AN ABSTRACT OF THE THESIS OF

Kendall Bailey for the degree of Master of Science in Computer Science presented on December 8, 2015.

Title: Out of the Mouths of Users: Examining User-Developer Feedback Loops Facilitated by App Stores

Abstract approved: ______________________________

Danny Dig

Application Stores, such as the iTunes App Store, give developers access to their users' complaints and requests in the form of application reviews. However, little is known about how developers are responding to application reviews. Without such knowledge developers, users, Application Stores, and researchers could make incorrect assumptions. To address this knowledge gap, in this study we focus on **feedback loops**, which are instances of application reviews where developers respond to a user concern.

To conduct this study we use both supervised and unsupervised methods to automatically analyze a corpus of 1752 different applications from the iTunes App Store consisting of 30,875 release notes and 806,209 application reviews. Our research examines the software changes propagated by developer-user interaction using Support Vector Machine classifiers and a semantic relatedness algorithm based upon release notes and user provided reviews. We found that 18.7% of the applications in our corpus contain instances of feedback loops. In these feedback loops we observed interesting behaviors. For example, (i) feedback loops with feature requests and login issues were twice as likely as general bugs to be fixed by developers, (ii) users who reviewed with an even tone were most likely to have their concerns addressed, and (iii) the proportion of reported bugs addressed by developers was static across all star ratings.

Out of the Mouths of Users: Examining User-Developer Feedback Loops Facilitated by App Stores

by

Kendall Bailey

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented December 8, 2015
Commencement June 2016

Master of Science thesis of Kendall Bailey presented on December 8, 2015.

APPROVED:

Major Professor, representing Computer Science

Bella Bose, Director of the School of Electrical Engineering and Computer Science

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Kendall Bailey, Author

ACKNOWLEDGEMENTS

I would like to thank my advisor, Danny Dig, for his mentorship during my graduate career and support of this research.

I also express my gratitude to Mei Nagappan for his expert guidance through the mobile software engineering.

I extend thanks to Michael Hilton, Mihai Codoban, Beatrice Moissinac, Sruti Srinivasa Ragavan, and Nicholas Nelson for their advice and perspective over the course of this research.

The annotation of both release notes and application reviews was necessary to conduct this research. A special thank you goes to Hongyan Yi, who annotated the release notes with me, and helped spark this line of research. I am also grateful to Sam Falconer and Asma Sattar for annotating the application reviews.

I thank my committee members Mike Bailey and Scott Sanner for providing constructive feedback on this thesis. I would also like to thank Nicole Thompson for her patience and guidance through thesis defense process.

Thanks also goes to Daniele Maddaluno for providing the data dumps needed for Wikipedia Miner.

This research was partially funded through a National Science Foundation grant CCF-1439957.

Finally, I would like to thank my family for their unwavering support both during graduate school and my life.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF TABLES

Chapter 1: Introduction

The methods used to sell software over the last decade changed dramatically. Software distribution platforms (such as Application Stores) toppled brick and mortar stores as the default means of acquiring software. Among software distribution platforms, mobile Application Stores host applications developed by a wide variety of companies. As of 2015 the iTunes App Store surpassed 100 billion application downloads [1] and the Google Play Store exceeded 175 billion downloads [2].

The services provided by Application Stores help lead to their success. Application Stores provide developers a location to host and manage their applications while the Application Store handles payments. For users, Application Stores offer a large selections of applications in one location presented in a ranked order with application reviews to guide users when making a purchasing decision. Acting as an third-party, Application Stores moderate both the applications and application reviews provided to ensure ethical treatment of both developers and users.

Not only is using an Application Store more convenient for both users and developers, but it has also affected the way developers interact with their users. Third-party moderation of reviews by the Application Stores allows all users a glimpse behind the curtain of software maintenance. Before, anonymous feedback forms and error reports cloaked the maintenance portion of the software development life. Users re-purpose Application Store reviews to provide developers valuable information about bugs and future requirements [3–5] in the free text portion of their reviews. Users can attempt to “coax” reactions from developers through their rating (such as a star rating) of their reviews. Using the rating as weapon provides developers with incentive to address the users’ needs as ratings can affect the ranking of an application in the Application Store and ultimately the developers revenue.

The user-developer relationship facilitated through Application Stores is not always adversarial. Users also provide developers information on what features to implement and prioritize. According to Vision Mobile’s State of the Developer Nation [6], 54% of application developers rely on advertisements and subscription models. These financial models require users to use the application over long periods of time to generate profits. This incentivizes developers to listen to feedback to both growth and maintain their user base. But are developers listening? Current research [3, 7–9] aims to help developers identify actionable information from application reviews, but we know little about how developers are currently responding to users. Ignoring developer's focus could lead to research and the development of tools that are ill-suited to developers' needs.

In this study we focus on **feedback loops**, which are instances of application reviews where developers respond to a user concern. We determine feedback loops using release notes and user reviews provided on the iTunes App Store [10]. We identify feedback loops by searching for instances where users gave their feedback in the form of users reviews addressed in the release notes of an application's subsequent release. To conduct this study we analyzed a corpus of 1752 applications from the iTunes App Store consisting of 30,875 release notes and 806,209 user reviews First, we need to label the specific statements that identify feature request and bug reports from application reviews and release notes. To do this we use Support Vector Machine [11] classifiers. Second, we need to determine whether the concerns expressed in the application reviews are related to the ones described in release notes. Thus, we need to determine whether two sentences talk about the same thing, even when they are using a rich vocabulary. To determine relatedness we use Wikipedia Miner [12]. Additionally, we explore the sentiment users express in application reviews to illuminate strategies users follow to elicit responses from developers. We use SentiStrength [13] to measure the sentiment expressed by users.

Using our corpus and the automated data analysis, we answer the following research questions:

RQ1: *Are developers responding to feedback provided in user reviews?* We found feedback loops present in 331 applications which represents 18.7% of the eligible applications in our corpus. (Section 4.1)

RQ2: *What are the characteristics of these feedback loops?* Among the feedback loops, Log-in Issues, Feature Requests, and Crashing information provided in application reviews were resolved by developers more often than general reports of bugs. When analyzing the categories of the feedback loops, we discovered developers of music and social networking applications responded to the most user feedback. We found that the economic model employed by the developer did not influence the likelihood of responding to user feedback. The largest amount of feedback is addressed by developers within a quarter of being reported. Developers responded to a significant amount of bugs and crashes within a week of being reported, particularly in the games category.(Section 4.2)

RQ3: *What do users do that enables a feedback loop?* Through analysis of user ratings, we found Crashes and Log-in Issues elicited developer response in lower star ratings while Feature Requests comprised most of the developer response in highly rated reviews. Interestingly, the proportion of bug fixes addressed was similar across both high and low star ratings. We also noted instances where users attempted to "coax" developers for responses by withholding star ratings and claiming that they will give a higher rating when the developer addresses their concerns. In all instances, none of the reviewers fulfilled their promise. An exploration of sentiment revealed a majority of the feedback addressed by developers expressed using a neutral tone. (Section 4.3)

Our study has implications for researchers, developers, users, and Application Stores. *Researchers* can better align their questions with the needs of developers and can focus on simplifying communication barriers between users and developers. *Developers* can discover where to focus their energy to minimize lower ratings each release. *Users* can become aware of the steps they need to increase the chances of the developers addressing their feedback. *Application stores* can further construct their platforms to help developers and users communicate.

This paper makes the following contributions:

1. **Research Questions.** We designed and answered three novel research questions to understand the extent to which developers respond to feedback in Application Stores.

2. **Mining Repositories.** We developed tools and metrics to identify feedback loops in Application Stores. We applied our tools on 1752 mobile applications.

3. **Implications.** We present our findings from the perspectives of researchers, developers, users, and Application Stores.

Chapter 2: Background

In order to be able to process our large corpus, comprising of 806,209 application reviews and 30,875 release notes from 1752 mobile applications, we use several machine learning techniques from two categories: supervised and unsupervised.

2.1 Unsupervised Techniques

SentiStrength

In RQ3 we look at the characteristics of feedback loops and one of the metrics we use is the sentiment expressed in the user review. Sentiment analysis, also called opinion mining, quantifies the emotion conveyed by the author in texts. Often, sentiment analysis is a binary (positive or negative) or trinary (positive, neutral, negative) classification [13].

For our research we elected to use SentiStrength [13], a lexicon based tool that calculates the sentiment expressed in short texts. SentiStrength performed well when tested on online social networks [14] and has been used to detect sentiment for similarly informal texts such as twitter, Yahoo! answers, and product reviews from Amazon [15–17].

SentiStrength operates as a single-scale classifier and simultaneously calculates both positive and negative sentiment in texts. When given a sentence, SentiStrength returns two values on a scale of 1 to 5 where 1 is weakly expressed and 5 is strongly expressed emotion. The first value returned measures the level of positive emotion expressed and the second measures level of negative emotion expressed in the given sentence. We simplify these values to a single score by subtracting the score for negative emotion from positive emotion to get the net sentiment of the sentence. The net sentiment of a sentence can have a value from 4 to -4 where 4 is the most positive sentiment and -4 is the most negative sentiment. SentiStrength addresses several features that commonly appear in informal texts, such as application reviews, like stronger emotion denoted by repeated letter use and repeated punctuation.

The following three sentences are examples from our corpus. The net sentiment from

SentiStrength is shown in parenthesis at the end each sentence.

- “I looooooooooove this app!!!!!!!” (4)
- “this app is a blessing and a curse.” (0)
- “Another ridiculous update without new levels.” (-2)

Wikipedia Miner

We use a semantic relatedness tool to identify release notes and application reviews that compose a feedback loop. Semantic relatedness quantifies the multitude of possible relationships between concepts such as “has-a”, “is-a”, “is-part-of”, or ”is-close-to” within a taxonomic scheme. To calculate the semantic relatedness we use Wikipedia Miner [12], which uses the link structure of Wikipedia to determine the relationships between concepts. Wikipedia Miner represents the relatedness between two concepts as a value from 0 to 1, with 1 being the most related. In addition to a relatedness score, Wikipedia Miner returns a confidence value for the relatedness score.

The confidence value exists because Wikipedia Miner must disambiguate the given topics. Consider the concept “papyrus”; papyrus a type of paper, but it is also a typeface, the name of a comic book series, the name of a UML2 tool, and the name of a race horse. As Wikipedia is a large, open-source encyclopedia, disambiguation is present when comparing most concepts. Like semantic relatedness, the confidence score is also a value from 0 to 1, with 1 being full confidence. Occasionally, Wikipedia Miner would not return a confidence value. We understood this to be a vote of no confidence and returned a value of 0.

Our installment of Wikipedia Miner was built on the 7/22/2011 dump of Wikipedia.

Alternatives to Wikipedia Miner we considered included Explicit Semantic Analysis(ESA) by Gabrilovich and Markovitch [18] and a search-engine relatedness measure [19]. ESA generates a relatedness vector of two texts based on entire Wikipedia articles and generally correlated more closely with human understanding. ESA’s use of the entire article increase the cost of the algorithm, whereas Wikipedia Miner offered slightly worse results but cut out expense by only using the link structure [20]. The web-based measure of semantic relatedness also correlated more closely with human judgment than Wikipedia Miner, but only using certain search engines [19]. Since the publishing of the article, Yahoo, the search engine with a higher correlation, stopped reporting the total number of search results needed in the calculation [21]. Wikipedia Miner bases the equation to determine relatedness between articles on the Normalized Web Distance proposed in [19], thus making it the best compromise.

2.2 Supervised Techniques

Support Vector Machines

We use support vector machines (SVMs) [11] to classify release notes and application reviews. An SVM uses a supervised learning model and is a binary classifier that several other researchers [22–24] found suitable for classifying texts. To perform classifications the SVM assumes the vectors are linearly separable by class and calculates hyperplanes to divide the classes. Each feature is a dimension considered while calculating the hyperplane that creates a very high dimensional problem for which SVMs are robust enough to handle. We use LIBLINEAR, a library developed by Fan et al. [25].

In this research we use two separate classifiers; one for the release notes and one for the application reviews. Two classifiers are necessary due to difference in language in the statements and classes used to label the two sets of data. To classify the sentences (in this section referred to as documents), we represent each document as a vector of words (in this section called features). We also weight each feature according to its significance within the data set(Section 3.4.1).

Our implementation of LIBLINEAR does not use multiclass classification, meaning each document is only allowed to belong to a single class [25]. Due to this restriction, we chose to work at sentence level granularity for both release notes and reviews because both application reviews and release notes are not restricted to atomic topics. For example, the release notes of a single version of an application can have a section for bug fixes and a section for new features, making assignment of the entire release notes to a single label impractical. By breaking the release notes into sentences we classify the sentences in the bug fixes section as bugs and in the new features section as features.

The following are examples from our corpus with the classification listed in parenthesis at the end of each sentence:

- Application reviews

  "It gets halfway through opening, hangs for a second and closes." (Crashing)

“Plz make a very easy tutorial for the app!!” (Feature Request)

“Please fix the broken badge notifications.” (Bugfix)

- Release Notes

  “- Songs can now be played in a choice of 7 keys” (Feature)

  “ Fixes for a number of cases where ’Save As’ could fail” (Bug)

Chapter 3: Experimental Setup

3.1 Feedback Loops

We define a **feedback loop** as an instance where a user review describes a short-coming or a request and at a later date the release notes of the mobile application addressed the short-coming or request. Feedback loops are only found within the user reviews and release notes of a single application.

