AppJoy: Personalized Mobile Application Discovery

Bo Yan and Guanling Chen
Department of Computer Science, University of Massachusetts Lowell
1 University Avenue, Lowell, MA 01854
byan@cs.uml.edu, glchen@cs.uml.edu

ABSTRACT

The explosive growth of the mobile application market has made it a significant challenge for the users to find interesting applications in crowded App Stores. To alleviate this problem, existing industry solutions often use the users' application download history and possibly their ratings to recommend applications that might interest them, much like Amazon's book recommendations. However, the user downloading an application is a weak indicator of whether the user likes that application, particularly if the application is free and the user just wants to try it out. Using application ratings, on the other hand, suffers from tedious manual input and potential data sparsity problems.

In this paper, we present the AppJoy system that makes personalized application recommendations by analyzing how the user actually uses her installed applications. Based on all participants' application usage records, AppJoy employs an item-based collaborative filtering algorithm for individualized recommendations. We discuss AppJoy's design and implementation, and the evaluation shows that it consumes little resource on the off-the-shelf Google Android phones. AppJoy has been available in the Android Market and used by more than 4600 users. The AppJoy's prediction algorithm provided reasonably accurate usage estimate of the recommended applications after they were installed. We also found AppJoy to be effective as the users interacted with recommended applications longer than other applications.

Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]: Information filtering; H.3.4 [**Systems and Software**]: Performance evaluation (efficiency and effectiveness)

General Terms

Design, Experimentation, Measurement, Performance

MobiSys'11, June 28–July 1, 2011, Bethesda, Maryland, USA.

Keywords

Mobile Application Discovery, Personalized Recommendation, Application Usage Pattern, Collaborative Filtering

1. INTRODUCTION

The mobile application market has seen explosive growth in recent years, with Apple's App Store boasting more than 400,000 applications and Google's Android Market also having well above 150,000 applications. According to a recent study, the mobile application market will reach $17.5 billion by 2012. By then, the number of mobile application downloads will have also grown to nearly 50 billion from just over 7 billion in 2009 [12].

While the application stores allow the users to search for applications by keywords or browse top applications in different categories, it is still a significant challenge for the user to find interesting applications that she will like. In light of this problem, many industry solutions have been proposed that mostly leverage the user's application download history or her ratings of other applications as the basis for *personalized* recommendations.

Whether the user has installed an application, however, is only a weak indicator of whether she actually likes that application. The user may simply want to try the application out, and may never use it again or may have uninstalled it. On the other hand, asking the users to explicitly rate each application they use can provide a better picture of their application taste. This approach, however, requires manual labeling and not many people are willing to or can remember to consistently provide their input. For example, we often see a widely popular application only receives 2–3% ratings from its users.

In this paper, we present AppJoy, a system that makes personalized mobile application recommendations. The novel feature of AppJoy is that it measures how the applications are actually used, and the usage scores are then used by a Collaborative Filter (CF) algorithm to make personalized recommendations. This is analogous to the "vote by your feet" approach, in which what the user does matters more when profiling her application needs. Compared with other solutions, AppJoy is completely automatic without requiring manual input and AppJoy is adaptive to the changes of the user's application taste. To the best of our knowledge, AppJoy is the first mobile application discovery system that leverages the user's actual application usage patterns.

AppJoy employs a client-server architecture, where its client collects the application's usage records and periodically uploads them to the server. The AppJoy server runs

the CF algorithm that calculates recommendations for all users on a daily basis. The user can use the AppJoy client to browse and install the applications recommended for her. To preserve the user's privacy, only the device ID was used to identify the application users.

Currently the AppJoy client is implemented on the Android platform and has been available in the Android Market since late February 2010. It has been used by more than 4600 users worldwide. Our evaluation results show that the AppJoy client consumes about 4% battery on off-the-shelf devices and the perceived response latency was reasonably low about 3–5 seconds on WiFi and 3G connections. About 5% of the recommended applications were installed and used by the AppJoy participants. For these applications, the predicted usage scores achieved more than 80% accuracy for more than 80% of users. We also found that the users interacted with the recommended applications longer than other applications.

Note that the AppJoy system is designed for "casual" discovery of interesting applications, instead of for searching applications that may meet the user's particular needs. For the later case, the user typically can search the App Stores with a couple of keywords and then read the description and other users' ratings and comments to decide whether to install an application. On the other hand, AppJoy allows a user to simply browse for recommended applications without using any keyword search. If the user does not like a particular recommendation, she can flag that application so it will not be shown again.

The structure of the paper is as follows. In Section 2 we discuss related work, and we present the AppJoy recommendation algorithm and implementation details in Section 3 and 4, respectively. The performance benchmarking results, recommendation effectiveness, and some statistics from the real-world use of AppJoy are presented in Section 5. We discuss the limitations and improvements of the AppJoy system in Section 6. Finally we conclude in Section 7.

2. RELATED WORK

Recommendation algorithms are well studied, and are widely used particularly in E-Commerce for personalized suggestions of products or services [14]. For example, Amazon makes 20-30% of its sales from recommendations. Collaborative Filtering (CF) is a popular approach for recommendations by aggregating data, such as purchase history or rating values, from many users and predicting the probability that an item will be liked by a particular user. User-based CF methods find similar users to the given user first before filtering desired items, while item-based CF methods find similarity among items first. Typically item-based CF is more scalable and can address better the data sparsity problem [13]. Thus AppJoy employs an item-based CF algorithm, Slope One [10], which is simple to implement, computationally efficient, and continuously updatable. The details are discussed in Section 3.2.

Given the significant challenge of helping the users to find interesting mobile applications, several industry solutions are proposed, such as the AppBrain for Android platform [1] and the AppsFire for iPhone platform [2]. AppBrain uses Android API to monitor what applications a user has installed recently, and recommend other applications in the same category. AppBrain also allows the users to browse hot applications that get many downloads. On the other hand, AppsFire takes a social approach that allows its users to form friendships and share what the applications they like. Different than these solutions, AppJoy takes an automatic approach that measures the application usage patterns, based on which a CF method is used to recommend related applications.

Woerndl et al. seeks to combine context information to recommend mobile applications [18]. Namely, the applications that others often used at the same location will be recommended. For example, a train timetable application may be recommended to a traveling user who is inside or near the train station, presumably many other traveling users have also used that application at that location. This is an interesting approach and could be effective under certain situations, though there lacks a real-world validation in the paper. As a future work, we will investigate how to integrate context with the AppJoy's CF framework to improve its recommendations.

The Cinematch recommendation system [4], developed by Netflix, automatically analyzes the accumulated movie ratings weekly using a variant of Pearson's correlation with all other movies to determine a list of "similar" movies. There also exist several music recommendation systems with a content-based approach. MusicStrands offers several different ways to discover music by browsing playlists, artists, albums, or by genre [8]. Pandora Radio [9] relies on a Music Genome database that consists of 400 musical attributes covering the qualities of melody, harmony, rhythm, form, composition and lyrics. These musical structures are used to analyze similarity among the songs for recommendations. AppJoy currently uses a CF method and has not integrated content, such as the application attributes, which can be an interesting extension.

Falaki et al. developed a logging tool for Android and Windows Mobile that records how much time a user spends interacting with each application, together with some other information [7]. Their purpose is to study the patterns of mobile application usage across many users, which can be used to predict, for example, the future energy drains. LiveLab is an iPhone-based application and network logger, though it requires to "jail break" the stock iPhones [15]. The results from both papers show that there exists significant diversity of application usage, which validates the usefulness of the personalized application recommendation system, such as AppJoy. Other studies of mobile application usages also exist [6, 16], which mostly use diaries and interviews at a smaller scale.

3. APPJOY RECOMMENDATION

Recommender systems are best known for their use on E-Commerce web sites, which recommend products to the visiting users based on their past purchase history or their ratings on other items [14]. Collaborative Filtering (CF) techniques are widely used to make automatic predictions (filtering) about the interests of a user by collecting many users' interests information (collaborating). The intuition behind CF is that those showed similar tastes (likes and dislikes) in the past tend to agree again in the future. For example, Amazon recommends books or music CDs to its users given what they have bought and their ratings on other items. These recommendations are specific to the user, but leverage data collected from many other users.

In the context of mobile applications, we expect that CF

algorithms will work in a similar fashion by clustering the applications into related groups based on similar personal tastes. For example, some users may use their phones primarily for entertainment to kill time, or for personal information and other productivity management applications, or for social and communication applications. Presumably people with similar tastes will like a similar set of applications. In addition, user behaviors may also influence how applications are grouped. For example, the users who commute with public transit may frequently use a transit information application that shows real-time bus schedule and may also likely use weather forecasting and game applications when on the bus. While not directly exposing the users' personal tastes or behavior patterns, CF algorithms do implicitly link related applications driven by user similarities to provide a foundation for personalized recommendations.

