and specific idiosyncrasies of the training data. Also, our vocabulary size plays a role here. As we move to bigrams and trigrams we expand the feature space, but the amount of input data stays the same (we always use the same reviews). It is more challenging for a model to handle a larger feature space but with the same data. Lastly, sentiment is often carried by single words (good, bad etc.) and it is likely most reviews contain such words. This information is captured easier by unigrams.

TF-IDF Vectorizer

This vectorizer seems to achieve the best performance of the Naive-Bayes models (although the best Count Vectorizer is pretty close to it, the TF-IDF vectorizer outperforms it on precision, recall, f1 and accuracy metrics). The TF-IDF Vectorizer and Count Vectorizer work in similar ways and that is why the results appear to be similar too. TF-IDF is, in general, expected to perform better and it does. The reason is that TF-IDF manages to encapsulate information not only about the frequency of appears of different words in a specific sentence/review text but uses a bit more information about the frequency of such words across our texts. This way it better captures the significance of more 'rare' words and, also, appropriately weighs more common words too.

LSTM embeddings

Our embeddings where of type (sequence_length, embedding_length) (each is implemented to be equal to 100 in our approach). These embeddings were flattened in order to be given as input to the Naive Bayes Model. This approach was by far the worst in performance. Embeddings capture complex, contextual relationships between words and these dependencies are a crucial part of the information contained in them. Naive Bayes assumes

independence of variables, which in this case is very much not true and thus performance is hindered. Also, embeddings are continuous dense vectors, while Naive Bayes is not typically used with such inputs and works well with discrete features (e.g., word counts or TF-IDF values). Also, our approach to reducing the embedding space dimensions does not help with any information gain since we simply flatten our embeddings into one sequence.

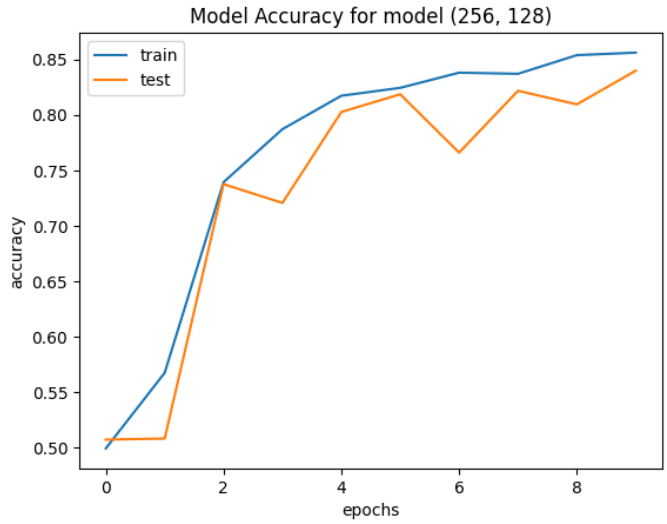
Due to the mismatch in assumptions, nature of features, dimensionality and training objectives Naive Bayes is not a suitable algorithm for leveraging the rich, contextual information encapsulated in embeddings.

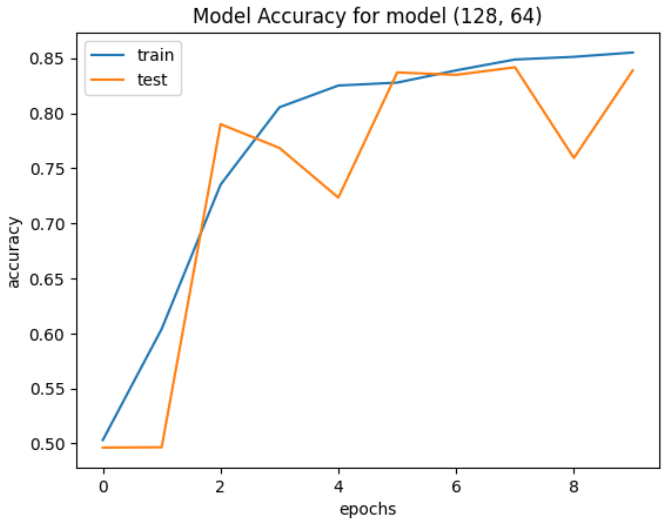
Classification with LSTM and GloVe embeddings

