CS182 Final Project: Predicting Yelp Reviews

Lawrence Wong, Tram Tran, Varun Srinivasan

May 2020

1 Problem Statement and Approach

In this project, we were given access to a dataset between the raw text of a Yelp review and the star rating that Yelp review was associated with. The dataset was pretty simply a list of raw Yelp reviews (misspellings and all) and their associated star ratings. Our goal was to design and implement a neural network that would predict the assigned star rating given the text of a Yelp review. Besides just being an interesting problem to solve, this task has a variety of potential benefits, such as allowing Yelp to "correct" the overall rating for a business to be more in line with the actual reviews than with the ratings which may be "out-of-sync" with the text of the reviews.

In solving this problem we assess models based on two metrics which we refer to in the future, the raw accuracy or percentage of correctly guessed stars (% ACC), and the average distance of a wrong answer from the correct numbers of stars (AD). We chose to approach this problem from many angles, with each member of our group exploring a different approach for this problem and attempting to optimize that approach. This gave us the maximum opportunity to find the best kind of model, and also gave us a lot of room to explore and present interesting findings about the nature of this problem. Because of this, we've chosen to present our findings approach-by-approach.

Overall, we found RNN-based models to work very well, and specicaly non-sequential models worked very well. Surprisingly for RNN, most perturbations we could come up with did not improve the results greatly, although for RNNs as with all of our approaches, a balanced data-set yielded improvements. Additionally, with our BERT models we found that using ordinal labels (something we hadn't heard of until this project) was beneficial in improving our model. Additionally, some "outside" the box approaches we tried like a Word2Vec model and a CNN+LSTM model, ended up not performing very well, but it was interesting to see the results nonetheless. Overall, our BERT model with ordinal labels turned out to perform the best.

2 Baseline

Before we delve into our neural network approaches for this problem, we needed to establish a baseline accuracy and average distance. We chose to do this using a simple bag of words (BoW) model. Our BoW model simply constructed the distributions of the most popular 5000 words for each star value review and then computed the KL-Divergence between the distribution of words in the review to be predicted, guessing the star value as the one with the lowest divergence from the review. This simple model managed to attain a 50.84 % ACC and 1.614 AD on our our validation set (a random sample of 2500 reviews from a set of reviews that were never used for training). From here we moved forward to our neural network solutions.

3 RNN Models

3.1 Tools and Data Preparation

For this style of model, we used Keras, for most of the model implementations as well as the model graph visualizations and we used scikit-learn to create our confusion matrices to analyze our model's predictions. As for data preparation, although we experimented heavily with spell-checking, stopword removal, and lemmatization of our inputs texts, we found that there was no practical positive effect on our model's results and any heavy preprocessing drastically increased the amount of time it took for our model to evaluate. As such, we mainly stuck to punctuation and extraneous white-space removal, as well as lowercasing, for preprocessing for this style of model.

For training we set aside 80% of the given set to be randomly sampled from for training data, and 20% of the given set to be randomly sample from for validation data. Across all of our RNN models we validated across the same set of 2500 validation reviews for consistency.

3.2 Initial Model

Our base RNN-model was built of an embedding-layer with embedding dimension 250, followed by an LSTM layer with an output size of 250, and a dense layer with an output size of 5 (one for each star review, since for RNN's we framed the problem as a categorization). Into this we fed a preprocessed (as described above) and tokenized (using Keras' tokenizer) version of the reviews we wanted to predict the ratings of, and one-hot encoded vectors representing the classification of the rating. This model achieved substantial improvements over our baseline with no fine-tuning or other techniques. Although this model did perform relatively well, we noticed issues with the training time (which we suspected was due to the high dimensionality of the LSTM output space), the relatively high AD, and additionally, the high confusion between certain star ratings categorizations. We attempted to address these problems with further iterations on our intial model.