For example, a user, Alice, installs an application on her phone. After a new update, Alice is unable to log in to her application. Frustrated by her experience, Alice turns to the iTunes App Store and posts a review for the application stating, ”This app always comes up with a login error disclaimer”, and gives the application 3 stars. Alice posting a review that identifies a bug potentially begins a feedback loop.

Bob, the developer of the application Alice downloaded, checks the iTunes App Store and reads Alice’s review. Bob, now aware of the bug when logging in, fixes the problem. In the next release, Bob includes ”Login issue fixes” in the release notes posted on the iTunes App Store. Bob’s response to Alice’s concern completes a feedback loop.

Alice and other users download the new update and enjoy a better experience with their application. Bob should no longer receive low rated application reviews for the fixed login issues. The iTunes App Store can generate more profit from more downloads of the now higher quality software. Completing feedback loops creates a win-win-win situation for all parties involved.

3.2 Corpus

We divided our corpus into three main parts, application metadata, release data, and review data. Within our corpus the submission dates of release notes range from July 7, 2008 to February 2, 2015 and the application reviews range from July 10, 2008 to January 28, 2015. A more detailed explanation of the information stored in the corpus can be found in Appendix A.

3.2.1 Data Collection

This section details the methodology for obtaining the data in our corpus.

3.2.1.1 Application Metadata and Release Notes

We collected the release notes for the iOS applications in batches of 999 applications per day from the iTunes App Store. We chose to limit the number of applications downloaded to prevent being locked out of the iTunes App Store. We found the iTunes App Store preferable to the Google Play Store [26] in this research because the iTunes App Store saves the release notes for up to the previous 25 releases while the Google Play Store only displays the most up-to-date release. The algorithm works in five steps.

Step 1: The algorithm generates a list of application ids that will be downloaded for the day. In this list we store:

1. The id number of the application on the iTunes App Store
2. The total number of reviews the application received last time we collected its information
3. The date we last collected the application's information
4. The number of new reviews submitted since the last time we collected the application's information

The algorithm chooses the top 999 id numbers with the earliest date previously downloaded and generates a daily id file.

Step 2: The collection algorithm then retrieves all the information from the iTunes App Store using the id numbers provided in the daily id file.

Step 3: After collecting the daily application information, the algorithm stores the application metadata in the database. The algorithm then extracts the release notes from each of the daily application and compares them with the existing release notes file. We add any new release notes to the data base and ignore release notes previously collected.

Step 4: This is the "growing" stage. The iTunes App Store often recommends additional applications to customers on each page; the recommended applications are one or more of the following categories:

1. The top iPhone applications
2. The top iPad applications
3. Other applications by the same developer
4. Applications that customers who bought this application also bought

Our algorithm collects all of the recommend application ids and queues them to the id bank with other applications the algorithm not yet scraped.
Step 5: The final step of the algorithm updates the complete list of ids by changing the last read date and the number of reviews. Our mining algorithm was originally seeded with the top 50 iOS applications as determined by Distimo [1] on December 30, 2014.

3.2.1.2 Application reviews

The iTunes App Store does not store a significant number of reviews on each page accessed by the release notes scraper so we deemed it necessary to create a second scraper to accomplish this. Using the number of new reviews as determined by the release notes algorithm, we use a modified version of Kent Bye's script [2] which scrapes a number of reviews for a single application from the iTunes App Store.

The iTunes App Store stores reviews on separate pages, with each page containing 10 reviews. Our scraper collects a maximum backlog of 100 pages of reviews per application each run, up to 1000 reviews at one time. Each day the review mining algorithm runs until it reaches 14500, a limit implemented to prevent the iTunes App Store from blocking requests, or until the request times out five times. As a result, reviews are a limiting factor in the size of our corpus. Presently, we have collected reviews for 1752 applications for a total of 806,209 application reviews.

This section details the methodology for annotating release notes and application reviews for training and test data for the SVM classifiers.

[1] http://www.distimo.com (now part of App Annie)
[2] https://gist.github.com/kentbye/3740357

3.3 Data Annotation

Training and test data for the SVM classifiers requires the creation of human annotated sets of data. We followed a separate procedure for release notes and application reviews due to differences in language and purpose.

3.3.0.1 Release Notes

The author and a graduate student in Computer Science completed the classification of a sample of release note statements using qualitative thematic coding [27] to build the data for training the classifier.

We took a random sample across a corpus consisting of release statements from 29 mobile applications and classified the same data in isolation to compare for agreement. The classifications were validated by reaching 79% agreement on 20% of the sample corpus using the Jaccard Coefficient [28]. The sample corpus contains 3667 release note statements.

We identified each document as belonging to one of the four classes:

Bugs: This class describes statements in release notes that correct flaws within the application. An example of a statement belonging in the bugs category is “Fixed a crash when closing documents while there was an active text insertion point.”

Enhacements: This class describes statements in release notes that build on previously introduced features or improve application performance, but do not correct overt flaws like those in the bugs category. An example of a statement belonging in the enhancement category is “Improved the performance of the Stroke and Fill layer styles.”

Features: This class describes statements in release notes that introduce new functionality or properties to an application. An example of a statement belonging in the features category is “Added preference setting for prompting to save session information on close.”

Miscellaneous: This class describes text we do not analyze that contains titles, headers, punctuation, signatures, or other organizational type text that are not considered in our analysis. Statements such as “thank you to all our fans!” also fall into the miscellaneous category.

3.3.0.2 Reviews

We divided user reviews into more diverse classes than release notes as the text has more varied purposes, such as admonition, praise, or requests, compared to release notes which generally only inform the readers of software changes.

We began with the set of twelve classes from Khalid et al. [4] and then adjusted the classes as we saw trends in the data to a total of 16 classes. The labels used and their description are found in Table 3.1.

When labeling the reviews we treat them similarly to unstructured interviews as application reviews are more open-ended than release notes. Instead of using inter-annotator agreement, we trained two computer science students on randomly selected reviews from the corpus and allowed them to label the reviews in isolation. In the event of disagreement, the labels were resolved by the annotators and the author using the negotiated agreement technique [29]. The author moderated the negotiation process to ensure that one coder was not incorrectly disproportionately deferred to, one of the threats recognized in [29]. The coders reached an agreement of 97% through negotiation over 6 iterations from an average inter-rater agreement using the Jaccard Coefficient [28] of 62% agreement.

The login class is one of the labels we added to our original list through the negotiation process. Amongst the data it was often hard to distinguish if a login problem was the result of network issues or just a general implementation bug. Login issues appeared so frequently in our training set of reviews that we decided to create a separate class for it rather than guess what was most appropriate in each instance.

In our corpus, we found no reviews concerned about privacy, but six concerned about business ethics or cheating. We feel the lack of privacy concerns in our data compared to [4] may be the result of the Khalid et al. focusing on more reviews of a smaller number of applications.

We adopted the "user is always right" philosophy when assigning a class to eliminate bias stemming from the labelers tech background. For example, one user complained:

> Horrible. I got the game and it was fun but when I waited too lomg[sic] the internet played for me and I lost the game. That is so stupid. Everybody in the game waited for some time but that did not happpen[sic] to them. Needs to be fixed.

Category	Label	Description
Fixes	Bugfix	A general category for errors that do not fit in the more specific types of bugs listed below.
	Response Time	Mentions of slowness within an application. This does not include complaints about network problems.
	Resource Heavy	Mentions of battery, energy, or memory consumption.
	Crashing	Mentions of crashing or freezing within an application.
	Network Issues	Mentions of connections or slowness of a network. This include problems with multi-player features on game applications.
	User Interface	Mentions of UI.
	Log-in Issue	Mentions of problems with login.
Fixes & Requests	Update Related	Mentions of problems due to updates, requests for updates, transfer problems, lost data after update.
Requests	Feature Request	Mentions of things that developers should add to applications.
	Crowdsource Q & A	Requests seeking to get answers for how to do things or to meet other people interested in the same things on reviews.
General Information	Content Related	Mentions of things related to what the application has or does that does not fall in the other categories.
	Not Specific	Statements that don't tell much about the application. Often very short.
Economic Models	Ad Removal	Complaints about the presence of ads and/or their interference with the functioning of the application.
	Additional Cost	Complaints about having to pay for things in the application.
	Use Limit	Complaints about limitations on use, such as can play for free 3 times every hour, etc.
Ideological	Privacy and Ethical Issues	Mentions that the application invades privacy or does/allows something objectionable, such as cheating.

Table 3.1: Labels used to tag review statements. The left columns shows the larger theme that the label falls under, the middle the label, and the right a description of the label.

From a developer's point of view, one can infer that the auto-play feature in question is both necessary and, most likely, functioning properly. However, the user felt that their version of the application malfunctioned resulting in a sub-par experience.

An automatic classifier may not differentiate between a buggy feature and a feature that upset a customer. We also recognized that users have more experience with the application than the people annotating the user reviews. We chose to err on the side of the user and therefore classified the above example as a bug.

3.4 Text Analysis

This section explains our approach to finding feedback loops.

3.4.1 Preprocessing the Data

Before classifying any of the data we first convert the raw text into vectors for the SVM classifier.

Our first step breaks the release notes or reviews into sentence-level documents. Then we remove formatting noise in the documents that make it easier for humans to read but serves no purpose for a computer. The linguistic noise our preprocessing focuses on removing is topic headers, distinct bullet points, different word forms, and stop words. The following are the steps executed to prepare the data for classification.

If the data source is release notes, we removed the text related to section headers within the release notes. Figure 3.1 contains an example of a section header in release notes found in the application Afterlight. The statement "BUG FIXES" should not be considered for classification under the Bug label because it does not represent a change made to the application, while the phrase "bug fixes" in "Other small bug fixes and improvements" should be considered as valid for classification. User reviews do not usually follow the list format common in release notes, therefore we skip removing section headers when processing user reviews.

When creating a vector, the algorithm considers the words "fixed", "fixes", "Fixed", and "fix" as four different features. The next steps standardizes language.

First, the algorithm converts all of the text to lowercase and expunges non-alphanumeric characters. This gets rid of the distinction between "fixed" and "Fixed" as well as re-

Before preprocessing

```
BUG FIXES

-A crash that occured when an image is sent to
Afterlight from another app has been fixed.
-Other small bug fixes and improvements.
-Fixed a small saving bug.
-a small fix to a location data bug
```

After preprocessing

```
crash occur imag fix
small bug fix improv
fix small save bug
small fix locat data bug
```

Figure 3.1: A sample of release notes. Notice the two different uses of the phrase “bug fixes” before preprocessing: one as a divider and one as something shipped in the release.

moving any extra bullets or symbols such as “➤”.

Next we remove stop words. Stop words are words that are necessary for the mechanics of natural languages, but do not further understanding of concepts such as “to”, “at”, or “the”. Stop words commonly include prepositions, articles, pronouns, and adverbs.

In addition to a general language stop word list, we also created a mobile stop word list. This list contains all the non-English names of the applications in our corpus (e.g. Facebook, TriviaCrack, and Afterlight) but does not remove the names of applications that use existing English words (e.g. Color Splash, Planets, and Alarm Clock Free) as these words may appear elsewhere in the corpus.

Removing non-English names prevents over-fitting the classifier on applications associated with a label and over-matching the semantic relatedness score, described in Section 3.6.1. For example, application “BuggyApp” has many reviews or release notes that mention “bug” or “fix” in the form of “BuggyApp has too many bugs. Don’t buy it!” or “fixed a bug that made BuggyApp slow”. When generating a model with the training set the classifiers will associate “BuggyApp” with the “bugfix” class. When classifying another application not in the training set, “AnotherApp”, the classifier will more weakly connect a bugfix and the “bugfix” class because the documents from “AnotherApp” do not contain references to “BuggyApp”.

The final step in standardizing the language stems the words using the Porter Stemming Algorithm [30] (Porter stemmer) from the Natural Language Toolkit (NLTK) [31]. The Porter stemmer transforms words from their conjugated or inflected forms to a reduced form of the word. Stemming the words allows the classifiers to make associations on the concepts conveyed by the words, not based on the tense or variation of the word. After transformation by the Porter stemmer the words “fixed”, “fixes”, and “fix” are all reduced to “fix” as shown in Figure 3.1.

3.4.2 Converting to Vectors

After cleaning the documents, the next step is to convert the documents into vectors for the SVM classifier. We use the GenSim libraries for creating dictionaries, storing corpora, and generating a tf-idf model [32].

First we need to make a **Term Frequency-Inverse Document Frequency** (tf-idf) model for the entire corpus which will give each feature in the vector a weight [33].

1	Original	“Bug fixes and improvements”
2	Preprocessed	“bug” “fix” “improv”
3	Create dictionary	“bug” “fix” “improv” dict. index 21 163 262
4	MM Corpus	(21,1) , (163,1) , (262, 1)
5	Tfidf Transform	(21, 0.429), (163, 0.662), (262, 0.614)
6	Liblinear SVM Format	“0 21:0.429 163:0.662 262:0.614”

Table 3.2: The steps for transitioning a release note into a corresponding tf-idf-scaled vector for use in the SVM classifier.

As a running example in this first section we use the common release note statement “Bug fixes and improvements”. The progress of this sentence through the next steps is shown in Table 3.2. At this point in our document preparation algorithm our example document is in the form of Table 3.2 Row 2.

We begin converting the documents to vectors by creating a dictionary from the all documents in the text corpus (either for release notes or reviews, the corpora remain separate). The dictionary maps all features to unique numeric identifiers.

In the example, we now have a dictionary that contains “bug”, “fix”, “improv” where the numeric identifiers of the features are 21, 163, and 262, respectively (Table 3.2 Row 3).

