Amazon Review Helpfulness Prediction

Cheng-Hao Tai

June 13, 2018

Abstract

This document exhaustively details my complete approach towards predicting the helpfulness of Amazon review scores on a curated train and test set. The datasets used in this Kaggle competition presented unique challenges that couldn't be resolved through brute-force computation or simple models. Throughout this process, I explored numerous machine learning models: the Multilayer Perceptron, Random Forest Regressor, Gradient Boosting Classifier, and Gradient Boosting Regressor. These models were used in conjunction with 2 radically different preprocessing strategies. Ultimately, I found that a lightweight, preprocessed dataset in conjunction with a model consisting of a **multi-stage Gradient Boosting Classifier** and a **multi-stage Gradient Boosting Regressor** performed the best. My final Kaggle standing is 9th place out of 28 participants with a mean absolute error score of 0.16703. *Please note that the submitted Jupyter Notebook code refers only to the model that yielded the best performance. Code snippets for unsuccessful attempts will be sprinkled throughout this report.*

1 Given Baseline vs. Hidden Baseline

The overarching goal of this project was to accurately predict the helpfulness of particular reviews for various Amazon products. *Accuracy* was evaluated using the metric of **mean absolute error (MAE)**. The lower the MAE, the more accurate the overall predictions were. A set of baselines for the public and private datasets were stipulated:

- Public Leaderboard Baseline: 0.26919
- Private Leaderboard Baseline: 0.25207

At the time of the project kick-off, the stipulated baselines were misleading. While my first model (Random Forest Regressor) easily surpassed the public baseline, its performance stalled at an MAE value of 0.23. It wasn't until I stumbled upon the *hidden baseline* (the **outOf** attribute) during my experimentation with Neural Networks that I made significant progress.

When submitting on the public leaderboard, an entry that consists purely of the **outOf** attribute easily surpasses the public baseline with a displayed MAE of 0.1988. Additionally, when evaluating MAE on the training set, using the **outOf** attribute resulted in an MAE of 0.1938. Consequently, I gleaned two important insights from this discovery:

1. The **outOf** field was the most important of the original, unprocessed features.
2. The fact that both the training set's MAE and the public leaderboard's MAE were so similar suggests that there was an underlying similarity to the training and test sets. This meant that optimizing on my training set would likely result in a corresponding performance increase on the test set and that the resultant model would be well-generalized (assuming that I didn't overfit).

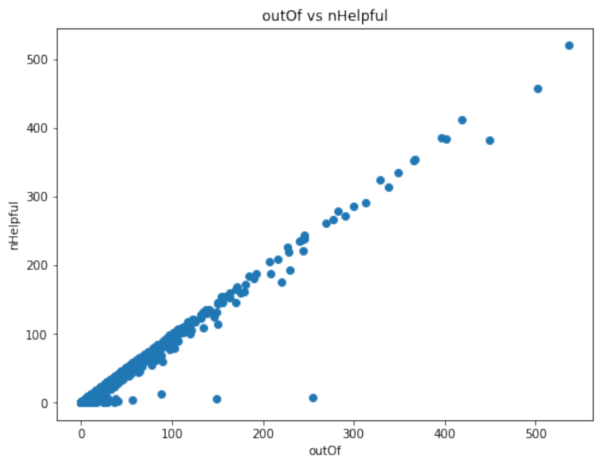


Figure 1: Linear Relationship between outOf and nHelpful in the Train Set

Figure1 is a visualization of the relationship between the **outOf** attribute and the **nHelpful** label value,

As can easily be seen from Figure 1, there is a clear 1:1 mapping relationship between most of the elements in the training dataset. From this plot, it can also be seen that there is a denser cluster of reviews with lower **outOf** values than higher ones. Though my initial models do not utilize this information, my ultimate successful model relied heavily upon the structure demonstrated in this plot.

2 Kick-Off / Random Forest Regressor

2.1 Initial Preprocessing

The first step in this project was deciding which features would be useful. However, rather than visualizing the distributions of each feature in the train set and cross-validating them with the corresponding features in the test set, I used a qualitative heuristic to aid my feature selection (asking myself whether or not a human would find this information intuitively useful). Unfortunately, my naive approach to feature selection resulted in bloated models, long training times, sub-par results, and a significant delay in arriving at an optimal solution. These initially selected features are summarized in Table 1.

