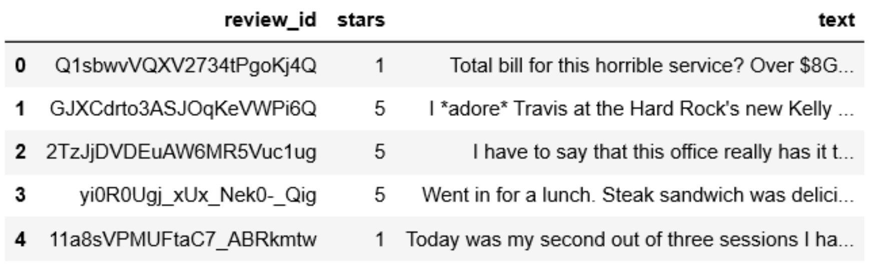
Yelp Review Prediction

By: Tanner Arrizabalaga and Tanvee Desai

In this article, we will be explaining our approach to a popular NLP problem in the deep learning world: Yelp Review Star Prediction. We explored different ways to structure the data, different losses to train on, various models to deploy, and methods for ensemble learning. We additionally explored how to modify our model to perform well on the challenge datasets. Overall, we finished in BLANK place for the extra credit competition and achieved a final BLANK test MAE and accuracy on our own test set of 30% of the data.

The Data

The yelp review dataset has three columns. There is a column for the review ID, the text, and the number of stars the reviewer rated their experience. Here are a few entries from the dataset:



	review_id	stars	text
0	Q1sbwvVQXV2734tPgoKj4Q	1	Total bill for this horrible service? Over $8G...
1	GJXCdrto3ASJOqKeVWPi6Q	5	I *adore* Travis at the Hard Rock's new Kelly ...
2	2TzJjDVDEuAW6MR5Vuc1ug	5	I have to say that this office really has it t...
3	yi0R0Ugj_xUx_Nek0-_Qig	5	Went in for a lunch. Steak sandwich was delici...
4	11a8sVPMUFtaC7_ABRkmtw	1	Today was my second out of three sessions I ha...

Figure 1. First 5 samples in our dataset

Before we began our model construction, we wanted to get a general idea of what our data was like. In terms of the distribution of the stars, we see that the majority of the reviews receive either a 1 or 5-star rating; and, in general, if we define a “positive” review as a being >= 4 stars and a “negative” review as <4 stars, we see that there are more positive reviews than negative reviews.

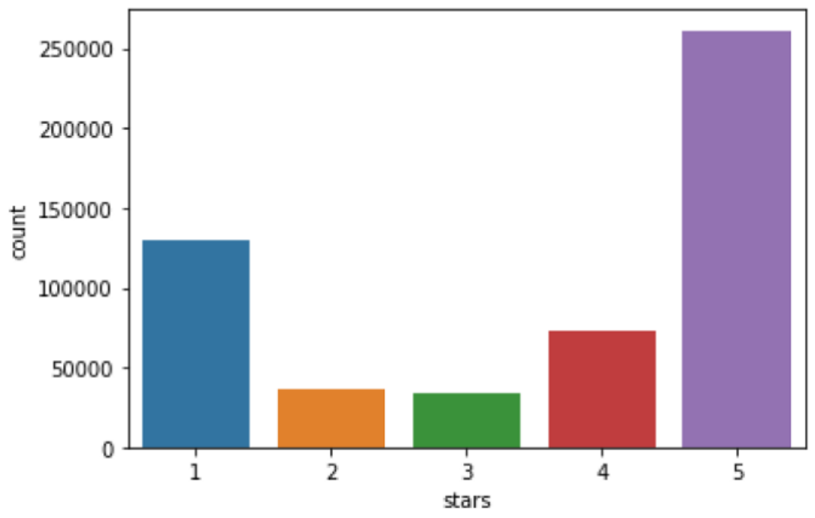


Figure 2. Distribution of ratings

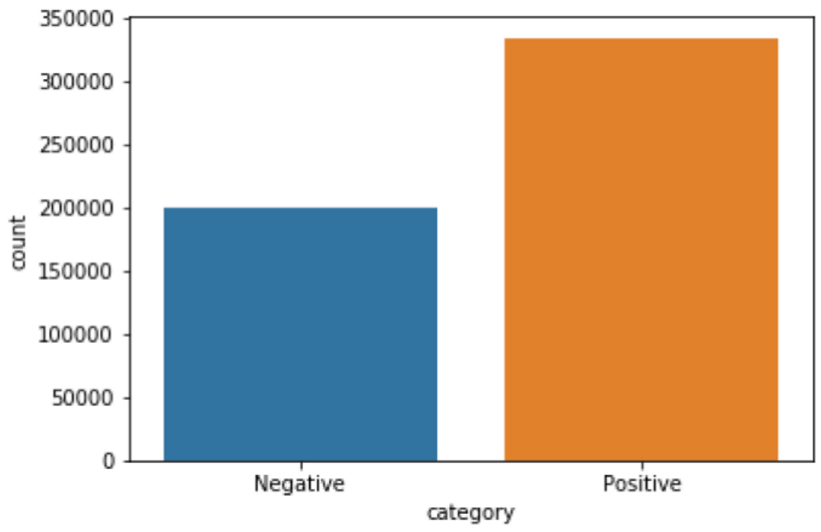
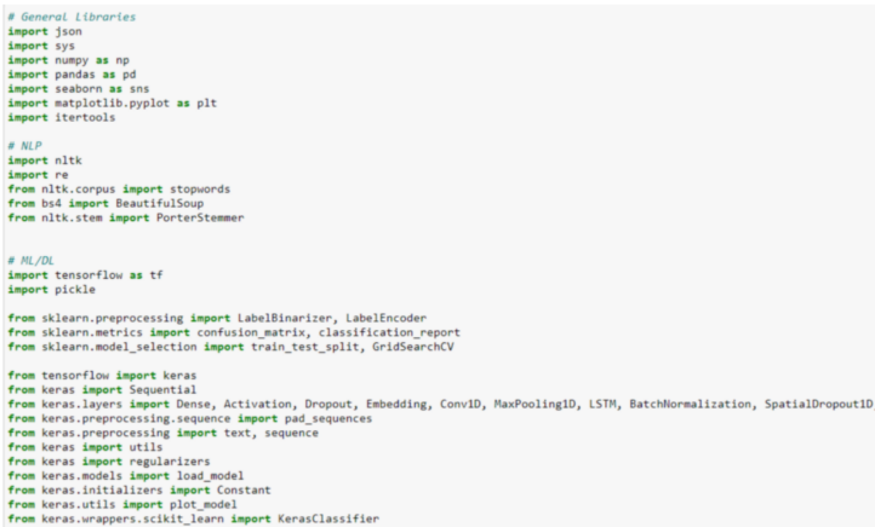


Figure 3. Bar chat of "positive" and "negative" reviews

Tools

To make reproducing our results easier, we have listed the libraries we utilized below:

- numpy, pandas, matplotlib, seaborn, counter → general use for data manipulation, analysis, and voting for the ensemble model
- nltk, re, bs4 → text cleaning and stemming
- sklearn, tf/keras, pickle → deep learning/modeling



```
# General Libraries
import json
import sys
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import itertools

# NLP
import nltk
import re
from nltk.corpus import stopwords
from bs4 import BeautifulSoup
from nltk.stem import PorterStemmer


# ML/DL
import tensorflow as tf
import pickle

from sklearn.preprocessing import LabelBinarizer, LabelEncoder
from sklearn.metrics import confusion_matrix, classification_report
from sklearn.model_selection import train_test_split, GridSearchCV

from tensorflow import keras
from keras import Sequential
from keras.layers import Dense, Activation, Dropout, Embedding, Conv1D, MaxPooling1D, LSTM, BatchNormalization, SpatialDropout1D
from keras.preprocessing.sequence import pad_sequences
from keras.preprocessing import text, sequence
from keras import utils
from keras import regularizers
from keras.models import load_model
from keras.initializers import Constant
from keras.utils import plot_model
from keras.wrappers.scikit_learn import KerasClassifier
```

Figure 4. Imported libraries

The Problem

How can we predict the number of stars a user will rate *only* given the text of their review?

We turned to techniques in sentiment analysis to figure that out. Sentiment analysis is a category within Natural Language Processing that uses data science to determine the underlying sentiment of a text[1].

We thought of several approaches within the realm of deep learning.