3.1 Usage Score

The input to CF algorithms is the "scores" of the (user, item) pairs. A score can be binary 0 or 1, indicating whether the user has purchased or liked the item, such as a book on Amazon or a news story on Digg. The score can also be a rating value given by the user to the item, often in a range such as between 1 to 5. In the context of mobile applications, we may assign 1 as the score to the (user, app) pair if the user has installed that application, or assign 0 if otherwise. While this approach is fairly straightforward, many users simply install some applications to try them out and may not bother to uninstall those applications that they do not like or feel so-so about. Thus whether the user has downloaded and installed an application is a weak indicator of the user's application taste.

Another approach is to have the users explicitly rate the applications they have installed, and assign the rating value as the score to the (user, app) pair. This method, however, requires the user's manual input and only a small percentage of users may be willing to consistently rate the applications they use. For example, a popular free Android game, Angry Birds Lite, gathered more than 1.3 million downloads but received only 4% ratings from those downloads. In addition, as the application gets updated with new versions or as competing applications become available, the user's preference of that application may change, requiring previous rating to be updated again manually to reflect the user's taste shift. The limited and possibly outdated rating data thus will reduce the recommendation quality.

Instead, AppJoy chose to passively observe how the applications are being used with an assumption that the more an application is being used suggests that the more the user likes it. This is analogous to "vote with your feet" and the users continuing to use the similar applications can be considered as like-minded users.

The next question is what metric can be used to appropriately quantify the usage of an application. AppJoy applies an RFD (Recency, Frequency, and Duration) model, which is a variation of the widely-used RFM (Recency, Frequency, and Monetary) model in marketing that measures a customer's behavior and loyalty [5]. Recency measures how recently a customer has purchased; Frequency measures how often the customer purchased in a given time period; and Monetary measures how much the customer spent in that period. In the context of using mobile applications, AppJoy focuses on the user "interacting" with the application through the application's user interface. Thus recency means how recently a user has interacted with the application. Frequency means how frequently the user interacted with the application in a given time period. Instead of using Monetary value, AppJoy uses Duration to measure how long the user actually interacted with the application. By combining these three values, RFD can provide a good estimate of how much a user "likes" to use an application.

We define recency v_R as the time elapsed since the last use of the application p by the user u, frequency v_F as the number of times u interacted with the application within a certain period, and duration v_D as the total duration time that u interacted with an application during that period. The usage score is thus represented as

$$v_{u\vdash p} = w_R v_R + w_F v_F + w_D v_D$$

where w_R, w_F, w_D are the weights based on their relative importance. The combination of these three measurements reflects the user's application taste. The applications that have been used more recently, more frequently and more time are likely to be favored more by the user.

3.2 Slope One Prediction

Given an application j that the user u has not used before, AppJoy uses the Slope One algorithm to predict the RFD score u_j reflecting how u will like j as follows. Let $S(u)$ be the set of applications u has used, and let $R_{u,j}$ be the set of applications u has used and is *relevant* to i (meaning some other user(s) used the application in $R_{u,j}$ together with j). So

$$R_{u,j} = \{i | i \in S(u), i \neq j, card(S_{j,i}) > 0\}$$

where $S_{j,i}$ is the set of the users who have used both i and j and $card(S_{j,i})$ is the number of the users in that set. The prediction of u_j thus is

$$P(u_j) = \frac{1}{card(R_{u,j})} \sum_{i \in R_{u,j}} (dev_{j,i} + u_i)$$

where

$$dev_{j,i} = \sum_{w \in S_{j,i}} \frac{v_{w\vdash j} - v_{w\vdash i}}{card(S_{j,i})}.$$

Basically the $dev_{j,i}$ is the average score difference between j and i from all the users who have used both of them. Then AppJoy predicts u_j based on u_i by adding $dev_{j,i}$ to u_i and taking an average for all relevant application i. The predictor is in the form of $y = x + b$, thus the name of Slope One. The simplicity of this approach also makes it easy to implement, and its predication accuracy is comparable to more sophisticated and computationally expensive algorithms [10].

One of the drawbacks of simple Slope One is that the number of scores observed is not taken into consideration. Assuming that we are given the scores of user u on applications i and k to predict the score of user u on application j. If 1000 users have used i and j whereas only 10 users have used k and j, the score of user u on i is likely to be a much better predictor for j than the score of user u on k is. Taking this into account, the prediction can be changed to

$$P_w(u_j) = \frac{\sum_{i \in R_{u,j}} (dev_{j,i} + u_i) card(S_{j,i})}{\sum_{i \in R_{u,j}} card(S_{j,i})}. \quad (1)$$

Algorithm 1 AppJoy Similarity Matrix

```
for ordered vector of app ids T_u by user u ∈ U do
  while size(T_u) > 1 do
    find an application p ∈ T_u with score v_{u⊢p}
    T_u = T_u − {p}
    for each application q ∈ T_u with score v_{u⊢q} do
      Diff(p,q) = v_{u⊢p} − v_{u⊢q}
      Count(p,q) = Count(p,q) + 1
    end for
  end while
end for
for each pair application p and q do
  Diff(p,q) = Diff(p,q)/Count(p,q)
end for
return Diff
```

This approach is called the Weighted Slope One and the Algorithm 1 shows how to compute the similarity matrix of the applications using this approach. Suppose there are N applications and M users, the computation of the similarity matrix table can be time intensive, with $O(N^2M)$ as the worst case. In practice, however, the complexity is closer to $O(NM)$, as most users use only a few applications.

Once the applications' similarity matrix is computed, AppJoy makes recommendations for the user u as follows. For an application j not used by the user u, AppJoy can calculate its weighted slope one score $P_w(u_j)$ using Equation 1, while the $dev_{j,i}$ can be looked up in the similarity matrix. AppJoy computes the scores for all applications j not in $S(u)$, if there is a $Diff(j,i)$ entry for any i in $S(u)$, and returns the top N applications with the highest scores.

4. APPJOY IMPLEMENTATION

AppJoy employs a client-server architecture, as shown in Figure 1. On the AppJoy client, the Usage Analyzer runs in the background and collects the application usage data, which is then periodically uploaded to the AppJoy server. The user may open the User Interface on the client to browse recommended applications, whose data is pulled from the AppJoy server. On the AppJoy server, the Web Service provides a RESTful API for the client uploads and requests. The uploaded usage records are stored in a Backend Database. AppJoy updates recommended applications for all of its users on a daily basis, as the Recommendation Engine pulls the usage data from the database and calculates the similarity matrix (Section 3.2). The recommended applications for each user are stored back in the database and are queried when the user's request comes in.

Next we discuss the implementation details of the AppJoy client and server components.

4.1 AppJoy Client

We have implemented the AppJoy client on the Google Android platform, and we have published it in the Android Market for others to use since late February 2010. The Android client has approximately 4,300 lines of Java code using Android SDK Version 1.5, also named Cupcake. The complete AppJoy application package, an archive file to be downloaded and installed by the users, is approximately 130KB including resource files.

Figure 2 shows the components of the AppJoy client. The

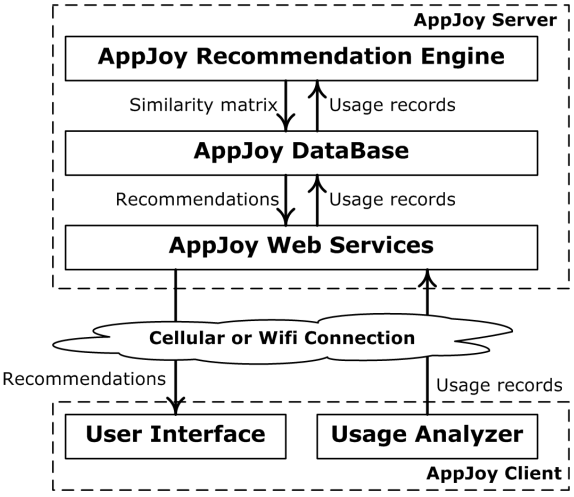


Figure 1: High-level view of AppJoy's architecture.

