Accuracy Analysis of Different ML Models on Amazon Fine Food Review Dataset

Masashi Omori
12/11/2017

1. Introduction

With the recent purchase of Whole Foods, an increasing number of consumers are relying on Amazon for their products and services. But with tens of thousands of products under each category, there needs to be a way to classify the products into ratings so popular items gets more visibility and the less popular ones are harder to find. The typical rating system used in many companies (Amazon, Yelp and Netflix) is based on the raw text review and the number of stars given by a consumer. The number of stars are averaged and shown alongside each product. Once the user digs deeper into the product, they are able to read the reviews people have put. There can be a problem with this approach because the stars are a very biased form of metric to base rankings off. Consider two customers who bought the same product. They could have had the same experience, but one of them tends to give a five star review, which is the highest rating possible, for any products with a good experience, whereas the other customer may only give out five star review on phenomenal products. For this particular instance, the latter customer gave a four star review. But both of them could have given very similar semantics in their review, something along the lines of “The product was great and worked as expected.” These biased star ratings adds up and overall provides for a biased rating system, based on a user’s expectation and/or mood. In order to remediate this problem, a machine learning approach will considered. The idea is to predict the positivity of a product (> 3 stars is a positively rated product, <=3 is a non-positive product) based on the text after having a predictive model trained on a dataset of reviews and stars. Multiple classification models were considered, and for this project a logistic regression, SVM, and a convolutional neural network based approach was tested. Also for the encoding of documents into numerical vectors, a bag-of-words and word2vec approaches were used.

2. Amazon Fine Food Review Dataset

The Amazon fine food review dataset is taken from kaggle which is a csv file. There are approximately 570k observations. The columns are explained below:

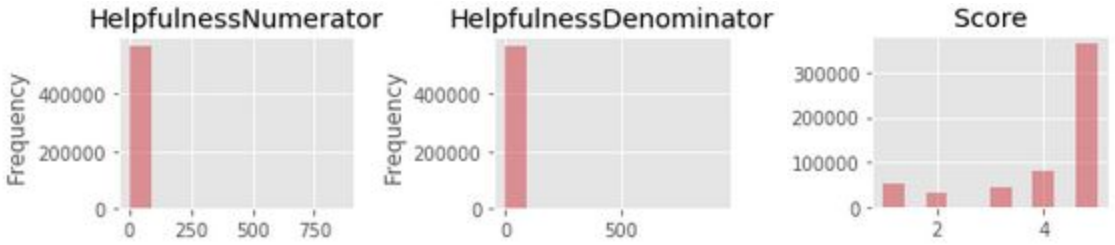
- Id - unique row Id
- ProductId - identifier for the product
- UserId - identifier for the user who reviewed the product
- ProfileName
- HelpfulnessNumerator - number of users who found the review helpful

- HelpfulnessDenominator - number of users who indicated whether they found the review helpful
- Score - rating between 1 and 5
- Time - timestamp for the review
- Summary - brief summary of the review
- Text - text of the review

In this project, the focus is to create a model which will predict the positivity of a product based on the review, but will also see if other features might add some value to the accuracies through statistical inferences and feature engineering.

3. Data Wrangling/Analysis

Initially, distributions of the features helpfulness numerator, denominator and score were checked for anomalies as seen below.



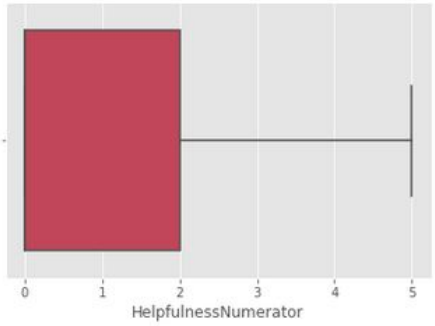
There are definite outliers in this dataset, as seen by the scaling of the histogram, and users tend to give feedback for positive reviews more than neutral or negative reviews. But since histograms do not deal with outliers very well, a box plot was chosen to

visually observe the distributions in detail.

