Robert Ervin
97170118
CIS 481 Computational Learning: Theory and Practice Winter 2016

App Store Review Analyzer

Throughout the world, there are millions of app developers, and even more millions of apps on mobile devices. Each of these apps is in a centralized system (the iOS App Store or Google Play). This centralization has proven to be crucial for both users discovering apps and for developers having a standard way to distribute their creations to the technologically-connected population. However, due to ease of distribution, these systems also have such a large number of apps that it makes it difficult for a standard user to discover apps which are not downloaded frequently. The app stores try to combat this with a ranking system, which takes into account the number of reviews users are allowed to leave on an app, the distribution of ratings (users can leave a 1, 2, 3, 4, or 5 star rating), and the velocity at which these are accrued. Therefore, it is advantageous for a developer to attempt to obtain a large percentage of high-rated reviews to a low percentage of low-rated reviews. In this paper, we will attempt to analyze why users leave low-rated reviews, and if we can do anything as developers to mitigate them.

Data Collection

Obtaining the reviews in the first place proved to be a bit tricky. Apple has methods in place to ensure people don't have easy access to the reviews since that will allow third parties to attempt to reverse-engineer their ranking algorithm. If the algorithm is successfully reverse-engineered, then those who have access to that information can potentially game the system. This would likely result in poor-quality apps ranked highly. Apple does not want this to happen. In order to get around this limitation, I was able to find an exposed REST API endpoint that returned JSON data for the top 500 reviews for a particular app which were labeled as *most helpful*. I then hand-selected 18 apps in the shopping category, and scraped the top 500 reviews for each. I decided it would be most beneficial to the algorithm if all reviews were relatively similar since I suspected the variance of reviews would be quite high at a baseline.

Methodology

If we want to analyze why users leave 1-star reviews, we first want group reviews by the type of review it is. For example, one user type may post a bubbly review that is upbeat and uses simple language. Another user may be a bit more thoughtful and take time to really analyze the design decisions behind the app, and hence will likely use more complex language. Going into this, I thought there may be anywhere from 10-20 different types of users in the shopping category from the apps selected. Once the reviews are grouped together, or clustered, we can analyze the words used in the groups that have a lot of low-rated reviews.

Algorithms

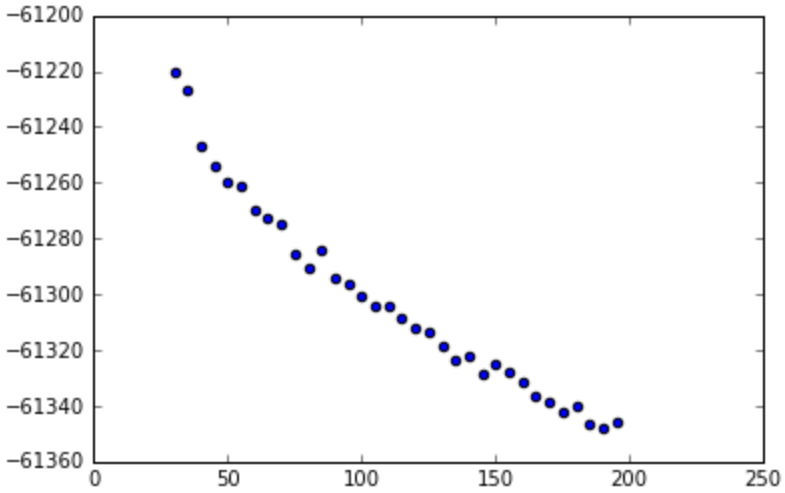
Given a review, we want to break it apart into the individual words used, and given a weight to each word. The weight represents the impact it has compared to other words. For example, a word such as *broken* should be considered more useful than *app*. To extract this data, I used the Term Frequency - Inverse Document Frequency (TF-IDF) algorithm. This algorithm can be shown simply as follows: *weight(term) = tf(term) * idf(term)*. Term Frequency (tf) is the number of times the term occurs in a given review. The Inverse Document Frequency (idf) weights terms that occur in more reviews lower than words that occur in fewer reviews. It is easy to think of idf as something that gives words which are more rare a higher weight. To group the reviews, I used a standard K-Means clustering algorithm. I chose this because it exposed all the necessary parameters I needed, and was a black box that I could trust to run correctly. Since the feature space was as large as the total number of terms, and I also wanted to graph the results visually, I used Principal Component Analysis (PCA) to select the 2 most important directions in the feature space in order to display the clusters in 2-D. It should also be noted that the TF-IDF implementation I used ignored all *stop words* (e.g. it, him, her, the), as well as words that either occurred more than 80% of the time or less than 2% of the time. This limitation was good since it limited the feature space, allowing for more focused clusters.