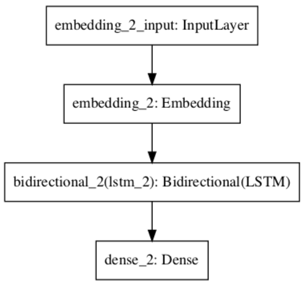
embedding_2_input: InputLayer
embedding_2: Embedding
bidirectional_2(lstm_2): Bidirectional(LSTM)
dense_2: Dense

Figure 1: Initial RNN Graph

3.3 Reducing Training Time

We reduced training time on our initial model simply by reducing the output space of the LSTM layer from 250 to about 50, which was the lowest we were able to go to in our experimentation while leaving our results pretty much the same. Additionally, while thinking about this, we stumbled upon the idea of using pre-trained word embeddings, and we decided to use GloVe embeddings (of size 200) published by Stanford which used data from Twitter. This didn't have a major affect on the speed or accuracy of the model, but we did notice some very minor improvements so we left this in.

3.4 Reducing AD

To reduce our AD, we employed a solution we called "mid-val". Essentially, since we used categorical crossentropy for our loss, the model would be optimizing to attempt to match its predicted distribution of probabilities of each star with our objective distribution, which would just be a zero vector with one entry of 1 for the correct star. Under mid-val, instead of structuring our objective value like this, we instead structure it using a "mid-val" with half of the "leftover" probability on the nearest two star values to the true value and the mid-val probability on the true value. Essentially for a 3-star review with mid-val 0.8, instead of the categorization vector looking like: [0, 0, 1, 0, 0], it would look like [0, 0.1, 0.8, 0.1, 0]. The goal here was to give the model a little more to "grab onto" around the true value to bring in the AD and potentially increase the accuracy. This paradigm succeeded in both respects. We also saw through shifting the mid-val that we selected that a mid-val of .334 created a very low AD but severely at the cost of accuracy and that the "sweet spot" lay around 0.9.

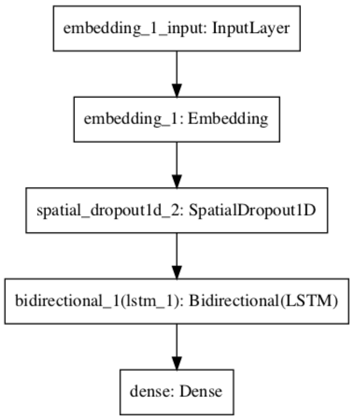
embedding_1_input: InputLayer
embedding_1: Embedding
spatial_dropout1d_2: SpatialDropout1D
bidirectional_1(lstm_1): Bidirectional(LSTM)
dense: Dense

Figure 2: Best "Regular" RNN

After doing this, we tentatively experimented with adding a spatial dropout layer in between the embedding layer and LSTM layer, as well as experimenting with stacked LSTM layers, ideas which we got from Alexander Burmistrov's Kaggle post about toxic comment classification [1]. Although the stacked LSTM layers basically proved to be useless outside of increasing time to train, the spatial dropout layers did provide a very small improvement, and so we added this to our model.

3.5 Confusion Matrices and Ensemble Modeling

After making the modifications above, we decided to "take a look under the hood" of our model and created a confusion matrix of what types of reviews our models misclassified the most. As it turns out, the model misclassified four star reviews as five star reviews and vice versa significantly more than any other star pairing. To address this, we tried two techniques, ensemble models and finetuning.

We attempted to create an ensemble model between our best model from our efforts thus far, and a model of the same structure that was purpose trained to distinguish between 4 and 5 star models. We created this ensemble by having our original model crosscheck with the other model when its predicted probability of the 4 or 5 star was below a certain threshold. After playing with the threshold for a bit, we were able to get a small improvement out of this ensemble strategy, but the marginal improvements over the base model did not seem worth it relative to the extra time to predict and the generally "finickyness" of the ensemble.

We also attempted to employ some finetuning tactics to mitigate model confusion. To do this, we took our best model as mentioned above, removed the dense layer, froze the other layers, and added in another LSTM layer and then added back the dense layer. We then trained this model again on the target reviews for finetuning (4 and 5 star reviews that were being confused). Although this method did mitigate confusion a bit, it drastically reduced the overall effectiveness of our model, so we ended up scrapping the idea, although our experiences with finetuning eventually led to some positive futures changes that are coming up.

*	1	2	3	4	5
1	N/A	22	13	3	16
2	99	N/A	27	11	8
3	40	13	N/A	38	21
4	13	6	38	N/A	185
5	33	4	18	69	N/A

Confusion Matrix For Best “Regular” Model (X: Predictions, Y: True)

3.6 Further Experimentation

At this point, we had explored improvement along multiple avenues and we were having trouble pushing up the performance of our RNN solution. Before making our final breakthrough and wrapping up RNN’s for good however, we did explore three more ideas: 1) Thesaurization, 2) “Even” Training, and 3) “Tree” RNN.