In [6]:
```
print 'Helpfulness Numerator quantiles'
print data['HelpfulnessNumerator'].quantile([0.7,0.8,0.9, 0.95, 0.99, 0.995])

ax = sns.boxplot(x=data['HelpfulnessNumerator'], showfliers=False)
plt.show()
```

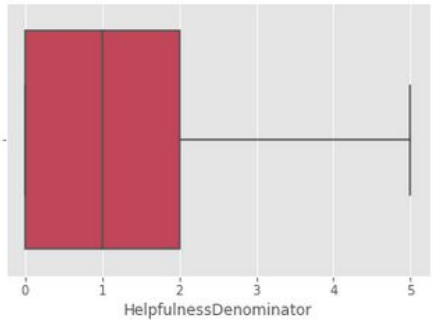
```
Helpfulness Numerator quantiles
0.700    1.0
0.800    2.0
0.900    4.0
0.950    7.0
0.990    19.0
0.995    30.0
Name: HelpfulnessNumerator, dtype: float64
```



In [7]:
```
print 'Helpfulness Denominator quantiles'
print data['HelpfulnessDenominator'].quantile([0.7,0.8,0.9, 0.95, 0.99, 0.995])

ax = sns.boxplot(x=data['HelpfulnessDenominator'], showfliers=False)
plt.show()
```

```
Helpfulness Denominator quantiles
0.700    2.0
0.800    3.0
0.900    5.0
0.950    9.0
0.990    23.0
0.995    35.0
Name: HelpfulnessDenominator, dtype: float64
```

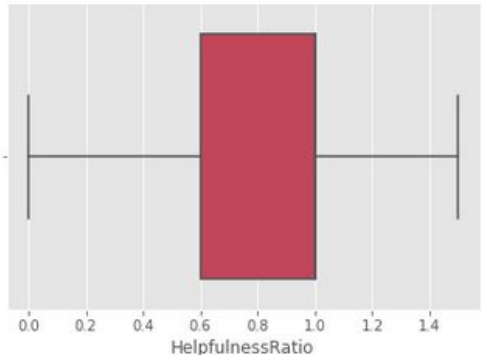


From the boxplots, notice that most of the data spread is within 0 and 5 for these features. Further a new feature called ‘Helpfulness ratio’ is created to assess the percentage of helpful tags a review received. The plot for that is found below:

```
In [8]: print 'Helpfulness Ratio quantiles'
        print data['HelpfulnessRatio'].quantile([0.7,0.8,0.9, 0.95, 0.99, 0.995])

        ax = sns.boxplot(x=data['HelpfulnessRatio'], showfliers=False)
        plt.show()
```

```
Helpfulness Ratio quantiles
0.700    1.0
0.800    1.0
0.900    1.0
0.950    1.0
0.990    1.0
0.995    1.0
Name: HelpfulnessRatio, dtype: float64
```



By definition of the ratio which is numerator divided by the denominator, values over 1.0 are outliers and should not be considered. Fortunately, there were only two rows with this problem and hence were removed.

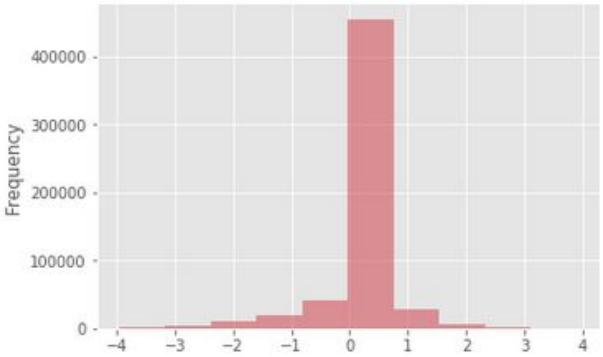
A second feature was engineered in an attempt to create values which would have less bias than the star ratings. Since different users can be biased towards certain ratings (users who tend to give 5 if they are satisfied vs users who give 4 for the same level of satisfaction), the data will be grouped into users and the average score per user will be calculated. The average score for each user will be used to calculate the deviation away from their average, and set as the column normalized_score. As an example, If a user gave three reviews, 2, 2, 5, which averages to an average_score of 3, the normalized_score would be -1,-1, 2 respectively. This gives an idea of how much satisfaction/dissatisfaction a user gained from using a certain product which may be more accurate to use than the star ratings. For example, the below user gained a satisfaction of 0.5 compared to his/her average level of satisfaction with the products.

Summary	Text	HelpfulnessRatio	avg_score	normalized_score
Candy was delivered very fast and was purchased at a reasonable price. I was home bound and unable to get to a store so this was perfect for me.		NaN	4.500000	0.500000

To assess if this new column would be useful in making predictions, the distribution was plotted on a histogram. A column with very little variance would most likely not add much value to the model.

```
In [14]: fig = plt.figure()

         tmp_data['normalized_score'].plot.hist(alpha=0.5)
         plt.show()
```



As can be seen above, the normalized score column has roughly 450k records (out of a total 570k) and very few scores spread out besides that. Upon inspection, it was found that there are very few occasions of a single user putting multiple reviews in this dataset, which explains for the distribution. Hence this feature will not be of much use in our modeling.

