Energy Management Based On Charging Behavior Prediction

Prashanth Swaminathan
NESL, UCLA

Mikhael Semaan
NESL, UCLA

1. ABSTRACT

As mobile applications transition into providing richer content and supporting more use-cases, it has become increasingly apparent that smart-phones are constrained by battery life. Industry has put great effort into energy-aware hardware platforms, but little attention has been given to designing energy efficiency around user charging behavior. If smart-phones were to become capable of accurately predicting the time and duration of charging events of specific users, task scheduling could be more intelligently catered towards the user. Our project examines whether machine learning can be reliably implemented on smart-phones to infer aspects of charging behavior, as well as the appropriate classification schemes required to accurately predict future charging events.

By implementing SystemSens, we have developed an alternative model to offload model training to a server and provide user-specific models to the phone. The new application, Tree-Diagram, manages classification on the client-side and keeps track of prediction accuracy. Many server side scripts also do additional processing that produce attribute-relation files that we can analyze externally.

Using the machine learning suite Weka\(^1\), we found that the “Random Tree” classifier yielded the most accurate results initially. Later analysis showed that classifier strength varies across different users and over time as the datasets grow. We found that the prediction capability was strongest when using the attributes of time, battery level, charging status, and statistics of the most recent charging session. Our application utilized our classifier and predicted the time until the user’s next charge accurately, and without high-resource use.

2. BACKGROUND & IMPACT

The impact of smart-phones in recent years has permeated through several facets of life. Industry is tending towards creating the idealistic all-purpose device and smart-phones are ever-expanding their capabilities to match. However, as the amount of dynamic content and complex tasks delegated to smartphones increase, it has become increasingly apparent that these devices are largely constrained by battery life. As a result, the smartphone is at odds with advancing technology; while it is able to provide the necessary processing power, it cannot maintain a state of heavy use.

In the context of wireless health, this limitation poses a serious barrier. In light of the need of patient monitoring for medical diagnosis, wireless health has become an increasingly important field of research that attempts to integrate wireless technologies into the medicinal field. One of the fundamental developments that have arisen out of wireless health is the creation of Body Area Networks (BAN), an application of wireless sensor networks to the human body. By using small, low-power units that a patient could easily wear, researchers can collect data regarding body motion and the general physiological state of the subject. In recent years, the pervasiveness of smartphones in society has shifted this research to the Android platform so that researchers may develop applications that serve as the Gateway Sensor Node that is a feature of many wireless sensor networks in its other uses. The smartphone essentially serves as a liaison between the patient’s sensors and their medical provider’s network.

However, the use of a smartphone as this liaison causes concern on the effect on battery consumption. Wireless health applications that are geared towards sending crucial patient data must be capable of doing so in real time. Continuous transfer of data from the sensor network to the device and then the medical provider demands heavy use of the Bluetooth and Wi-Fi communication channels available on
the smartphone. Comparative studies have already shown that Wi-Fi especially has a high connection maintenance energy cost. This naturally leads us to a problem in the practical use of wireless health applications: users experiencing fast battery drain may let the phone power down and stop collecting data or simply uninstall the application altogether. Either case results in a loss of information to the doctor and hinders the diagnostic process.

Machine learning has great potential to solve this problem. While many efforts have been made to tackle battery efficiency by energy-aware hardware solutions, not much attention has been given to building energy management around a specific user. If machine learning can be utilized to accurately predict the charging behavior of the individual, a phone’s operating system could perform intelligent task management. More CPU-intensive tasks or tasks that require large data transfers could be scheduled to occur when the user charges their phone for longer periods of time. Applications could interface with the task manager to indicate the priority of their tasks to avoid battery drain.

3. RESEARCH QUESTION & GOALS

Our project set out to determine whether we can reliably predict future charging events by implementing machine learning on smart-phones. There are quite a few obstacles that stand between current machine learning and the final goal, but we are concerned with two of the more practical issues that face this research.

First is the issue of choosing a classifier. This project started with the assumption of some feature based pattern existing across all charging behavior patterns. That assumption was confirmed through our results, but our data set is only a small window into very diverse range of charging behaviors. It was unlikely from the outset that we would discover a perfect classification scheme, but it was this project’s intention to explore some of the schemes that perform well across the users we had accessible to us. We also wanted to delve into what features seem to be most descriptive of a user’s charging behavior. An established feature set and classifier will provide solid footing for future research to explore variations on classifiers as well as provide a baseline for additional statistical features.