Our idea for Thesaurization was based on the idea that we could replace commonly used adjectives with less used synonyms, which would hopefully create connections in the model making it more robust to similar sentiment reviews with different word choice. Although in theory this may have worked out, in practice it was doomed for failure as doing so reduced our accuracy and increased our AD. We believe this is mostly likely either because our thesaurization used synonyms that were too obscure or because generally reviews used relatively similar word choice. Either way this avenue of exploration was fruitless.

Our second idea, “even” training, came from a completely accidental observation. When playing with our mid-vals, we accidentally created a training set and validation set of only five star reviews, which led to us getting a model with “100%” accuracy. This reminder that a skewed data set would lead to a skewed model, made us reevaluate our training set creation. Instead of just random sampling from our training set, we tried random sampling a certain amount of reviews from each star value from our set. The idea behind this was that we would get a model that is more evenly trained on each kind of review, rather than one that is more heavily trained on one kind or another. This model had a relatively low accuracy, but the best AD of all our model’s thus far and was a catalyst for some of the final touches we made to our RNN-style models.

Our final idea, “Tree” RNN, was an offshoot of our attempts at minimizing confusion and ensemble models. The general idea was that of a “binary search tree of models”. Essentially we would have one model that decided if a review was a 1 - 2 star review or a 3 - 5 star review. Depending on the results at this level, the review would propagate down the tree to increasingly specific models till the final rating was assigned. The idea behind this was that this ensemble might theoretically mitigate confusion, as it isn’t trying to make a single broad categorization, but is instead making a series of granular categorizations. However, we didn’t even need to complete this model in order to see the flaw with it after some experimentation. Even with our best “initial-split” model, we could only attain a bit above our normal overall accuracy, and we realized the compounding effect of the series errors in the models would make the “Tree” impractical and inaccurate. This, along with its incredibly slow time to train and execution time, meant that the “Tree” never got its day in the sun.

Model Description	Mid-Val	Dropout	% ACC	AD
Baseline Model	N/A	No	50.84	1.614
Base RNN	N/A	No	64.12	1.754
Base RNN	0.9	No	72.4	1.519
Base RNN	0.9	Yes	72.92	1.461
Ensemble RNN	0.9	Yes	71.63	1.773
Finetune RNN	0.9	Yes	17.84	1.181
Thesaurize RNN	0.9	Yes	69.34	1.638
“Even” RNN	0.9	Yes	61.04	1.316
Non-Seq RNN	0.9	Yes	74.08	1.424

Table 1: Model Performance Data (After 5 Epochs Of Training)

3.7 Non-Sequential Models

The final frontier we explored with RNN's was non-sequential models. We decided on this approach after realizing that a combination of our best "regular" model and our best "even" sampling model would give us a theoretically ideal combination between accuracy, AD, and general robustness. To create a model that combined the two effectively, we removed the dense layers from both, froze all of their current layers, and then added a dense layer on top that was still trainable in order to tune our outputs categorization based on the new combination. When we fit this model using random training samples initially, we got relatively poor results which we re pretty much equal to our best "regular" model. We realized that this was likely because the dense layer learned to prioritize information coming from the "regular" model branch of the model. To remedy this we created a training set composed of 25% reviews that only the "even" sampling model was able to guess correctly, 25% reviews that only the "regular" model was able to guess correctly, 25% reviews that neither were able to guess correctly, and 25% reviews that were randomly sampled from the training set. This combination finally achieved what we were looking for, a high accuracy, a low AD (on par with models with a mid-val of 0.5), and relatively resistant to perturbations (as evidenced by its good performance on the challenge data sets that were released).