- Simple Deep Neural Network (our baseline model)
- CNN LSTM
- Bi-Directional LSTM
- CNN with Word2Vec
- RNN with Word2Vec
- One vs. All Ensemble Model

After establishing a baseline model, we chose to experiment with our data transformation techniques:

- No stopword removal
- Full stopword removal

- Selective stopword removal

For selective stopword removal, we decided to keep the following words in our reviews, as we believed they carried sentiment that was important in our final prediction. The addition/subtraction of many of the stopwords makes a dramatic difference in the meaning of many texts.

> *Words kept: [“nor”, “not”, “very”, “no”, “few”, “too”, “doesn”, “didn”, “wasn”, “ain”, “doesn’t”, “isn’t”, “hasn’t”, ‘shouldn’, “weren’t”, “don’t”, “didn’t”, “shouldn’t”, “wouldn’t”, “won’t”, “above”, “below”, “haven’t”, “shan’t”, “weren”, “but”, “wouldn”, “mightn”, “under”, “mustn’t”, “over”, “won”, “aren”, “wasn’t”, “than”]*

We also chose to experiment with the loss function training our models; we changed the loss various times, as seen in the experiments below.

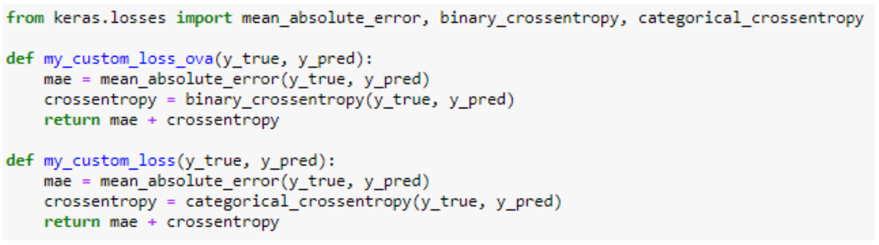
Experiments

We set up experiments with different kinds of data cleaning methods combined with different losses for model training:

1. Adjusted stop words, MAE as loss
2. Adjusted stop words, Cross entropy as loss
3. Adjusted stop words, Combined loss
4. Removal all stop words, MAE as loss
5. Remove all stop words, Cross entropy as loss
6. Remove all stop words, Combined loss
7. Keep all words, MAE as loss
8. Keep all words, Cross entropy as loss
9. Keep all words, Combined loss

For these experiments, the train/test split was 70:30, and the training/validation split during training was 80:20

The “combined loss” was built with the two following functions—one for the baseline model and CNN LSTM, and the other for “One vs. All” model



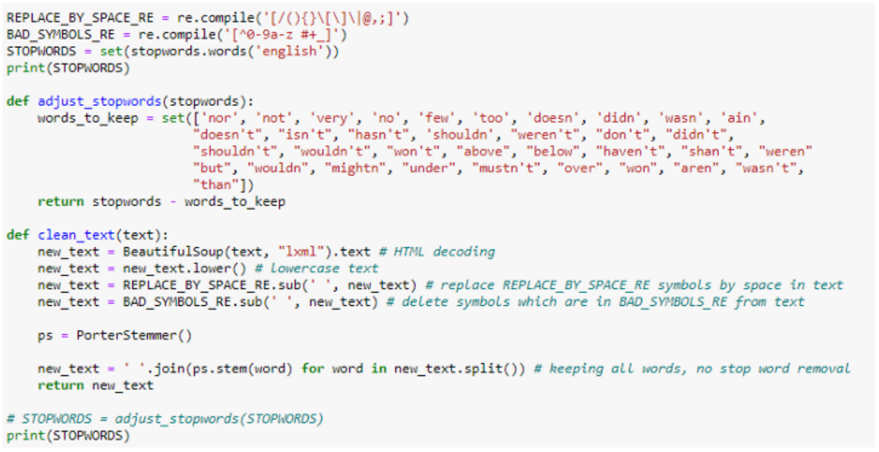
```python
from keras.losses import mean_absolute_error, binary_crossentropy, categorical_crossentropy

def my_custom_loss_ova(y_true, y_pred):
    mae = mean_absolute_error(y_true, y_pred)
    crossentropy = binary_crossentropy(y_true, y_pred)
    return mae + crossentropy

def my_custom_loss(y_true, y_pred):
    mae = mean_absolute_error(y_true, y_pred)
    crossentropy = categorical_crossentropy(y_true, y_pred)
    return mae + crossentropy
```

Figure 5. Custom loss functions for training

Cleaning and Tokenization of Text

Throughout each experiment, there were some text cleaning methods we kept constant as we believed they were necessary for the model.

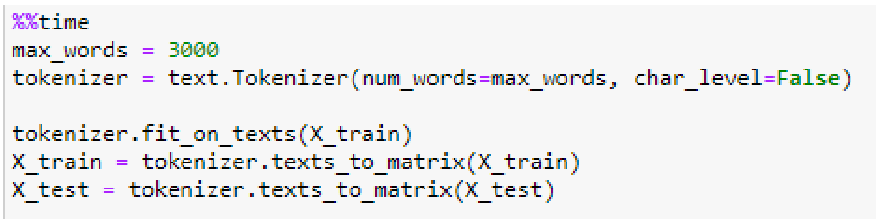
1. Account for potential HTML encoding/decoding issues via bs4
2. Lowercase the text
3. Remove all non-sensical symbols such as ‘\n’ with space
4. In addition, remove any symbols we may have missed



```python
REPLACE_BY_SPACE_RE = re.compile('[/(){}\[\]\|@,;]')
BAD_SYMBOLS_RE = re.compile('[^0-9a-z #+_]')
STOPWORDS = set(stopwords.words('english'))
print(STOPWORDS)

def adjust_stopwords(stopwords):
    words_to_keep = set(['nor', 'not', 'very', 'no', 'few', 'too', 'doesn', 'didn', 'wasn', 'ain',
                         "doesn't", "isn't", "hasn't", 'shouldn', "weren't", "don't", "didn't",
                         "shouldn't", "wouldn't", "won't", "above", "below", "haven't", "shan't", "weren"
                         "but", "wouldn", "mightn", "under", "mustn't", "over", "won", "aren", "wasn't",
                         "than"])
    return stopwords - words_to_keep

def clean_text(text):
    new_text = BeautifulSoup(text, "lxml").text # HTML decoding
    new_text = new_text.lower() # lowercase text
    new_text = REPLACE_BY_SPACE_RE.sub(' ', new_text) # replace REPLACE_BY_SPACE_RE symbols by space in text
    new_text = BAD_SYMBOLS_RE.sub(' ', new_text) # delete symbols which are in BAD_SYMBOLS_RE from text

    ps = PorterStemmer()

    new_text = ' '.join(ps.stem(word) for word in new_text.split()) # keeping all words, no stop word removal
    return new_text

# STOPWORDS = adjust_stopwords(STOPWORDS)
print(STOPWORDS)
```

Figure 6. Data cleaning function

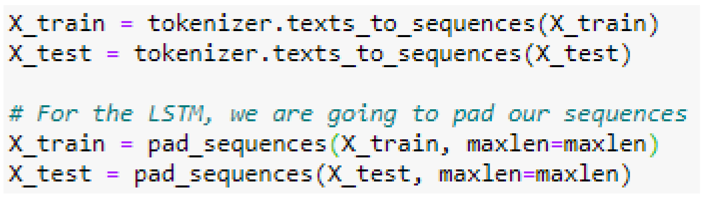
From there, we fit a simple tokenizer using Keras to convert our sentences into vectors.



```
%%time
max_words = 3000
tokenizer = text.Tokenizer(num_words=max_words, char_level=False)

tokenizer.fit_on_texts(X_train)
X_train = tokenizer.texts_to_matrix(X_train)
X_test = tokenizer.texts_to_matrix(X_test)
```

Figure 7. Fitting of the tokenizer onto our text

For the LSTM based models, we also needed to pad the sequences, as seen below. For the experiments, the max length for the sentences were 400 words.



```
X_train = tokenizer.texts_to_sequences(X_train)
X_test = tokenizer.texts_to_sequences(X_test)

# For the LSTM, we are going to pad our sequences
X_train = pad_sequences(X_train, maxlen=maxlen)
X_test = pad_sequences(X_test, maxlen=maxlen)
```

Figure 8. Code for padding the sequences for the LSTM models.

Architectures Initially Built

Baseline model—Simple Deep Neural Network

The sequential model is simply a linear stack of layers, where the input is a vector with a length of the size of our dictionary. We introduced dropout to avoid overfitting, batch normalization in order to have a smoother training process and to avoid "internal covariate shift", and had several activations to connect the network (4th layer → ReLU, Final layer → SoftMax).