The second, and more pressing issue, is whether classification can be done on the phone at all. While smart-phones are capable of performing classifications without too much resource use, the process of building a model is a poor fit for the Android platform. Machine learning relies on being fed large amounts of data and model training utilizes the CPU heavily. Both requirements are luxuries unavailable to the average user. Our project explores the method and architecture that can be used to achieve fast classification on smart-phones with limited resource use.

4. METHODOLOGY

In this section, we discuss the path through which we acquired data, decided on a classification scheme, and performed classification on the Android client. We also discuss the architecture of Tree-Diagram and the reasons for our design decisions. Our application utilizes SystemSens[^2], a researching tool developed by Dr. Hossein Falaki at UCLA’s Center for Embedded Network Systems. SystemSens was convenient for polling the phone’s sensors and met our requirements of low resource consumption.

4.1 GROUND WORK

In order to decide upon a classifier to build models on, we initially needed raw sensor data. SystemSens performed wonderfully for this purpose. We were able to direct SystemSens to populate a database on our local server from which we extracted JSON objects tagged as battery data. While SystemSens collects data from various other sensors available on the phone, we predicted that charging behavior would largely be dictated by the time and battery level. We make the assumption here that a majority of users will charge during specific times (i.e. at night) or for specific reasons (i.e. when battery level gets too low). A python script processed this data and converted this data to a format accepted by Weka, our chosen machine-learning suite. We discovered that the raw feature set provided by Android (as shown in Fig.1) was simply inadequate in predicting the status of the battery. Often accuracy was too low (below
60%) or depended on future values (the battery level at time of prediction). In order to remedy this, we expanded our feature set to include attributes that directly make predictions about future charging times.

<table>
<thead>
<tr>
<th>Old/Raw Feature Set</th>
<th>New Feature Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Battery Level</td>
<td>Battery Level</td>
</tr>
<tr>
<td>Charging Status</td>
<td>Charging Status</td>
</tr>
<tr>
<td>Timestamp (Unix)</td>
<td>Time (since beginning of week)</td>
</tr>
<tr>
<td>Temperature</td>
<td>Last Charging Time</td>
</tr>
<tr>
<td>Voltage</td>
<td>Last Charging Level</td>
</tr>
<tr>
<td>Battery Health</td>
<td>Duration of Last Charging Session</td>
</tr>
</tbody>
</table>

▲ Figure 1. Changes to original feature set provided by the battery sensors on Android.

Our new feature set assigned information to each instance about previous and future charging sessions in relation to the instance. Timestamp was converted to the time in relation to the start of the week to account for periodic behaviors. In addition, we removed the temperature, voltage and health features as they showed no variation or were very unlikely to affect charging behavior. Prediction of future charging behavior improved, but our choice of classifiers was too limited to make any concrete decision.

Since the prediction of time until a future charging session was numerical, many tree-based classifiers were out of reach. The feature list was once again modified to represent the time to next charge as a nominal value. By grouping future charging times into “quarters” with a granularity of 15 minutes, Weka was made to predict a nominal quarter number. After several tests comparing the new tree classifiers with previous classifiers, we settled on the Random Tree. The Random Tree works by selecting an arbitrary set of attributes from the full feature set and then splitting at each node based on each attribute. Since it managed to predict 98% of the instances correctly across multiple datasets, we tentatively held it as our chosen classifier as we proceeded to build the client-server model.

4.2 TRANSITIONING FROM SYSTEMSENS TO TREE-DIAGRAM

The application was built as a specialized tool around SystemSens, to address some of the problems surrounding machine-learning on phones. While many machine-learning suites like Weka are designed with Java, attempting to collect and train classifiers completely on a smartphone is extremely taxing on its resources. Since high resource use would obviously be counter-productive to our goal of battery efficiency, the client-server architecture emphasizes delegating complex tasks to the server.

4.2.1 A BRIEF OVERVIEW OF SYSTEMSENS

The SystemSens client is an Android service that runs continuously in the background. On its invocation, it registers broadcast receivers to catch intents that Android releases on sensor events. These compose the event-based sensors on Android and include sensors that generally do not change state at high frequencies. For instance, Android broadcasts an intent whenever the battery level decreases or increases and when the phone is plugged in. SystemSens also uses the Alarm Manager class to poll other sensors on a fixed interval. These polling sensors change too frequently to broadcast, so they are accessed through Manager classes.