Making the initial corpus large is crucial because the dictionary will define the number of dimensions for the SVM Classifier. Words excluded from the dictionary will not be considered features the SVM classifier. If you want to add documents that have new features to the corpus, the new features will either be ignored or the process to convert the statements to vectors will have run from the beginning.

After creating the dictionary, the algorithm transforms the documents using the dictionary to a matrix market (MM) formated corpus. The MM formatted corpus stores the sparse text data efficiently [34].

The algorithm represents documents retrieved from the MM formated corpus using the GenSim library as tuples. The first value in the tuple is the index of the feature in the dictionary and the second value is the number of times that feature appears in the document (Table 3.2 Row 4). In the example, each feature appears once in the document so the second value in each tuple is 1.

Label	precision	recall	f1-score	support
Bug	0.78	0.92	0.84	62
Feature Request	0.45	0.55	0.49	33
Enhancement	0.80	0.75	0.77	84
Misc.	0.41	0.28	0.33	40
avg / total	0.67	0.68	0.67	219

Table 3.3: Evaluation metrics of the release note SVM classifier by class.

Next, the algorithm uses the GenSim libraries on the numerical corpus to generate a tf-idf model. After the tf-idf model transforms the corpus, each tuple contains the index of the feature and the tf-idf real-value weights (Table 3.2 Row 5).

With each additional document added to the corpus the tf-idf model and the processed corpus must be updated; this also means the algorithm must recalculate the tf-idf for each document in the processed corpus.

The final step in preparation for classification converts the vectors of tuples into a string vectors for use by the SVM Classifier (Table 3.2 Row 6). We add a field to the beginning of the string that indicates the class to which the document belongs. The SVM classifier considers the class when training and testing on the data, but will ignore the class when predicting classes for documents.

3.4.3 Classifier Evaluation

Release Notes SVM We trained the release notes SVM classifier on some of labeled release notes discussed in Section 3.3.0.1. To prevent the classifier from over-fitting the data we divided the documents to include 80% of each application's documents in the training data and 20% in the testing data. The training data consisted of 889 documents and the testing data consisted of 219 documents.

The release note SVM classifier reached 67% precision over 4 classes summarized in Table .

Application reviews SVM We trained the application reviews SVM classifier on labeled reviews discussed in Section 3.3.0.2. Unlike the release notes, the labeled documents represent a random selection from all of the applications, not from a set of selected applications. To balance the training and testing data for the user reviews, we chose

Label	precision	recall	f1-score	support
Not Specific	0.62	0.70	0.66	205
Privacy & Ethical Issues	0.00	0.00	0.00	2
Bug Fix	0.55	0.42	0.48	40
Update Related	0.65	0.79	0.71	14
Content Related	0.50	0.48	0.49	120
Feature Request	0.44	0.26	0.33	27
Network Problem	0.29	0.50	0.36	4
Additional Cost	0.33	0.43	0.38	14
Response Time	0.50	0.50	0.50	6
Ad Removal	0.86	0.67	0.75	9
Compatibility Issues	0.67	0.33	0.44	6
Resources Heavy	0.50	1.00	0.67	3
Crashing	0.74	0.74	0.74	23
User Interface	0.33	0.20	0.25	10
Not English	1.00	0.86	0.92	21
Use Limit	0.00	0.00	0.00	2
Login Issue	0.00	0.00	0.00	2
avg / total	0.58	0.58	0.57	508

Table 3.4: Evaluation metrics of application reviews SVM classifier by class.

80% of the labeled documents of each class for the training data and the other 20% for the testing data to prevent over-fitting or exclusion of some classes. The training data consisted of 1996 documents and the testing data consisted of 508 documents.

The user review SVM classifier reached 58% precision over 17 classes summarized in Table . Our results are on par with Panichella et al. [7], who achieved 59% precision following similar methods.

3.5 Sentiment Analysis

This section describes the additional processing on the raw data to maximize the effectiveness of SentiStrength [19]. Subsection 3.5.1 details the handling of emoji and subsection 3.5.2 explains how we correct spell to include the presence of repeated letters.

Emoticon	Emoji
:D	😁 😃 😸 😺
>:(	😠 😡 😾
’-_-	😓 😥 😰

Table 3.5: A sample mapping of emoji to emoticons.

3.5.1 Emoji

Emoji are pictures used to convey emotions available for use in electronic messages, like less abstract emoticons. Originally from Japan, technology providers such as Microsoft, Apple, and Google added emoji as default characters sets to their mobile devices as their popularity grew and eventually standardized the Unicode codes for a set of common emoji in 2010 [35]. As a result, emoji appear in many application reviews. Due to the platform-dependent meaning of emoji before standardization in October 2010, we discard all emoji used in reviews before standardization.

SentiStrength includes emoticons in its sentiment calculations, but, unfortunately, does not include emoji. Nevertheless, research [36, 37] has shown that emoticons contribute to more accurate sentiment analysis and as emoticons and emoji serve similar purposes, we created a mapping between common emoji and emoticons to improve sentiment analysis.

Our mapping matches each emoji to a similar emoticon from SentiStregth’s emoticon library as shown in Table 3.5. If the emoji did not correspond to SentiStrength’s emoticons we added a new entry from Wikipedia’s list of emoticons[3] if the sentiment and appearance were similar. At the same time we also expanded the emoticons parsable by SentiStrength. Finally, if an emoji did not correspond to any emoticons in Wikipedia’s list we created three general categories to encapsulate “generally positive”, “generally neutral”, and “generally negative” sentiment. Table 3.6 shows some examples of each of the general categories. In SentiStrength’s original library emoticons only have trinary values (-1, 0, and 1) as opposed to the single-scale values possible in text. We followed this template and therefore do not distinguish different degrees that may appear to be present between 😠 and 😡, for example.

[3] https://en.wikipedia.org/wiki/List_of_emoticons

Emoticon	Unmatched Emoji
∼:)	👏 👌 👍
∼:(	👎 💩
∼:|	👊

Table 3.6: A sample mapping of unmatched emoji to emoticons. A '∼' precedes each emoticon to indicate the emotion attached with that emoji is equivalent to a positive, neutral, or negative sentiment.

SentiStrength allows for multiple emotional associations between words and so most emoji tend to be either positive, negative, neutral, positive or neutral, or negative or neutral. There was one emoji that did not fall into this scheme. The "face with look of triumph" emoji, 😤, appears in both a positive and negative context. Unable to initially place this emoji, we looked through tweets and found usage split between those using it to express a triumphant emotion and those using it to express an angry emotion (as a person with steam coming off of their head). Due to the extreme duality of the triumph/anger emoji, we decided the textual context that this emoji should represent a neutral value, not affecting the sentiment of the text.

3.5.2 Sentiment Spell Check

Research in sentiment analysis [38] shows that increased levels of sentiment relates to repeated letters. SentiStrength operates under the assumption that a single repeated letter indicates a misspelling, but two or more repeated letters denote an intensification of the sentiment. For example, when calculating the sentiment of "I love this app!", "I loove this app!", and "I looooooooooove this app!" , the first two sentences represent a net sentiment of +3 and the last, a net sentiment of +4.

As part of processing the data, we needed to correct spelling mistakes, particularly for user reviews, because misspellings are common and may interfere with the classification and sentiment calculation. Correcting misspellings becomes more complicated when repeated letters for sentiment because we want to retain the sentiment and ensure, apart from the extended letters, the word is spelled correctly.

Consider the word "Hirribbbble", a misspelling and elongation of the word "Horri-

Spelling	Hirribbbble	Horrible	Horribleness	Horribbbble
Net Sentiment	1	-3	-3	-4

Table 3.7: Different sentiment based on the spellings and miss-spellings of “Horrible”

ble” which unaltered has a sentiment of -3. When run corrected in its original form “Hirribbbble” becomes “Horribleness”, which is both a different word and has the same sentiment of “Horrible”. When corrected with our spell check “Hirribbbble” becomes “Horribbbble” which has the corrected net sentiment of -4 (Table 3.7).

To retain the sentiment information, we developed a spell checker based on the PyEchant Dictionary, an open source spell-checking library[4], that allows for more than one repeated letter. In the sentiment spell checker algorithm, we used the PyEnchant United States English Dictionary that has some added non-standard mobile-related words such as “OS”, “iPhone”, and “WiFi” as well as all of the acronyms and abbreviations recognized by SentiStrength such as “ttyl”, “g2g”, and “ur”.

The algorithm for the sentiment spell checker starts by removing any “?”, “!”, or “.” attached to the word. The algorithm stores the punctuation and replaces it after correcting the word. The spell checker also removes any non-English characters. Next, the dictionary checks the word to see if it is a correctly spelled word. If the word is spelled correctly or the dictionary has no suggestions for correctly spelling the word, the algorithm will return the word with its original punctuation back in place. If the word is not spelled correctly, we store the best match to the given word as the suggestion.

Then, the spell checker determines the number of duplicate character sets that the word contains. For example “Horrible” contains one group of duplicated characters “rr”) and “Hirribbbble” contains two groups “rr”, “bbbb”). The spell checker stores the location of the duplicated character set by index. The algorithm then generates two versions of the string, one with only single instances of the duplicated character groups and one with double instances of the duplicated characters groups and chooses the best match among the two. When checking “Hirribbbble” the generated strings are “Hirribble” and “Hirible”.

The best match at this stage is one of the two generated strings if it appears in the dictionary. If neither string is a correctly spelled word, the algorithm queries the

[4] https://pythonhosted.org/pyenchant/

Spell Checker	the sentiment spell checker	PyEnchant
Correct	3397	3380
Incorrect	871	888
Percent Correct	79.2%	79 .2%

Table 3.8: Results for the sentiment spell checker and default PyEnchant on Wikipedia's list of common misspellings.

dictionary for new spelling suggestions for both generated strings. If there are no spelling suggestions the generated string remains unchanged. In the example, the suggestions for each generated string are "Scribbler" for "Hirribble" and "Horrible" for "Hirible". The new suggestions are then compared to the original string using a Levenshtien distance [39] similarity ratio from the FuzzyWuzzy Python library[5]. The algorithm selects the generated string with the highest ratio of similarity to the original string.

After choosing the correctly spelled word the duplicated letter groups are restored to their original location. However, if a repeated letter group was part of the misspelling, for example if the word had been "Hiirribbbble", the duplicated letter group is not replaced.

The sentiment spell checker was evaluated with the lists of common misspellings from Wikipedia[6]. Table 3.8 shows that results of the sentiment spell checker did not degrade the quality of the PyEnchant spell checking.

3.6 Finding Feedback Loops

To find feedback loops we begin by selecting the earliest release we stored for an application. Next we pull all the reviews for the application with a submission date earlier than that of the selected release. Then, we iterate through the sentences of the release notes, comparing each to the user review sentences with our text comparison algorithm described in Section 3.6.1.

Here we pose three restrictions on the comparison of the sentences. First, our algorithm only compares statements in equivalent classes; bugs with bugs, crashes with bugs, feature requests with features requests, etc. This not only reduces false positives, but also reduces the computational cost of comparison across the corpus.

[5] https://github.com/seatgeek/fuzzywuzzy

[6] https://en.wikipedia.org/wiki/Wikipedia:Lists_of_common_misspellings

Second, our algorithm excludes general classes from comparison as they do not contain concrete actionable items for developers. General classes include the “Not Specific”, “Privacy and Ethical Issues”, “Content Related”, “Additional Cost”, and “Crowdsource Question and Answer”.

Finally, we restrict each review statement to only be included in a single feedback loop. This restriction limits the feedback loop to the soonest response to a user and assumes the developer then completes the mentioned short-comings or requests within a single release. The limitation applies to the first set of release notes if they describe a feature or bug in more than one sentence, which sometimes occurs when developers categorize their release notes. For example, release notes that contain: “Two new features: AUTO-FRAMES and TEXTURES” and “APPLY TEXTURES to your Photos for FREE - When applying effects, there is now a toolbar option to apply an additional texture”. If a user review requests textures in the application our algorithm would only identify a feedback loop with the first release note sentence, but not the second.

After comparing the release note statements with all the selected review statements, the algorithm pulls the next release and selects all the reviews submitted between the dates of the previous release and the current release as well as all the reviews from before previous release. Pulling all previous reviews allows the algorithm to capture feedback loops that are fulfilled in more than one release.

The algorithm repeats for all releases stored for the application.

3.6.1 Text Similarity

To determine the similarity of a release note sentence and a review sentence we begin by once again removing stop words and converting the statements to lowercase. Unlike the preprocessing for labeling classification, we do not stem the words as this invalidates the words for the semantic relatedness tool later in the algorithm. We consider a sentence a **bag of words**, where the text similarity algorithm ignores word order and grammatical structure.

Next we filter the bags of words based on their length. If either bag of words contains less than 3 words after preprocessing, we do not calculate the semantic relatedness. Sentences with less than 3 non-stop words are unlikely to provide meaningful information in both reviews and release notes.

We then compare each word with all the words in the other bag using a semantic relatedness algorithm based on Wikipedia Miner [12] (Section 2.1). Wikipedia Miner has been used to determine the relatedness between words and short phrases [12, 40], but not for sentences. We developed an equation to score the relatedness between bags of words so we could determine the relatedness of application reviews and release notes.

To formulate this equation, we selected four applications from outside the corpus with at least eight releases and reasonable distribution of application reviews across the releases. We then manually labeled application review and release note pairs that created feedback loops. We then preprocessed the manually labeled application reviews and release notes using the steps detailed above. Our sentence semantic relatedness equation then scored the labeled sample. Finally, we compared the relatedness scores with the human-labeled set.