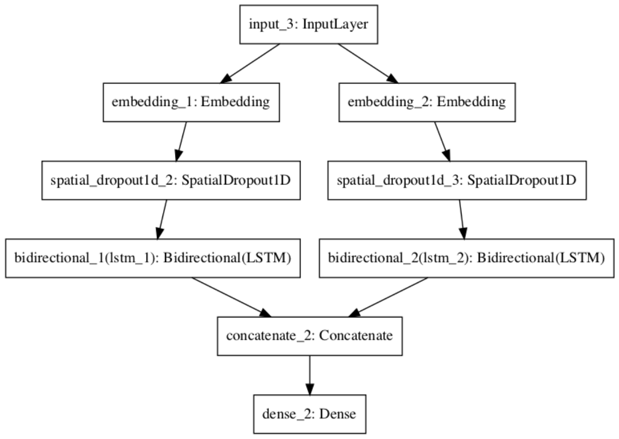
input_3: InputLayer
embedding_1: Embedding
embedding_2: Embedding
spatial_dropout1d_2: SpatialDropout1D
spatial_dropout1d_3: SpatialDropout1D
bidirectional_1(lstm_1): Bidirectional(LSTM)
bidirectional_2(lstm_2): Bidirectional(LSTM)
concatenate_2: Concatenate
dense_2: Dense

Figure 3: Final Non-Seq RNN

3.8 RNN Conclusion and Further Experimentation

We were pretty satisfied with how our RNN experimentation worked out. We managed to attain competitive accuracies and AD's with a relatively simple model architecture through intensive experimentation and "playing around" with the data. Even though our performance improvements plateaued after we surpassed about 72% accuracy, we were able to make some more improvements on AD, accuracy, and on the challenge datasets through this experimentation, and our final model represents those improvements. Our experimentation and research also raised some interesting possibilities for future expansion of our pure RNN models. For example, although we kept our model architecture pretty light for the sake of quick execution and training, given more execution time and more time for training and experimentation, we would experiment with more complex model architectures. We could do this by trying out things like pooling layers and additional concurrencies besides just "even" sampling, to try and eek out some more performance on reviews where our model gets confused. Some additional non-model concurrencies we could see being useful are counts of certain key words, number of times price is mentioned, etc. Overall, we consider our RNN experimentation a success, both from a raw metrics point of view and from a purely experimental point of view.

4 BERT Models

Many advances in the field of NLP have been led by pre-trained models. Therefore, we decided to further explore pre-trained BERT, a model proposed by Google. We fine-tuned (using Tensorflow) the pre-trained BERT model provided on TF-Hub and created multiple BERT-based models that are similar to a Google's BERT Colab example [2], but each with some modifications in the hidden layer's size, activation and the loss function.

4.1 Preparation

We shuffled the given data set and giving 90% for training and 10% for validation. Due to limitation in computation resources, we further split the training dataset into two equal size training sets. Each model is trained with a batch size of 16, dropout rate 0.1, learning rate 2e-5, for 2 epochs for one of the training set, and another 2 epochs for the other training set. We used BERT base uncased with max sequence length of 128 and the default optimizer from the BERT module. All text reviews fed are lowercased, tokenized, broken into WordPieces based on the pretrained BERT's vocabulary, and added special tokens "CLS" and "SEP".

4.2 Initial Model

This BERT model is almost identical to Google's example model on the right. The only modification we made was changing the output dimension of the fully connected layer to 5 as we have 5 categories. This model uses cross entropy loss and predicts the label that has the highest probability among the 5 categories.

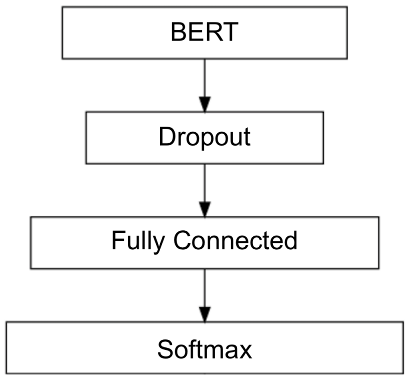

Figure 4: Google's example Model

4.3 Modified Loss

This model has the same architecture as the above BERT model and a modification in the loss function. The new loss function for each sample is

$$-\frac{1}{2}\mathbf{log}(\hat{y}_y) - \frac{1}{2} \times \prod_{i=-1}^{1} \mathbf{log}(\hat{y}_{y+i})$$

where $\hat{y}$ is the output of softmax and y is the index of the true label. If $y+i < 0$ or $y+i > 4$ then $\mathbf{log}(\hat{y}_{y+i}) = 1$. The idea behind this loss was that we want to increase both the probability of the correct label and the sum of the probabilities of the correct label, its left neighbor, and its right neighbor.