Technology

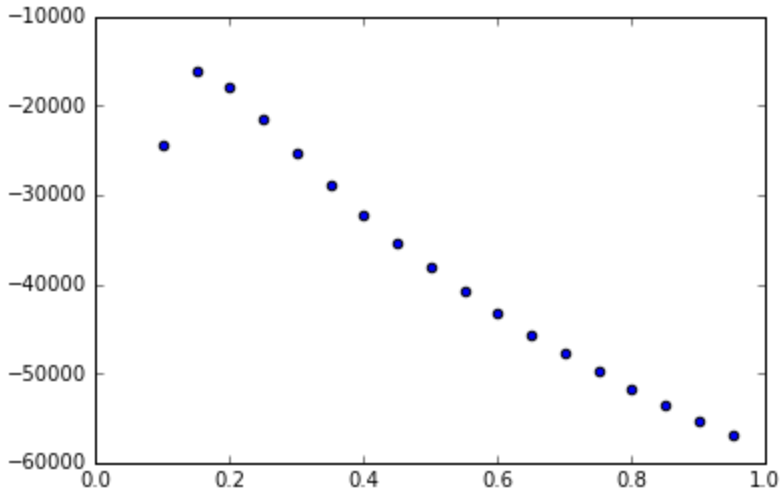
Python was the selected technology for this project. It was suitable for the needs because it came with libraries that contained all of the algorithms I was going to use. I have also been working in it professionally for 2 years, and am experienced with it. I used a number of popular libraries such as *numpy*, *scikit-learn*, and *matplotlib*.

Implementation

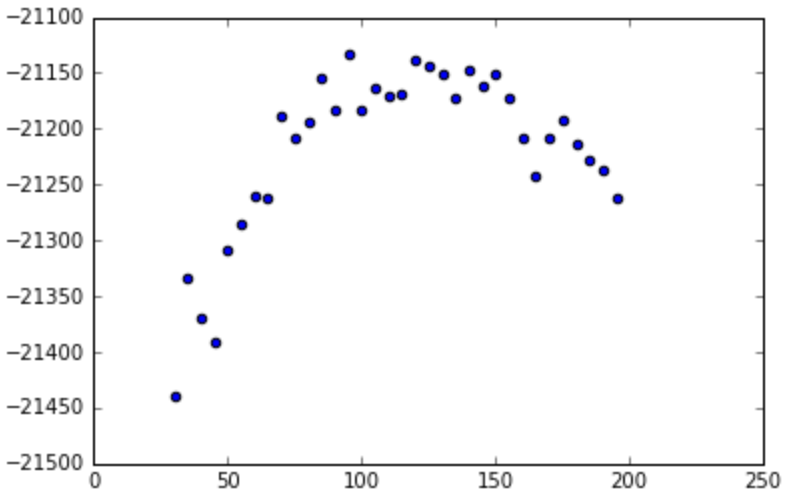
As with most machine learning implementations, there are some parameters that must be tuned for. In the case of K-Means, that parameter is the number of clusters (k) that will be fitted to the data. In order to optimize k, I maximized the log-likelihood over the hold-out set for each cluster center (centroid). The hold out set was set as 20% of the total training data. When maximized, the k that the maximum occurred at is the optimal number of clusters to use for that dataset. Following is a graph for the results of this implementation.



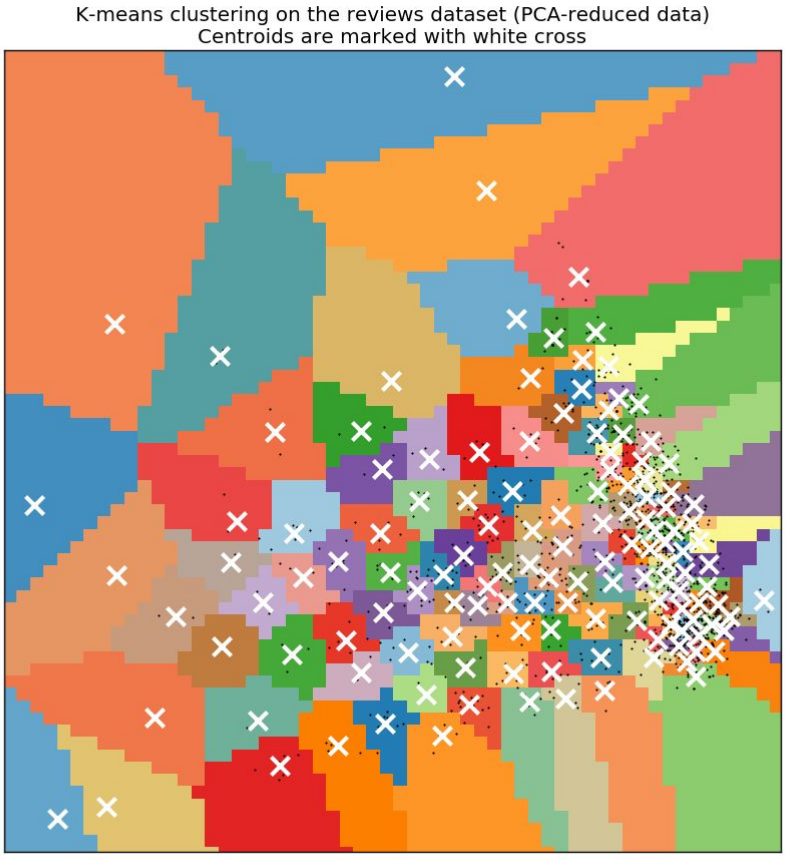
As you can see, the log-likelihood is continually decreasing. This should not happen. After experimenting a bit, it was discovered that one of the main assumptions that the k-means algorithm makes was wrong. This assumptions assumes that the variance of the data is always equal to 1. With this new knowledge, I now had to optimize for sigma as well. It should be noted that sigma is simply the square root of the variance. I optimized for sigma by fixing the number of clusters at 20, and maximizing the log likelihoods over possible sigmas. The results of that are the following:



From this, it is clear that the optimal sigma is around 0.15. With this new information, I took this into account when computing the log likelihoods for K-Means. The results are as follows:



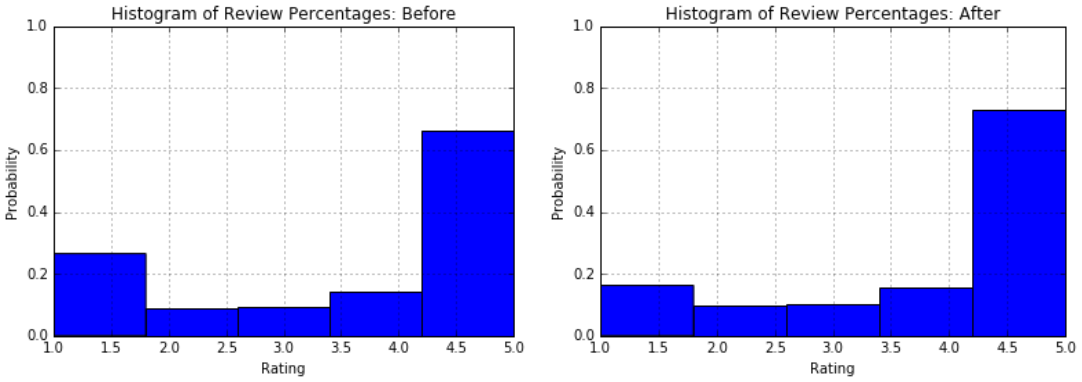
Jackpot! The optimal number of clusters for this dataset is around 135. I then ran the dataset through PCA to select the best two directions, and ran that through K-Means. The graphed results are as follows. Note each white *X* is a centroid, and each colored box is a cluster.



These results are a bit surprising. If the data were perfectly separable, we should see an “elbow” in the graph, where each centroid lies along either the vertical or horizontal axis. In this dataset, there is a smattering of points in the middle. This is not entirely good, and could use some further investigation.

Analysis

Now that the data is grouped into clusters, it is possible to easily find those clusters which represent low-rated reviews. I noted that 131/135 clusters contained at least one 1-star review. This was a bit surprising since one would expect those reviews to be in a small percentage of total clusters. In light of this, I decided that any cluster that contained at least 2% of the total amount of 1-star reviews (or about 10 1-star reviews), was to be considered a low-rated cluster. Once these clusters were found, it was easy to find the terms closest to the center of the cluster. These terms should represent the cluster the best. Some of these top terms were very insightful. These terms included *fix, broken, update, needs, crashes, slow, tried,* and *buggy.* These can be classified into two groups: *unintended* and *intended*. The *unintended* words were those which can be thought of as the cause of an app experience which was designed to work correctly, but didn’t. These include *fix, broken, update, crashes, slow,* and *buggy*. The *intended* words can be thought of as design-decisions made by the creators which may have resonated negatively with the user. These include *needs,* and *tried*. When I selected all reviews (not just the 500 per rating in the training set), and removed all those reviews which both were 1-star and contained a *unintended* word, 45% of all 1-star reviews were deleted. From this we can conclude that if there is an app that is not shipped with unintended features (e.g. bugs, glitches, etc), then that app will receive about 45% less 1-star reviews than if they had shipped with those unintended features. The following histograms show the gain in ratings-ratios before and after these reviews were removed.



Next Steps

Development teams are always in a push to ship, so this must be taken into account when implementing a positive change such as mitigating 1-star reviews. In order to balance the speed of shipping with quality of delivery, a new recommended approach would be to focus the scope of a development sprint to a very small, almost rudimentary set of features. The

development team should then write unit and integration tests, and send the build to QA and UAT for them to test. In the time it takes QA and UAT to either pass or fail the build, the developers can work on fixing old bugs in order to support a greater percentage of their users who are currently using a version which contains bugs on their device. At the end of the day, this will result in significantly less low-rated reviews, which will result in a higher app ranking, which will result in more organic downloads. And organic downloads, ladies and gentlemen, is what it’s all about.