Attribute	**Preprocessing Strategy**
categoryID	Extract directly from dataset
rating	Extract directly from dataset
itemID	Removal of 1st letter and casting to *int*
reviewerID	Removal of 1st letter and casting to *int*
outOf	Extract *outOf* field from *helpful* dictionary entries
reviewText and *summary*	Word2Vec embedding and TFIDF document embedding
reviewLength	Length of characters in *reviewText*
reviewTime	Parse string representation into 3-element list [month, day, year]
category	Parse categories to individual words and encode using multi-hot encoding

Table 1: Description of Selected Features and Applied Preprocessing

2.1.1 Preprocessing: reviewText and summary

A couple of the attributes and accompanying preprocessing strategies merit some elaboration. Of the features listed in Table 1, the most involved preprocessing was required on *reviewText and summary.*

These two attributes represent the majority of textual data provided in the training and test sets. I took two approaches in preprocessing this text data: **(1)** word embedding using Word2Vec and **(2)** document embedding using TFIDF.

Starting with strategy **(1)**, I first trained a Word2Vec model on the training corpus:

```
# Extract all textual information to assemble training corpus
reviews = list(train_df.reviewText.values)
summaries = list(train_df.summary.values)
corpus_paired = zip(reviews, summaries)
# Join reviews and summaries together
corpus = []
for pair in corpus_paired:
    corpus.append(' '.join(pair))

# Remove punctuation and uncase all text
print('Removing punctuation and uncasing text...')
replace_punctuation = str.maketrans(string.punctuation, ' '*len(string.punctuation))
docu_corpus = [text_processor(doc, replace_punctuation) for doc in corpus]
word_corpus = [w for doc in docu_corpus for w in doc]

# Train and save a Word2Vec embedding model
print('Training Word2Vec model on training set corpus...')
model = Word2Vec(docu_corpus, min_count=1, size=50, window=6)
```

This trained Word2Vec model was then used to encode words into 50D numerical vectors. For each individual review entry, these 50D word vectors was averaged to obtain a 50D representation of each corpus document. To ensure consistency between the training and test corpus, the test corpus was trimmed to exclude words that were not encountered in the training set.

As for strategy **(2)**, I transformed each review's text information into a TFIDF vector representation to obtain a sparse-matrix representation of the training and test set's corpuses. Subsequently, I reduced the dimensionality of this sparse matrix down to 100D using truncated SVD (which was necessary since sklearn's PCA library was not compatible with the sparse matrix data format). The method used to perform this TFIDF transformation is as follows:

```
def tfidf_fit(self):
        """Fits a model of the tfidf transform"""
        review_and_summary = list(zip(self.df.reviewText.values, self.df.summary.values))
        corpus = [' '.join(x) for x in review_and_summary]
        validated_corpus = self.vocab_validate(corpus)
        vectorizer = TfidfVectorizer(stop_words=stopwords.words('english'), lowercase=True,
                                     use_idf=True, smooth_idf=True)
        vectorizer.fit(validated_corpus)
        return(vectorizer)
```

2.1.2 Preprocessing: category

There were a total of 1042 unique categories present in both the training and test sets. However, product reviews often included overlapping categories (i.e. "Women, Clothing, Fashion" would be considered separate from the "Clothing, Fashion" category). As such, I decided to parse each category field down to individual words and subsequently performed a multi-hot encoding process followed by a PCA dimensionality reduction down to 800 dimensions:

```
def categories(self):
        """Returns a binarized representation of the categories column"""
        categs = self.df.categories.values
        categs_list = self.parse_categories(categs)
        master_placeholder = np.zeros((len(categs_list), len(self.categs_dict)))
        for i, sentence in enumerate(categs_list):
            master_placeholder[i, :] = self.convert_to_index(sentence)
        # Perform a PCA dimensionality reduction
        transformed_cats = self.pca_model.fit_transform(master_placeholder)
        return(transformed_cats)
```

2.2 Random Forest Regressor

My first idea was to utilize a random forest regressor due to its ease of implementation and relatively fast training speed. This stage of the project was meant to provide insights into the design of a subsequent multilayer perceptron deep learning model. I utilized sklearn's Random Forest Regressor library with the following parameter initializations:

```
rf_model = RandomForestRegressor(n_estimators=600, max_depth=10, verbose=1, n_jobs=-1)
print('Fitting random forest regressor model...')
rf_model.fit(train_df, train_labels)
```

Those initializations were the final parameters that I settled upon after running a GridSearch on the following parameters:

- Number of estimators: 10, 50, 100, 300, 600
- Maximum depth: 3, 5, 7, 9
- Maximum number of features: 3, 5, 7

The preprocessed training and test datasets that were fed into the Random Forest Regressor had over 1000 features and trained in around 5 minutes.