4.4 Ordinal Labels

So far, our models do not take into account that the output labels are ordinal. After doing a little research on ordinal regression and deep learning, we decided to implement a new BERT model inspired by a paper that we read [3]. First, we modified the fully connected layer output dimension to 4, used sigmoid activation, and removed softmax. Secondly, we changed the loss function of each sample to

$$loss = -\sum_{i=0}^{i=3} e_i\mathbf{log}(\hat{y}_i) + (1-e_i)\mathbf{log}(1-\hat{y}_i)$$

where e_i is the i+1th element of the true label's embbedding $\{\mathbf{label} \rightarrow \mathbf{embedding} : 1 \rightarrow [0,0,0,0], 2 \rightarrow [1,0,0,0], 3 \rightarrow [1,1,0,0], 4 \rightarrow [1,1,1,0], 5 \rightarrow [1,1,1,1]\}$. The predicted label is (1 + the index of the first element of the sigmoid output that has value less than 0.5) or (5 if all values $\geq$ 0.5). With this implementation, a review with a rating of k is classified automatically as having ratings 1,...,k-1 too. Therefore, we impose an ordering on the output labels of the model.

4.5 Ensemble of Binary Classsifiers

Our final idea for BERT was an ensemble of binary classifiers. This model was created in an attempt to reduce the effect of imbalanced ratings distribution in the dataset (1 and 5 have a lots more samples than other labels). We created five binary classifiers, one per rating category. They were almost identical to the initial BERT model, the only difference was the fully connected layer's output dimension was 2 and not 5. Each classifier was assigned a different rating category and was trained to predict the probability of a review is in its assigned rating category. Moreover, all classifiers were trained on a balanced dataset, 50% samples in the classifier's assigned category and 50% are not in the category. At inference time, the ensemble model predicts the category assigned to the classifier that outputs the highest probability.

4.6 Results

As mentioned above, the given dataset is imbalanced because there are a lot more ratings with 1 and 5 stars than other numbers. Therefore, we decided to test all BERT models on 2 datasets, one with imbalanced ratings distribution (similar to the given dataset) and another one with balanced distribution (equal number of samples per category)

4.6.1 Imbalanced Set

Label 1: 12988
Label 2: 3609
Label 3: 3429
Label 4: 7228
Label 5: 26105
Total: 53359 samples

Model	% ACC	AD
Initial Model	79.60	0.246
Ordinal Labels	79.47	0.244
Modified Loss	79.52	0.244
Binary Classifiers	76.74	0.288
RNN	73.27	0.386

Table 2: Model Performance Data

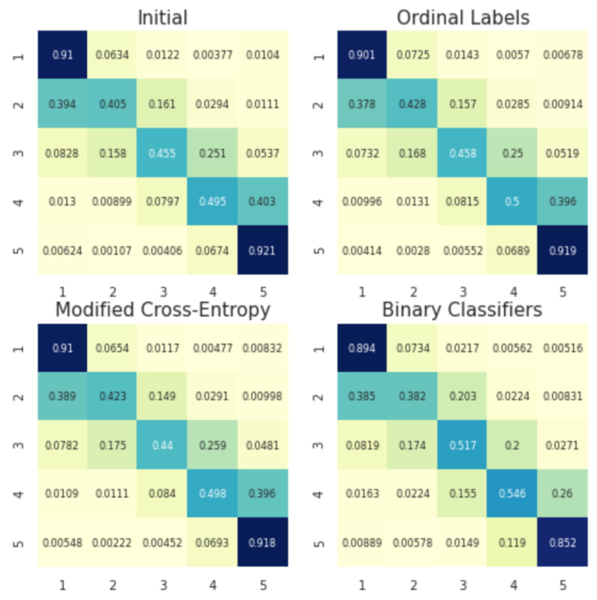

Figure 4: Normalized Confusion Matrices

For this dataset, the initial model has highest accuracy and a little higher AD. We noticed that other models have much less "extreme" misclassified samples (1 predicted as 5 and vice versa) and more off-by-1 samples. Moreover, the initial model has lower precisions for labels 2,3,4 than other models. The ensemble of binary classifiers (EBL) has the worst accuracy and AD. We suspect this is because we trained EBL on uniform distributions. Surprisingly, EBL does a lot better on labels 3,4 but worse on 2.