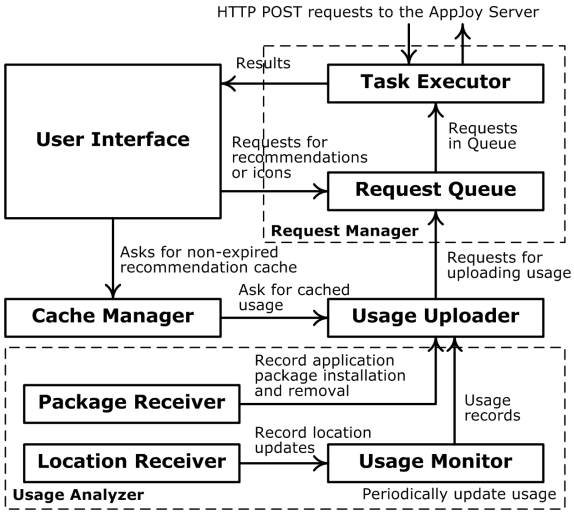


Figure 2: The components of AppJoy client.

Package Receiver monitors the changes of the device's *installed* applications using the Android notification API. The Usage Monitor collects the *running* applications' interaction statistics (Section 4.1.2). The Usage Uploader aggregates one hour's data and asks the Request Manager to upload the data to the AppJoy server. When the user opens the AppJoy client interface and starts browsing recommended applications, the User Interface looks for the cached recommended applications from last request through the Cache Manager. If the cached results are expired, it issues a request to download the newly recommended applications and the response will be cached for 24 hours, as the recommendations are updated on a daily basis. Both the recommendation download and data upload requests are put in the Request Queue and the Task Executor processes these requests by issuing HTTP requests to the AppJoy server.

4.1.1 Android Application Model

Android application can have several types of components, including *activity* and *service*. An activity presents a visual

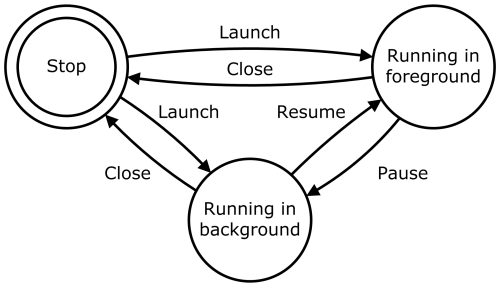


Figure 3: State transition diagram of Android applications.

interface with some focused purpose, such as showing a list of menu items the user can choose from or displaying photographs along with their captions. An application can have one or more activities, one of which is marked as the first one when the application launches. Moving from one activity to another is accomplished by having the current activity start the next one. A service, on the other hand, does not have a visual interface and runs in the background. For example, a media player application may have one or more activities allowing the user choosing songs and starting to play them. To keep the music playing while the user switching to another application, the media player activity could start a service in the background.

Android allows an application activity to start another activity that belongs to a different application. When an application is launched from the home screen, a *task* is created to maintain an activity stack. As the user navigates forward and backward through activities, which may belong to different applications, the task pushes and pops activity objects into and out of the stack. When the user presses the home button and launches another application, the previous task stack is put into the background and a new task is created.

By default, an application is in the *Stop* state. When the user taps to open this application, the application's first activity is started. The application then transitions to the *Running in Foreground* state. The active window may lose its focus and switch to the *Running in Background* state, if the user launches another application, short presses the power button that turns off the screen (long pressing the power button will turn off the device), or if the screen saver turns on when lacking user interaction. Once the user presses back button to navigate or unlock the screen to the application's activity, that application is resumed and returns to the *Running in Foreground* state. The application will return to the *Stop* state when it is terminated by the user or killed by Android OS (such as for reclaiming resources). Some service-centric applications may always run in the background after they are launched, and their state will directly move from the *Stop* state to the *Running in Background* state. Figure 3 summarizes the state-change diagram.

4.1.2 Monitoring Application Usage

To compute the applications' RFD usage scores, AppJoy needs to monitor when the application is launched and how long the user interacts with the application. This can be achieved by having the AppJoy client count the number of times the application's activities become active in the foreground (facing the user) and calculate the duration of the application's activities stay in the foreground. Unfortunately Android does not provide an API to notify AppJoy client whenever the application activity's state changes.

Instead, AppJoy client checks every one second which application's activity is in the foreground. Thus an application's *session* can be defined as the time period of consecutive checks showing that application's activity is in the foreground. AppJoy then calculates the application's hourly duration as the total length of all its sessions in the past hour and the number of sessions as the active times. These hourly application usage records are then uploaded to the AppJoy server (Figure 2). Of course, such discretized sampling is only an approximation as the application's state may change multiple times between the samples. However one second interval is quite short and the results should be reasonably close to the actual usage statistics.

One may argue that an application's background service, such as the one used by the music player, should also be considered when calculating duration time even if the user is not interacting with any of its activities. In practice, however, a long duration time caused by background service significantly increases the application's usage score and other applications without background service will be unfairly penalized. Instead, AppJoy focuses on interaction time and the application with background services may still have good usage score if the user interacts with its activities a lot. We saw this kind of applications, such as Pandora Radio, are still often recommended even when their background service time is not considered.

The AppJoy client itself runs a service in the background that checks the top activity's package name every one second, which allows AppJoy to track each application's state transition as shown in Figure 3. Although AppJoy client samples current tasks and activities fairly frequently, the evaluation results show that it only consumes modest energy (Section 5.4).

4.1.3 Usability Considerations

Today's smartphones have larger and often touch-enabled screens, have faster CPU and more memory, and support faster 3G and WiFi networks than feature phones. The usability of the mobile applications, however, remains as a thorny issue due to small screens and lacking of a full-scale physical keyboard. A recent study shows that 73% of the users experienced the slow-to-load problem when using the mobile Web, and 48% of the users found mobile applications difficult to read and use [11].

When the user launches the AppJoy client, he is greeted with an interface to provide a sense of immediacy and a background thread is spawned to retrieve the list of recommended applications for this user from the AppJoy server. The information downloaded includes the name, publisher, description, price, average rating, and download count of these applications. The download count is the number of downloads maintained by the Android Market. Once the response from the AppJoy server is received, the user interface will be updated with the downloaded data and the user can start to browse recommended applications. The icons of these applications are then downloaded in the background and the placeholder icons will be replaced with real ones after each finishes downloading, as shown in Figure 4. By pre-

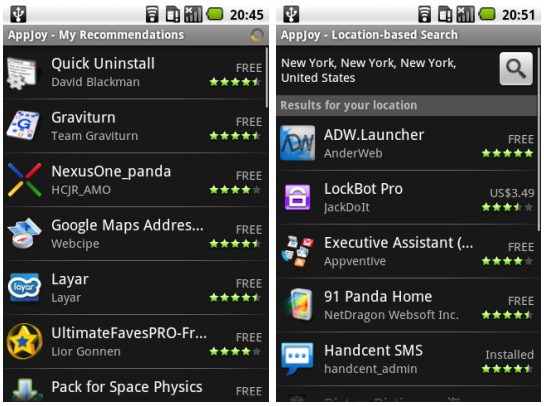


Figure 4: AppJoy client on T-Mobile G1.

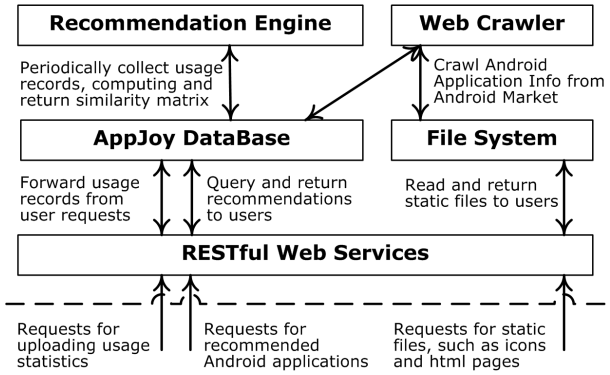


Figure 5: The components of AppJoy server.

senting pieces of content to the user whenever they become available, instead of waiting after all content is downloaded, the perceived wait time is shortened with better usability.

Instead of issuing a single request to download all recommended applications, the AppJoy client further divides the request to smaller ones by downloading data of only 10 applications for each request. Smaller downloads will more likely succeed even under a weak wireless connection, and will again shorten the time before the user sees something useful. The application icons are all downloaded in the background by the Request Manager (Figure 2), which maintains a thread pool with 10 maximum threads.

Besides making the AppJoy client more responsive, we also used bigger fonts and larger areas for each application so they are easily tappable. As a future work, we plan to do a formal usability study of the AppJoy client.

4.1.4 Location-Based Application Search

AppJoy also provides location-based search that allows the user to discover popular applications in a geographic region. We expect that people in different areas have different tastes of mobile applications, such as due to culture preferences. Also some applications are more likely to be used in a place than in other places. For example, the NYCMate application showing the bus and subway maps in the New York City is most likely used by people when they are in the city. Thus location-based search adds an interesting dimension to application discovery.