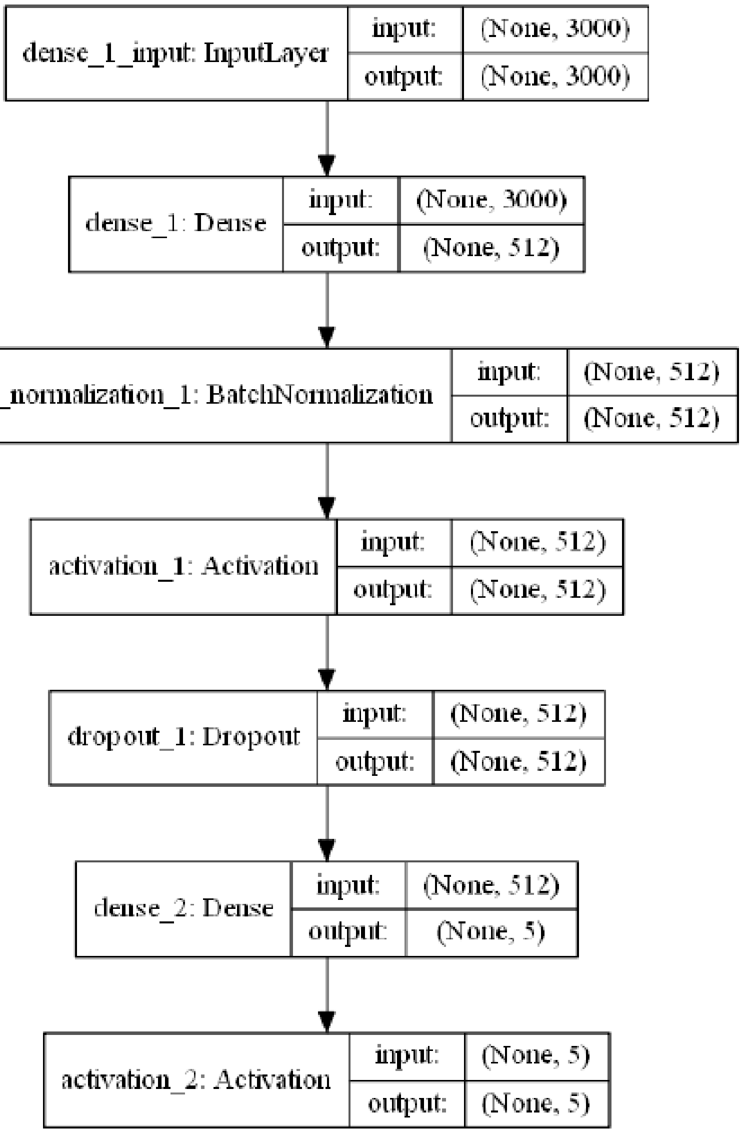


Figure 9. Baseline architecture

CNN LSTM

The LSTM is a type of RNN capable of retaining information for long periods of time, unlike typical RNNs, which usually struggle to learn “long-term dependencies”. This is because in addition to standard RNN “units”, LSTM has “memory” units that can maintain information in memory for a long time. Special gates control when information enters memory when it is supposed to be output, and when it is forgotten. This structure allows LSTM to learn long-term dependencies[2].

This LSTM is slightly enhanced, however, as it is a CNN LSTM.[3] In this model, we introduce an additional convolutional layer **on top** of the LSTM, as studies show the CNN LSTM as having a slight improvement over the standard LSTM. In addition, the model also includes dropout and batch normalization.

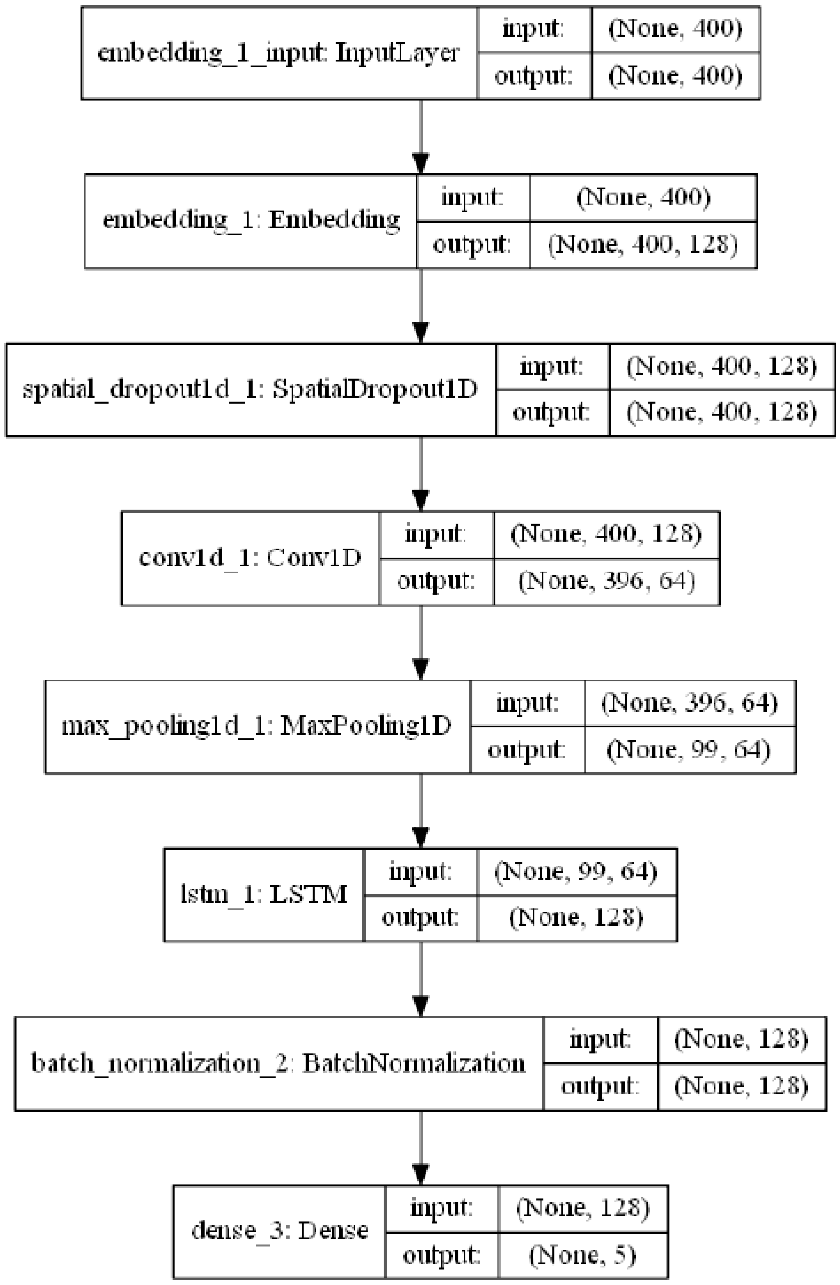

embedding_1_input: InputLayer | input: (None, 400) | output: (None, 400)
embedding_1: Embedding | input: (None, 400) | output: (None, 400, 128)
spatial_dropout1d_1: SpatialDropout1D | input: (None, 400, 128) | output: (None, 400, 128)
conv1d_1: Conv1D | input: (None, 400, 128) | output: (None, 396, 64)
max_pooling1d_1: MaxPooling1D | input: (None, 396, 64) | output: (None, 99, 64)
lstm_1: LSTM | input: (None, 99, 64) | output: (None, 128)
batch_normalization_2: BatchNormalization | input: (None, 128) | output: (None, 128)
dense_3: Dense | input: (None, 128) | output: (None, 5)

Figure 9. CNN LSTM architecture

One vs. All

The "one vs. all" approach was inspired by a section in the machine learning crash course from Google.[4] The main idea is that we create multiple learners—one for each class.

- We will have ***N*** learners for the multi-class classification problem, where ***N*** is the number of classes
- For each learner ***L***, we will train ***L*** on our training data X_{train} and y_{train}. However, y_{train} consists of only one label, making it a binary classification problem instead of multinomial
- For instance, learner L_i will still use all of X_{train}, but y_{train} will now be transformed to be a binary vector v_i where *i* denotes the star rating we are attempting to predict
- Once we have concluded our training, we will then create an ensemble model (bagging) that does the following

1. *L1, L2, …, L5* all assign p_i to each record in X_{test}, where p_i is the likelihood observation x_n belongs to class *i*
2. From there, our prediction is the following: p_n *= argmax(p1, p2, p3, p4, p5)*

After observing the challenge datasets 5 & 6 (challenge datasets 5 and 6 contained imbalanced classes—5 had only 2 star reviews and 6 had almost all 1 and 5 star reviews), we believe this approach is a clever way to tackle the challenges while still having a strong model.