4.6.2 Balanced Set

Each Label: 3429
Total: 17145 samples

Model	% Accuracy	AD
Initial Model	63.68	0.424
Ordinal Labels	64.12	0.416
Modified Loss	63.62	0.423
Binary Classifiers	63.73	0.424
RNN	52.06	0.643

Table 3: BERT Model Performance Data

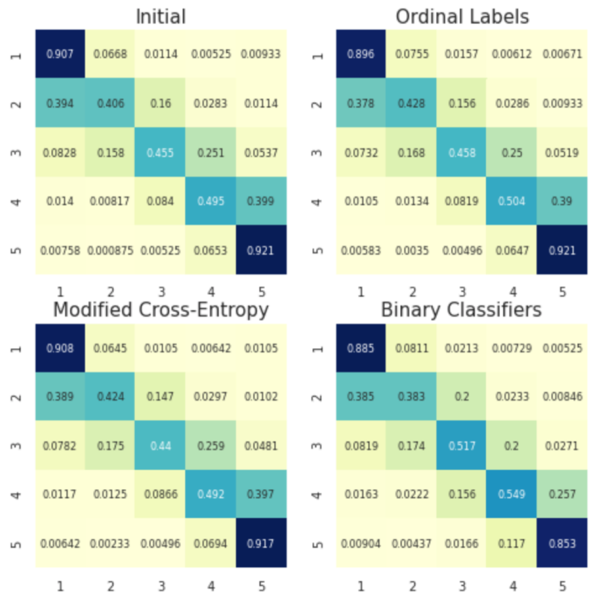

Figure 5: Normalized Confusion Matrices

The model with ordinal labels (MOL) works the best for this dataset. Also as expected, EBL is on par with other models if the data's distribution is uniform. After looking at the results, we concluded that MOL is the most robust among all BERT models. It has lowest ADs and good accuracies for both test sets.

5 Additional Models

5.1 Word2Vec Model

The goal of a Word2Vec model is to learn word embeddings. By taking in a large corpus of text, the model, ideally, will be able to learn what words are typically used for each star level.

5.1.1 Data Preprocessing

First, we cleaned the reviews to only contain adjectives, adverbs, and their variations. This is because what truly constitutes a user's star rating is what words they use to describe "xyz" rather than "xyz" itself. To do this, we utilized the nltk package, specifically "nltk.pos_tag" and "nltk.word_tokenize". The key feature of the word2vec model that we take advantage of is its ability to calculate the "distance" between two words (i.e. cosine similarity). However, the reviews as is do not have any notion of star rating. Thus, we then insert the rating itself (5, 4, 3, 2, and 1) in between each pair of words. By doing this, we force the model to learn what rating is associated with certain words. Here is how some sample words compare with each rating:

```
excellent:  [(1, -0.53788555), (2, -0.14643644), (3, -0.0060804435), (4, 0.22294004), (5, 0.46282187)]
horrible:  [(1, 0.638587), (2, 0.19604206), (3, 0.05769393), (4, -0.24640091), (5, -0.36491573)]
okay:  [(1, 0.060839493), (2, 0.46249625), (3, 0.45996273), (4, 0.14601853), (5, -0.37550485)]
aight:  [(1, -0.23911157), (2, 0.14746639), (3, 0.18935823), (4, 0.15836397), (5, -0.121525794)]
```

As expected, "excellent" most closely aligns with 5 and "horrible" with 1. Furthermore, both "okay" and "aight" most closely align with central ratings (2, 3, and 4) and even have negative similarity with 5 as well as 1 in the case of "aight".

5.1.2 Model

In order to create the word embeddings, we used the gensim package, specifically "gensim.models.Word2Vec". The hard part is finding a good method to generate predictions. The first idea we tried was a simple voting system. For each word in a cleaned review, we added a point for the star rating it most closely aligned with. Then, after tallying up all the votes, we simply chose the star rating with the most votes. This resulted in a test accuracy of 56%. Another attempt was to average the votes and pick the middle star rating. However, this resulted in an appalling 16%. After looking why we got reviews wrong, we realized that some words such as "first" were pretty evenly similar with all star ratings. Thus, instead of a max of the cosine similarities, it would be better to take the average. This resulted in a test accuracy of 55%. The key breakthrough come with the idea of "Total Cosine Similarity". What this means is that instead of worrying about how each word compares, we look at the review holistically. By adding all of the cosine similarities and then taking the max, we got a test accuracy of 65.6%. A small improvement then came from realizing that because some cosine similarities were negative, we could add its absolute value to the inverse rating (i.e. If 1 had a similarity of $-.2$ we would add .2 to 5's similarity). This resulted in a accuracy of 68.4%.