We began with equation presented by Mihalcea [41] and adjusted to compare short texts using Wikipedia Miner over our test data. The final version of our sentence semantic relatedness equation is shown in Equation 3.1. We cubed semantic relatedness score and the confidence score because when we looked at the distribution scores of the labeled sample the feedback loops, though on average higher than then not-feedback loops, were not a distinct group. We capitalized on the difference in Wikipedia Miner scores by cubing them to polarize the data and introduce fewer false positives.

We multiply the Wikipedia Miner Scores by the tf-idf of each word to ensure more important words are given a stronger weight in the sentence relatedness score. Finally, we divide the result by the total number of semantic relatedness computations performed, giving us a value between 0 and 1.

$$R_{rev,rn} = \frac{\sum_{i=0}^{n}\sum_{j=0}^{m} WM_{rel}(rel_i, rev_j)^3 * WM_{conf}(rel_i, rev_j)^3 * tfidf(rel_i) * tfidf(rev_j)}{len(rev) * len(rel)} \tag{3.1}$$

3.6.1.1 Evaluation

We evaluated our sentence relatedness equation using three human reviewers. The reviewers backgrounds included two graduate students and one undergraduate student in

Computer Science.

We generated random set of 74 review statement-release note pairs. Using our relatedness equation, our algorithm identified 34 of these pairs as feedback loops.

We gave the human reviewers a user review and the set of release notes pair and asked “Do the release notes address a concern or request expressed in the user review?” We gave the reviewers 4 options: “Yes, the release notes address a concern or request in the user review.”, “No, the release notes do NOT address a concern or request in the user review.”, “No, there is no concern or request expressed in the user review, but they mention a similar feature or functionality.”, and “No, there is no concern or request expressed in the user review, they are about completely different things.”

We then created a single answer for each question based on the reviewers’ agreement. If all three reviewers selected a different response for a pairing, then no answer was chosen. All three reviewers disagreed on 3 of the pairs. Of the 74 pairs, the reviewers selected all the same choice for 38 of the pairs.

We use the Rand Index [42] to calculate the accuracy of our sentiment relatedness equation. Among the choices given to reviewers, we consider only the first statement a true positive as it indicates a feedback loop. Our sentence relatedness equation has 55% accuracy in identifying feedback loops.

Chapter 4: Results

We applied the tools and methodology presented in Chapter 3 on our rich corpus. In this chapter we present the main results, as driven by answers to our research questions.

4.1 **RQ1:** *Are developers responding to feedback provided in user reviews?*

First, we wanted to determine if developers were responding to feedback submitted in application reviews. We found feedback loops present in 331 applications which represents 18.7% of the eligible applications in our corpus. We set the threshold for sentence semantic relatedness intentionally high to reduce false positives. Therefore the number of feedback loops we show here is a lower bound to the feedback loops that exist. Additionally, when discussing number of application reviews, we remove all reviews with a non-actionable label "Not Specific", "Content Related", and "Use Limit"). This removes close to 75% of the application reviews from analysis, but is similar to findings by Chen et al. [3], who report a majority of application reviews are not informative for developers.

4.2 **RQ2:** *What are the characteristics of these feedback loops?*

Now that we established the presence of feedback loops, we begin our analysis of the characteristics of feedback loops on the application level. This research question is divided into several sections based on focus: category of the application, label of the application, economic model of the application, and length of time for developer to respond.

4.2.1 Feedback Loops by Category

We looked at the distribution of feedback loops by category of the application, summarized in Table 4.1 and shown in Figure 4.1. We found "Games", "Photo & Video", and "Social Networking" to have the highest absolute number of feedback loops.

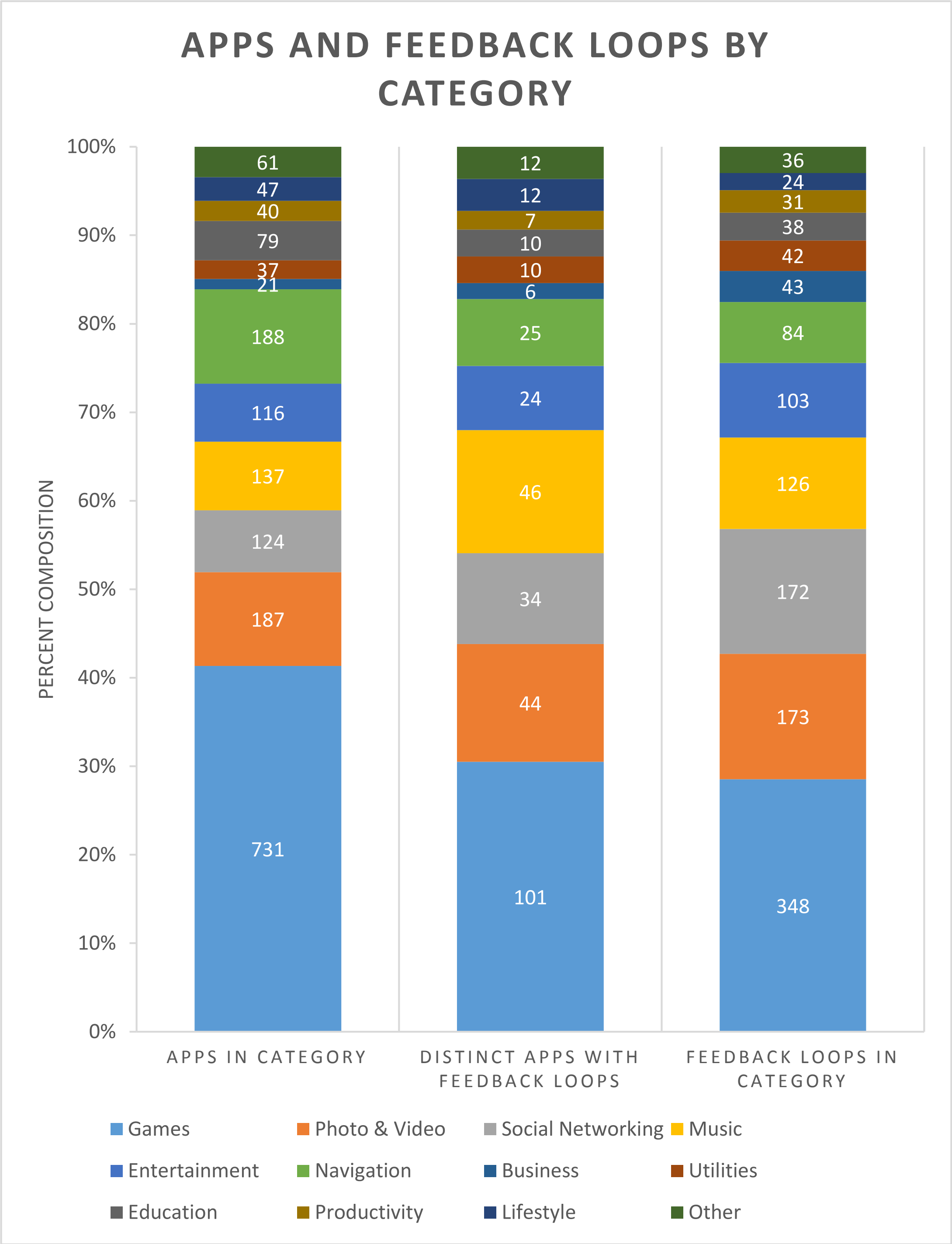

Figure 4.1: The composition of the corpus and the feedback loops by category. The left bar represents the composition of the corpus by category of application, the middle bar the composition of applications found to have at least one feedback loop by category, and the right the composition of absolute feedback loops by category. We normalized the data in each bar to use the same percentage scale. The “Photo & Video” and “Social Networking” have the largest application to feedback loop ratio. “Games”, “Navigation”, and “Education” have the smallest application to feedback loop ratio

Category	Total Apps	Apps with Feedback Loop	Number Feedback Loops
Games	731	101	348
Photo & Video	187	44	173
Social Networking	124	34	172
Music	137	46	126
Entertainment	116	24	103
Navigation	188	25	84
Business	21	6	43
Utilities	37	10	42
Education	79	10	38
Productivity	40	7	31
Lifestyle	47	12	24
Travel	23	6	18
News	7	2	10
Reference	12	3	7
Finance	1	1	1
Books	4	0	0
Health & Fitness	8	0	0
Medical	2	0	0
Sports	3	0	0
Weather	1	0	0
Total	1768	331	1220

Table 4.1: The distribution of feedback loops by category of application. The 2^{nd} column shows the absolute number of applications in a category. The 3^{rd} column shows the absolute number of applications containing at least one feedback loop in a category. The 4^{th} column shows the absolute number of feedback loops completed in a category.

However, in comparison to the category composition of the entire corpus, the number of game applications with feedback loops are underrepresented. The “Education” and “Navigation” category also had a smaller ratio of applications with feedback loops compared to other categories.

One reason “Games” may be underrepresented may involve developers incentivizing reviews by offering in-game power-ups and additional content in exchange for reviews. This practice could encourage less thoughtful reviews. The “Education” category may be underrepresented due to the target audience. We looked over the top 10 paid applications in the category and found 9/10 of the applications we targeted at children under the age of 12. Our results suggest that children are less likely to provide a critical review of an application.

“Navigation” applications may have fewer feedback loops because “Navigation” applications often rely on information or functionality outside of the software and tend to use GPS as part of the services provided. For example, an application that shows public transportation routes will rely on information from the public transportation company for scheduling or location, something the developer may not be able to control. To illustrate this one user review complains, “This app never has current and correct info. It didn’t even update the schedule on MLK day and I was late for work.” If the company does not post changes where the application draws its data or the application does not dynamically update it’s information, then users could encounter problems if developers are not vigilant. “Navigation” applications not only encounter external complications, but also contend hardware limitations. “Navigation” applications application reviews contain 15.72% of Resource Heavy complaints, second only to the “Games” category. Many of the “Navigation” applications in our corpus make use of GPS, such as an application that gives real time directions or a phone tracker. While developers can take steps to mitigate the battery consumption due to GPS, “Navigation” applications will consume resources at a faster rate than applications that do not provide services with GPS.

We found the “Music”, “Business”, and “Social Networking” applications to contain a higher proportion of applications with feedback loops. One reason for this may be the dominance of certain applications in these categories, such as Facebook, Twitter, and iTunes. Familiarity with well-known applications may provide users with expectations for applications with similar functionality.

The “Business” Category not only contains a larger percentage of applications that have feedback loops, but also has the largest ratio of feedback loops to application at 7.2 : 1. The applications responsible for this high ratio are Citrix Receiver, Genius Scan, and SignEasy with feedback loops numbering 15, 12, and 9 respectively. Further analysis of the feedback loops in Citrix Receiver found that 11 of the feedback loops were tied to a single release. The users had encountered a bug that caused the Citrix Receiver application to show a black screen shortly after starting the application. This bug was both critical, as it prevented users from using the application, and easy to describe which is likely the reason that so many feedback loops are connected to this bug.

The high number of feedback loops in Genius Scan demonstrate a limitation in our current implementation for finding feedback loops. Genius Scan is an application centered around scanning, organizing, and sending documents from smart phones. Genius Scan saw a spike of 7 feedback loops in one of its releases, unfortunately this was caused by a number of false positives in our algorithm. For example, the release notes state “[Added ability] to scan from the Photos when adding a page to a document” which our semantic relatedness algorithm mistakenly connected to “I have to scan many documents and email them to different vendors.” and “This was the only app to scan all of my documents clear without any problems.” EasySign suffered from the same language problem.

EasySign and Genius Scan are both applications with specific functionality that focuses the language used by developers and users. The problems caused by language usage in niche purpose applications could be fixed in future work by restricting the tf-idf used in the semantic relatedness algorithm to each application. This would limit the impact words like “scan” and “document” would have on the relatedness score in an application whose main service is scanning documents.

4.2.2 Feedback Loops by Label

Next we looked at the trends in feedback loops by the labels assigned by the application review SVM classifier (Section 3.4). Table 4.2 shows bug fixes are the most frequent type request and type of feedback loop, followed by Feature Requests and Crashing. All other types of feedback loops comprised a small portion of the total number of feedback loops (less than 25%).

Label	Number Reviews		Number Feedback Loops		Number Completed Loops per 10,000 reviews
Bug Fix	121573	(40.24%)	505	(41.39%)	42
Feature Request	57781	(19.13%)	308	(25.25%)	53
Crashing	51949	(17.20%)	323	(26.48%)	62
Compatibility Problems	24617	(8.15%)	5	(0.41%)	2
User Interface	18028	(5.97%)	3	(0.25%)	2
Response Time	11056	(3.66%)	14	(1.15%)	13
Network Problem	10459	(3.46%)	22	(1.80%)	21
Resource Heavy	3949	(1.31%)	3	(0.25%)	8
Log-in Issue	2703	(0.89%)	37	(3.03%)	137
Total	302115		1220		339

Table 4.2: The distribution of feedback loops by the type of feedback the user provided. The 2^{nd} column shows the number of times feedback with a label appeared in the application reviews in the corpus. The 3^{rd} column shows the number of feedback loops completed of a label. The 4^{th} column shows the number of feedback loops completed relative to 10,000 statements of feedback with that label.

