Classifying and Categorizing App Store Reviews

Emma Gardner
University of Cincinnati
Advanced Software Engineering (CS 6030)
Dr. Yu Zhao
Due December 5, 2023

Classifying and Categorizing App Store Reviews

Gathering, analyzing, and implementing user feedback in software has five main advantages (Frank, 2022). First, it can improve the user experience of a product by allowing designers to learn what users like and dislike about the design of the current product. Second, the quality of the final product is increased because developers can implement the features desired by the users and remove or modify those that the users dislike. Third, it is important to know which features the users want because, if a product does not contain their key desired features, users will search for alternative products. Fourth, risk is reduced because time and money are not wasted on unneeded or undesired features. Lastly, agility is promoted by continually learning what users want and updating the product requirements to match.

For software applications, user feedback is readily available both to the public and to the company responsible for developing the applications. One example of this increased availability is the ratings and reviews section on the various app stores (e.g. Google Play Store, Apple App Store, Microsoft Store). Although ratings allow for an estimation of the general satisfaction of the users of an application, written reviews provide a forum for users to give specific feedback. These reviews have the potential to greatly improve the quality and popularity of an application. Unfortunately, manually sorting through reviews to find those that suggest modifications or new features is incredibly time-consuming. Developers rarely have free time in their schedules that they can spend sorting through hundreds, if not thousands, of application reviews.

The time-consuming aspect of manually analyzing app store reviews leads to the problem I set out to solve: automatically classifying app store reviews as actionable or unactionable and categorizing them based on their contents. Note that, in this project, an actionable review is considered a review that requests a specific feature, complains about a specific existing feature, or requests a particular modification of an existing feature. For example, “The lack of an undo feature in the text editor is frustrating.” would be considered an actionable review, because the review’s author is indirectly requesting the addition of an undo feature in the text editor. However, “I appreciate the daily motivation quotes that inspire me.” would be considered unactionable because this review simply mentions a feature of the application that the user likes; no suggestions for improvement can be inferred. Regarding the second part of the problem (categorizing reviews), I designated eleven categories: app, GUI, contents, pricing, feature/functionality, improvement, updates/versions, resources, security, model, and company. Examples and descriptions of each category are available in the README file of my project (https://github.com/egardner2019/App-Review-Classifier#readme).

To solve the aforementioned problem, my approach was to implement and evaluate four different classifiers and one categorizer using Node.js, which is a JavaScript runtime environment. The four classifiers implemented were the Naive Bayes algorithm, a long short-term memory neural network, a gated recurrent unit neural network, and the class-feature-centroid algorithm. The class-feature-centroid algorithm was also used for the categorization.

Before implementing the classifiers and categorizers, I generated training data with ChatGPT (OpenAI, 2023). I requested 1,000 unique actionable reviews and 1,000 unique unactionable reviews. Unfortunately, I realized later that duplicate reviews were provided. After removing these duplicate reviews, I was left with 681 actionable reviews and 621 unactionable reviews, for a total of 1,302 training reviews. After gathering this training data, I collected the testing data. To do so, I manually gathered 100 actionable and 100 unactionable reviews from various applications across the Google Play Store, Apple App Store, and Microsoft Store. After collecting all 1,502 of the training and testing reviews, I manually assigned them to categories.

After the necessary training and testing reviews were collected, classified, and categorized, I implemented the classifiers. I first implemented the Naive Bayes algorithm using the information in Jonathan Lee's paper titled "Notes on Naive Bayes Classifiers for Spam Filtering." I manually implemented this classifier; no external coding projects were included. Because the exact required mathematics are available in that paper, I will provide only a brief overview of the Naive Bayes algorithm. First, using the training data, 4 probabilities are computed: 1) each word in the actionable reviews appearing in an actionable review, 2) each word in the unactionable reviews appearing in an unactionable review, 3) a review being actionable, 4) a review being unactionable. Then, for each review in the testing data, those four probabilities are used to compute the probability of the review being actionable. If that probability is greater than 0.5, it is marked as actionable. Otherwise, it is marked as unactionable.

Next, I implemented the long short-term memory (LSTM) and gated recurrent unit (GRU) neural networks, which served as the second and third classifiers. The LSTM neural network is a type of recurrent neural network (RNN) that is better able to access contextual information than the original RNN design. It contains one input layer, a hidden layer with memory cells and corresponding gate units, and an output layer (Brownlee, 2017). The GRU neural network is a variation of the LSTM neural network that has gating units that contain the flow of information but does not have separate memory cells (Chung et al., 2014). To implement both of these neural networks, I used the brain.js npm (Node.js package manager) package, which is a GPU-accelerated library for neural networks (*Brain.js*, 2023). I used brain.js' corresponding constructor methods to create the neural networks, trained them on the training data for 500 iterations each (about 16 hours total), and ran the neural networks on the testing data.