The AppJoy client's Location Receiver (Figure 2) polls the device's location every 15 minutes, and attaches the latest location to the usage records when the hourly upload is due. As AppJoy only needs a coarse location, it uses Android's Network Location Provider (NLP) that obtains current location using cell tower and WiFi signals instead of using GPS to determine location. NLP works both indoors and outdoors, responds faster, and uses less power [3]. The AppJoy client converts the coordinate returned by NLP to the name of current city by calling Google's reverse geocoding API. Thus the AppJoy server understands how and where (at the city level) the applications are used.

The user can use the AppJoy client to search nearby applications or he can input the name of desired city or a zip code. The input is again reverse geocoded to a city, and a request is sent to the AppJoy server asking for a list of applications that have been used in that city, sorted in descending usage scores. Again the AppJoy client implements the paging behavior that retrieves first 10 applications and requests next 10 as the user scrolls to the bottom of the list. If the user is the only one who uses AppJoy in that city, the list of return applications are the ones used by that user. Note that the location-based application search is not personalized. Rather, it returns popular applications based on RFD scores in a geographic region.

4.2 AppJoy Server

We implemented the AppJoy server using the Tornado web framework,[1] which is an open source and non-blocking web server. Tornado uses epoll-based I/O event notification facility and thus is reasonably fast. We deployed the AppJoy server on a HP server with a 2.6 GHz AMD Opteron six-core processor and 4G memory. The server runs Debian Linux 5.4 with kernel version 2.6.26. A nginx 0.6.35 web server is installed as a front end to offer HTTP access to the AppJoy server over RESTful web services. If needed, it is possible to deploy additional AppJoy servers on separate machines and use nginx as a load balancer. The current version of the AppJoy server has about 1,500 lines of Python code.

The components in the AppJoy server are shown in Figure 5. The AppJoy server accepts application usage records uploaded by the AppJoy clients through HTTP POST requests, and the data is forwarded to and stored in a MySQL 5.1 database. The usage records also come with additional information, such as the installed applications on the client devices and the last seen device location. The AppJoy server responds to HTTP GET requests of downloading application recommendations and other static files, such as the application icons.

To improve the performance of handling requests by decreasing the response time of querying the database, we let the database server cache query results in memory by setting the parameter *query cache limit* as 1M, and the parameter *query cache size* as 128M in the MySQL. This allows at most 1 million rows from a single query result and 128 million in total to be stored in the memory to avoid repeated queries.

We implemented an offline recommendation program using Python to compute the applications' similarity matrix (Section 3.2). On a daily basis, the recommendation program calculates the usage scores based on the application usage history of all users, and then compute and update

[1]http://www.tornadoweb.org/

the similarity values of any two different applications in the database.

4.2.1 Calculating Usage Scores

An application usage record uploaded from the AppJoy client indicates the interactive behavior between a user and an application in a certain period. We can represent a usage record r as a tuple $r(u, p, d, f, l, t)$ that is identified by the user u of the uploading AppJoy client and the corresponding application p, the interaction time d between the user and the application, the interaction count f of the application. It is also tagged with a location l to indicate where the user used this application. Furthermore, to describe which time period this usage is recorded for, a timestamp t is attached to this usage record. For example, the usage record

$$(\text{“}a04t987de0e792b3''\text{, “}swin.system.memfree\text{”}, \\ 296, 8, [42.6548, -71.3267], 1273863714)$$

comes from a user whose ID is “a04t987de0e792b3”. It records the usage within one hour before the unix time 1273863714 at the location with geographical coordinate [42.6548, -71.3267]. During this period, the application MemFree, identified by its package name [2] “swin.system.memfree”, is used 8 times in the foreground. The total interaction time between the user and this application is 296 seconds within the past hour.

There can be a lot of usage records for an application p from a user u over a long period of time. To compute the usage score $v_{u \vdash p}$, AppJoy calculates the recency v_R, the frequency v_F, and the duration v_D, and then adds them together based on their weight of importance. Here we simply set the weight of w_R, w_F and w_D to be 0.5, 0.3, and 0.2, respectively. The assignments of these weights reflect our bias towards more recently used applications, but they can be easily adjusted if other kinds of weight assignments are desired, without any modification to the client as all the calculations are done on the server.

The recency v_R is defined as 100, 50, 20 and 10 if there exists usage records with the past month, 2 months ago, 3 or 4 months ago, and 5 or more months ago, respectively. While we could have used the number of days since last use for v_R, the 4-value approach is much simpler and allows AppJoy only need to update the recency values once a month if no new usage records of p are found from u. The frequency v_F is computed as the average number of times an application was active in the foreground during an hour, and the duration v_D is computed as the average amount of time an application remained in the foreground during an hour. We normalize v_F and v_D to the range between 0 and 100 by dividing them with their possible maximum values (1800 and 3600, respectively). If an application has not been used recently, AppJoy further penalize its v_F and v_D by multiplying them with 0.5, 0.2, and 0.1, depending on the recency value. This will allow the difference of two applications' usage scores to decrease over time if they have not been used for a while, thus their usage scores become insignificant when calculating Slope One predictions.

To understand how AppJoy computes a usage score, we consider the following example. The user u_1 continuously uses two application p_1 and p_2 over six months (July through December). The summary of their usage records are shown in Table 1. The usage scores of the two applications are calculated using the definition given above. Based on these values, the application p_2 has a higher score 73.15 than the application p_1's score 25.93, since p_1 gets a recency penalty for the reason that there is no usage record for p_1 during December. Suppose that the user u_2 installed two applications p_1 and p_3 with usage scores 36 and 54, respectively. If the user u_3, who installed and used the application p_1, is asking for recommendations, the application p_2 will be recommended to u_3 due to the higher similarity between p_1 and p_2.

[2]The package name serves as a unique identifier for an application. It is also used to identify an selling application in Android Market.

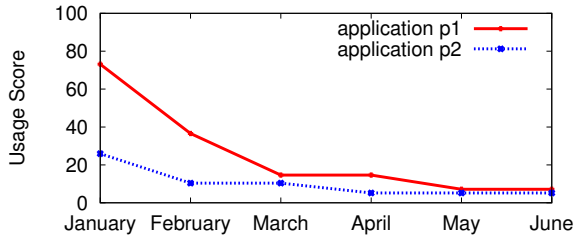


Figure 6: Usage scores decrease over time once the applications are no longer used.

As mentioned above, the difference of usage scores for these two applications p_1 and p_2 will decrease if the user u_1 stopped using them, such as by uninstalling them. Figure 6 shows their changing usage scores over the next 6 months. Since there is no new usage record for these two applications is received from u_1, they will get more recency penalty on their frequency and duration values, which causes the difference of their usage scores to decrease. As shown in the figure, the difference of usage scores between p_1 and p_2 decreases to 4.72. This decreased difference weakens the u_1's contribution power on p_1 and p_2 to the Slope One prediction, which makes sense since u_1 has changed application preference and no longer uses p_1 and p_2. Now suppose the usage scores for p_1 and p_3 from the user u_2 do not change. If the user u_3 asks recommendations again, p_3 will be recommended instead of p_2.

4.2.2 Updating Similarity Matrix

The AppJoy server updates the applications' similarity matrix when new usage records are received. If the newly received usage record for an application p from the user u is received, the usage score $v_{u \vdash p}$ will be re-calculated based on its RFD values (Section 4.2.1). If p is an application newly installed on u's device, the similarity $Diff(p, q)$ between p and another already installed application q will be calculated and entered into the matrix as a new entry. Otherwise, the $Diff(p, q)$ will be re-computed based on the change of $v_{u \vdash p}$.

Figure 7 shows the examples of similarity updates. Here the user u_1 has usage scores for applications p_2, p_8, p_9, and p_{10}. Assuming new usage records indicate that a new application p_6 is installed on u's device and the usage score for p_8 changed from 32 to 28. New similarity entries will be inserted to the matrix for p_6 and each of the other three applications used by u, and the counter is set to 1 if no other users have used any pair of these applications together. Previous similarity for application pair p_2 and p_8 will be up-

Month	July	August	September	October	November	December	Total
Number of records	5	12	7	8	4	0	36
Number of Launching	16	38	19	24	15	0	102
Opened Lasting(s)	426	2019	836	947	361	0	4589
$v_R = 50, v_F = 0.08, v_F = 1.77, v_{u_1 \vdash p_1} = 25.93$							
Month	July	August	September	October	November	December	Total
Number of records	35	42	27	48	34	29	215
Number of Launching	172	214	126	287	171	146	1116
Opened Lasting	58736	72902	43991	80174	55503	48250	359556
$v_R = 100, v_F = 0.29, v_F = 46, v_{u_1 \vdash p_2} = 73.15$							

Table 1: Examples of usage records for two application p_1 and p_2 from user u_1.