We suspect the reason most of the labels are so infrequently completed is due to users' familiarity with the type of problem. Bugs and crashing have crept into the vernacular such that most people can understand how to describe the error. Far fewer users will be able to attribute their short battery life or slow response time to a specific application, and may just assume their phone or battery is old. However, the expansion of bugs and crashing terms into general English could cause over-reporting of general bugs and crashes when other problems to be under-reported. For example, an application takes several minutes to download something from the Internet and the developer provides no indicator that the application is currently working (such as a progress bar), the user then thinks that the application has frozen and crashed because nothing is happening. The user could report this as a crash, when in reality the problem was Network Issue.

Compatibility issues encompass performance across different types of iPhones (e.g. 4s, 5S, IOS8) and different platforms (e.g. iPhone, iPad, Macbook). One reason the response to compatibility issues is so low may be due to cross platform compatibility. Some applications, such as Evernote, maintain a single application for the iPhone, iPad, and iTouch versions of their applications. Other applications maintain different applications for different platforms, such as CNN's "CNN App for iPhone" and "CNN App for iPad". Reviews that complain about issue for the application on the other platform will not be updated in the release notes for the application the user is submitting a review for. In the (now defunct) application "HopStop Transit Directions for iPhone" one user asks "I wish it would enlarge when used on an iPad/have landscape view." The changes made would be reflected in the release notes for "HopStop Transit Directions for iPad."

User Interface feedback loops are also sparse. The low number of completed loops could be attributed to the classification of User Interface statements, which included both positive and negative mentions. A filter all User Interface application review statements shows that just by searching for "clean", "simple", "easy", and "friendly" and excluding the word "not" returns 2,500 statements; close to 14% of the User Interface statements. Compliments such as this review from "Mobile Mouse Pro", "Crisp, clean interface, easy to use, and so much fun." are not actionable for developers.

Next we determined the number of feedback loops completed compared to the total number of feedback with the same label provided in application reviews (shown on the far right of Table 4.2) We found that both Log-in Issues and Feature Requests were almost twice as likely as bug fixes to be addressed by developers. Log-in Issues may have a high response rate because they are both high on the triage list and can be quickly localized to the bug. Users being unable to login would preclude their use of application which could raise the importance to developers. Crashing problems are also more likely than Bug Fixes to be addressed by developers. One reason for this may be severity, small bugs may be tolerable to users whereas crashing would greatly inhibit users' ability to use the application.

Finally, developers may not want to admit how many bugs they have introduced and simply add a blanket statement along the lines of "fixed some bugs". Being vague allows the developer to both please the customer, who will find that the bug they experienced is gone when using the application, and save face. These vague Bug Fixes are ignored by our semantic relatedness algorithm and thus under-report the number of Bug Fix feedback loops.

4.2.3 Feedback Loops by Economic Model

We originally suspected the developers of paid applications or applications with in application purchases would be more responsive to user requests than in free applications. So we checked to see if the number of feedback loops was skewed to a certain economic model. However, Figure 4.2 shows that the ratio of feedback loops to the corpus loosely equivalent.

We were surprised that completely free applications did not suffer from lack of response from developers, but the meta-data provided by the store does not list whether advertisements are present. Using advertisements for revenue would give incentives to improve the application. Also, some free applications are connected to paid services, such as the previously mentioned Citrix Receiver application. Citrix provides services purchased by businesses for secure cloud management, to meet the needs of their users they publicly provide an application for free on the iTunes App Store. Without the paid service behind the application, it would not function on a users' phone. Having a paid service connected to the application would contribute to incentive to address the users' concerns.

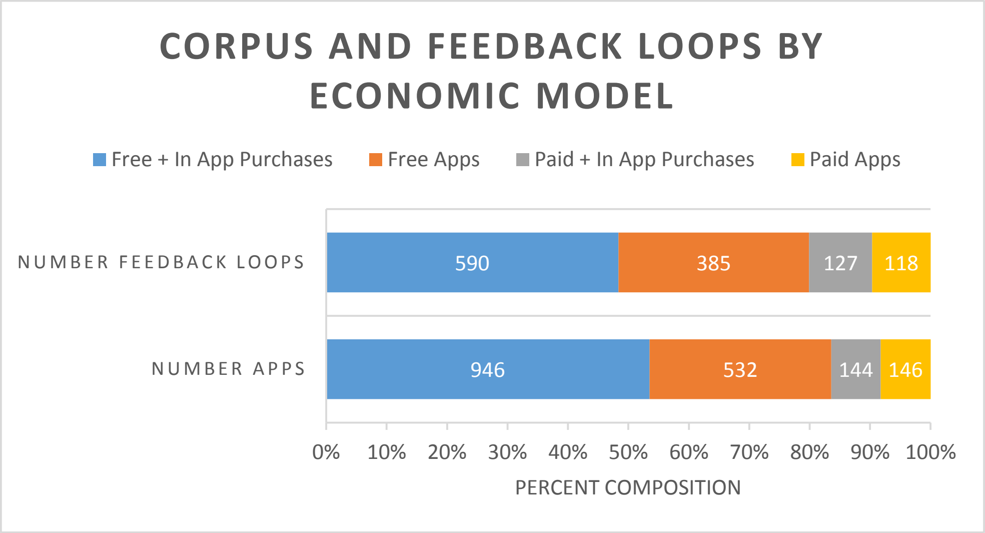

Figure 4.2: The composition of the main corpus and feedback loops by economic model. The figure shows they the presence of feedback loops does not significantly vary between models.

4.2.4 Feedback Loops by Time to Completion

The length of time in days that developers took to complete a feedback loop is the final feature of application-feedback loop characteristics we analyze. Figure 4.3 shows the number of feedback loops completed within a certain period of time (e.g. "within 1 week", "within a month", and "within a quarter"). We found that the number of feedback loops completed remained fairly static over the time periods. We also observed developers had the highest number of most feedback loops completed within the 30-90 day period (within a quarter). Surprisingly, a large number of feedback loops were completed within a 7 day time period, even more than within 90-180 day time period. Later in this section we look at the characteristics of feedback loops completed within a week.

To gain a better understanding of what type of feedback loops are completed over time we broke each of the time periods up by type. Figure 4.4 shows that although still composing a significant portion of the feedback loops, in general Bug Fix feedback loops decrease over time. This could be because developers eradicate found bugs over time so

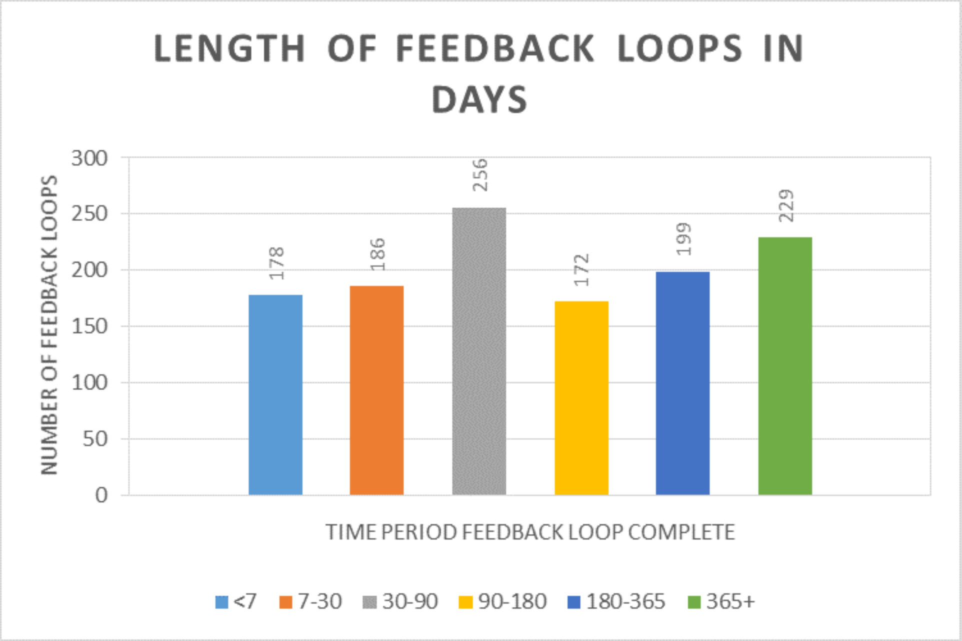

Figure 4.3: The time in days it takes for developers to complete a feedback loop. Completion is fairly consistent across all time periods. Surprisingly, a large number of feedback loops are completed in a week.

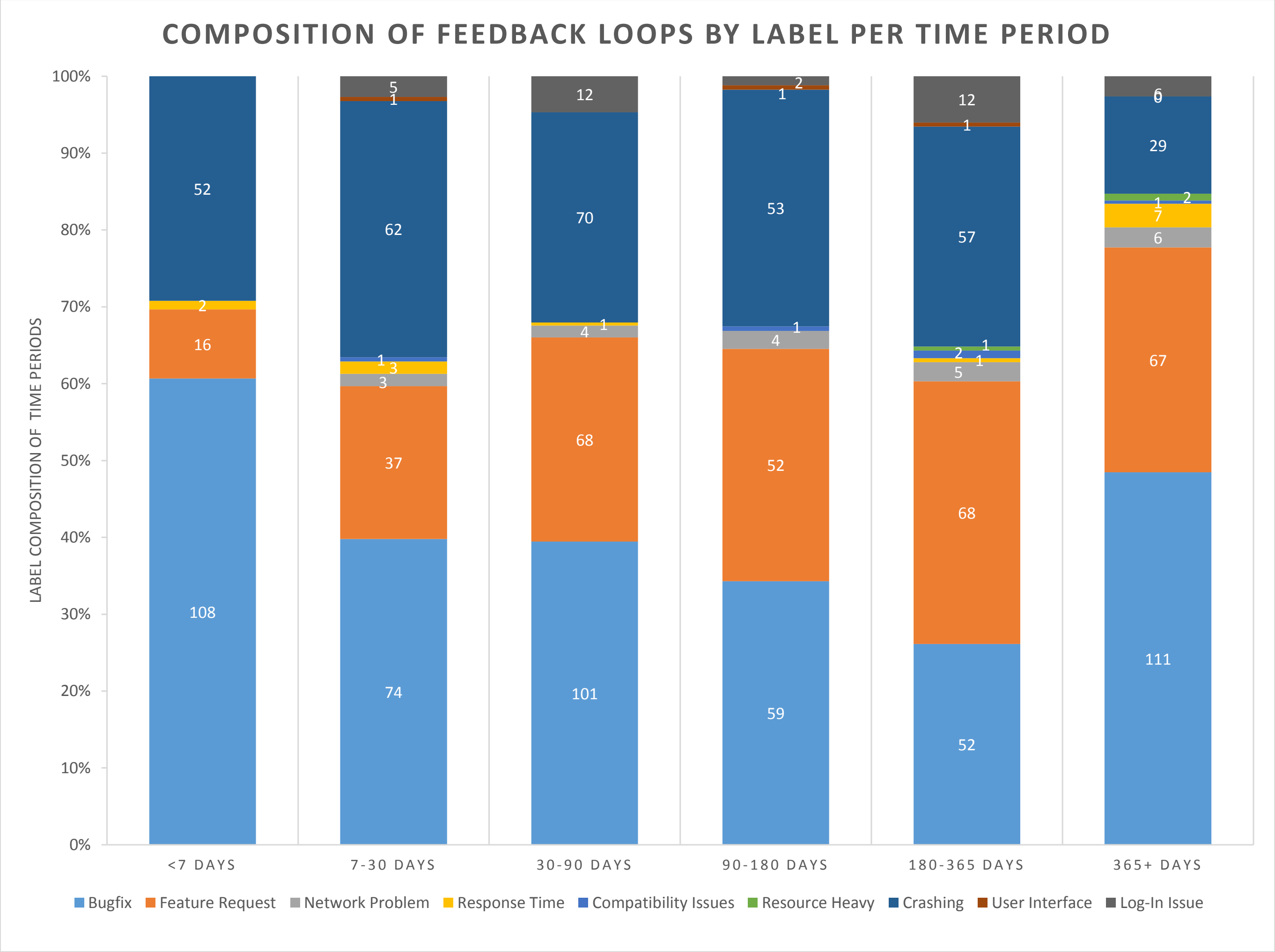

Figure 4.4: The composition of the time periods in Figure 4.3 by label. Bug fixes decrease over a period of a year while feature requests increase of the same period. Crashes maintain a fairly significant portion of feedback loops over all but the longest of time periods.

there are less to fix later. The exception for this trend is feedback loops completed in more than 1 years time. The reason for this behavior could be that as new releases are pushed, newer, more visible bugs take priority over old bugs.

Unlike Bug Fixes, Crashing feedback loops comprise roughly the same percentage of feedback loops in all time periods except for those over 1 year in length. One of the reasons crashes may be so prevalent across all time periods can be attributed to the description provided by the user. For example, in the application “Slow Down Music Trainer”, one user submitted “It crashed once in a while, but functioned perfectly.” The developer now knows that their application crashes, but is not provided information on when or where the crash occurs. This could make it difficult to locate the crash until another user provides more information on where the crash is occurring. Crashes can also be the result of Compatibility Issues or problems interacting with other applications running on the same phone. One user of the “NYC Subway Map” application states “This app, if kept running in the background, may crash the iPhone4 running iOS 4.1.” In this case, there could be several sources of the crash; the application could have compatibility problems with iOS 4.1, the application could have compatibility problems with other applications concurrently running on the phone, or the application could be crashing due to any problem in its programming. Finally, though crashes are a critical problem, they may only occur when the user takes certain actions. If those actions are not central to the functionality of the application, they may be placed on lower level of triage than other reported problems. For example, in the application reviews of the application “Muzy”, one user commented “Only crashes when I pick a photo.” and another user submitted “It crashes when I click on my camera roll...” If choosing a photo from the camera roll is not the only way to import pictures to the application, the developers at Muzy may give this crash bug a lower priority than other bugs. The feedback loop mentioned by the two users was completed in just over 180 days.