4. Inferential Statistics

In an attempt to quantify if the helpfulness numerator and denominator will improve the prediction accuracy of our model, the pearson correlations were compared against the score variable. The pearson correlation is defined as follows:

$$\rho(X,Y) = \frac{cov(X,Y)}{\sigma_X \sigma_Y}$$

Where $\rho(X,Y)$ is the pearson correlation coefficient, $cov(X,Y)$ is the covariance of the two random variables, and σ is the standard deviation. Since the magnitude of the covariance is unbounded, by dividing it by the product of the standard deviations, pearson coefficient correlation becomes a value between -1 and 1. By using the corrcoef function of numpy, the calculated pearson correlation coefficients were:

- Helpfulness numerator: -0.0326
- Helpfulness denominator: -0.098

From this observation, a hypothesis testing was done. The following null and alternative hypothesis was set:

h_0: There is no correlation between the two features and score variable.

h_1: There is some correlation between the two features and score variable.

Then a permutation sampling was done on the helpfulness variables and scores were kept in the same order, resulting in a dataset where each score was linked with a random helpfulness numerator and denominator. Then the pearson coefficient was calculated, and these steps were repeated 5000 times. Once the outputs were stored, a 99% confidence interval of the pearson coefficient was calculated. This was found to be the below:

- Helpfulness numerator: [-0.00517265 0.0029612]
- Helpfulness denominator: [-0.00476361 0.00298221]

Since the observed correlation is outside of the ranges found, it is concluded that the null hypothesis can be rejected. However, the observed correlation value is so small that this feature may not add value to the prediction output, so this was discarded.

5. Text Encoding: Bag of Words

Once the data cleaning steps were done, the texts needed to be massaged into a format such that machine learning models can understand. As a first approach, the bag-of-words method was used. This happens by first creating a list of all vocabularies used by the entire dataset. Then for each data point, the above list is initialized with all zeros, then the corresponding index's entry is incremented for each word in the data point. For example, consider the sentence 'I eat an apple' with a vocabulary list of 'I', 'eat', 'an', 'apple', 'orange'. The cardinality of the vocabulary list is five, so there will be a 1x5 vector representation of each datapoint. In the above example's case, the representation will be [1,1,1,1,0]. Similarly, for the sentence 'I eat eat an orange,' the representation would be [1,2,1,0,1]. The benefit of this representation is that document similarities can be calculated via cosine similarity defined as below:

$$similarity = \frac{dot\_product(d_1, d_2)}{||d_1|| * ||d_2||}$$

Where $d1, d2$ are the encoded vectors. Intuitively, this measures the closeness of the two vectors in n-dimensional space, where n is the number of vocabularies used to encode the text. However, the main issue with this encoding is that the order of words are not kept, so by this standard, the sentences 'cat eat rat' and 'rat eat cat' are identical. Regardless, it is a common encoding used in NLP and will be used to create the initial baseline models. Another issue with this representation which requires some preprocessing of the data. First off, the existence of stopwords heavily bias the resulting vectors. For example, most sentences will have very common words such as 'the', 'a', 'an', punctuations and the likes. These words are removed from the original

data source so the models will only look at significant terms, and the list of stopwords are provided by the NLTK library. Lastly, the vectorizer class used to create the bag of words representation will differentiate between terms of different cases, such as 'apple' and 'Apple.' Hence, before we start the vectorization process, the datasource will be turned into all lowercase.

Once the above steps were done, the end result is a numerical vector representation of the documents, a matrix with 568454 rows (1 for each observation) with 110979 columns (number of vocabularies identified by the vectorizer class). With this, a logistic regression and SVM model were trained.

6. Baseline Modeling: Logistic Regression

Since the problem at hand is a classification problem, logistic regression fits the needs of this project. A logistic regression assumes that a linear combination of the features can be modeled by the log-odds:

$$log\frac{p}{1-p} = b_0K_1 + b_1k_1 + ... + b_nk_n$$

Where P is the probability of a successful outcome (positive review, hence $1-p$ is the probability for negative), b are the coefficients, k are the parameters, and n is the number of parameters, which in this case would be the number of vocabularies, 110979. Then the goal of logistic regression model is to find optimal values of b such that the log-odds are maximized. This is done through sklearn's logistic regression model api.

Through some hyperparameter tuning using gridsearch, l2 penalty and a max_iteration count of 10 was found to be the optimal parameters, and a confidence interval of accuracies was created through 1000 repeated testings. The 99% confidence interval for accuracies found was [0.83733333, 0.84466667].