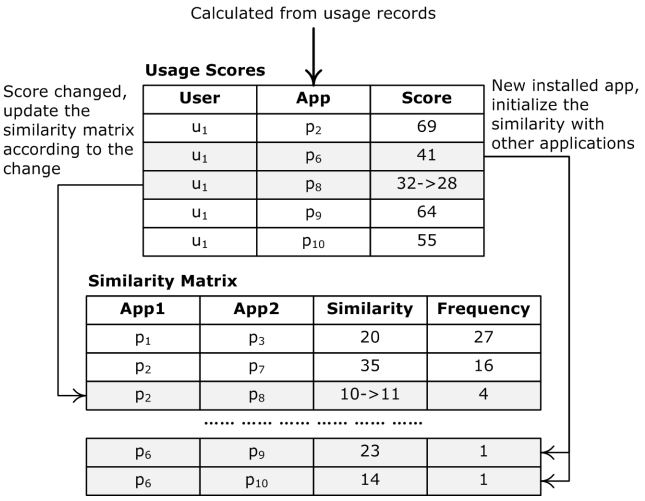


Figure 7: Example of online similarity calculations.

dated from 10 to 11, meaning their average score difference increased because the usage score for p_8 decreased.

Suppose that there are N applications and M users, the computational cost of updating the similarity matrix for each newly received usage record is $O(N)$ as the worst case. In practice, however, it is closer to $O(1)$ since most users use only a relatively small number of applications. On the other hand, the AppJoy database stores all of the hourly application usage records. Thus it is possible to recompute the usage scores and the similarity matrix from scratch. The complexity of re-calculating usage scores is $O(M)$ and re-creating the similarity matrix is $O(NM)$, which is reasonable for offline recommendations. The AppJoy database currently has 4606 users, 16,950 applications used by these users, and 10,554,319 total hourly records. It takes about 86 minutes to calculate recommendations for these users.

4.2.3 Cookie-Based User Authentication

What applications a user uses can be sensitive information that the user may not be willing to reveal publicly. To provide some privacy and anonymity, AppJoy does not require its users to register an account and to sign in for each session. Instead, AppJoy identifies the device by its Android ID, which is a hex string that is randomly generated at the device's first boot. The Android ID remains constant for the lifetime of the device. The tradeoff is that the device may change ownership and AppJoy cannot distinguish the data from different users of the same device. The recommendation algorithm, however, will eventually catch up with the new user's application taste and make better recommendations over time.

A malicious user may want to manipulate AppJoy by submitting fake usage records to influence recommendation results. For example, the developer of some application may want to upload customized usage records so it looks like his own application is used a lot from many devices. When AppJoy client is launched at the first time, it sends through HTTPS its own Android ID to AppJoy server that responds with a secure cookie. Future HTTP requests from this device will carry this cookie for authentication. Not using HTTPS on all communications reduce crypto overhead on the server, which can be significant with many clients. This approach, however, does not prevent a snooping attacker that obtains the cookie and injects fake usage records with that cookie. The AppJoy server expects hourly uploads from any one device, identified by individual cookies, and thus can easily flag the devices who send excessive usage records.

5. EVALUATION RESULTS

In this section, we first present some AppJoy statistics after it was released to the Android Market. We also analyze how AppJoy participants use mobile applications, and we discuss the effectiveness of the AppJoy recommendations. Finally we show the energy consumption results of AppJoy on three off-the-shelf Android smartphones and then evaluate the perceived latency when requesting recommendations from the AppJoy server.

5.1 AppJoy Characteristics

We released the first version of AppJoy client to the Android Market in late February 2010 (v1), and later published two additional updates (v2 and v3 in late June 2010 and December 2010, respectively). The results in this section were obtained using the AppJoy data from February 2010 to the end of March 2011.

The Android Market allows the user to browse 800 new applications or updates to the existing applications sorted by their release date. As Figure 8 shows, there was an initial jump on the number of AppJoy users after its immediate release as it attracted attention of many browsing users. As more and more new applications entered into the market, however, it became difficult for the users to find AppJoy, particularly after AppJoy dropped out of the top 800 list. We did see another small jump of user population after AppJoy v2 was released 110 days after the first release. Overall, AppJoy has seen more than 4600 unique users from 10190 cities in 99 countries.

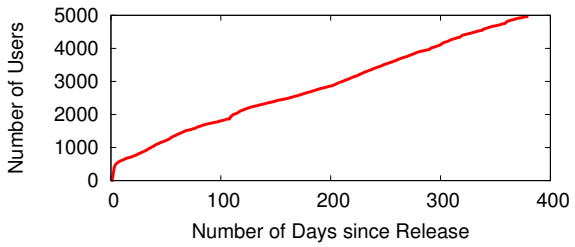


Figure 8: AppJoy user growth since February 2010.

Device Type	Number	Percentage
Motorola Droid	618	13.4%
HTC Hero	302	6.56%
HTC Desire	246	5.34%
T-Mobile MyTouch 3G	226	4.91%
HTC Droid Eris	188	4.08%
HTC EVO 4G	187	4.08%
Sony Ericsson Xperia X10i	171	3.71%
Samsung Galaxy S	165	3.58%
T-Mobile G1	145	3.15%
Samsung Moment	133	2.89%
Google Nexus One	132	2.87%
Motorola Cliq	126	2.73%
Others	1,967	42.7%
Total	**4,606**	**100%**

Table 2: Number of different Android devices running AppJoy.

Table 2 shows the device distribution of all AppJoy users. We see that a lot of users have smartphones with at least 512 MB onboard memory, such as Motorola Droid, HTC Hero, and T-Mobile MyTouch 3G. This amount of memory is often enough to allow the users to install many applications.

To the end users, AppJoy is just another Android application that they download and install. To understand how well AppJoy was received by its users, we tracked the AppJoy's *usage duration*, which we define as the time interval between the first and the last records uploaded to the AppJoy server from a particular device. Figure 9 shows the cumulative distributions of the usage duration for three AppJoy versions. The v1 curve represents the users who installed AppJoy before the v2 release; v2 curve represents the users who installed AppJoy before v3 release but after v2 release; and v3 curve represents the users who installed AppJoy after v3 release.

The AppJoy's usage duration results in Figure 9 show that

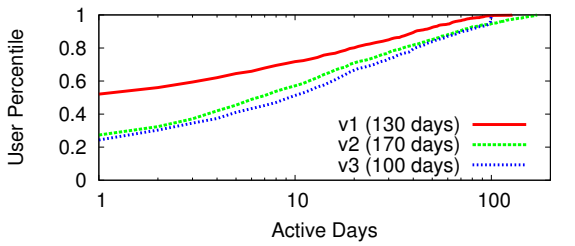


Figure 9: The usage duration of AppJoy users.

40% of the v1 users used AppJoy for only one day, which suggests that those users did not have a good experience of using the first version. Based on the feedback from some users, we found that there existed high latency when trying to view recommended applications through the 3G connections. To address this issue, we made improvement mentioned in Section 4.1.3 in the second version, which resulted in 50% of the v2 users staying with AppJoy for at least one week. In the third version, we refactored the code base and redesigned the UI, which further improved the user experience that resulted in 50% and 28% of the v3 users using AppJoy for at least 10 and 30 days, respectively.

When active, the AppJoy client uploads collected application usage records once an hour. If the upload for some reason failed, such as due to the network problems, the client will retransmit after the time out. If the upload failed three times, the client will wait until next uploading hour. If the AppJoy client is active the whole day, the server should receive 24 records per client per day if there are no persistent network or server problems. Table 3 lists the number of days, new users, and average daily records per user for the three AppJoy releases. These results show that AppJoy maintained a relatively stable organic growth without any advertisement (as also shown in Figure 8), and its user retention has significantly increased over its three releases as reflected by the received daily records.

Version	Days	Users	Records	Average
v1	130	1,529	88,646	0.45
v2	170	1,544	379,755	1.45
v3	100	1,059	327,741	3.09

Table 3: The number of usage records for three versions of AppJoy.

After the release of AppJoy, we got the error reports from Android Market Developer Console, which showed that AppJoy was force closed because of *Activity Not Responding* (ANR). We found that the error occurred when the downloading and uploading threads did not receive the response from the server after a long time, as we set the HTTP request timeout to be 60 seconds. We thus reduced the timeout value to 10 seconds in the third version.