5.1.3 Improvements

A key idea to take note of is that of prior probabilities. In the case of the given data, we realized that 5s were given 48% of the time and 1s were given 24%. By leveraging this fact, our model could have seen additional improvements. Additionally, another idea that might have increased model accuracy would be give weights to each word. Rather than treating each word equally, we note that the stronger the similarity between a word and a rating, the more likely that that word matters more to the true rating than other words.

5.2 CNN+LSTM

Additionally, we did some slight experimentation with a convolutional layer before the recurrent layer (LSTM). On an imbalanced dataset, the best validation accuracy obtained was 72%. On a balanced dataset, the best validation accuracy obtained was 52%. We quickly realized this didn't work as well as the models described in earlier parts.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
embedding (Embedding)        (None, 200, 32)           160000    
_________________________________________________________________
dropout (Dropout)            (None, 200, 32)           0         
_________________________________________________________________
conv1d (Conv1D)              (None, 196, 64)           10304     
_________________________________________________________________
max_pooling1d (MaxPooling1D) (None, 49, 64)            0         
_________________________________________________________________
lstm (LSTM)                  (None, 100)               66000     
_________________________________________________________________
dense (Dense)                (None, 5)                 505       
_________________________________________________________________
activation (Activation)      (None, 5)                 0         
=================================================================
Total params: 236,809
Trainable params: 236,809
Non-trainable params: 0
_________________________________________________________________
```

6 Wrapping Up

To wrap up our report and our final project experience we're going to give information about our submission, summarize our lessons learned, and outline responsibilities in the project.

6.1 Submission Details

On the leaderboard, we submitted out BERT model with ordinal labels, as it performed best in our testing, and we submitted under team number: "1", and team name: "this team". All the relevant code to our project can be found at our GitHub repository: https://github.com/vsrin1/CS182-Final-Project. The README gives an overview about how to run our final submission as well as our other models, and we have included some of the code used to generate the models as well, although this is less well documented in the README.

6.2 Lessons Learned

We learned a lot of interesting lessons during this project. Firstly, and least specifically, we learned that training a model that performs well on a new task is really difficult, and its important to get creative with trying to come up with different ways to structure a network to solve a problem. For example, our mid-val idea and our ordinal labels idea are perfect examples of this kind of creativity. We learned a lot about getting creative with the data and thinking about the kind of data we were training on, for example, noticing that there wasn't an even distribution of stars in our training set was an interesting revelation for us that made us think about our models' robustness. Toying with our loss functions was also very useful as it allowed us to get a better hold on what to "train for" (ordinal labels are an example of this). Overall, we learned through experimentation that although we were able to get pretty good results by "messing with" RNN structure and input data format as well as with other strategies like Word2Vec and CNN+LSTM, a pre-trained model like BERT is often objectively superior right out of the gate, and combining that with a good loss function makes the model even better. In the end, although our custom models put up a good fight, the BERT pre-trained model combined with ordinal labels performed the best in our testing, and that ended up being our final submission.

6.3 Group Responsibilties

We split group responsibilties and contributions as such:

Lawrence Wong:
Contribution: 20% Responsibilities: Creating "Additional models" and conducting experimentation with that approach, as well as writing that section of the report (section 5).

Tram Tran:
Contribution: 40%
Responsibilities: Creating BERT models and conducting experimentation with that approach, as well as writing that section of the report (section 4).

Varun Srinivasan:
Contribution: 40%
Responsibilties: Creating RNN models and conducting experimentation with that approach, as well as writing that section of the report (section 3).

Additionally, we all collaborated with each other to come up with ideas in our experimentations.

7 References

[1] https://www.kaggle.com/c/jigsaw-toxic-comment-classification-challenge/discussion/52644

[2] https://colab.research.google.com/github/google-research/bert

[3] http://orca.st.usm.edu/ zwang/files/rank.pdf