Text Classification of Amazon Review Data

Robert Hall
University Of Tennessee at Chattanooga
Engineering, Math & Computer Science Department
615 McCallie Avenue
Chattanooga, TN 37403

CPSC4430: Introduction to Machine Learning
Dr. Yu Liang
Spring 2021

***Abstract*— Amazon Review data text used in processing for classification models. Using Logistic Regression and Random Forest Classifiers, models were created to predict the sentiment and category of the review. Tokenization of the text documents was performed to create word-count vectors to store the information to be used in the classification models. Models were fine-tuned using the hyperparameters present in the Sklearn packages to set the Logistic Regression model's best accuracy at 94%, and the Random Forest classifier model highest at 90%.**

Keywords— Text Classification, Amazon Reviews, Logistic Regression, Random Forests, Sentiment Prediction, Classifiers, Scikit, Sklearn

Motivation

The motivation behind this project is more along the lines of a guide on how to perform basic text classification. The classifications will be used is a simple linear regression analysis and a Random Forest classifier. Some information about the Random Forest model that will be used is based on the decision tree classifier, except instead of using just one decision tree to classify the text, an N-number of trees is used to have each tree vote to see which is the more likely classification for our text. More motivation behind this project is also to see how well fine-tuning the hyperparameters of our model changes the outcome of the metrics and accuracy. Upon working on the models, it was shown that changing the parameters changed accuracy anywhere from a fraction of a percentage to large accuracy differences and it made the project more interesting to work with. The Amazon data-set was selected due to how plentiful the selection was in the data set. The repository where the data was sourced from had the full 233.1 million reviews, but our experiment used the smaller subset to reduce time-complexity.

I. Introduction

Amazon is the world leader in online shopping consisting of billions in dollars in revenue, with their large presence in online shopping, the Amazon review system is used to recommend products, inform purchasers the quality of the items, and to create a brand following. In this article, the Amazon review data is used to create a model that will classify a given review document into one of the five selected categories. A Logistic Regression model and a Random Forest classifier will be created and trained from the processed review data to be able to correctly predict the respective category.

II. Sourcing the Data and processing

II.A Sources

The data used to the text classification was sourced from Jianmo Ni, Jiacheng Li, Julian McAuley's repository containing over 233.1 million reviews. However, for this implementation we are using the smaller sub-sets they provide. In this classification we are using reviews from the following categories:

- Electronics
- Grocery and Gourmet Food
- Home and Kitchen
- Pet Supplies
- Office Products

The review data was processed to filter out common word occurrences and select based on word count in the review such that the model can be trained with longer reviews. This need arises from reviews that are very short do not contain much information regarding our categories. Inside the model creation will be the use of "stop words" that are common to the English dictionary such as: if, and, but. These "stop words" are not used due to the reason of the word's frequency in other review documents as well. Because of the frequency in other documents, we are not including them in the model due to the words not representing or providing key clues for document classification origin.

II.B Initial Processing techniques

The data used in the model creation was read from the main review data set and filtered out based on our selected categories. Jupyter Notebooks was used for the initial data prepossessing using Python 3.8.1. The methods for subsetting the data consisted of reading the source .gzip file to retrieve the proper categories. Each review had its word count calculated and. if the review was at or above the thirty words, it was decided to include it in the model. However, reviews that fell under the thirty word-limit were not included in the model creation. Other instances of data-preparation methods included checking to see if the words inside the review were correctly formatted, ensuring they are lower case, and removing any special characters from the review to have a cleaned data set to work with. After the document processing has occurred, the review data was written to a .txt file that would be in the directory labeled the same as its category. This reason for having each review file in its certain folder is for the load_files function from the Scikit library allows us to treat the data in the directory as its own category and allows to use the data and have the target variables. One of the biggest challenges with this project was the amount of time to filter, clean and place the cleaned reviews into .txt files and into their respective categorical folders. An estimated time calculated would-be couple hours for the retrieval, cleaning, and placing in correct directory for each category used in the project.

III. Creating the Models

III.A Loading Required Data

After all the data has been pre-processed, the data is then loaded into a dictionary using the load_files function from the Scikit library. This function takes the listed directory from the parameter and treats each folder inside said directory as its own category. For our purpose it pulled all five of the categories and saved it in a variable. The data was also pickled and saved to a file so we could effectively save time by not having to read the data using the Scikit function each time. Using the created dictionary from the function, we initialize two variables that store the content of our actual data, and the target variables of the categorical classification of the review data. With both the review data and target classifications loaded we can then move onto the next section and use Tokenization to create meaningful values based on the word occurrences and word categories as well. Our selected method of Tokenization will be a count vectorizer that embeds each word as a value based on the frequency of it. The data is then split into a randomized 80 / 20 training and testing set, respectively. The training and testing set are used for both models.

III.B Logistic Regression Model

For the first model used to classify the data was the Logistic Regression model. Source code located on Section I. on included Jupyter Notebook for reference. From the previously tokenized review data we split them into an initial test-train split at 80% and 20$ respectively and set the random state parameter to 1000. Using the Sklearn package we can use the logistic regression model. With no prior modifications and just fitting our classifier with the tokenized training data our calculated accuracy with the Logistic Regression model is: 0.9376% or rounded to 94% accurate. The Sklearn package implementation of the Logistic Regression classifier gives us hyper-parameters to change to fine-tune our model to see if we can get a better accuracy. A hyper-parameter we can change is the solver parameter which can change the algorithm used to optimize the problem. The default one used in the implementation is lbfgs (Broyden-Fletcher-Goldfarb-Shanno). Using the other optimizer of “newton-cg” which is more used for multi-class problems and might be beneficial netted us a marginal increase of: 0.9382% at a change of 0.0005% setting it as the current best model.