Figure 10 presents the cumulative distributions of the time interval between two consecutive records from a device. It shows that there were only about 40% of 1-hour intervals, with other irregular intervals caused by network retransmission and AppJoy client being shut down, such as when the device was turned off or ran out of battery. Note that v3 release had a shorter retransmission timeout, for the purpose of reducing ANR errors, which resulted in slightly longer average intervals due to the higher probability of transmission timeouts.

5.2 Application Usage

The application installation and usage records collected by the AppJoy clients can provide insights of what kinds of applications are installed and how they are used by the Android users. We first analyzed the number of applications installed by AppJoy users and the results show that this number varied significantly, from 3 to 910 across all users, as shown in Figure 11. The median of the user's number of installed applications was 61. The extreme case is that

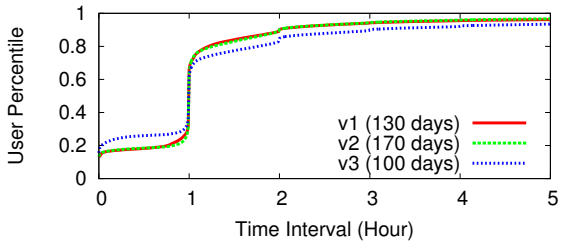


Figure 10: The time interval between two consecutive usage records for three versions of AppJoy.

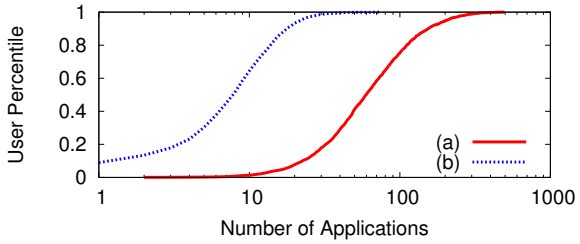


Figure 11: (a) Number of installed applications (b) Number of frequently-used applications.

someone with a DROID X phone, released in July 2010, installed 910 applications. This result suggests that many Android users do actively seek to install applications that they are interested in, and discovery tools like AppJoy can provide valuable help so the users do not have to sift through the huge number of applications in the market.

The next question is how frequently the users actually use their installed applications. Based on the application usage records we collected, we define frequently-used applications as follows. Assuming that there are n hourly usage records uploaded from the user u's device, which means that there are n hours that u interacted with at least one application. If an application p appeared in more than half of the n records, we say p is one of the u's frequently used applications. As Figure 11 shows, the number of frequently-used applications ranged from only 1 to 73, with the median to be 8. It is clear that despite the availability of many applications, the users tend to only interact frequently with a small number of applications as their favorites. This results also confirm with other researchers' findings [15].

We also checked the categories (e.g. games, shopping, lifestyle, etc.) of the applications installed by the users. The

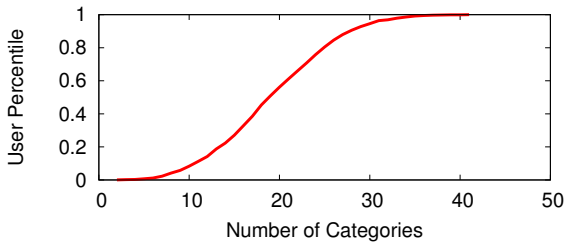


Figure 12: Number of categories of installed applications.

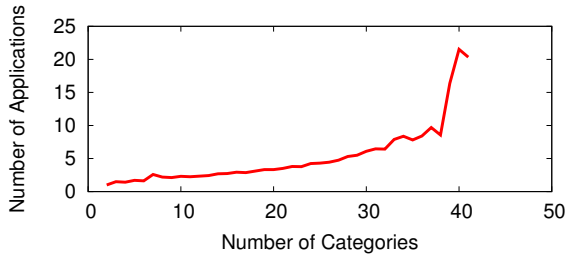


Figure 13: Number of installed applications per category.

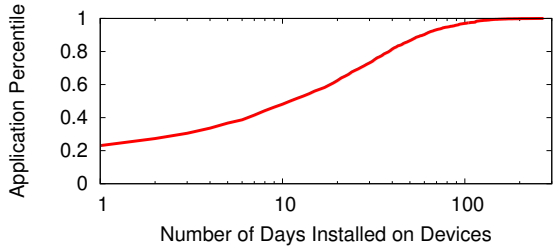


Figure 14: Number of days an application stays on user's device.

results in Figure 12 show that 40% of the users installed less than 14 of the total 42 categories defined by the Android Market, which suggests that the users do have a preference over different types of applications. We also counted the average number of installed applications per category, using the number of categories to divide the number of total installed applications. Figure 13 shows that the users who downloaded applications from 18 categories installed about 3 applications per category. The results also suggest that the more categories a user installed applications from, the more "exploratory" she is and likely will install more applications in each category as well.

Finally we analyzed how long an application stayed on the device: the time interval after an application is installed until it is removed from the device by the user. We took the AppJoy data from February 2010 to March 2011 and retrieved 753 users whose usage duration (Section 5.1) is longer than 30 days. The result in Figure 14 shows that 50% and 27% of the applications are installed on user's device for at least 11 and 30 days, respectively. Note that this result is similar to AppJoy (Figure 9) that stayed on a user's device for 10 days on average.

5.3 Recommendation Effectiveness

Our data shows that all AppJoy users have requested the list of their recommended applications, and the AppJoy server responded to all the requests. For new users without usage records yet, the AppJoy server makes recommendations based on their installed applications. Once the AppJoy client updated the user interface to show the list of recommended applications (including their icons, name, developer, price, and rating), the user can then tap to see more information about individual application, such as detailed descriptions and number of downloads.

The action of the user tapping a recommended application is also recorded and uploaded together with the hourly

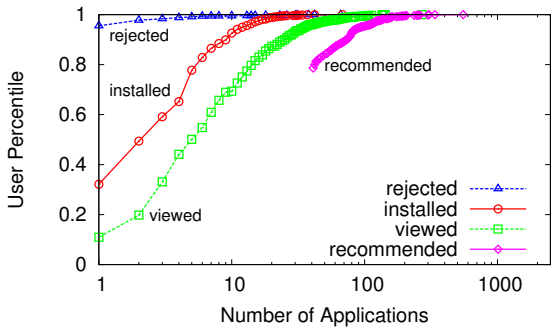


Figure 15: Recommendations statistics for 2,603 AppJoy users after June 2010.

Viewed ratio	24.2%
Rejected ratio after viewing	3.57%
Installed ratio after viewing	20.5%

Table 4: View, reject, and installation ratios for recommended applications.

usage records. We then consider this application is *viewed* and three actions can happen next. The user can proceed to Android Market to read detailed application description and other users' comments before he decides to install the application (*installed*) or not (*no action*). If the user does not like this recommendation, he may flag this application so AppJoy will no longer recommend this application (*rejected*).

The reason we know an application is installed through AppJoy recommendation is because we know the user has viewed the application through the AppJoy client and we know the application is installed afterwards given the uploaded list of installed applications. We studied AppJoy data of 2603 users who installed AppJoy after June 2010 (the v2 and v3 users in Table 3). Figure 15 shows the cumulative distributions of the number of the users' received recommended applications, and their viewed, installed, and rejected applications from these recommendations.

The results show that these users received at least 40 recommended applications, and 50% of them viewed the details of at least 5 of the recommended applications. There were 67.9% of the users who installed at least one recommended application and 7.39% of the users installed more than 10 recommended applications. In addition, Table 4 shows that 24.2% of recommended applications were viewed by the users, among these viewed applications 20.6% were actually installed and 3.57% were rejected. Thus about 4.96% of all recommended applications were installed.

If a user received X recommended applications and only installed Y of them, we can calculate an error rate E as $1 - Y/X$. By considering all users, we calculated the Root Mean Squared Error (RMSE) to be 0.9749. In comparison, Netflix has a Cinematch algorithm that predicts a user's potential rating over a movie that achieves 0.9514 RMSE [4]. In an open competition, the team of *BellKor's Pragmatic Chaos* later achieved a 10.09% improvement over Cinematch with an RMSE of 0.8554 [17]. These results show that AppJoy algorithm's recommendation effectiveness is reasonable

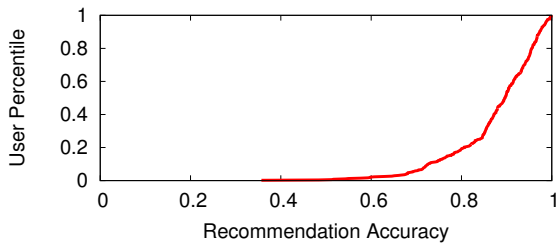


Figure 16: Usage score prediction accuracy after users installed recommended applications.