Every recorded event is recorded in a unique JSON format and then passed to a database adaptor. The adaptor rewraps it in a standard JSON indicating the record type as well as the timestamp at which it was recorded. Records are stored in the database until the phone is plugged in, at which point SystemSens attempts to upload the data to a server. Successfully uploaded data will be flushed from the database. The server side will receive these JSON objects from the phone and will place their contents in an SQL database. Record data will remain in their unique JSON formats.

4.2.2 USER INTERFACE

While SystemSens was very useful in data collection, integrating it into a larger application proved difficult. As part of its principle of keeping a low profile, SystemSens does not provide any user interface. Errors in
communication to the server or mishandling of data simply caused the service to crash, requiring a full reinstall to start again. Our initial user interface remedied this problem by allowing the user to toggle the state of SystemSens. While turning the service off seemed to be an unwanted feature for data collection, it provided some insight into how sensor polling should be handled in the future. Phones can be completely powered down, so it was necessary to see how gaps in data collection would affect the training of a model, as well as how to creatively fill those gaps with virtual sensor data.

Additions to the user interface appeared as more interaction options for developers and users. SystemSens was not created with customization in mind and several important actions (especially the posting of sensor data to the server) were done only in specific conditions and at a very slow rate. We included a buttons on the UI to force the service to upload the phone data and indicate the number of records still held by the service. Later additions sent requests to the server to update the current classification model and classify random instances to test the strength of the model.

The ultimate goal of the user interface was to push towards a more specialized tool for modeling research. We wanted to produce an application that researchers could easily test without having to interact with the phone through an separate interface like Eclipse, because actions taken by developers (like plugging in the phone for debugging) weaken the integrity of the collected data.

![Figure 2. User interface and general architecture of Tree-Diagram (adapted from SystemSens).](image)

### 4.2.3 CLIENT-SERVER ARCHITECTURE

The greatest modification to SystemSens is the inclusion of a model handling class. Acting as a helper class to SystemSens, the Model Manager class periodically performs analysis on the predicted times to the next charge with the actual ground truth observations. If the error of the model exceeds a preset threshold, it is invalidated and a request is sent to the server for an updated model. This offloading of model training to the server is advantageous to the primary goal. It eliminates the need for the phone to keep its records as well as avoid performing very CPU-intensive tasks.

The server-side code that handles model creation also adheres to the principle of prioritizing the client. Connections to the server are maintained for very short amounts of time. After a request to update the model is sent, the server starts a thread to run python scripts to convert the SQL table data to ARFF and then feeds that file to a Weka model training program. The process of creating this model can take two to three minutes on about a week’s worth of data and up to nine minutes for data that spans a month. Wi-Fi expends power to maintain a connection but is efficient for large throughputs of data. Simply letting the connection hang for several minutes while model creation occurs is a poor model for our needs. This is the motivation for running these processes as a separate thread, while the server immediately responds with a confirmation message. The client's Model Manager will close the connection and wait for some time before another attempt.
The Model Manager also is responsible for the most important step: classifying data in real time. Model files returned from the server are written to a directory on the SD card and are then loaded into a classifier. When battery states change, SystemSens requests Model Manager to classify the instance and stores the prediction for later analysis.

### 4.2.4 MODIFICATIONS TO WEKA

On the first attempt to return a Random Tree model from the server to the Android client, we discovered an inconsistency with method of serialization between the Java Virtual Machine (JVM) and Dalvik, the virtual machine upon which Android is based. Mismatching between the Serial Version UID on these platforms rendered the Android client unable to deserialize the model and use it for classification. As Weka is an open source tool, we were able to address this problem by adding our own methods to the Random Tree class. The Random Tree class was fitted with a recursive method to create a JSON representation of the model. The structures of the JSON and Java tree node are shown in Fig. 3.

![Figure 3. JSON tree (left) and TreeNode class in Model Manager (right)](image)

This solution actually led us to an interesting design choice. JSON trees, while requiring some extra implementation on the server and client to process, provided many benefits in terms of resources. The tree structure meant including the Weka libraries on the phone was no longer required and the removal of unnecessary header information and model statistics cut the size of model files in half. Classification speeds also improved by 3 - 4 milliseconds.

### 4.3 FINAL STEPS

The goal of implementing the Tree-Diagram application was to show that machine-learning could indeed be reliably used on a smartphone without causing battery drain. However, there is still the matter of how accurately it could be able to predict user behavior. As Tree-Diagram went through multiple iterations, there are very few days of data we acquired of classification done on the phone. In order to verify that the chosen classifier was still performing strongly, we created the model again from Weka on a local machine and packed it into a JSON. After confirming that the tree structures in the phone and the locally generated JSON were the same, we ran cross-validation tests to observe the accuracy of the model. Random Tree was significantly less accurate (91%) but observing the incorrectly classified instances showed us that predictions were often only off by one interval. This meant that the ground truth was occasionally within 15 minutes of our predicted time.