Some further hyper-parameter tuning that was attempted was changing the max iterations that the solver will take before converging. Source code location on Section I.B on the included Jupyter Notebook for reference. An array of values was created with the contents: 100, 150, 200, 250, and 300 to be the maximum iterations for the model. The created array was looped through to have each index created a model with that set maximum iterations to see if increasing the iterations netted a better accuracy. Upon iterating through the created array and using each value as the maximum iteration for the model gave us values corresponding on the table in the figure below:

Maximum Iterations	Model Accuracy
150	0.9379
200	0.9382
250	0.9386
300	0.9384

It is shown that as larger iterations are chosen the accuracy does not keep going up, rather it flattens out around the 225 – 300 number of iterations. Knowing that the maximum iterations set at 250 gives us the best accuracy the model was also ran using the previous “newton-cg” optimizer. Our reported accuracy from setting the iterations to 250 and optimizer to Newton-CG was a 0.9382. At this point our best performing model is using the default optimizer with the iterations at 250. There are other parameters to tune, however. Another hyper-parameter the Logistic Regression package includes access to the “multi_class” parameter that changes the loss minimization, where the default is the "ovr". For our data-set it is multinominal we can see if changing the parameter will help the model fit our data-better. Section I D. shows the code to reproduce this accuracy. By changing over to default multi_class to the "multi-nominal" and another model changing it to "ovr", one can see in the chart below the various accuracy changes with our maximum iterations at 250:

Multi_class parameter	Model Accuracy
Auto	0.9386
Ovr	0.9433
Multinominal	0.9433

One can see from the chart above that at maximum iterations at 250 and using either "Ovr" or "Multinominal" gives us the best performing accuracy at 0.9433% or the same 94% accuracy at a difference of 0.0047%. From all the hyper-tuning of our Logistic Regression model it was declared at a maximum iterations of 250, using the "Multinominal" multi_class parameter to minimize the loss by fitting it across the entire distribution netted us our best performing model at: 0.9433 or 94% accurate.

III.C Random Forest Model

For the next modeling technique, we implemented a Random Forest classifier to fit the data and see if we can find any improvements with this classifier. Source code located on Section II. On included Jupyter Notebook for reference. Like how we handled the data using the logistic regression, we use the same 80% and 20% data-split to use in our created Random Forest classifier. The N-estimators for the model was set at the default of 100 and the random state to 0. Upon fitting our model with the training data, we predicted based on the test data and our model metrics came out to an accuracy of 0.8961% or rounded to 90% accurate. The Sklearn implementation of the Random Forest classifier gives us hyperparameters to fine-tune our model. The implementation lets us change the criterion from the default Gini-index to entropy to measure the split. Section II. A in the Jupyter Notebook shows the change from Gini-Index to Entropy. Changing the hyperparameter to use entropy as the criterion and keeping the other parameters the same, our model gives us metrics of 0.881 which is a minimal difference from the Gini-Index criteria at a 0.0151% difference.

One of the most important hyper-parameters for the Random Forest Model is the N-Estimators where the N is the number of trees the algorithm creates before calculating the votes for predictions. At the expense of larger number of trees, it does slow down computation, however. Sampling each N-Estimator at a time would be to time intensive, on Section II. B in the included Jupyter Notebook loops over a set of N-Estimators adding 20 to the parameter each time from the default of 100. Using this technique, we can see if there is a better N-Estimator for our Random Forest classifier to achieve a better accuracy for our model. Located below is the table to show the accuracy at each N-Estimator step:

N-Trees	Model Accuracy
100	0.8961
120	0.8982
140	0.8983
160	0.8989
180	0.9029
200	0.9018

Just like with the Logistic Regression model whenever we kept increasing the number of maximum iterations the accuracy seems to hit a wall and flatten out around the 180-200 range. Selecting the best N-Estimator number of trees for our model has been chosen as: 180. As one can see from the previous chart at the N-Estimators number of trees of: our model performs at an accuracy of 0.9029, making it our best performing model for the Random Forest classifier.

IV. Conclusion

After cleaning the data, filtering based on our selected five categories two models were created using Logistic Regression and Random Forest classifiers to correctly predict the category of new-review documents at an accuracy of 94.33% and 90.29% respectively. One of the most important information learned from this experiment is how important changing hyper-parameters of the models can be at changing the overall accuracy. In the Logistic Regression model, more specifically, changing which optimizer was used changed the accuracy anywhere from a minuscule 0.005% to lesser than the default optimization method. On the Random Forest classifier, it was beneficial to see that increasing the number of our trees increased the overall computational time and model accuracy to set our highest accuracy at: 0.9029.

Overall, the most time-consuming part of this project involved the data-filtering and creating the text documents in the correct directories. Numerous times in the project whenever the method to create the text files in the directory from the source file had to have files be deleted and take up computation space to remove them. If one wants to recreate this project the included Jupyter Notebook gives access to the same methods used to pull the data from the source gzip, create the required directories and run the models. It is noted that both model creation and data-filtering methods in the Notebook may take a different amount of time depending on your system's configuration and processing speed.

References

[1] Justifying recommendations using distantly-labeled reviews and fined-grained aspects, Jianmo Ni, Jiacheng Li, Julian McAuley, Empirical Methods in Natural Language Processing (EMNLP), 2019

[2] Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011.