As the time from submission grows, Feature Request feedback loops comprise a larger portion of the feedback loops as the time between application review submission and developer completion grows. This is not very surprising, features can be complex and take longer to implement and may have lower priority than fixing bugs. Additional features may also be planned according to development calendar. For example, a team plans to release new content for their application every quarter, but release every few weeks for maintenance on existing features.

Feeback Loops Completed Within 7 days

Due to the developers' surprisingly quick response some feedback loops we provided further analysis of feedback loops completed within a week. Figure 4.4 shows that Bug Fixes and Crashes compose most of the feedback loops completed within a week of being submitted. The types of feedback loops completed within a week suggests quick releases of versions following a large release that was not thoroughly tested and introduced a number of smaller or critical bugs. We filtered the data to get a closer look at the behavior by looking at release date of the release the application review was for instead of the submission date of the application review. 78% of Bug Fix feedback loops were still addressed while only 21% of Crashing and 6% of Feature Requests were still addressed from the previous release. The prevalence of Bug Fix feedback loops emphasizes that a majority of feedback loops completed in less than 1 week perform maintenance to the previous release.

Next, we looked at the category distribution of the applications that completed feedback loops within 7 days shown in Figure 4.5. Applications in the “Games” category far exceeded any other category, composing 61% of the feedback loops completed within 7 days. We added the percentage of total feedback loops to Figure 4.5 to compare with the corpus at large. Despite “Games” applications composing a majority of the feedback loops, “Games” feedback loops completed within 7 days are more than twice as common than in all of the feedback loops. The prevalence of quickly completed feedback loops may point to poor testing practices or higher complexity in “Games” applications. It could also indicate gaming application development is competitive to the point that releasing new content faster is more important than releasing well tested content slower.

4.3 **RQ3:** *What do users do that enables a feedback loop?*

Our third research question looks for what users can do to encourage developers to respond to their feedback. We divide this research question into three parts: feedback loops by star rating, coaxing the developer, and feedback loop by sentiment.

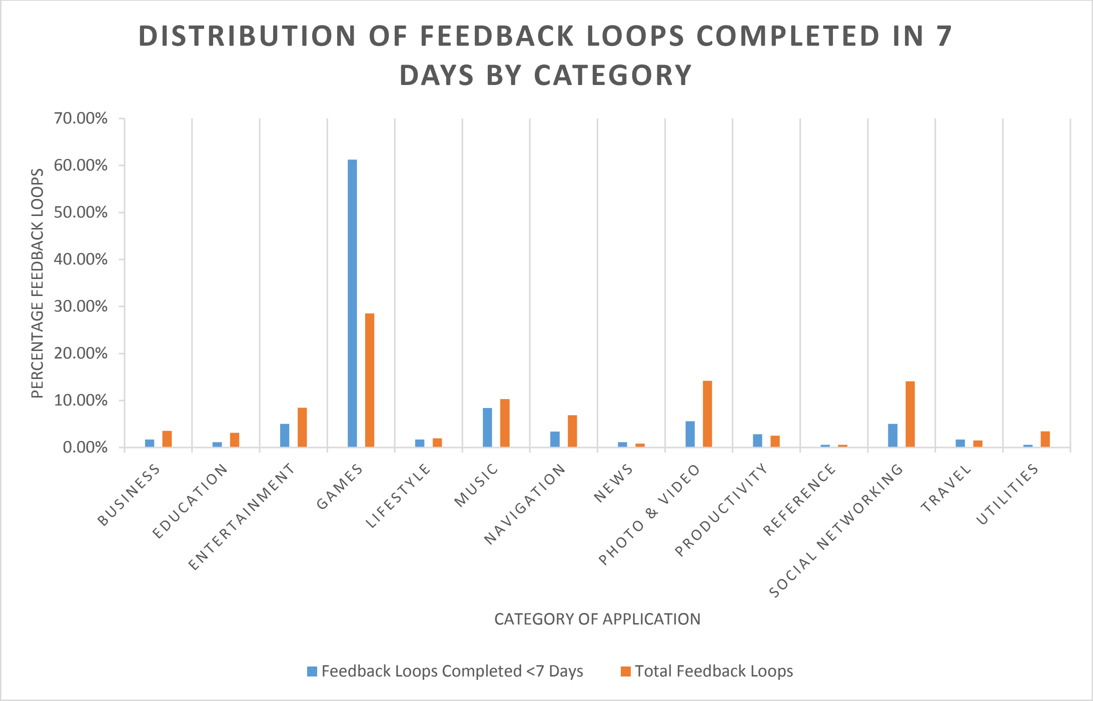

Figure 4.5: The distribution by category of the feedback loops that are completed within 7 days. The “Games” category accounts for a majority of the week long feedback loops.

4.3.1 Feedback Loops by Star Rating

First we looked at the distribution of star ratings given by users across the corpus and throughout feedback loops. Figure 4.6 shows that across the corpus (left) the majority of the user ratings are 5-star ratings. The number of 5-star ratings may seem high, but our corpus formed by collecting information from the top ranking applications from the iTunes App Store, so the ratings are probably slanted towards the higher end. The star distribution also agrees with findings in Fu et al. [43].

Feedback loop user ratings (right) are significantly more negative in comparison to the corpus. We observed the number of 1-star reviews to be the most frequent user rating in feedback loops. When considered with distribution of type of feedback loop we found, this is not surprising as Bug Fixes and Crashing comprised a majority of feedback loops observed. More surprising was the number of 5-star ratings. In theory, the 5-star rating should be used for perfect applications, not applications that are in some way lacking.

To further explore the user ratings discrepancy, we looked at the distribution of feedback loop type within star ratings, shown in Figure 4.7. We noticed that as the star rating increases the percentage of Feature Requests feedback loops completed also increases.

Conversely as the star rating decreases the number of Crashing feedback loops also increases. Log-in Issue feedback loops follow the same trend as Crashing feedback loops. These two types of feedback loops may follow such a strong trend because they are issues that will prevent the user from using the application. We further examined the Crashing feedback loops with 5-star ratings. Of the 27 application review statements, 6 mentioned the application “never crashing”, 2 thanked the developers for fixing crashes in previous releases, and the remaining reviews discussed crashes experienced by users. One user states, “Luv it, BUT. It crashes a lot and has too many ads.” but gives “Breakfast Maker” a 5-star rating. We suspected 5-star ratings for Crashing feedback loops may be an anomaly for “Games” applications, but found that games comprised 37% of the 5-star which is consistent with the distribution of feedback loops by category.

Surprisingly, Bug Fixes remain a noticeable percentage of the feedback loops across all star ratings where we expected them to follow the same trends as Crashing and Log-in Issue feedback loops. However, Khalid et al. [4] found that while users complain about bugs in different star ratings, other issues eclipse the presence of bugs. In addition to Khalid et al.’s findings, our survey of reviews suggests that if users are provided the functionality they want, they will be able to overlook small problems. A 4-star rated application review illustrates this phenomena:

> Very nice when it involves studying. It does glitch, though. Sometimes it deleted your stuff, or just randomly shuts down. If they could improve those minor glitches, this would have five stars, and this would be a pretty nice app. No major glitches for me, though!

Sporadically deleting data and shutting down, may or may not be considered “minor glitches”, but this user is willing to overlook these problems because the application, “Flashcards*”, fulfills the purpose the user downloaded it to fulfill.

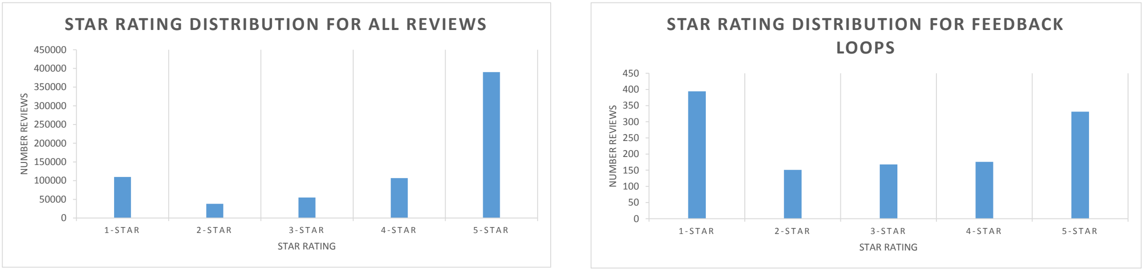

Figure 4.6: The overall star distributions of between the corpus and the feedback loops. The left graph displays the star distribution for all reviews and the right graph displays the star distribution for feedback loops.

4.3.2 Coaxing Developers

Among the reviews, we noticed instances where the users would use star ratings as bartering chip with the developers. When trying to coax the developer, the user will claim that if the developer adds or fixes some feature of the application then the user will return and replace the low rated review with a higher one. We wanted to see if the user made good on their promises, so we searched the reviews found in feedback loops for the words “star”, “update”, “edit”, or “ rat” (for “rate” or “rating”) and compared the request in the review with the rating. In searching for coaxing reviews, we only selected those that explicitly mention changing their rating, not what the application would “be worth”. For example, the following application review has a user rating of 3 stars:

> They need to add more dogs like Chihuahuas, German Shepherds, Boxers, Dalmatians, Shih Tzus, Cocker Spaniels, Huskies, Jack Russel Terriers, Pinchers, Bulldogs, Pugs, Great Danes,Poodles, Foxhounds, Beagles, Rottweilers, Collies, Bloodhounds, &Schnauzers!!!!!!!!! GIVE ME A BIG SELECTION OF DOGS THEN I WILL GIVE YOU 5 STARS!!!!!! I think EVERYONE would like a bigger selection of dogs!!!!! !Shetland Sheep Dogs,

The developer answers:

> 3D room has been added
> New puppies have been added

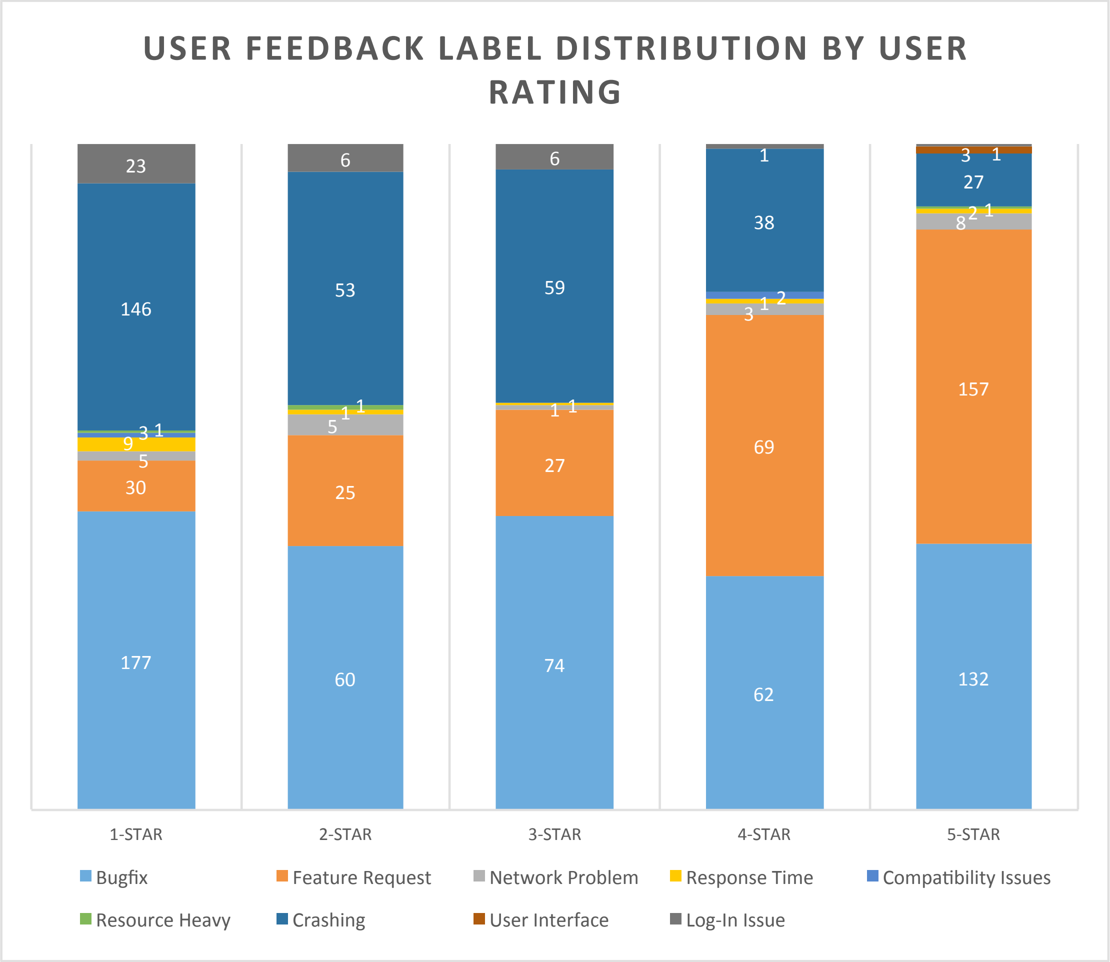

Figure 4.7: The figure shows distribution of type of feedback loop by star rating. Crashes and login issues decrease as the star rating increase while bugs remain fairly constant.