7. Baseline Modeling: SVM

Similar to logistic regressions, a gridsearch methodology was used to tune the hyperparameters of SVM, and found that a l2 regularization along with a squared_hinge loss function achieved the highest accuracy. The SVM classification function can be defined as follows:

$$classification = \begin{cases} Positive, w^TX + b > 0 \\ Negative, w^TX + b <= 0 \end{cases}$$

Where the w are the weights (coefficients), X is the inputs matrix (parameters) and b is the bias. Training a SVM will create a hyperplane which separates the data into classification. From this, the 99% confidence interval for accuracy was found to be [

0.83666667, 0.84466667]. Logistic regression and SVM both achieve a decent result of around 83%. The interesting fact to note here is that both models seems to converge rather quickly (ie: max_iter parameter has little effect on the models) and to a similar accuracy for this dataset. This is good bases to believe that further tuning of these models may not improve the accuracy much further. Also note that even after running the simulation 1000 times, the models have very little deviation from each other in terms of accuracies. This implies that there is very little error from variance and most of the error term comes from bias.

8. Another approach: Word2vec and CNN

Now that the baseline models are in place, a convolutional neural network was created and accuracies measured. But before that, a different approach to word embeddings into numerical vectors was considered, word2vec. The main difference between word2vec and bag-of-words is that the representation of vectors are not as sparse and the context of the words are stored.

In terms of sparsity of a bag-of-words representation, imagine there is a vocabulary size of 1000 and the individual documents are 5 words long. Than each sentence would at most have 5 non-zero entries and 995 zero entries. A lot of additional space is required to encode these without adding much value. The word2vec approach will generate much more dense representations as will be seen later.

Another advantage of the word2vec approach is the notion of word context. It assumes that given a sentence and a target word, then the words surrounding the target word will be highly correlated to the context of the target word. As an example, consider the sentence ‘quick brown fox jump over the lazy dog.’ With a window size of 1 and target word fox, then the two tuples (brown, fox) and (jump, fox) is created. These co-occurences of words are kept as a probability distribution to predict the context of a target word. From the above, it can be seen that fox and brown goes together along with jump and fox, but fox and lazy would have low probability that they are correlated. The end result of a word2vec encoding is a matrix such that given a target word, it will output a vector with probabilities of that word being associated with all other words, hence the vector is also not sparse.

Once the encodings are in place, the shape of the CNN was considered. A basic form CNN will have an input layer, an embedding layer, convolutional layer, a pooling layer, flatten layer and a dense layer which would be the output layer. Different hyperparameters for the layers were considered based on the article "A Sensitivity Analysis of (and Practitioners' guide to) Convolutional Neural Networks for Sentence Classification." https://arxiv.org/pdf/1510.03820.pdf.

9. Hyperparameter Tuning

The goal of a convolutional layer is to apply transformations to the original embedding and extract multiple features from it. The parameters to concern here are the size of the convolutions, number of feature maps, and the list of the convolutions to use.

The size of the convolution can be thought of as a window size of each transformation. For example, given an 5x5 matrix representation of words and a convolution size of 2x5, it will create 4 transformed images. In general, given a nxk matrix and a window size of mxk, it will create $n - m + 1$ image mappings.

Number of feature maps defines how many times these convolutions will be applied to the incoming data. For the project's case, multiple values ranging from 10 to 700 was tested and found that 700 was the value which had the best accuracy rate.

Since more than one type of convolution can be applied to the input data, a list of convolutions were provided to the input also. Multiple combinations of filters were tried and found that using 3 of the same single best kernel size worked the best in this project's case.

The authors of the sensitivity analysis paper recommends to use 1 max-pooling layer, which was utilized, and a ReLU or TANH activation function. Through trials, it was observed that sigmoid function works best for our case. The result of the multiple trials can be found on the tables below:

filters used	training accuracy	testing accuracy
9	88.50	86.9
9,9	88.75	87.19
9,9,9	92.01	90.56
8,9	91.31	89.80
9,10	89.56	87.69
8,9,10	91.94	90.42