While this Random Forest Regressor easily surpassed the public leaderboard's baseline, it struggled to achieve an MAE score below 0.23. When compared with the hidden baseline value of 0.1988, this Random Forest Regressor's poor performance suggested that there was a fundamental error with either the preprocessing, the model structure, or choice of machine learning model.

In hindsight, the root cause of the RF regressor's poor performance is obvious: due to the "long tail" distribution of reviews in the training and test datasets, the regressor tried to reconcile data from fundamentally disparate distributions. This becomes clear from looking at a plot of the distribution of **outOf** values (which is a fair estimate of the actual target variable). By trying to fit parameters to the entirety of the dataset, the RF regressor most likely averaged out characteristics from the sparse end of the long tail with the dense end, thereby resulting in a worse MAE value.

At this stage, I had become aware of significant nonlinear characteristics within the datasets but pushed forward with the multilayer perceptron with the expectation that a deep neural network would be capable of resolving these granular intricacies.

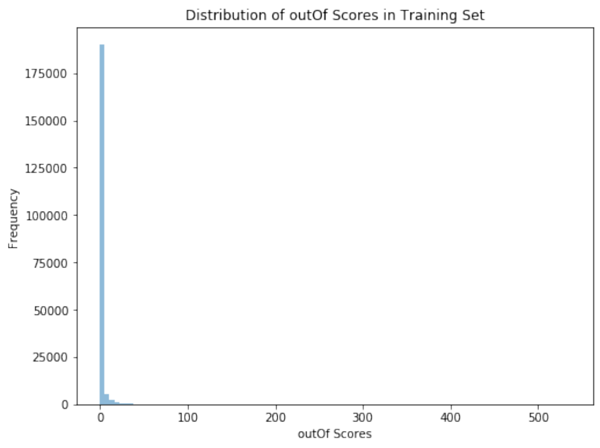


Figure 2: Distribution of Training Set outOf Values (bins=100)

3 Important Discoveries / Neural Networks

3.1 Multilayer Perceptron

Following my results using the RF regressor, I decided to utilize a deep learning model for the following reasons:

- I understood deep neural networks as structures capable of fitting nonlinear behaviors
- I wanted to develop familiarity with neural nets through a practical application

My multilayer perceptron model (coded using Keras) had the following structure:

```
 # Fully connected network (multilayer perceptron)
def multilayer(size):
    inputs = Input(size)
    # 1st hidden layer
    dense1 = Dense(2048, name='dense1')(inputs)
    dense2 = Dropout(0.2)(dense1)
    # 2nd hidden layer
    dense3 = Dense(512, name='dense3')(dense2)
    dense4 = Dropout(0.2)(dense3)
    # 3rd hidden layer
    dense5 = Dense(128, name='dense5')(dense4)
    dense6 = Dropout(0.2)(dense5)
    # Final layer
    densef = Dense(1, activation='linear', name='densef')(dense6)

    model = Model(inputs=inputs, outputs=densef)
    # Optimizer
    opt = SGD(lr=0.00001)
    model.compile(optimizer=opt, loss='mean_absolute_error', metrics=['accuracy'])
    return(model)
```

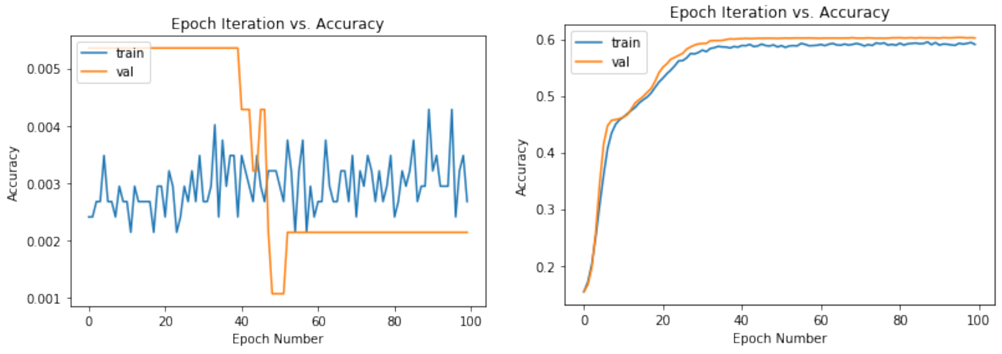
My network's first hidden layer had a size of 2048 to accommodate the number of features in my preprocessed train and test dataset. In an effort to prevent overfitting, I included dropout layers after each hidden layer. As a whole, this was a relatively simple deep neural network with 3 layers of depth, built-in dropout, and a stochastic gradient descent optimizer.