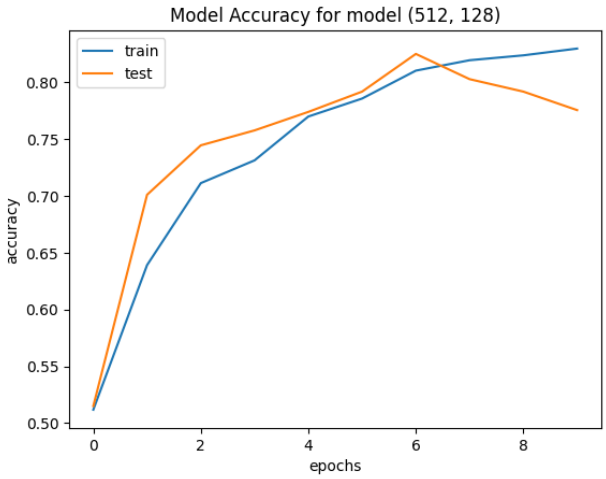
We originally intended to use BERT model's word/token embeddings as the embedding weight matrix used in the LSTM Embedding layer, but due to compatibility issues between the two models we decided to use embeddings that are more commonly associated with LSTM models (such as GloVe) and not try to refactor the BERT embeddings to work with our LSTM model. The model structure can be seen in our LSTM model definition in the code. In this report we should mention that: all LSTM models take as inputs `text` that has been turned into token sequence. And the main layers are the two LSTM layers. One larger and bidirectional which aims to capture more information from our data and one smaller, which aims to capture more high-level information than the first one. Also, we include some dropout layers because we experienced a certain amount of overfitting to the data (model performance peaked and then seemed to drop during subsequent epochs in training). Dropout is a usual technic to mitigate overfitting and it seems to aid us greatly. We implemented 4 different LSTM models with different neuron counts in the two LSTM layers, while keeping the condition that the second layer be smaller than the first. All models were trained on 10 epochs using a 20% validation set (taken from our original training set).

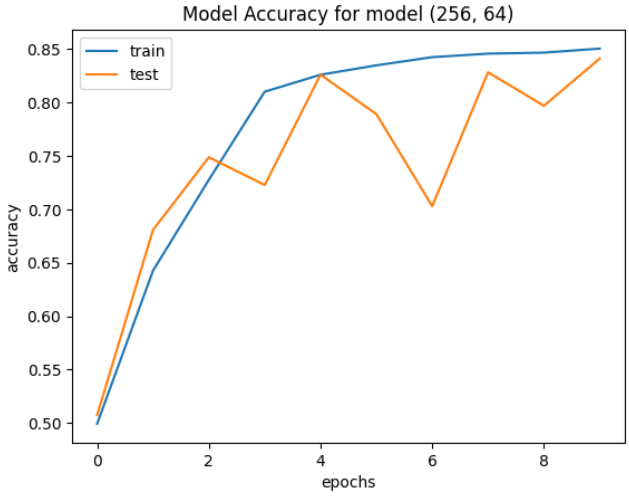
Below we are going to represent the models as tuples with (neurons_in_layer_1, neurons_in_layer_2). Also, `test` refers to the validation set in these charts.

Accuracy charts:





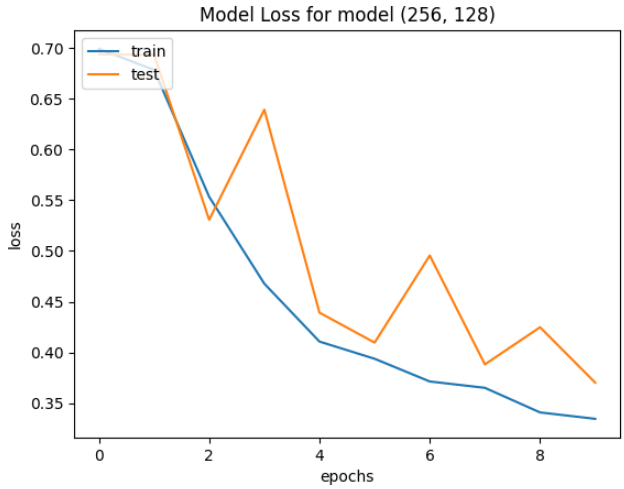


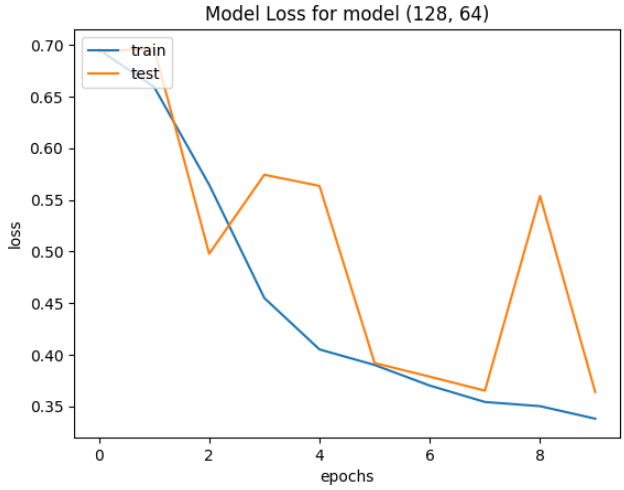


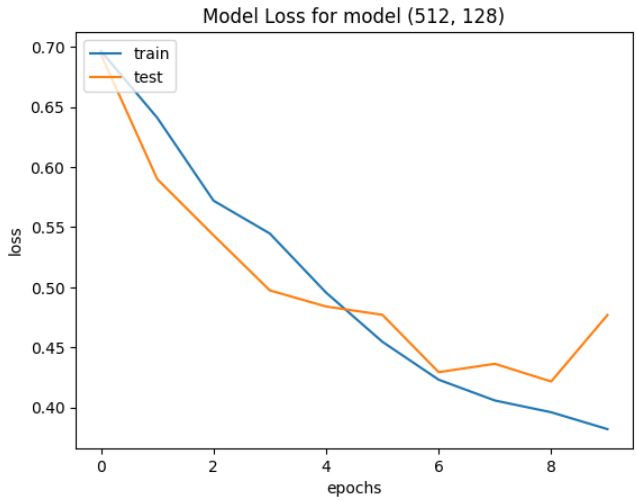
From the above tests we can draw certain conclusions on the LSTM parameters. First of all, when both layers were kept big (256, 128) we observe significant dips in accuracy in the epochs. However, the model manages to survive and perform relatively well. Using layers that both have a lot of neurons likely forces the model to better adapt to the data itself (overfitting). In order to mitigate that effect, we construct the rest of our models in way that there is larger discrepancy between the sizes of the two layers. The (512, 128) model suffers

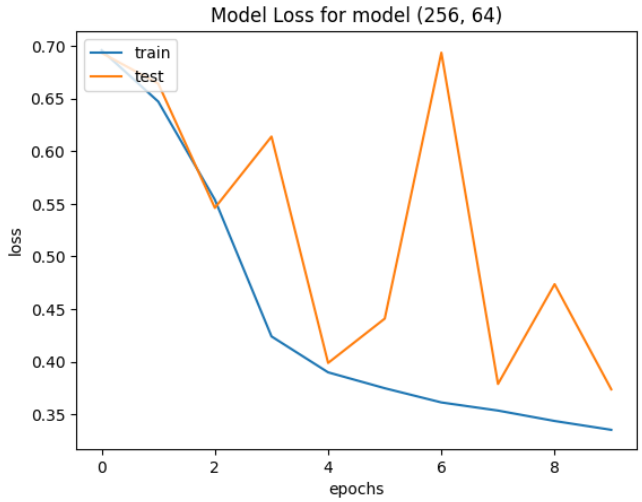
from a similar issue with its performance on the validation/test set dropping after the 7th epoch, while the (128,64) and (256, 64) seem to manage more stable results both during and after training.

Loss Charts: (we observe practically similar behaviour as in the accuracy charts which is expected)









Comparing metrics on the test set (original test set):

```
Metrics for model (256, 128)
              precision    recall  f1-score   support

           0       0.82      0.87      0.84      1945
           1       0.87      0.82      0.84      2045

    accuracy                           0.84      3990
   macro avg       0.84      0.84      0.84      3990
weighted avg       0.84      0.84      0.84      3990

[[1686  259]
 [ 371 1674]]

Metrics for model (128, 64)
              precision    recall  f1-score   support

           0       0.85      0.83      0.84      1945
           1       0.84      0.86      0.85      2045

    accuracy                           0.85      3990
   macro avg       0.85      0.85      0.85      3990
weighted avg       0.85      0.85      0.85      3990

[[1613  332]
 [ 282 1763]]
```

```
Metrics for model (512, 128)
              precision    recall  f1-score   support

           0       0.71      0.95      0.81      1945
           1       0.92      0.64      0.75      2045

    accuracy                           0.79      3990
   macro avg       0.82      0.79      0.78      3990
weighted avg       0.82      0.79      0.78      3990

[[1839  106]
 [ 742 1303]]
```

```
Metrics for model (256, 64)
              precision    recall  f1-score   support

           0       0.83      0.87      0.85      1945
           1       0.87      0.83      0.85      2045

    accuracy                           0.85      3990
   macro avg       0.85      0.85      0.85      3990
weighted avg       0.85      0.85      0.85      3990

[[1694  251]
 [ 357 1688]]
```

The third model was the worst, as is indicated by all metrics. The (256, 128) model (which is our second model with relatively large parameters) manages to achieve a good accuracy score, however, both its accuracy and its f1-scores are pretty close to the last (128, 64) model, which as we observed during training appears to be more stable. Thus that model would be better in this task.

Lastly, the (256, 64) model appears to perform the best on all metrics. The smaller second layer it uses manages to capture just enough information for it to be able to make good predictions on data while avoiding overfitting. This model appears to provide a good balance among the training issues we faced earlier.