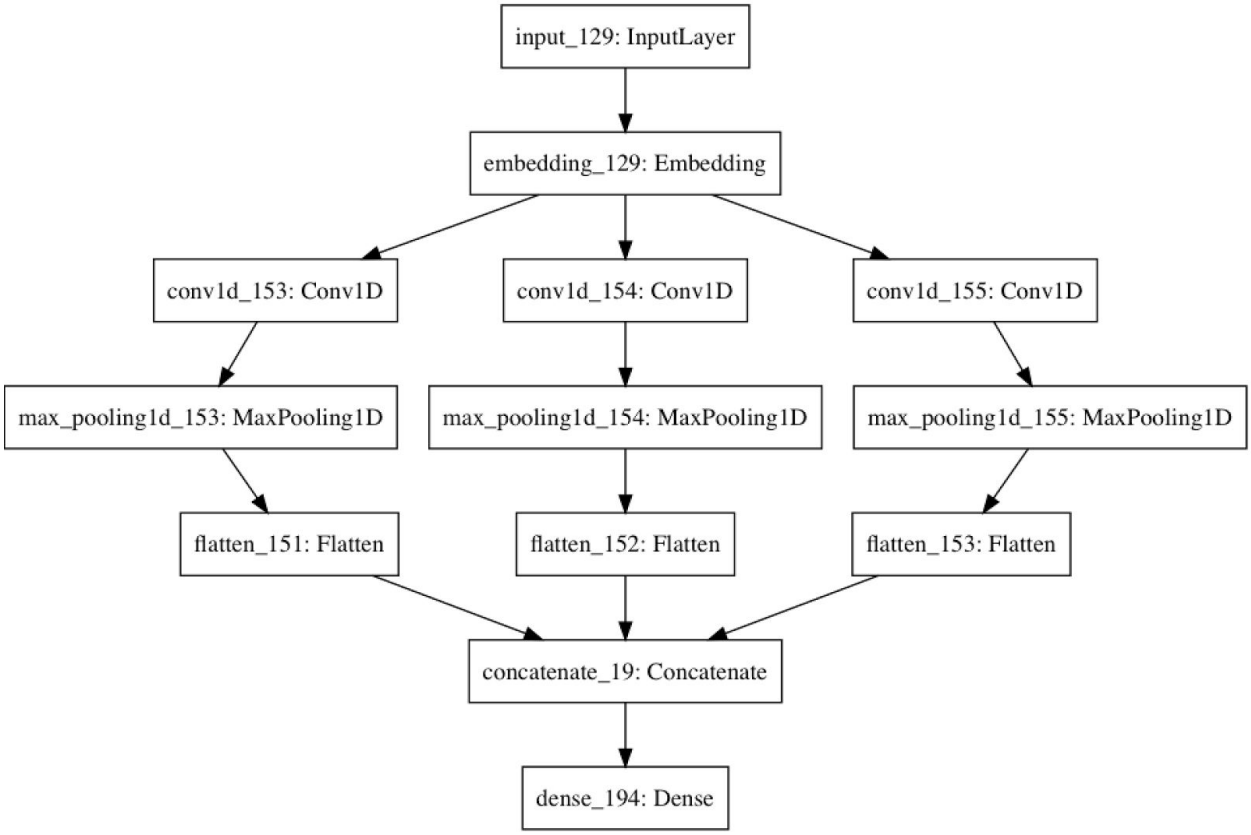
number of features	training accuracy	testing accuracy
10	92.59	91.20
50	93.79	92.22
100	94.10	92.48
200	94.35	92.64
400	94.51	92.69
600	94.47	92.69
700	94.85	92.91

output dimension	training accuracy	testing accuracy
2	85.44	84.54
4	86.42	85.51
8	87.75	86.47
16	87.77	86.41
24	87.90	86.40
30	87.43	85.92

activation function	training accuracy	testing accuracy
relu	86.36	85.20
tanh	86.79	85.53
sigmoid	86.78	85.63

10. Creating the Optimal Model

Once the optimal hyperparameters were found, the final CNN was trained. The below is a visual representation of the model.



For the testing data, it correctly labeled the data 93.28% of the time compared to 83% of the baseline models. This is a very significant increase (with a 99% confidence) and shows that a correctly tuned CNN can outperform a more standard machine learning model by a very large amount. With further tuning of the parameters and adding more dense layers, there is a good chance that there will be even more increase in accuracy for this model.

11. Further Notes

In terms of training time, a single logistic regression or SVM took less than 5 minutes on my laptop (macbook pro 8GB RAM), whereas training a CNN took close to 1 hour per epoch (2 epochs were used for a total of ~2 hours. Multiply this by the number of models created during hyperparameter tuning, it would've taken days for the process to finish).

To speed up the process, an AWS p2.xlarge instance was used, which has a builtin GPU. Since most calculations of a neural network is matrix computation, a GPU sped this process up significantly (~10~20x). With this, training the final model took roughly 15 minutes. The total amount of time took to do the hyperparameter tuning was around 7 hours, and costed about 6 dollars in AWS EC2 fees.