After implementing the LSTM and GRU neural networks, I implemented the class-feature-centroid (CFC) algorithm using the cfc-classifier npm package (*Cfc-Classifier*, 2018). This package includes the implementation of a machine learning algorithm proposed in Hu Guan et al.'s research paper titled "A class-feature-centroid classifier for text categorization." As previously mentioned, I used the CFC algorithm for classification (actionable/unactionable) and also for categorization (e.g. app, GUI, contents). For both, I used the package's constructor method and passed in the necessary classified/categorized training data as the parameters. I then ran the algorithm on the testing data to get the classifications/categorizations.

For all four classifiers and the categorizer, I printed specific reviews to .txt files. For the classifiers, the following were printed to separate files: all reviews marked as actionable, all reviews marked as unactionable, all unactionable reviews incorrectly marked as actionable, and all actionable reviews incorrectly marked as unactionable. For the categorizer, all reviews were printed to a file by their marked categories. Additionally, all incorrectly classified reviews with their marked category and real categories were added to another file.

In addition to printing reviews to .txt files, evaluation metric values were printed to the console. These evaluation metrics are accuracy, precision, recall, and F-score. The following table (Table 1) contains the results of the classifiers in percentages. As is evident by Table 1, CFC resulted in the highest accuracy and F-score, while Naive Bayes had the highest precision and LSTM had the highest recall. The accuracy and precision of each classifier were fairly consistent, ranging from 56-66% and 54.29-67.27%, respectively. On the other hand, recall and F-score varied widely, ranging from 29-95% and 39.73-71.19%, respectively.

	Average of Classifiers	**Naive Bayes**	**CFC**	**GRU**	**LSTM**
Accuracy	59.75	59.5	66	56	57.5
Precision	61.59	67.27	61.76	63.04	54.29
Recall	61.25	37	84	29	95
F-score	59.94	47.74	71.19	39.73	69.09

Table 1: Classifier Results (%)

The values in the below tables (Table 2 and Table 3) are those of the categorizer, also represented in percentages. Note that the average of all categories is relatively high, however, the precision, recall, and F-score have a lot of room for improvement. In Table 2 and Table 3, N/A means that the metric could not be calculated due to a division by zero error. Interestingly, the Feature/Functionality and Security categories did not result in any true positives.

	Average of Categories	**App**	**GUI**	**Contents**	**Pricing**	**Feature/ Functionality**	**Model**
Accuracy	81.45	66.5	77	65	81.5	71	89.5
Precision	38.37	52.78	8.33	47.06	93.48	N/A	25
Recall	23.35	27.54	2.78	11.59	55.84	0	20
F-Score	31.42	36.19	4.17	18.6	69.92	N/A	22.22

Table 2: Categorizer Results Part 1 (%)

	Improvement	**Security**	**Company**	**Updates/ Versions**	**Resources**
Accuracy	82.5	93.5	94	87.5	88
Precision	55.56	0	44.44	37.04	20
Recall	13.89	0	36.36	55.56	33.33
F-Score	22.22	N/A	40	44.44	25

Table 3: Categorizer Results Part 2 (%)

The evaluation metric values from the classifiers and categorizer suggest possible improvements to this project. Collecting more training and testing data would likely ensure more true positives and allow all metrics to be calculated (no division by zero errors). Unfortunately, due to the time and computational resources available for this project, training the neural networks for more than 500 iterations was not feasible. However, it is logical to assume that training the neural networks for longer would result in increased metrics. Additionally, because app reviews are created by real people, having real app users (instead of ChatGPT) generate the training data would more accurately represent the real-world scenario in which the classifiers and the categorizer would be used. Another potential improvement would be to implement multiple categorizers and compare their metrics in the same way that was done for the classifiers. This would help to determine which method is best for the categorization of reviews. Finally, when classifying and categorizing training and testing reviews, it would be beneficial for multiple people to agree upon the labeled classification and categories. Other developers could disagree with my labeling, so having a stricter standard for classification and categorization could help ensure the legitimacy of the evaluation metric values.

References

brain.js. (2023, April 12). npm. https://www.npmjs.com/package/brain.js

Brownlee, J. (2017, July 19). *A Gentle Introduction to Long Short-Term Memory Networks by the Experts*. Machine Learning Mastery. https://machinelearningmastery.com/gentle-introduction-long-short-term-memory-networks-experts/

cfc-classifier. (2018, November 18). npm. https://www.npmjs.com/package/cfc-classifier

Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*. https://arxiv.org/pdf/1412.3555v1.pdf

Frank, J. (2022, May 31). *Council Post: Why User Feedback Is So Important For Software Development*. Forbes. https://www.forbes.com/sites/forbesbusinesscouncil/2022/05/31/why-user-feedback-is-so-important-for-software-development/

Guan, H., Zhou, J., & Guo, M. (2009). A class-feature-centroid classifier for text categorization. *WWW '09 Proceedings of the 18th International Conference on World Wide Web*. https://doi.org/10.1145/1526709.1526737

Lee, J. (2018). *Notes on Naive Bayes Classifiers for Spam Filtering*. https://courses.cs.washington.edu/courses/cse312/18sp/lectures/naive-bayes/naivebayesnotes.pdf

OpenAI. (2023). *ChatGPT*. Chat.openai.com; OpenAI. https://chat.openai.com/