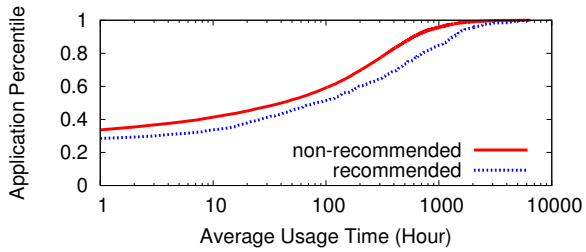


Figure 17: Average usage time of 14,927 recommended and non-recommended applications among 2,603 users.

compared to other recommender systems, though still has much room to improve.

We also measured the actual usage scores for those recommended applications that were installed by the users and compared with the predicted usage scores by the AppJoy's recommendation algorithm. We simply define the recommendation accuracy as the actual usage score divided by the predicted score, and Figure 16 shows the cumulative distribution of the recommendation accuracy. The results show that AppJoy produced fairly accurate estimate on how the applications may be used by the users, with more than 80% accuracy for more than 80% of the users.

In total there were 14,927 applications used by those 2,603 AppJoy users. Among which 597 applications were installed by some of these users through the AppJoy recommendations, and 14,330 applications were installed without the help of AppJoy. We computed the usage duration of these recommended and non-recommended applications. The result in Figure 17 shows that 50% of users interacted with the recommended applications for more than 87 hours while only interacted with the non-recommended applications for 41 hours.

Next we focused on the 597 recommended applications, which were used by 2,335 users and 839 of these users installed these applications through AppJoy. We call these 839 users "recommendation" users and the other 1,496 users "self-directed" users. Figure 18 shows that most recommendation users interacted with these applications much longer than the self-directed users. On the other hand, there were about 30% of the self-directed users interacted with these applications longer than recommendation users (see the cross over in the figure). This was most likely because they found the applications that meet their particular needs that led to longer usage than the applications casually discovered.

Finally we studied the 839 recommendation users and

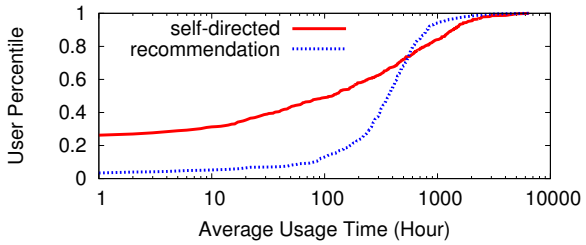


Figure 18: Average usage time of 597 applications among 2,335 users.

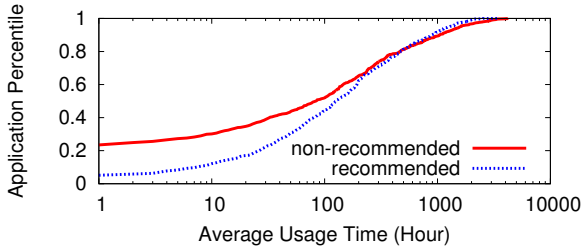


Figure 19: Average usage time of 839 recommendation users.

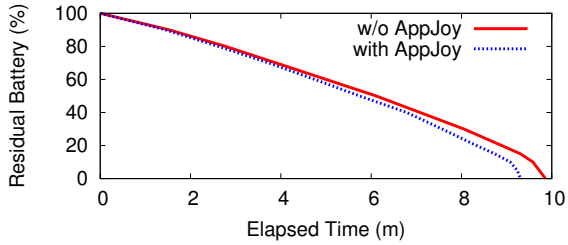
compared how they used the 597 AppJoy-recommended applications and the applications they installed by their own. Figure 19 shows that these users interacted more with the recommended applications, again demonstrating the effectiveness of AppJoy recommendations.

5.4 Energy Consumption

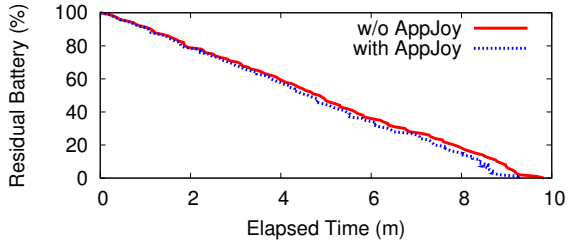
The energy consumption of the AppJoy client was evaluated using three types of Android phones. The Motorola Droid (a SIM-locked and hardware-locked version of Motorola Milestone) comes with a 1400 mAh Battery, Verizon voice plan and unlimited data plan, and runs Android 2.2. The Android Dev Phone 2 (a SIM-unlocked and hardware-unlocked version of HTC Magic) comes with a 1340 mAh Battery, T-Mobile prepaid voice plan, and runs Android 1.6. The Android Dev Phone 1 (a SIM-unlocked and hardware-unlocked version of HTC Dream) comes with a 1150 mAh Battery, and no SIM card, and runs Android 1.5.

To measure AppJoy's impact on the maximum battery life of the phone, we monitored the rate of energy consumption when no applications were running on these phones. We changed the system settings of screen timeout to prevent the screen from automatically turning off. Figure 20(a)-20(c) shows these smartphones lasted roughly 9 hours when the screen was always on. We then measured the battery depletion over time when only AppJoy was running, also shown in the same figure. The additional consumption by AppJoy reduced the battery life by roughly 3-6%.

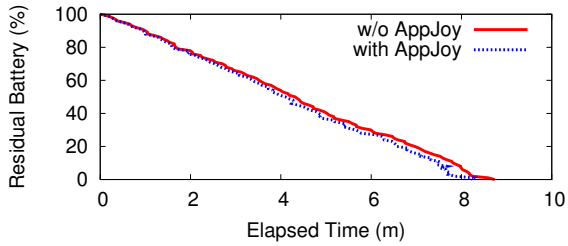
Since the version 1.6, the Android OS provides a utility that shows the individual application's energy consumption to let the user know which application has used how much of the battery. Unfortunately, there is no programmable API to directly access this information. The battery utility stores the application's power consumption data as a file in a directory only used by Android's built-in apps. On the other hand, Android Dev Phone 2 is hardware unlocked and



(a) Motorola Droid



(b) Android Dev Phone 2



(c) Android Dev Phone 1

Figure 20: Energy consumption of AppJoy client on three Android phones (controlled tests).

we can root access the Android system from its Linux shell, allowing us to copy out that data file to see how much power (in percentage) AppJoy has used.

Figure 21 shows the energy usage of AppJoy averaged over 30 times total discharge from a full battery. The x axis is the total elapsed time from last fully charge to empty battery. The y axis is the total battery consumption by AppJoy running on the Android Dev Phone 2. These experiments were uncontrolled as the user simply used the phone in her daily life, thus the battery lasted ranging from 18 to 36 hours depending on how the phone was used. The results show that on average AppJoy consumed about 4% of battery during the normal use of the phone, similarly to the results from the controlled experiments (Figure 20).

5.5 Perceived Latency

As discussed in Section 4.1.3, the AppJoy client requests 10 recommended applications from the AppJoy server each time to reduce the perceived response delay. The AppJoy server responds with detailed information of the 10 applications, such as name, publisher, average rating, download count, and so on. The AppJoy server compresses the data before sending it to the client, reducing the size of the responses to about 3–4 kilobytes. Once received the application list, the AppJoy client requests the application icons, each of which has a size of about 4 kilobytes. All these

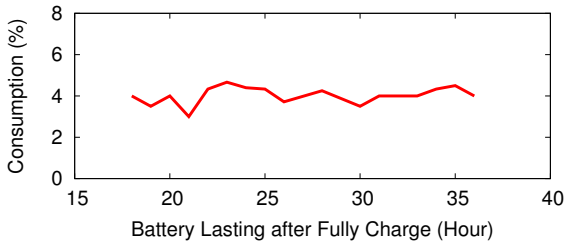


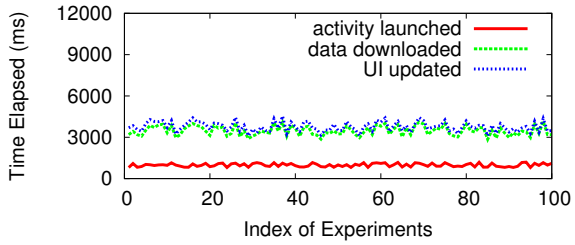
Figure 21: Energy consumption of an AppJoy client (uncontrolled test).

requests go to a thread pool and icons are downloaded in parallel.