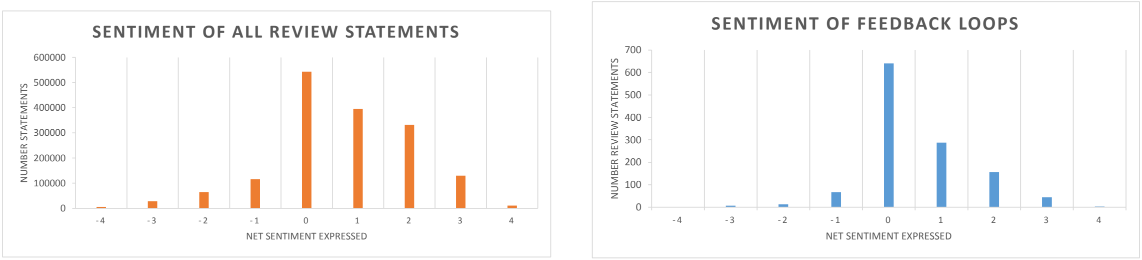

Figure 4.8: The distribution of net sentiment across all reviews and the net sentiment across feedback loops. The net sentiment expressed over all reviews has positive sentiment, while feedback loops are mostly neutral in tone.

> The kinds of Dog Clothes has been added

It is possible that the user did not feel that the developer added enough to the application to warrant a rating change, but of the 15 coaxing reviews we found in the feedback loops, none of the users returned to change their ratings. In addition to the developer not doing enough for the user, it is also possible that the user forgot or did not care enough to return to update their review.

Not returning to change ratings could indicate that developers have no reason to acquiesce to users' demands as none of the users updated their reviews.

4.3.3 Feedback Loops by Sentiment

The final user dimension we examined is sentiment expressed in the application reviews. Figure 4.8 shows the distribution of application review statements by net sentiment. Similar to the star ratings, application review statements in the overall corpus (left) express more positive than neutral or negative sentiment; i.e. if you summed the number review statements with a sentiment score greater than one, it contain the majority of sentiment expressed. However, neutral sentiment describes the largest portion application reviews in the corpus. Many users keep the head when writing a review, they describe the benefits or short-comings of the application in level tones. Only a small subset users write reviews in all capital letters with excessive exclamation points, which would represent the extreme ends of the sentiment scale.

The sentiment expressed in feedback loops followed similar trends to total corpus; most of the application reviews statements in feedback loops were neutral with a larger number of positive reviews than negative ones. However, while the total corpus could be described as overall positive in sentiment, feedback loops are overall neutral. The neutrality of statements can be attributed to users communicating enough information for developers to act on. For example, a user of the application “SKI TRACKS” posed a feature request in the application review,“Always use this when I ski. Only could want two things:1. iCloud Backup for tracks 2.Maps for popular ski resorts. Even without these features it’s a great app.” The user gave the application 5 stars, but any emotion expressed was in the final sentence, which has a net sentiment of 2, while the first sentence and the feature request in the second sentence have a neutral net sentiment of 0. Feedback loop application reviews with extreme sentiment contained colorful words such as “despise” or “unbearable”, multiple exclamation points as in “loved it!!!”, or repeated letters as in “When i try to play all i see is a black screen ughhhhhhhhhhhhh”.

To further our understanding, we divide each level of sentiment into the label of the feedback loop (Figure 4.9). Feature Requests follow the same trend as in star rating, increasing in proportion as the sentiment becomes positive. Log-in issues also follow the trend established in star rating, increasing in proportion as sentiment becomes negative. Bug Fixes seem to follow a more downward trend as sentiment becomes more positive, which is different from the star ratings which remained more static across ratings. This shows that while users may be willing to overlook bugs in their rating, they will not be happy about it.

Crashing feedback loops seem to increase with the positivity. However, crashing feedback loops illustrate a limitation of SentiStrength, which was developed using comments from a social networking site. The sentiment scores given by SentiStrength is not calibrated for mobile applications. For example, the word “crash” has a sentiment score of 0. In the mobile application world, crash has a more negative connotation than “can I crash on your couch?” Addressing the domain limitations of languages is part of our future work to improve the accuracy of sentiment analysis for mobile applications.

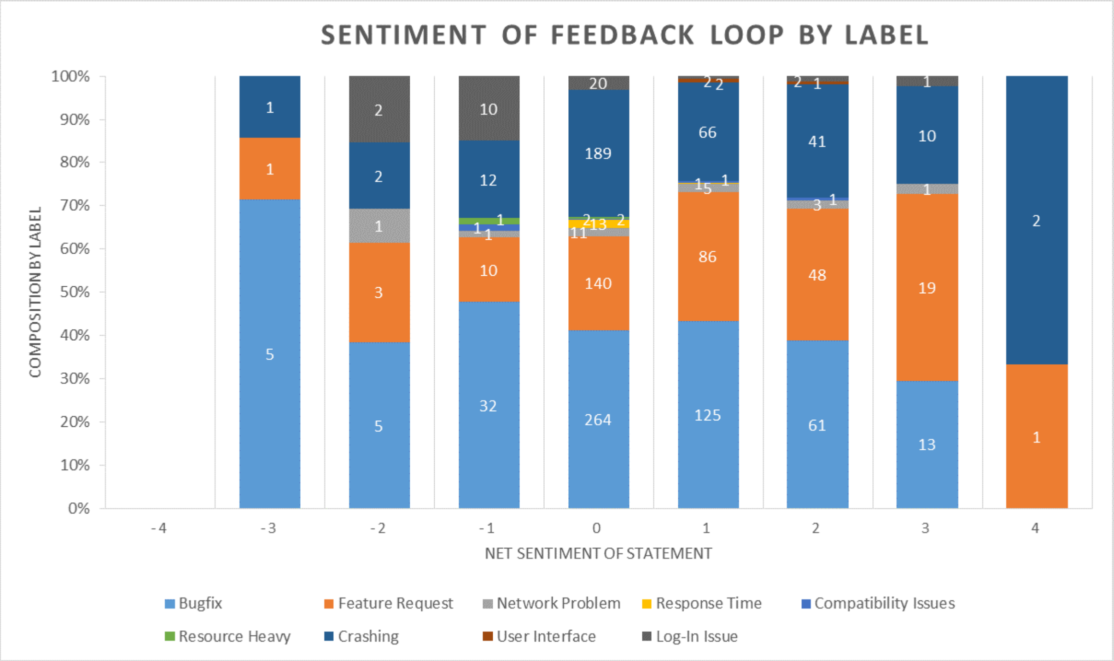

Figure 4.9: The distribution of net sentiment across Feedback Loops by label.

4.4 Implications

Here we present several implications for developers, end users, application stores, and researchers.

For Developers: Communication with users does matter; users want to know that they count. Developers need to create an infrastructure for managing the feedback in user reviews to handle the sheer volume of application reviews some applications receive. The presence of bugs at all star ratings in feedback loops suggests that developers should handle exceptions as users are more tolerant of application's unexpected behavior than of an application that crashes.

For End Users: We found evidence that developers are indeed responding to user requests, thus users should continue to provide information. Addressing the developers in a neutral and professional tone is more likely to elicit a response than ranting and raving. If the user is very frustrated, they may benefit by waiting a few minutes to cool off before submitting their problem in a application review. Users who chose a 1- or 5-star rating had the highest likelihood of the developers addressing their concerns. Users should not be on the fence with their reviews when trying to make their voice heard.

For Application Stores: We found that users have hijacked the application reviews as channels to report bugs and request new features. This calls for a dedicated channels to enable feedback and communication between users and developers. This would not only allow users to highlight what they want, but could concentrate the information for developers such that they can process volume of feedback received. Application Stores could also create more accountability for developers by visibly showing users how likely a developer is to address their feedback. If Application Stores can detect when a user is reporting an issue as they are typing a application review, it could be beneficial to prompt the user for more information. Finally, the Application Stores may be able to trace power users who provide developers informative, actionable feedback.

For Researchers: There is room for HCI research on alternative models for structuring casual users to submit feedback to developers. Additionally, there is another dimension to feedback loops not addressed in this paper, how other users respond to the presence of feedback loops.

Chapter 5: Threats to validity

Construct Threats *Are we asking the right questions?*

Our goal was to examine the developer-user interaction facilitated by applications stores and try to provide some characteristic of the effects of this relationship of the development of mobile applications. Thus, we think our research questions are able to provide insights of value to developers, researchers, users, and Application Stores.

Internal Threats *Is there something inherent to our processing of data that could skew results?*

The accuracy on the classifiers and sentence semantic relatedness are not perfect. The examples given in the paper show very clear cut classes, but the lines between classes are very blurred. For example, "Minor changes for improved application compatibility" could be a bug if the compatibility was not functioning properly to begin with or it could be an optimization of certain facets of the different platforms. The training and testing data was created through many renegotiating and refinements of classes. However, the accuracy of the classifiers is similar to the initial untrained agreement of humans, which was 71.6% for the release notes and 56% for the review statements in comparison to the 68% and 58% accuracy achieved by the SVM Classifiers, respectively. The reviewers of sentence semantic relatedness equations all agreed (by selecting the same choice) on only 38 out of 74, or 51%, the release notes-application review pairings.

Additionally, we use an older build of Wikipedia to construct the taxonomy for Wikipedia Miner from 2011, however we do not expect the jargon used to change dramatically in 4 years.

In gathering the applications and application reviews there was often a time delay because reviews had to be collected at a slower pace. As a result, popular applications could have had an excess of 1000 reviews submitted between the last release and the time the application reviews were collected. The most successful applications for our algorithm were those of modest popularity where application reviews were spread across several releases.

We recognize when labeling the training data for the application reviews considering the user as omniscient can introduce some error into the classifications, but instances where users were discernibly complaining about features as bugs were rarely found in the training data.

Wikipedia miner can calculate the relatedness of short phrases, but we only gave it individual words, this caused us to miss thing like “sign in”. Also, specialty apps, like a moon phase watching app had higher than average scores likely because there was less disambiguation to be done

External Threats *Are our results generalizable to general mobile applications and Application Stores?*

We conducted our research on iOS mobile applications and the iTunes App Store. We cannot guaranteed that the results will generalize to Android applications and the Google Play Store because it adds a dimension to the developer-user interaction by allowing developers to directly respond to user reviews. The ability to respond to application reviews could allow developers to extract extra information from users to better address their needs. However, we took a broad sample of applications and the iTunes App Store is one of the largest Application Stores on the market so the results could be generalizable for a large portion of mobile applications.

Reliability Threats *Can others reproduce our results?*

We will provide the code used to complete our process on Github. We will not be able to provide all of the data used in the experiment, but may provide a small sample for others to use.

Chapter 6: Related Work

We group the related work in three parts: (i) studies on text analysis, (ii) studies on Application Stores, and (iii) studies on user feedback.

6.1 Studies on Text Analysis

This section describes alternate methods that could be applied in our experimental methods or in future work.

Garg [44] developed an equation to extrapolate the number of downloads of an application in the iTunes App Store based on its rank in the top paid lists. The iTunes App Store doesn't allow users to see the total number of downloads for an application unless said user is the owner of that application. This measure could be used to compare our feedback loops with number of downloads for applications that fall in the top ranking category.

Novak et al. [45] developed an Emoji Sentiment Ranking using tweets to find a language independent resource for measuring the sentiment. The study is conducted across multiple languages and ranks 751 emoji with a sentiment score between -1 and 1. In our study, we just mapped emoji to emoticons based on their appearance or sense based on tweets for use in SentiStrength.

Mihalcea et al. [41] provide an equation to compare short texts using corpus based measures. Their equation was the inspiration for our sentence semantic relatedness equation. However, unlike the corpus measures used by Mihalcea et al., Wikipedia Miner was not bi-directional in assigning relatedness scores and provided a confidence score that needed to be considered. The examples provided by Mihalcea et al. showed an overlap in many words when comparing text pairs. Our purpose in using Wikipedia Miner was to compensate for the different language used by developers, which could result in fewer overlapping words.

6.2 Studies on Application Stores

Most prior work focuses on identifying and managing actionable feedback in application reviews, while our research focuses on how that feedback is used. This section is divided into prior empirical studies and tools to manage applications.

6.2.1 Empirical Studies of Application Reviews

Pagano and Maalej [46] performed an empirical study on user feedback found in Application Stores. Like our research, Pagano and Maalej look at the type of feedback users provide and relate it to star-ratings. They found that more than 50% of application reviews contain a mixture of topics in text, supporting our choice to use sentence level granularity. Despite looking at the distribution of reviews across categories of applications, they did not look at the type of feedback by category. Additionally, Pagano and Maalej did all of their classification manually where we use SVMs. Our research goes one step further than looking at the composition of application reviews by correlating them with release notes with developers.

Sarro et al. [47] use release notes to track feature migration through applications. The paper found that features tend to spread within similar categories. Like our research, Sarro et al. are able to track software evolution through Application Stores. However, our research focuses uses both release notes and application reviews to track changes on the application level. As future work, a combination of our research presents the opportunity to trace features as they spread through release notes and user requests. For example, when an application adds a popular feature, do users of other applications take notice and make requests? Do characteristics for a viral feature in an application exist?

Panichella et al. [7] develop an algorithm to classify the content of application reviews in Application Stores. Unlike our approach, their classifier uses a combination of text analysis, natural language processing, and sentiment analysis to derive a class. Our classifiers only consider text analysis. Panichella et al. achieved a higher precision of 75.2% using the combination of methods. However, when using only text analysis alone Panichella et al. reached similar precision to our classifiers (59.2% vs. 58.1%). The similar precision measures show that using text analysis is too naive and future work should look beyond text analysis.