I ran this network model for a total of 150 epochs until I verified that the training and validation accuracies converged. When I evaluated the results of this model, I found that its performance was more or less equivalent with the given baseline value of 0.26. After seeing these results, I realized that I must've been overlooking defining characteristic(s) of the datasets that weren't being captured during preprocessing or within the data pipeline.

3.2 Split Multilayer Perceptron

I identified this overlooked characteristic as the **outOf** attribute while analyzing the training errors of my multilayer perceptron model. In my analysis, I found that there were two distinct clusters of misclassified reviews: those that corresponded with outOf scores less than 15 and greater than 15. It wasn't enough to account for the **outOf** feature during preprocessing - I had to integrate this separation into my data pipeline. The importance of this discovery was validated when I submitted the test set's **outOf** scores and obtained my best results yet - thereby stumbling upon the "hidden baseline" value.

Using this information, I decided to split my training and test datasets using the **outOf == 15** boundary. Each half would then be fed into 2 separate multilayer perceptron models. My hope in doing this was to manually segregate the differences between the above 15 and below 15 distributions and thereby enable my two networks to learn the idiosyncracies of both sets. This split multilayer perceptron was trained for a total of 100 epochs.



(a) Below 15 (b) Above 15

Figure 3: Epoch Iteration vs. Training and Validation Accuracy

However, despite implementing the dataset split, I was still unable to achieve better performance on the public leaderboard. Looking at the epoch iteration vs. accuracy plot for the two split datasets, it can be seen that the **below 15** set behaved much differently compared with the **above 15** set. While the **above 15** set's plot looks quite standard in terms of rising accuracy with epoch iteration, the **below 15** set's plot depicts an incredibly noisy training accuracy curve and an abnormal validation accuracy curve. The disparity between these two plots led me to hypothesize that there was a great deal of nonlinearity within the reviews that have **outOf** scores below 15 which meant that yet another strategy must be implemented.

4 Success / Gradient Boosting

At this point in the project, I had yet to make significant advancements on the public leaderboard. That being said, I had honed my understanding of the datasets and gleaned several insights that would ultimately yield my most successful model:

- A regression model for **outOf** values greater than 15 performs at a satisfactory level (refer to Figure 3b under the Split Multilayer Perception section)
- In my Split Multilayer Perceptron model, a significant contributor to my MAE score was the review entries that had **outOf** values less than 15
  - Out of this set, a majority of incorrect results came from the values [0, 1, 2, 3, 4, 5]
- It was difficult to assess the relationship between model performance and feature distribution when I had over 1000 preprocessed features
- Neural networks required specialized expertise to tune while I simply needed a robust machine learning model that could be optimized with transparency

With these lessons in-hand, I proceeded to overhaul my preprocessing and also created a multi-stage machine learning model based on Gradient Boosting which is known to be a stable ensemble machine learning framework that is relatively unsusceptible to overfitting.

4.1 Preprocessing Overhaul

My initial preprocessing included text processing and multi-hot encoding that contributed over 1000 features to the overall preprocessed dataset. While there was most likely useful information embedded into those 1000 features, the resultant dataset was unwieldy and slow to process - resulting in long training times and an inability to rapidly tune parameters. As such, after the failures encountered during the Neural Network phase, I proceeded to trim the number of features generated during preprocessing. This overhaul resulted in the features listed in Table 2. *(Note: None of the features listed in Table 2 required any additional preprocessing other than that detailed in the Initial Preprocessing section.)*

Attribute	**Number of Contributed Features**
categoryID	1
rating	1
outOf	1
reviewLength	1
reviewTime	3

Table 2: List of Features After Preprocessing Overhaul

All in all, **I selected only 5 original features which, after preprocessing, ended up being a total of 7 features**. This dramatic reduction in the feature space enabled my machine learning models to train much faster and allowed for efficient parameter tuning. I was finally ready to proceed with constructing an optimal data pipeline and model.

4.2 Multi-Stage Gradient Boosting

After my experience with random forest regressors and neural networks, I knew that I needed a model that could combine the simplicity of random forest with the flexibility and adaptiveness of neural networks. Such a specification required that I choose an ensemble model and I ultimately went with Gradient Boosting which neatly synthesizes the multiple-tree structure with an update method similar to gradient descent.