Each model had the following architecture, very similar to the CNN LSTM above but with a few caveats (we went with a 500 length vector instead of 400 like the previous models) in order for it to succeed with binary classification.

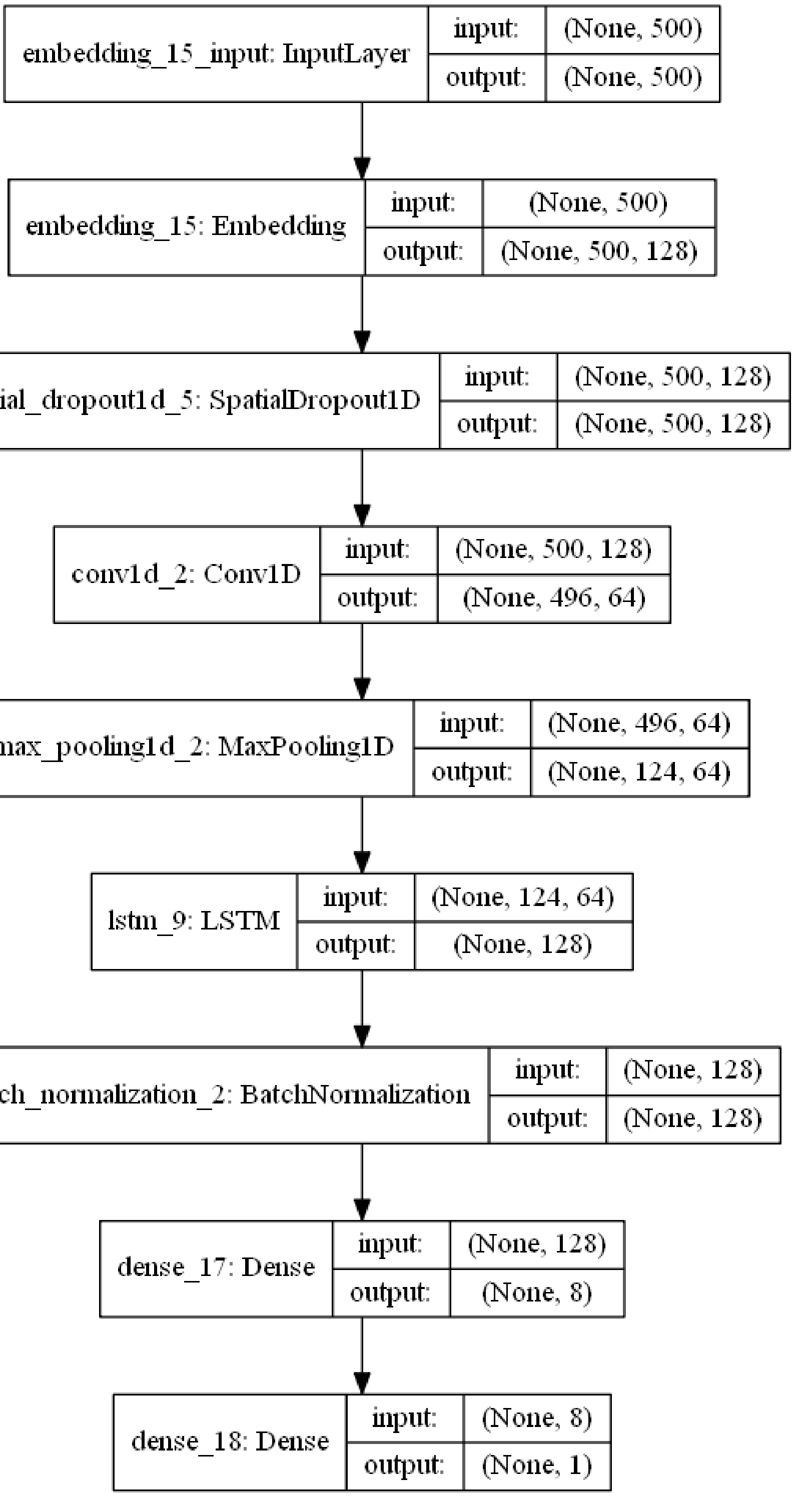
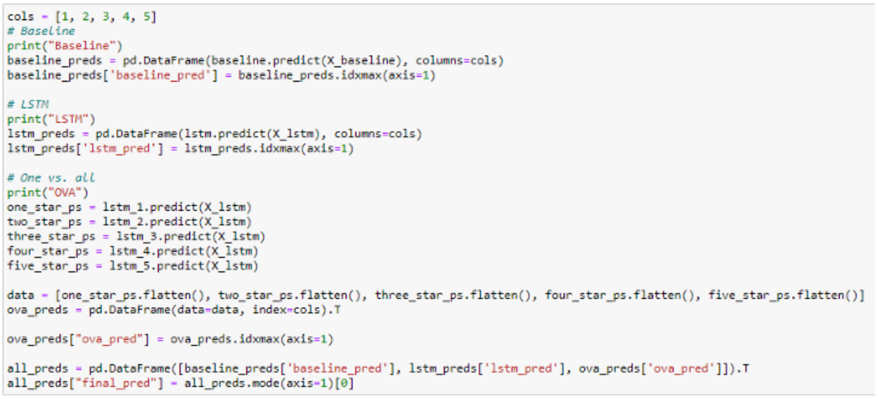


Figure 10. One vs. All architecture

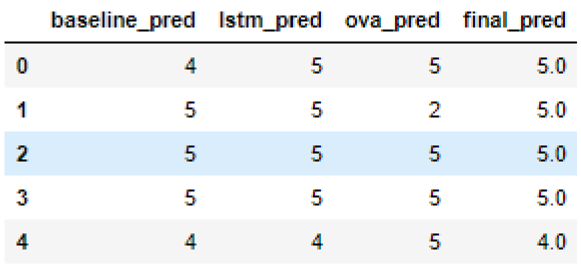
Ensemble Model

The ensemble model we create includes all of the above models; the three models vote ("bagging") on the final prediction. The code on how we did that is provided below. In python, the mode() function will choose the highest value if it cannot find a common value among a list. This works out in our favor when the predictions are high, as the distribution of the stars tends to follow that anyway (revisit “The Data” section if you are unsure why). However, it does not work in favor if the predictions are low.



```
cols = [1, 2, 3, 4, 5]
# Baseline
print("Baseline")
baseline_preds = pd.DataFrame(baseline.predict(X_baseline), columns=cols)
baseline_preds['baseline_pred'] = baseline_preds.idxmax(axis=1)

# LSTM
print("LSTM")
lstm_preds = pd.DataFrame(lstm.predict(X_lstm), columns=cols)
lstm_preds['lstm_pred'] = lstm_preds.idxmax(axis=1)

# One vs. all
print("OVA")
one_star_ps = lstm_1.predict(X_lstm)
two_star_ps = lstm_2.predict(X_lstm)
three_star_ps = lstm_3.predict(X_lstm)
four_star_ps = lstm_4.predict(X_lstm)
five_star_ps = lstm_5.predict(X_lstm)

data = [one_star_ps.flatten(), two_star_ps.flatten(), three_star_ps.flatten(), four_star_ps.flatten(), five_star_ps.flatten()]
ova_preds = pd.DataFrame(data=data, index=cols).T

ova_preds["ova_pred"] = ova_preds.idxmax(axis=1)

all_preds = pd.DataFrame([baseline_preds['baseline_pred'], lstm_preds['lstm_pred'], ova_preds['ova_pred']]).T
all_preds["final_pred"] = all_preds.mode(axis=1)[0]
```

Figure 11. Code that conducts the bagging



	baseline_pred	lstm_pred	ova_pred	final_pred
0	4	5	5	5.0
1	5	5	2	5.0
2	5	5	5	5.0
3	5	5	5	5.0
4	4	4	5	4.0

Figure 12. First 5 predictions for our test set, where each column represents a certain models predictions

Results from our Experiments

For our experiments, we **did not** do any hyper parameter tuning beforehand and utilized exponential decay for our learning rate in our optimizer (ADAM).