6.2.2 Tools to Manage Application reviews

Several tools [3,5,8,43] have been introduced to help developers source their user reviews for features. Paloma et al. [5] showed an increase in user ratings when the developers of Android applications responded to requests by users. This research included using issue tracking systems, which users of Application Stores lack. Chen et al. [3] focus on extracting informative application reviews from Application Stores. While highly accurate, the A.R. Miner only divides the application reviews into non-informative (equivalent to our "Not Specific" and "Content Related" label) and informative reviews. WisCom [43] is a tool that traces user sentiment on three levels and helps identify sources of user complaints. Most of the tools focus on helping developers find actionable items within user reviews while our research focuses on what feedback the developers act on. Vu et al. [8] propose MARK, a tool designed to help analysts sift through a large volume of reviews. MARK focuses on the expansion and clustering of keywords in application reviews. Our research examines feedback loops classified with an SVM, but as shown by Panichella et al. [7] is naive. MARK [8] could be used in future work to help automatically classify reviews using keywords.

Iacob and Harrison [9] developed the tool MARA, which identifies feature requests in application reviews. MARA uses a set of 237 linguistic rules to classify reviews where our research uses a text analysis approach, comparing text content to classify. The results of MARA are of interest because both the precision and recall of our SVM classifier is low for feature requests (0.44 and 0.26). MARA was able to achieve 0.85 precision for identifying feature requests. This suggests that in the future, a rule-based classifier may help labels that are more difficult to identify through text analysis alone.

6.3 Studies on User Feedback

Several studies explore untrained users' ability to document software requirements both in situ and through social networking sites [48,49]. Both Seyff's and our research explore user's competence in communicating with developers, though our research looks at a more ad hoc method of communication. Their research found that given opportunity, the users were able to provide understandable requirements to software engineers.

Chapter 7: Conclusions and Future Work

In this paper, we analyzed a corpus of 1752 applications by comparing 30,875 release notes with 806,209 application reviews to find feedback loops. We found developers responded to feedback provided by their users in 18.7% of the applications. Our most surprising findings include (i) log-in Issue, feature requests, and crashes had the highest likelihood of being addressed by developers, (ii) applications in the "Games" category comprise 61% of the feedback loops completed within a week of receiving feedback, (iii) users attempt to coax developers using star ratings as bartering chips, but do not return to fulfill the promised rating change. We hope that this initial study of developer response to user feedback can provide information to facilitate the growth of developer-user interaction in Application Stores.

Future work includes refining the current semantic relatedness algorithm to better handle niche applications. We also plan to expand the accuracy of results provided by WikipediaMiner by identifying and comparing key phrases found in application reviews and release notes.

Another direction for future work is tracing the source and life cycle of bugs using user reviews. User reviews could also be used to trace the tolerance of users for certain bugs before they ultimately remove the application from their devices.

Following the social network line of research, future work could examine the users providing the reviews. We could track users who provide actionable information for developers and use their acuity to map the flow of features in mobile applications. Studying the flow of features by user could lead to the discovery of power reviewers in Application Stores whose feedback may be of greater use for developers.

As this paper show, developers are listening to feedback provided by users in Application Stores, but as reviews are unstructured it can be hard for developers to find the useful information needed to address the feedback. We need to provide assistance for both developers and users to continue to improve developer-user interaction for the benefit of both parties.

Bibliography

[1] T. Cook. Apple's worldwide developers conference keynote address. [Online]. Available: https://developer.apple.com/videos/wwdc2015/

[2] A. A. Index, "App annie index–market q31 2015," *Online at:(http://blog.appannie.com/app-annie-index-market-q3-2015/)*, 2015.

[3] N. Chen, J. Lin, S. C. Hoi, X. Xiao, and B. Zhang, "AR-Miner: mining informative reviews for developers from mobile app marketplace," in *36th International Conference on Software Engineering*, 2014, pp. 767–778.

[4] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan, "What do mobile app users complain about?" *IEEE Software*, vol. 32, no. 3, pp. 70–77, 2015.

[5] F. Palomba, M. Linares-Vásquez, G. Bavota, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "User reviews matter! tracking crowdsourced reviews to support evolution of successful apps," in *IEEE International Conference on Software Maintenance and Evolution.* IEEE, 2015, pp. 291–300.

[6] M. Wilcox and C. Voskoglou. State of the developer nation q1.

[7] S. Panichella, A. D. Sorbo, E. Guzman, C. A. Visaggio, G. Canfora, and H. C. Gall, "How can i improve my app? classifying user reviews for software maintenance and evolution," in *2015 IEEE International Conference on Software Maintenance and Evolution*, 2015, pp. 281–290.

[8] P. M. Vu, T. T. Nguyen, H. V. Pham, and T. T. Nguyen, "Mining user opinions in mobile app reviews: A keyword-based approach," *CoRR*, vol. abs/1505.04657, 2015.

[9] C. Iacob and R. Harrison, "Retrieving and analyzing mobile apps feature requests from online reviews," in *10th Working Conference on Mining Software Repositories*, 2013, pp. 41–44.

[10] Apple. itunes store. [Online]. Available: https://itunes.apple.com/us/store

[11] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[12] D. N. Milne and I. H. Witten, “An open-source toolkit for mining wikipedia,” *Artif. Intell.*, vol. 194, pp. 222–239, 2013.

[13] M. Thelwall, K. Buckley, G. Paltoglou, D. Cai, and A. Kappas, “Sentiment strength detection in short informal text,” *Journal of the American Society for Information Science and Technology*, vol. 61, no. 12, pp. 2544–2558, 2010.

[14] P. Gonçalves, M. Araújo, F. Benevenuto, and M. Cha, “Comparing and combining sentiment analysis methods,” in *First ACM Conference on Online Social Networks*, 2013, pp. 27–38.

[15] M. Thelwall, K. Buckley, and G. Paltoglou, “Sentiment in twitter events,” *Journal of the American Society for Information Science and Technology*, vol. 62, no. 2, pp. 406–418, 2011.

[16] O. Kucuktunc, B. B. Cambazoglu, I. Weber, and H. Ferhatosmanoglu, “A large-scale sentiment analysis for yahoo! answers,” in *Fifth ACM International Conference on Web Search and Data Mining*, 2012, pp. 633–642.

[17] M. Ando and S. Ishizaki, “Analysis of travel review data from reader’s point of view,” in *3rd Workshop in Computational Approaches to Subjectivity and Sentiment Analysis*, 2012, pp. 47–51.

[18] E. Gabrilovich and S. Markovitch, “Computing semantic relatedness using wikipedia-based explicit semantic analysis,” in *20th International Joint Conference on Artificial Intelligence*, 2007, pp. 1606–1611.

[19] J. Gracia and E. Mena, “Web-based measure of semantic relatedness,” in *9th International Conference on Web Information Systems Engineering*, 2008, pp. 136–150.

[20] I. Witten and D. Milne, “An effective, low-cost measure of semantic relatedness obtained from wikipedia links,” in *AAAI Workshop on Wikipedia and Artificial Intelligence: an Evolving Synergy*, 2008, pp. 25–30.

[21] A. van den Bosch, T. Bogers, and M. de Kunder, “A longitudinal analysis of search engine index size,” in *15th International Society of Scientometrics and Informetrics Conference*, 2015.

[22] T. Joachims, “Text categorization with suport vector machines: Learning with many relevant features,” in *Machine Learning: ECML-98, 10th European Conference on Machine Learning*, 1998, pp. 137–142.

[23] G. Ganu, Y. Kakodkar, and A. Marian, “Improving the quality of predictions using textual information in online user reviews,” *Inf. Syst.*, vol. 38, no. 1, pp. 1–15, 2013.

[24] B. Pang, L. Lee, and S. Vaithyanathan, "Thumbs up?: Sentiment classification using machine learning techniques," in *ACL-02 Conference on Empirical Methods in Natural Language Processing*, vol. 10, 2002, pp. 79–86.

[25] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin, "Liblinear: A library for large linear classification," *The Journal of Machine Learning Research*, vol. 9, pp. 1871–1874, 2008.

[26] Google. Google play. [Online]. Available: https://play.google.com/store/apps

[27] J. Saldaña, *The coding manual for qualitative researchers.* Sage, 2012, no. 14.

[28] P. Jaccard, *Etude comparative de la distribution florale dans une portion des Alpes et du Jura.* Impr. Corbaz, 1901.

[29] J. L. Campbell, C. Quincy, J. Osserman, and O. K. Pedersen, "Coding in-depth semistructured interviews problems of unitization and intercoder reliability and agreement," *Sociological Methods & Research*, 2013.

[30] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, 1980.

[31] S. Bird, E. Klein, and E. Loper, *Natural Language Processing with Python*, 1st ed. O'Reilly Media, Inc., 2009.

[32] R. Řehůřek and P. Sojka, "Software Framework for Topic Modelling with Large Corpora," in *LREC 2010 Workshop on New Challenges for NLP Frameworks*, 2010, pp. 45–50.

[33] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," *Inf. Process. Manage.*, vol. 24, no. 5, pp. 513–523, 1988.

[34] R. F. Boisvert, R. Pozo, K. A. Remington, R. F. Barrett, and J. Dongarra, "Matrix market: a web resource for test matrix collections," in *IFIP TC2/WG2.5 Working Conference on the Quality of Numerical Software, Assessment and Enhancement*, 1996, pp. 125–137.

[35] U. Consortium *et al.*, "The unicode standard, version 6.0. 0, defined by:," the unicode standard, version 6.0. 0",(Mountain View, CA: The unicode consortium, 2011. isbn 978-1-936213-01-6)."

[36] J. Read, "Using emoticons to reduce dependency in machine learning techniques for sentiment classification," in *ACL Student Research Workshop*, 2005, pp. 43–48.

[37] A. Hogenboom, D. Bal, F. Frasincar, M. Bal, F. de Jong, and U. Kaymak, "Exploiting emoticons in sentiment analysis," in *28th Annual ACM Symposium on Applied Computing*, 2013, pp. 703–710.

[38] M. Thelwall, "Myspace comments," *Online Information Review*, vol. 33, no. 1, pp. 58–76, 2009.

[39] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8, 1966, pp. 707–710.

[40] A. Joorabchi and A. E. Mahdi, "Automatic keyphrase annotation of scientific documents using wikipedia and genetic algorithms," *Journal of Information Science*, pp. 410–426, 2013.

[41] R. Mihalcea, C. Corley, and C. Strapparava, "Corpus-based and knowledge-based measures of text semantic similarity," in *The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference*, 2006, pp. 775–780.

[42] W. M. Rand, "Objective criteria for the evaluation of clustering methods," *Journal of the American Statistical association*, vol. 66, no. 336, pp. 846–850, 1971.

[43] B. Fu, J. Lin, L. Li, C. Faloutsos, J. I. Hong, and N. M. Sadeh, "Why people hate your app: making sense of user feedback in a mobile app store," in *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA, August 11-14, 2013*, 2013, pp. 1276–1284.

[44] R. Garg and R. Telang, "Inferring app demand from publicly available data," *MIS Quarterly*, vol. 37, no. 4, pp. 1253–1264, 2013.

[45] P. K. Novak, J. Smailovic, B. Sluban, and I. Mozetic, "Sentiment of emojis," *arXiv preprint arXiv:1509.07761*, 2015.

[46] D. Pagano and W. Maalej, "User feedback in the appstore: An empirical study," in *21st IEEE International Requirements Engineering Conference*, 2013, pp. 125–134.

[47] F. Sarro, A. A. Al-Subaihin, M. Harman, Y. Jia, W. Martin, and Y. Zhang, "Feature lifecycles as they spread, migrate, remain, and die in app stores," in *23rd IEEE International Requirements Engineering Conference*, 2015, pp. 76–85.

[48] N. Seyff, F. Graf, and N. Maiden, "Using mobile RE tools to give end-users their own voice," in *18th IEEE International Requirements Engineering Conference (RE)*. IEEE, 2010, pp. 37–46.

[49] N. Seyff, I. Todoran, K. Caluser, L. Singer, and M. Glinz, "Using popular social network sites to support requirements elicitation, prioritization and negotiation," *Journal of Internet Services and Applications*, vol. 6, no. 1, pp. 1–16, 2015.

APPENDICES

Appendix A: Data Stored

This appendix outlines the information used from the iTunes App Store for each of the three divisions of the corpus: metadata, release notes, and application reviews.
Metadata:

1. The name of the application
2. The id number of the application in the iTunes App Store
3. The overall star rating of the application
4. The total number of application reviews submitted for the application
5. Whether the application provides in-app purchases
6. The price of the application
7. The first category to which the application belongs (e.g. “Games” and “Sports”)

Release Notes:

1. The date that the release was published
2. The version number of the release
3. The list of sentences that describe the release

Each of the release notes sentences also store a label (Section 3.3.0.1).
Reviews:

1. User rating as number of stars
2. The date that the user submitted the review
3. The percentage of people who thought the review was helpful
4. The number of people who thought review was helpful

5. The user who submitted the review's name and id number

6. The id number given to the review by the iTunes App Store

7. A list of the sentences in the review which contain:

   (a) A version of the sentence we corrected for spelling mistakes

   (b) The net sentiment of the statement (Section 3.5)

   (c) A label (Section 3.3.0.2), and potentially

   (d) An id number for the release note addressing a concern or request expressed in the sentence (Section 3.6)