Even better, sklearn already had a library that wrapped both Gradient Boosting Classifier and Gradient Boosting Regressor into an easy-to-use interface.

Now, I needed to revamp my data pipeline such that it matched the nonlinear structures within the train and test dataset. Taking the lessons I had learned in the previous two stages, my optimal solution consisted of 3 separate, cascaded models:

1. Gradient Boosted Classifier (Cascade of 6 Binary Classifiers)
    - Binary Classifier for nHelpful == 0 and All Else
    - Binary Classifier for nHelpful == 1 and All Else
    - Binary Classifier for nHelpful == 2 and All Else
    - Binary Classifier for nHelpful == 3 and All Else
    - Binary Classifier for nHelpful == 4 and All Else
    - Binary Classifier for nHelpful == 5 and All Else
2. Gradient Boosted Regressor (for $5 < outOf \leq 15$)
3. Gradient Boosted Regressor (for $outOf > 15$)

My rationale for utilizing final Gradient Boosted Regressor is simple: it mirrors the strategy used in the Split Multilayer Perceptron which saw excellent results for **outOf** values greater than 15. The remaining 2 stages require a bit of explanation.

First, notice that the Gradient Boosted Classifier trains on the **nHelpful** attribute - equivalent to the train label values. On the other hand, the Gradient Boosted Regressor trains on the **outOf** attribute. This difference between the Classifier and the Regressor models is deliberate. While the **outOf** attribute has been demonstrated to be a reasonably accurate reflection of the training labels (and by extension, the unknown test labels), it is not 100% accurate. Furthermore, **I am assuming that the training and test sets share an underlying similarity in their feature distributions**. In addition, I have already found that the greatest occurrences of error are between the labels 0 through 5. Therefore, by cascading 6 unique binary classifiers, I was able to surgically target the most problematic and error-prone reviews in the training and test sets. Furthermore, the design choice of creating separate binary classifiers allowed me to tune each classifier's performance so to minimize the risk of overfitting.

With each successive binary classifier, the training set reduces in size. Only the "All Else" portion of each classifier is fed into the successive classifier. Naturally, misclassifications occur and by the time that the training set is passed onto the first Gradient Boosting Classifier, there is no longer any guarantees that a coherent structure exists in the **nHelpful** feature distribution.

Therefre, I make the design choice of switching to the **outOf** attribute for the Gradient Boosting Regressor instead. This decision was motivated by confidence in the preceding classifier cascade's ability to successfully identify the granular nonlinear defining characteristics of any label between 0 and 5. Assuming that this was accomplished, the **outOf** attribute could once again be used to artificially embed structure into the data pipeline.

5 Final Words and Takeaways

The Multi-Stage Gradient Boosting combination of cascaded classifiers and regressors worked. Upon implementation, I was able to immediately obtain MAE scores in the 0.19 range. With careful parameter tuning, I was able to further lower that to 0.17 and ultimately to 0.166 on the public leaderboard. When the private leaderboard scores were released, I found that my model was indeed able to generalize well - resulting in my final Kaggle score of 0.16703.

Through a process that spanned various configurations of 4 machine learning models, I was ultimately able to obtain a firm-enough understanding of the intricacies of this Amazon dataset to create a good generalized model. Of course, my 9th place standing proves that there is room for improvement. That being said, it's important to keep in mind that **I was able to obtain my result with a preprocessed dataset that only had 7 features**. With regards my data pipeline and machine learning model design, I am confident that I was approaching the upper limit in terms of utility gained from the selected features.

Any further improvements would therefore lie in the realm of preprocessing and data analysis. Additional possibilities to explore on the preprocessing front might involve reconsidering text data and potential methods for extracting useful information from the corpus without introducing a plethora of extraneous features. There may have also been correlations between the training and test sets that was unexplored and would've greatly improved my model's ability to generalize beyond the training set.

All in all, this project taught me valuable lessons in applied data science:

- Initial data analysis is absolutely critical and must be as exhaustive as possible
- Find every possible similarity between the training and test features to create a generalized model
- Random Forest is a good place to start data exploration but it's not a good idea to dwell on it for extended periods of time
- Neural networks are complicated structures that do not perform miracles
- Only utilize a feature if it has tangible benefits for model training
- Parameter tuning to validation sets is absolutely critical to obtain good generalization