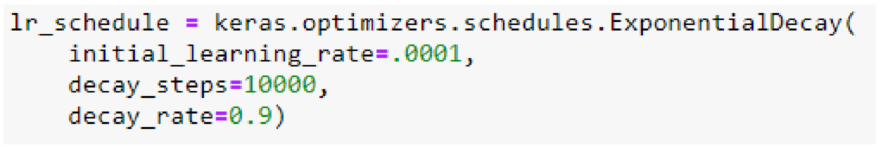
Here are the parameters we used for the experiments

Batch size: 512

Epochs: 5 for the LSTMs, 10 for the baseline, and a mix for each LSTM within the “One vs. All” ensemble (we used a mix because each model would overfit if we trained to long, photo attached)

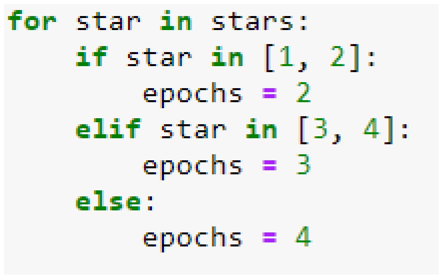
Optimizer: ADAM w/a learning rate with step decay

In addition, the “ensemble” results are simply us “bagging” the results from the other 3 models.



```
lr_schedule = keras.optimizers.schedules.ExponentialDecay(
    initial_learning_rate=.0001,
    decay_steps=10000,
    decay_rate=0.9)
```

Figure 13. Learning rate decay python object for training



```
for star in stars:
    if star in [1, 2]:
        epochs = 2
    elif star in [3, 4]:
        epochs = 3
    else:
        epochs = 4
```

Figure 14. Custom epoch design after noticing overfitting during the experiments

Some notes before our experiments:

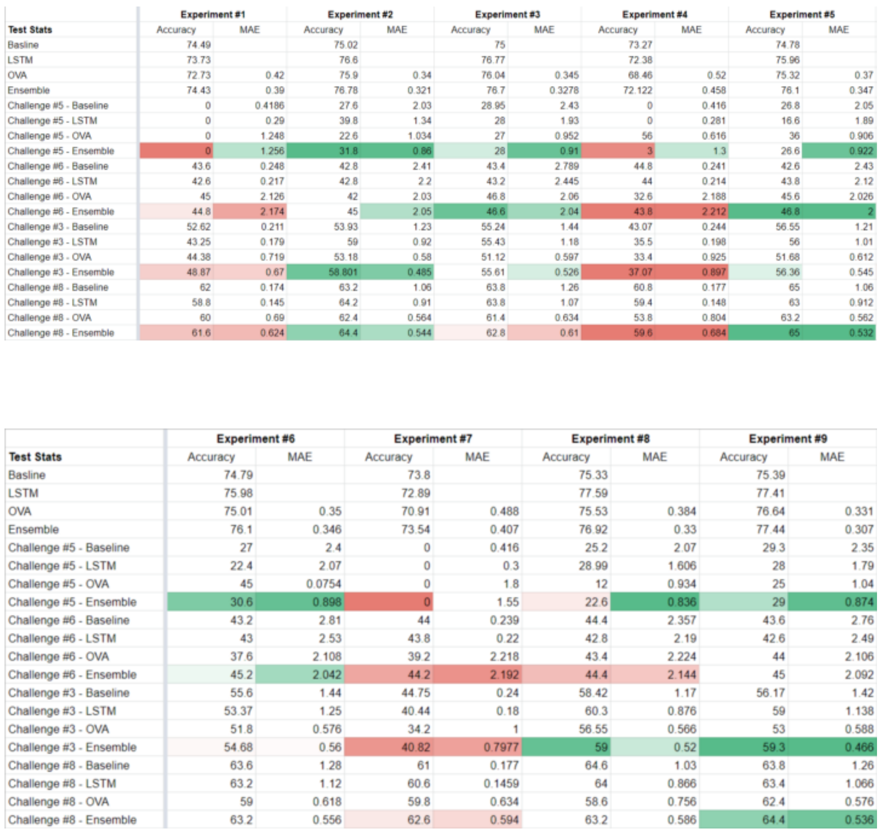
- The Bi-Directional LSTM worked but was slow to train and had no clear superiority to the CNN LSTM model. We did not include it in our experimentation due to its long training time. However, we chose include it in our final model
- The CNN model with Word2Vec embedding had low accuracy and slow train time, which we decided was not worth pursuing. We determined the same for the RNN with Word2Vec.

From our experiments, we found the following results:

- We found that the CNN LSTM model was the best approach
- In addition, training on just MAE ended up with fascinating confusion matrices and metric results. The majority of models (including the One vs. All ensemble model) ended up never predicting 2 or 3 stars (view figure 2).
- Our models trained with MAE as loss struggle especially on challenge #5, where all of the reviews had a true rating of 2 stars. This could’ve potentially have been solved by class balancing, but

we decided to avoid this approach as a whole, as other losses performed much better, and we did not want to further increase the size of our dataset

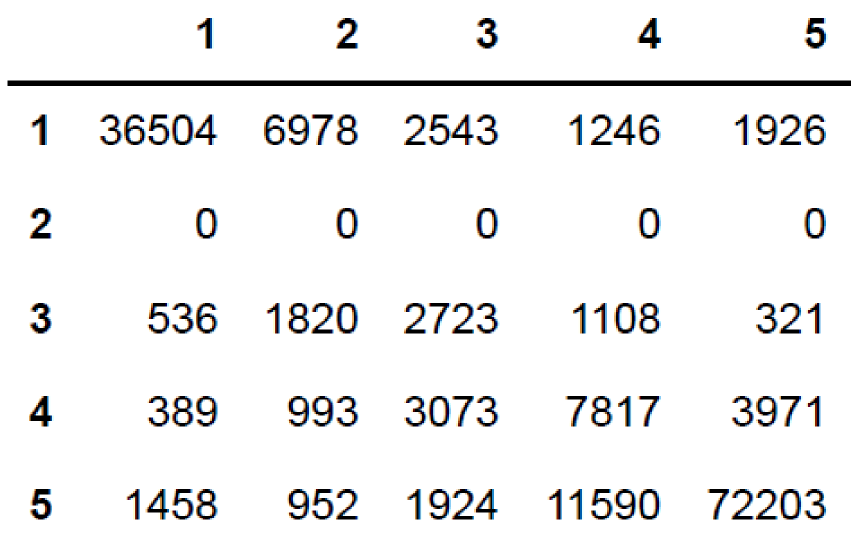
- When we combined all of our models, we had the most *stable* (where stable is defined as a good accuracy **and** MAE) results instead of choosing one model.
- Although experiment #5 had the best results on the challenges, we believe the results from experiment #9 were more promising for tuning due to the excellent results we obtained from our test set, which was not included in the challenge.



	Experiment #1		Experiment #2		Experiment #3		Experiment #4		Experiment #5	
Test Stats	Accuracy	MAE	Accuracy	MAE	Accuracy	MAE	Accuracy	MAE	Accuracy	MAE
Basline	74.49		75.02		75		73.27		74.78	
LSTM	73.73		76.6		76.77		72.38		75.96	
OVA	72.73	0.42	75.9	0.34	76.04	0.345	68.46	0.52	75.32	0.37
Ensemble	74.43	0.39	76.78	0.321	76.7	0.3278	72.122	0.458	76.1	0.347
Challenge #5 - Baseline	0	0.4186	27.6	2.03	28.95	2.43	0	0.416	26.8	2.05
Challenge #5 - LSTM	0	0.29	39.8	1.34	28	1.93	0	0.281	16.6	1.89
Challenge #5 - OVA	0	1.248	22.6	1.034	27	0.952	56	0.616	36	0.906
Challenge #5 - Ensemble	0	1.256	31.8	0.86	28	0.91	3	1.3	26.6	0.922
Challenge #6 - Baseline	43.6	0.248	42.8	2.41	43.4	2.789	44.8	0.241	42.6	2.43
Challenge #6 - LSTM	42.6	0.217	42.8	2.2	43.2	2.445	44	0.214	43.8	2.12
Challenge #6 - OVA	45	2.126	42	2.03	46.8	2.06	32.6	2.188	45.6	2.026
Challenge #6 - Ensemble	44.8	2.174	45	2.05	46.6	2.04	43.8	2.212	46.8	2
Challenge #3 - Baseline	52.62	0.211	53.93	1.23	55.24	1.44	43.07	0.244	56.55	1.21
Challenge #3 - LSTM	43.25	0.179	59	0.92	55.43	1.18	35.5	0.198	56	1.01
Challenge #3 - OVA	44.38	0.719	53.18	0.58	51.12	0.597	33.4	0.925	51.68	0.612
Challenge #3 - Ensemble	48.87	0.67	58.801	0.485	55.61	0.526	37.07	0.897	56.36	0.545
Challenge #8 - Baseline	62	0.174	63.2	1.06	63.8	1.26	60.8	0.177	65	1.06
Challenge #8 - LSTM	58.8	0.145	64.2	0.91	63.8	1.07	59.4	0.148	63	0.912
Challenge #8 - OVA	60	0.69	62.4	0.564	61.4	0.634	53.8	0.804	63.2	0.562
Challenge #8 - Ensemble	61.6	0.624	64.4	0.544	62.8	0.61	59.6	0.684	65	0.532