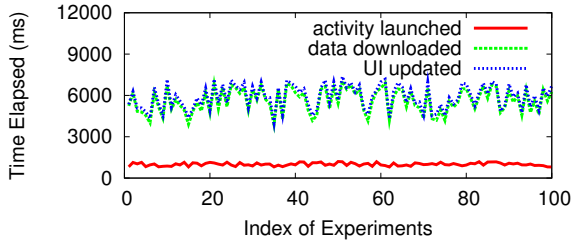
We measured the response latency of each request using Motorola Droid, and we tested through both WiFi and cellular connections. When the AppJoy client accessed the AppJoy server through a cellular connection, we also recorded the network type (2G or 3G). The response latency and the network type were recorded as the author used AppJoy throughout the day at different locations, to get a general sense of the perceived response latency. Here the *perceived latency* is measured starting when the AppJoy is launched from the home screen until the client UI is updated with the downloaded data. The main components of this latency include starting the main activity, handing off download requests to the thread pool and the thread scheduling time, round-trip network delay, server processing time, and client processing time until data is rendered in the UI. The server processing includes nginx web proxying, cookie authentication (Section 4.2.3), database query, and data compression, while the client processing includes data decompression and UI rendering.

Figure 22(a)- 22(c) shows on y axis the time needed for the activity to be launched, the round-trip delay for data to be downloaded including server and client processing, and the time for UI of that activity to be updated, over three types of connections for about 100 tests for each connection type. It can be seen that it took about 1 second to launch AppJoy activity before any request can be issued. The round-trip delay and client/server processing took about 2, 5, and 8 seconds for WiFi, 3G, and 2G connections, respectively. The latency variation over 2G network was also significantly higher. Once the data was downloaded, however, the UI rendering was quite fast.

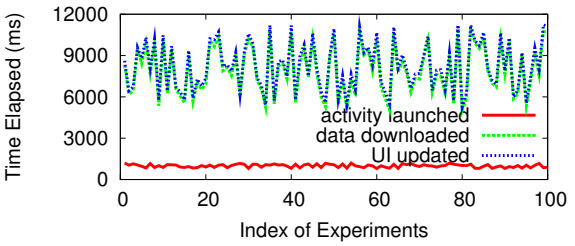
We also monitored the number of uploads to the AppJoy server over 8 months. The results are shown in Figure 23, suggesting that the number of uploads per hour mostly stayed around 70. Since AppJoy is released to international users, there is no significant load difference between day time and night. Thus at this time the overhead of the AppJoy server is relatively small and regular. This also suggests that in any hour there are about 70 active users who left the AppJoy client running in the background to report application usage records.



(a) WiFi Connection



(b) 3G Connection



(c) 2G Connection

Figure 22: Perceived latency of successful requests over WiFi, 3G and 2G connections.

6. DISCUSSIONS

In addition to Android, iOS is another popular mobile platform. Although Apple recently allowed background process on iOS, there is unfortunately no APIs for an application to monitor other applications' running state unless the device is "jail broken." On the other hand, Windows Mobile is becoming an important platform that may take a large market share due to the partnership between Microsoft and Nokia. As a future work, we plan to port AppJoy client to Windows Mobile that seems to have open APIs that allow application monitoring.

Some applications are designed to run mostly at the background with only limited foreground user interactions. For example, an automatic WiFi scanning and association application or a music streaming application often runs in the background while allowing the user to use other applications simultaneously. AppJoy currently employs a usage score tracking the user interaction time, thus is biased to penalize those background applications. This can be addressed by categorizing the applications into mostly-background or mostly-foreground groups to make separate recommendations, which we will investigate in future work.

Some mobile applications are often used in certain context. For example, a desktop powerpoint remote control application on mobile phone may only be used when the user is giving a presentation. Despite that this application may not be used often if the user does not give frequent talks, the user may actually like this application that can be recommended to other users who will give presentations. Currently AppJoy does not incorporate context information in its recommendation algorithms, which is an interesting

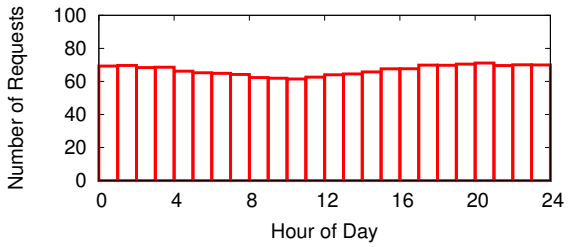


Figure 23: The average number of requests per hour received by the AppJoy Server.

research direction to explore. AppJoy has collected location information when the applications are used, and we plan to first study location-aware recommendations [18] that may notice that a user is traveling and may recommend applications frequently used by other people from out of town.

The AppJoy client shows a list of top recommendations for the user. The recommended application will be removed from the list if the user has installed it or flagged it not to show again. Otherwise, the list of recommended applications does not change frequently if the user's application usage pattern remains relatively stable. As the Figure 15 shows, few users took the effort to flag the applications that do not interest them. Thus they will see more or less the same list of recommended applications, which may make AppJoy less appealing. In the future releases, we plan to discount the applications that are shown multiple times but have not been installed so new applications can move up in the recommendation list.

Currently the AppJoy server uses a simple cookie-based authentication mechanism. This, however, does not prevent a malicious attacker from abusing the client to send in a large number of records with manipulated values, such as for the purpose of promoting certain applications. We implemented a simple filter that accepts only one upload per hour from a particular device. In each upload, the total running time of all applications should be at most 3600 seconds. In the future, we plan to investigate better integrity protection to minimize distortion of the recommendation integrity.

Our evaluation results show that the users interacted longer with recommended applications than others. We are interested in further conducting a comparison study by segmenting user groups, each employing a different recommendation strategy, such as based on installation, ratings, download popularity, or usage scores.

7. CONCLUSION AND FUTURE WORK

In this paper we present AppJoy, a novel system that uses collaborative filtering to make personalized mobile application recommendations based on the user's actual application usage patterns. Unlike other approaches that uses the user's application download history or ratings, AppJoy is completely automatic without requiring manual input and is adaptive to the potential changes of the user's application taste. The evaluation results show that the AppJoy Android client consumes about 4% battery on off-the-shelf devices, the perceived response latency was reasonably low about 3–5 seconds on WiFi and 3G connections, the recommendation's predicted usage scores achieved more than 80% accuracy for more than 80% users, and the users interacted with the recommended applications longer than others.

In the future work, we plan to improve the usability and conduct a user study of the AppJoy client, which is an important factor that the users often consider whether to continue using AppJoy. We also plan to improve the recommendation algorithm, such as by integrating the user context [18]. Finally, the application usage records AppJoy collected will allow us to perform detailed analysis of the real-world mobile application usage patterns at a much larger scale not done before.

8. REFERENCES

[1] AppBrain. http://www.appbrain.com/.
[2] AppsFire. http://www.appsfire.com/.
[3] Google Android Developers. http://developer.android.com/.
[4] J. Bennett and S. Lanning. The Netflix Prize. In *The ACM SIGKDD Cup and Workshop*, 2007.
[5] J. R. Bult and T. Wansbeek. Optimal selection for direct mail. *Marketing Science*, 14(4), 1995.
[6] K. Church and B. Smyth. Understanding mobile information needs. In *Proceedings of MobileHCI*, 2008.
[7] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin. Diversity in smartphone usage. In *Proceedings of MobiSys*, 2010.
[8] G. Holmberg and M. Torrens. Musicstrands: A platform for discovering and exploring music. *University of Michigan Library*, 2005.
[9] J. Layton. How Pandora Radio works. 2006.
[10] D. Lemire and A. Maclachlan. Slope one predictors for online rating-based collaborative filtering. In *Proceedings of SIAM on Data Mining*, 2005.
[11] Why the mobile Web is disappointing end-users. Equation Research Report, Oct. 2009.
[12] S. Perez. Mobile app marketplace. ReadWriteWeb.com, Mar. 2010.
[13] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of WWW*, 2001.
[14] J. B. Schafer, J. Konstan, and J. Riedi. Recommender systems in E-commerce. In *Proceedings of the 1st ACM Conference on Electronic Commerce (ACM-EC)*, 1999.
[15] C. Shepard, C. Tossel, A. Rahmati, L. Zhong, and P. Kortum. LiveLab: Measuring wireless networks and smartphone users in the field. In *Proceedings of HotMetrics*, 2010.
[16] T. Sohn, K. A. Li, W. G. Griswold, and J. D. Hollan. A diary study of mobile information needs. In *Proceedings of CHI*, 2008.
[17] A. Toscher, M. Jahrer, and R. M. Bell. The BigChaos solution to the Netflix Grand Prize, 2009.
[18] W. Woerndl, C. Schueller, and R. Wojtech. A hybrid recommender system for context-aware recommendations of mobile applications. In *Proceedings of the IEEE ICDE*, 2007.