Note: although it is not indicated above, what led us try different configurations, was that the first model (256, 128) was rather unstable on different seed values (executions with conditions). We attribute this behaviour to the overfitting problem due to large layers, that retain more information about the dataset than is wanted, as discussed earlier.

Classification with DistilBERT

We initially planned to use the BERT model for sentiment analysis due to its state-of-the-art performance in various NLP tasks. However, during implementation, we encountered significant runtime issues. Training BERT on our dataset required substantial computational resources and time, which was not feasible within our project constraints. Therefore, we opted for DistilBERT, a distilled version of BERT that retains 97% of BERT's language understanding while being 60% faster and smaller.

We used the `DistilBertForSequenceClassification` model classifier for the sentiment analysis task. The model was fine-tuned on our dataset for 3 epochs with a batch size of 16. The following training parameters were set:

- Learning rate: 5e-5

- Weight decay: 0.01

- Warmup steps: 500

The training process was monitored, and the model showed a decreasing trend in loss over the epochs, indicating effective learning. Below is the training loss at various steps:

Step	**Training Loss**
500	0.440000
1000	0.313500
1500	0.211500
2000	0.192400
2500	0.095900

Since loss approached values very close to 0 with our initial setup, it was decided to not implement alternative models (with different parameter values).

The fine-tuned DistilBERT model was evaluated on the test set. The evaluation metrics, including precision, recall, F1-score, and confusion matrix, are as follows:

```
DistilBERT:
              precision    recall  f1-score   support

           0       0.90      0.91      0.91      1991
           1       0.91      0.90      0.91      2000

    accuracy                           0.91      3991
   macro avg       0.91      0.91      0.91      3991
weighted avg       0.91      0.91      0.91      3991

[[1816  175]
 [ 202 1798]]
```

The DistilBERT model demonstrated excellent performance on the sentiment analysis task, achieving 91% accuracy on the test set. Despite initial challenges with BERT, the use of DistilBERT proved to be a pragmatic solution, delivering state-of-the-art results within our resource constraints.

Cumulative Comments

Naive Bayes vs LSTM:

The TF-IDF Naive Bayes seemed to match our best LSTM model performance. In practice LSTM models are considered better for such tasks. To interpret this behaviour we may notice that we used a small amount of data, which helps the TF-IDF with performance, and our LSTM models were relatively simple including only a few layers and having very basic normalization and overfitting mitigation (dropout) techniques. In a real world system, we would handle way more reviews (the original dataset had 4.000.000 reviews and we ended up utilizing only 20.000 of them) which work much better on systems such as LSTM. The Naive Bayes TF-IDF model would have more issues in handling this scale of data. Lastly, another reason for the similar performance was that we set sequence length to 100 in all of our experiments. Longer sequences seemed to hinder performance (on Google Colab), but they could possibly give better results since they would contain more information from the review text and thus carry more information concerning sentiment inference.

DistilBERT vs Naive Bayes and LSTM:

The DistilBERT model demonstrated superior performance compared to both Naive Bayes and LSTM models, with a clear advantage in terms of accuracy, precision, recall, and F1-score. This improvement can be attributed to DistilBERT's ability to capture contextual nuances in the text more effectively due to its transformer-based architecture, which makes it particularly suitable for large datasets and real-world applications involving extensive data. The training process showed a consistent decrease in loss over the epochs, indicating effective learning. DistilBERT's efficiency and effectivenes has shown, as expected, to be the most practical and high-performing model for our sentiment analysis task on the Amazon Reviews dataset.

Discussion

Challenges

- Runtime: runtime remained a challenge especially in the LSTM, BERT models and their respective trainings. To mitigate the issue we used: google colab which allows the use of GPUs for such tasks that greatly speeds them up. This was quite helpful in our approach. We, also, tried to use models/embeddings that have been pretrained to a greater degree than we did in our first approach. This saves us the cost of most of the training and we can better prioritize/emphasize fine-tuning. Lastly, as indicated by the instructed we greatly shrunk the size of our data. Instead of using 100.000 reviews we restricted ourselves to just 20.000. All of these allowed us to experiment in a more user-friendly environment and make better adjustments to our code. Perhaps, if we used a larger dataset, our results would be better (LSTM and Naive Bayes do not seem to exceed 85% accuracy) but that would be at the cost of performance and time. Since this project is for educational purposes we felt that shrinking the dataset was a good decision.
- Data cleaning: language detection didn't always perform properly (e.g. when a lot of uppercase words were used, detection was inclined to classify it as German and other such observations we made). Also, upon inspecting the dataset, misspellings were quite common in the review texts. Because of the nature of this classification problem and the discussed issue of runtime, it is hard for us to test different data cleaning/text preprocessing methods and pick a more consistent methodoly.