	Experiment #6		Experiment #7		Experiment #8		Experiment #9	
Test Stats	Accuracy	MAE	Accuracy	MAE	Accuracy	MAE	Accuracy	MAE
Basline	74.79		73.8		75.33		75.39	
LSTM	75.98		72.89		77.59		77.41	
OVA	75.01	0.35	70.91	0.488	75.53	0.384	76.64	0.331
Ensemble	76.1	0.346	73.54	0.407	76.92	0.33	77.44	0.307
Challenge #5 - Baseline	27	2.4	0	0.416	25.2	2.07	29.3	2.35
Challenge #5 - LSTM	22.4	2.07	0	0.3	28.99	1.606	28	1.79
Challenge #5 - OVA	45	0.0754	0	1.8	12	0.934	25	1.04
Challenge #5 - Ensemble	30.6	0.898	0	1.55	22.6	0.836	29	0.874
Challenge #6 - Baseline	43.2	2.81	44	0.239	44.4	2.357	43.6	2.76
Challenge #6 - LSTM	43	2.53	43.8	0.22	42.8	2.19	42.6	2.49
Challenge #6 - OVA	37.6	2.108	39.2	2.218	43.4	2.224	44	2.106
Challenge #6 - Ensemble	45.2	2.042	44.2	2.192	44.4	2.144	45	2.092
Challenge #3 - Baseline	55.6	1.44	44.75	0.24	58.42	1.17	56.17	1.42
Challenge #3 - LSTM	53.37	1.25	40.44	0.18	60.3	0.876	59	1.138
Challenge #3 - OVA	51.8	0.576	34.2	1	56.55	0.566	53	0.588
Challenge #3 - Ensemble	54.68	0.56	40.82	0.7977	59	0.52	59.3	0.466
Challenge #8 - Baseline	63.6	1.28	61	0.177	64.6	1.03	63.8	1.26
Challenge #8 - LSTM	63.2	1.12	60.6	0.1459	64	0.866	63.4	1.066
Challenge #8 - OVA	59	0.618	59.8	0.634	58.6	0.756	62.4	0.576
Challenge #8 - Ensemble	63.2	0.556	62.6	0.594	63.2	0.586	64.4	0.536

Figure 15. Results from our experiments



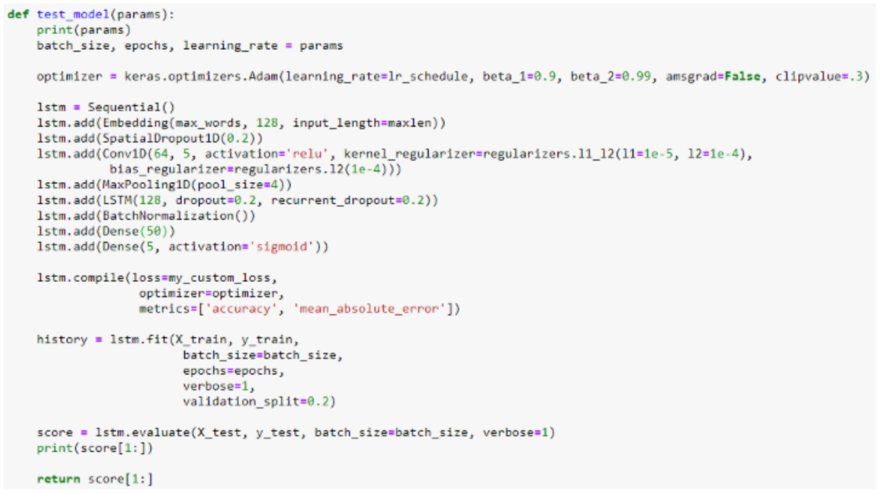
	1	2	3	4	5
1	36504	6978	2543	1246	1926
2	0	0	0	0	0
3	536	1820	2723	1108	321
4	389	993	3073	7817	3971
5	1458	952	1924	11590	72203

Figure 16. A sample confusion matrix we witnessed while training with solely MAE

Hyperparameter Tuning

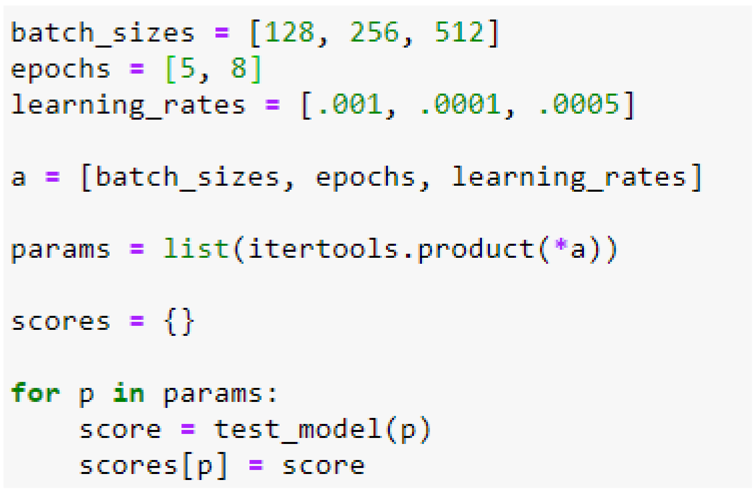
After we ran our experiments and found the best data formation, we began hyperparameter tuning for our models. We initially were going to use the built-in GridSearchCV from the scikit-learn library, but we had some difficulty utilizing the GPU in Tanner's desktop, thus the training was **very** slow.

Thus, we decided to write our own function to figure out the optimal parameters for our model.



```
def test_model(params):
    print(params)
    batch_size, epochs, learning_rate = params

    optimizer = keras.optimizers.Adam(learning_rate=lr_schedule, beta_1=0.9, beta_2=0.99, amsgrad=False, clipvalue=.3)

    lstm = Sequential()
    lstm.add(Embedding(max_words, 128, input_length=maxlen))
    lstm.add(SpatialDropout1D(0.2))
    lstm.add(Conv1D(64, 5, activation='relu', kernel_regularizer=regularizers.l1_l2(l1=1e-5, l2=1e-4),
              bias_regularizer=regularizers.l2(1e-4)))
    lstm.add(MaxPooling1D(pool_size=4))
    lstm.add(LSTM(128, dropout=0.2, recurrent_dropout=0.2))
    lstm.add(BatchNormalization())
    lstm.add(Dense(50))
    lstm.add(Dense(5, activation='sigmoid'))

    lstm.compile(loss=my_custom_loss,
                 optimizer=optimizer,
                 metrics=['accuracy', 'mean_absolute_error'])

    history = lstm.fit(X_train, y_train,
                       batch_size=batch_size,
                       epochs=epochs,
                       verbose=1,
                       validation_split=0.2)

    score = lstm.evaluate(X_test, y_test, batch_size=batch_size, verbose=1)
    print(score[1:])

    return score[1:]
```

Figure 17. Function to test the model with different parameters (this was the function for the LSTM)



```
batch_sizes = [128, 256, 512]
epochs = [5, 8]
learning_rates = [.001, .0001, .0005]

a = [batch_sizes, epochs, learning_rates]

params = list(itertools.product(*a))

scores = {}

for p in params:
    score = test_model(p)
    scores[p] = score
```

Figure 18. Building the set of parameters we were going to test for each run

From the hyperparameter tuning, we found the following parameters for each model:

Baseline → Batch Size = 256, Epochs = 5, Learning Rate = .007

LSTM → Batch Size = 256, Epochs = 8, Learning Rate = .0001

One vs. All → Customized Epochs (as seen in figure 2) but decided to use the Learning Rate Scheduler/Decay instead because it had the best results and hyperparameter tuning for each of the 5 models would've taken a long time.

Additional Models

Bidirectional LSTM

After we did more testing, we decided to introduce a new model into our ensemble, a Bidirectional LSTM. The Bi-LSTM, or Bidirectional LSTM, is frequently used in sequence classification. Bidirectional LSTMs train two LSTMs on the input text. One on the input sequence as-is and the second on a reversed copy of the same input. This provides more context to the network and can result in a more robust and accurate model.[5]

With the complex structure of Yelp reviews (and reviews in general), we believed this model would help output accurate predictions in the face of complex sentence structure ("it's a **mistake to not** come here", "every **other place** is **terrible**", etc.).

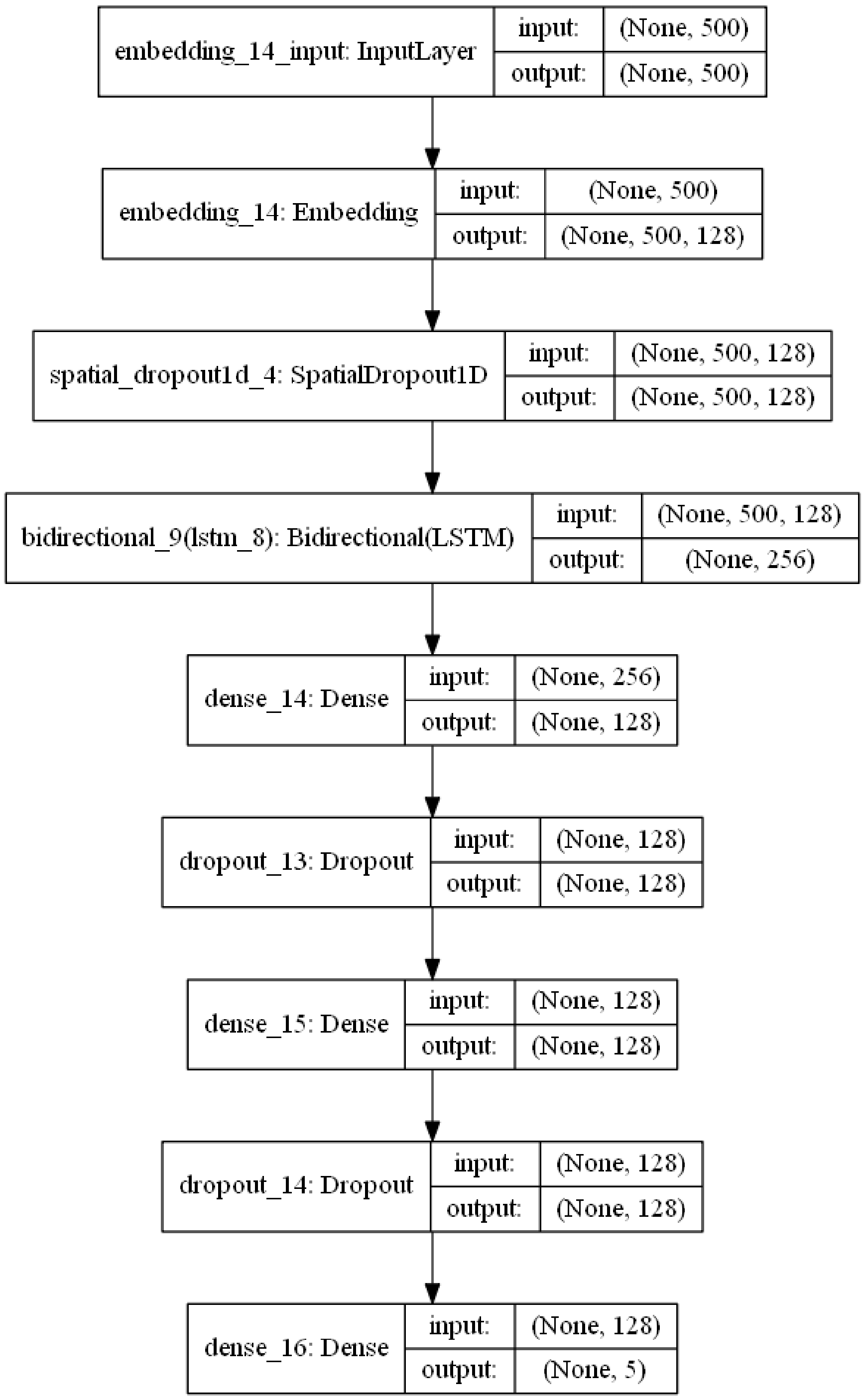


Figure 19. Bi-LSTM architecture

Figure 7. Bidirectional LSTM Architecture

For this model, we trained with batch size = 128 for 6 epochs and also used a Learning Rate Scheduler/Decay for the learning rate due to time constraints since the model took roughly 3 hours to train (Tanner's 1060 GTX 3GB was not enough). This model **was used** in the final submission, as it showed a slight improvement in test MAE and accuracy. The results of the model are the following, in the format [MAE, Accuracy / 100]:

[0.2810370138997345, 0.7835951897548025]

Boosted LSTM

Later on in the project, we also wanted to see if we could create a boosted LSTM with the entries we got wrong. For the test set, we misclassified roughly 20% of the entries, thus we trained a new model on those records. After we added this model to our ensemble, we saw a not-surprising spike in the test accuracy, but the true test was whether or not it would perform well on the challenge datasets. The results of the new ensemble model were the following, in the format [MAE, Accuracy / 100]:

[0.26952990785569264, 0.7906168983289084]

Final Model Selection

Our final architecture was an ensemble model with our Baseline, CNN LSTM, Bi-LSTM, and One vs. All models using uniform voting.

Each trained model predicted a class, and the ensemble picked the mode. We attempted **many** different ensemble methods and did extensive analysis of both bagging and boosting. We had slightly different results through each method, but ultimately found out that using just .mode() for voting and avoiding the inclusion of a boosted model in our ensemble was the best route

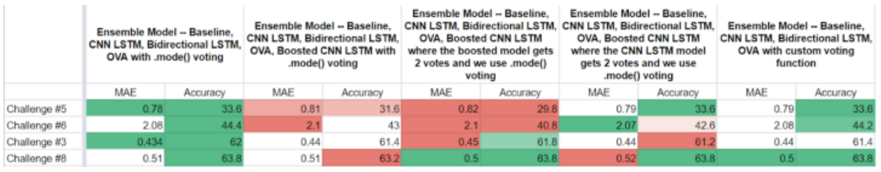
Bagging and Boosting

As stated above, we created a boosted CNN LSTM after initial testing. We were curious as to how our ensemble model would perform **with the inclusion** of the boosted model against the released challenges.

We discovered that this model **did not** help. When evaluating the test set, the boosted model, **on its own**, achieved the following poor scores as expected [MAE, Accuracy / 100]:

[0.1980009377002716, 0.252422571182251]

Testing on the challenges also did not go well, as the three experiments we ran with the inclusion of the boosted model and unique voting ended up with poor results (view columns 2, 3, and 4 in figure 20).



	Ensemble Model -- Baseline, CNN LSTM, Bidirectional LSTM, OVA with .mode() voting		Ensemble Model -- Baseline, CNN LSTM, Bidirectional LSTM, OVA, Boosted CNN LSTM with .mode() voting		Ensemble Model -- Baseline, CNN LSTM, Bidirectional LSTM, OVA, Boosted CNN LSTM where the boosted model gets 2 votes and we use .mode() voting		Ensemble Model -- Baseline, CNN LSTM, Bidirectional LSTM, OVA, Boosted CNN LSTM where the CNN LSTM model gets 2 votes and we use .mode() voting		Ensemble Model -- Baseline, CNN LSTM, Bidirectional LSTM, OVA with custom voting function	
	MAE	Accuracy	MAE	Accuracy	MAE	Accuracy	MAE	Accuracy	MAE	Accuracy
Challenge #5	0.78	33.6	0.81	31.6	0.82	29.8	0.79	33.6	0.79	33.6
Challenge #6	2.08	44.4	2.1	43	2.1	40.8	2.07	42.6	2.08	44.2
Challenge #3	0.434	62	0.44	61.4	0.45	61.8	0.44	61.2	0.44	61.4
Challenge #8	0.51	63.8	0.51	63.2	0.5	63.8	0.52	63.8	0.5	63.8

Figure 20. Results from several types of bagging/boosting methods on the challenges

After the ensemble model performed poorly on the challenges because of the additional boosted model we wondered if the model would perform better if we attempted different voting styles. Initially, we set out to double the number of votes the boosted model got, but that resulted in even worse results (column 3).

Seeing that the additional votes for the boosted model didn't improve the ensemble, we attempted to give multiple votes to our best, all-around performing model, the CNN LSTM. The results from that were also disappointing. However, giving the CNN LSTM two votes for our **test set** performed slightly better than our original ensemble with the normal vote:

[0.27154146493831016, 0.7873371856942059]

Additionally, giving the One vs. All model two additional votes gave us slightly better performance on our **test set**.

[0.28724035608308607, 0.7841199437763549]

Custom Voting

At this point, we believed that instead of utilizing additional votes to better-performing models, we could analyze **ties** that occurred in our predictions and utilize the model that performed for those types of ties. For instance, if one model performs particularly better when a 2 and 3 star tie occurs, we should utilize that model's prediction.

Roughly **3%** of our incorrect predictions resulted in ties; the most common ties in our test set were the following:

1. 4 & 5 stars—2356 occurrences

2. 1 & 2 stars—902 occurrences
3. 2 & 3 stars—619 occurrences
4. 1 & 5 stars—399 occurrences
5. 3 & 4 stars—365 occurrences

After identifying these occurrences, we analyzed each model to identify its performance during such a tie break—the results of our analysis are below (each number represents the model's **individual accuracy** when the ensemble was incorrect).

(4, 5) → Baseline: 42.31, CNN LSTM: 48.18, **Bidirectional LSTM: 48.33,** One vs. All: 39.56

(1, 2) → Baseline: 38.52, **CNN LSTM: 43.63,** Bidirectional LSTM: 40.92, One vs. All: 42.44

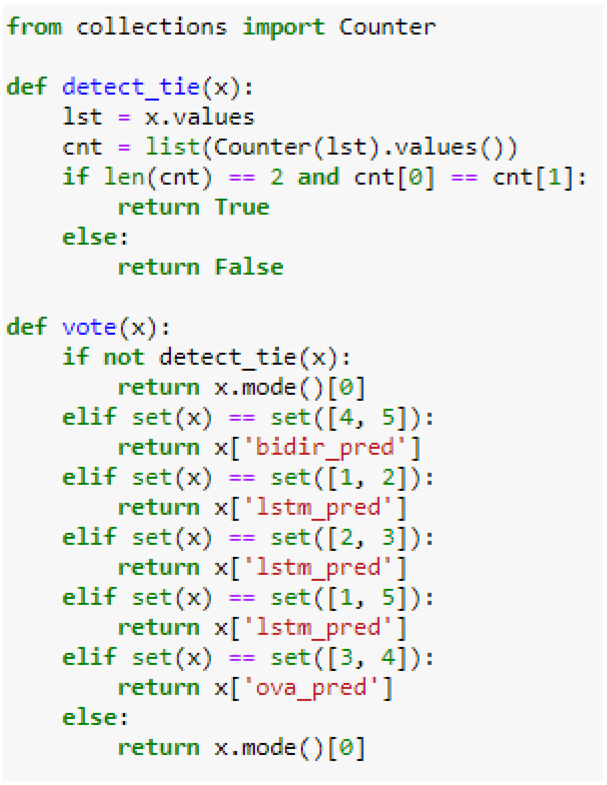
(2, 3) → Baseline: 38.86, **CNN LSTM: 47.60,** Bidirectional LSTM: 39.83, One vs. All: 41.84

(1, 5) → Baseline: 34.15, **CNN LSTM: 39.02,** Bidirectional LSTM: 38.88, One vs. All: 36.58

(3, 4) → Baseline: 44.49, CNN LSTM: 45.36, Bidirectional LSTM: 47.25, **One vs. All: 47.54**

Unfortunately, we do not see any accuracies above 50%, and this would give a direct buff to our test set results. Although we did not see the promising results we hoped for, we still wanted to try out custom voting on our challenges, and the model performed well, but not as well as the original ensemble with .mode() voting (view column 5 in figure ???).

With this data, we created the following functions to detect ties and decide tiebreakers:



```python
from collections import Counter

def detect_tie(x):
    lst = x.values
    cnt = list(Counter(lst).values())
    if len(cnt) == 2 and cnt[0] == cnt[1]:
        return True
    else:
        return False

def vote(x):
    if not detect_tie(x):
        return x.mode()[0]
    elif set(x) == set([4, 5]):
        return x['bidir_pred']
    elif set(x) == set([1, 2]):
        return x['lstm_pred']
    elif set(x) == set([2, 3]):
        return x['lstm_pred']
    elif set(x) == set([1, 5]):
        return x['lstm_pred']
    elif set(x) == set([3, 4]):
        return x['ova_pred']
    else:
        return x.mode()[0]
```

Figure 21. Tie detection and custom voting function, where x is a row in our prediction data frame

After running these experiments, we decided to avoid both custom voting and boosting. Here is the final model's MAE and accuracy on our test set:

[0.2826112759643917, 0.7832078713103233]

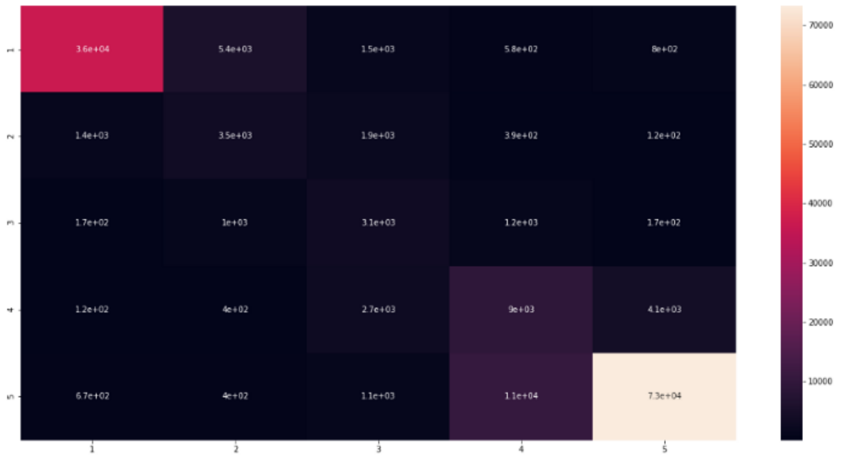
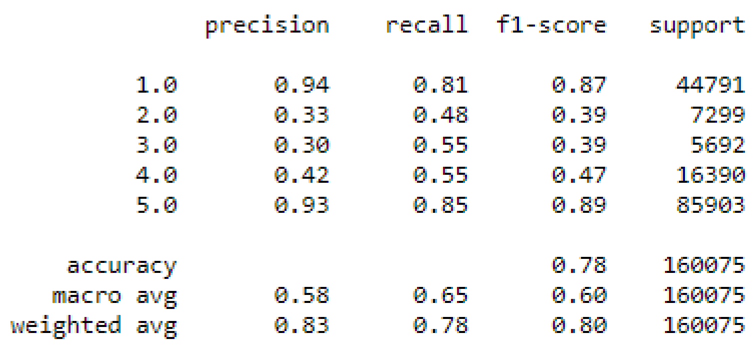


Figure 22. Confusion matrix for our final model



```
              precision    recall  f1-score   support

         1.0       0.94      0.81      0.87     44791
         2.0       0.33      0.48      0.39      7299
         3.0       0.30      0.55      0.39      5692
         4.0       0.42      0.55      0.47     16390
         5.0       0.93      0.85      0.89     85903

    accuracy                           0.78    160075
   macro avg       0.58      0.65      0.60    160075
weighted avg       0.83      0.78      0.80    160075
```

Figure 23. Classification report for our final model

Lessons Learned

We discovered that sentence structure is very important for sentiment analysis (as opposed to other NLP problems like topic modeling), so *not* removing stopwords improved the accuracy of our models. Intuitively, this makes sense as removing the stopword "not" from the following statements would completely change their meaning:

- "The food was not good"
- "The food was not bad"

With this lesson, we experimented with selective stopword removal, choosing to remove words like "the" but keeping words like "not". We also experimented with no stopword removal.

We also learned the importance of data cleaning. Most data scientists spend 80% of their time in this stage; just making sure that our dataset was clean and usable made huge differences in our model.

Although we did not use Word2Vec in our final model, we did learn how useful it could be for future applications. Instead of training an embedding layer, we first separately learn word embeddings and then pass to the embedding layer. The additional step of "pre-training" could help the model train faster overall.

Finally, we thoroughly examined methods of both bagging and boosting, and found that both methods produced slightly better results for our test set, but didn't perform as well as we wanted on the challenges.

Next Steps

For this project, we focused on ensemble learning. We trained a combination of LSTM, RNN, Bi-LSTM, and a boosted LSTM. The trained model uses a combination of voting and tiebreaking to predict the number of stars.

In the future, we would like to train a separate model, which is only called to predict in the event of a tie. We would train it on the “problem classes” which the different models in the ensemble sometimes disagree on. We chose not to pursue this option choosing instead to optimize our voting process.

In addition, we would live to look into models that utilize attention, specifically extending BERT to this problem and potentially building a HAN for the ensemble.

Sources

1. https://medium.com/@aariff.deen/creating-a-real-time-star-prediction-application-for-yelp-reviews-using-sentiment-analysis-9c7e94978bf6
2. https://arxiv.org/abs/1412.3555
3. https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/43455.pdf
4. https://developers.google.com/machine-learning/crash-course/multi-class-neural-networks/one-vs-all
5. https://machinelearningmastery.com/develop-bidirectional-lstm-sequence-classification-python-keras