A naïve check of the phone’s classifications also showed that the machine-learning had discovered patterns in our charging behavior. Since a large portion of lab work required observing data stored on the phone, the model had discovered that our phone was charged quite often in the middle of the day. Classifications near the late evening predicted that the future charging time would be around midday tomorrow, which matched our expectations.

There was a strong likelihood that future researchers may want to test customized classifiers, so implementation was also added on the server side to allow quick swapping of classifier builders. Some other extra features were added to the Python scripts on the server for the addition of more features. This will prove essential when classifier research in this area looks toward finding patterns in other sensor data like CPU usage and network state.
5. RESULTS

At the completion of our research venture, we have compiled an application and framework model for deploying machine learning on phones, as well as a short study into the performance of different classifiers. The source code for the Android client, server model handling, and modified Weka files will all be made available to other researchers.

5.1 TREE-DIAGRAM

The created application Tree-Diagram is primarily a tool for future research in this direction to allow researchers convenient methods to test classifier models. We wanted to emphasize customizability as much as we could so that classifiers could be easily switched out and data could be quickly extracted from the SQL database for analysis on local machines. Depending on future needs, much of the structure of the workflow was designed to make it easy to extract new features and quickly add/remove elements. Further documentation is also included that highlights the general design for systems that wish to perform low-resource classification on phones.

5.2 CLASSIFICATION

At the beginning of this research, our lack of data put us on unstable grounds when it came to choosing a classifier. Multiple iterations of cross-validation tests showed that the Random Tree classifier was consistently accurate and had a fairly short build time. Unsurprisingly, after collecting several weeks of data, the best-performing classifier had changed. We found that the Lazy IBk classifier suddenly began performing with similar accuracy levels and built a model several times faster. The results of these accuracy tests on the most recent data sets are summarized in Fig. 4 below.

<table>
<thead>
<tr>
<th>Classifier</th>
<th>Raw Feature Set</th>
<th>New Feature Set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Model Size (MB)</td>
<td>Build Time (s)</td>
</tr>
<tr>
<td>Lazy.IBk</td>
<td>1.1</td>
<td>0.01</td>
</tr>
<tr>
<td>Trees.J48</td>
<td>70.2</td>
<td>3.06</td>
</tr>
<tr>
<td>Trees.DecisionStump</td>
<td>0.06</td>
<td>0.35</td>
</tr>
<tr>
<td>Trees.REPTree</td>
<td>62.6</td>
<td>1.76</td>
</tr>
<tr>
<td>Trees.RandomTree</td>
<td>31.3</td>
<td>1.8</td>
</tr>
</tbody>
</table>

▲Figure 4. Table comparing accuracies of old and new features performance, speed and size on various classifiers.

This figure also draws attention to the model size in megabytes. Size did not present itself as an issue of great importance at the beginning of the project, but posed a serious difficulty when transferring the model from the server to the Android client. Some classifiers create an immense series of complicated rules to perform classifications, while others rely on copying entire sets of instances. These types of models, such as REPTree and J48, were fairly accurate but over 60 MB in size. Transfers took several minutes and occasionally were aborted due to lack of space on the device. The space limitation became a necessary consideration for future models.

This discovery re-emphasizes the decision to compose tree-based models into JSON tree objects. Generating a JSON tree took negligible extra time and brought the model size to slightly less than 0.6 MB. The JSON representation of tree-based models can be up to 40 times smaller than the model file generated by Weka. This is likely attributed to the erasure of header, statistical information and proprietary formatting done by Weka.

6. DISCUSSION & FUTURE WORK

The research on classifiers that can accurately model user charging behavior is still at its beginning stages. We have discovered that optimal classifiers vary depending on the user and may change over time. It should be put into consideration whether multiple classifiers could be used as a means of
predicting charging behavior. There is a vast opportunity for future work in this portion of the research. Applications like Tree-Diagram should be tested across a much wider variety of users to get a more realistic understanding of the various patterns by which phones are charged. We would then use this data to build a more reliable classifier, as well as examine whether the feature set we chose really does represent the factors that affect charging. The feature set may also need to be expanded to account for their habits based on location (i.e. charging is done only at home) or activity (i.e. charging is done after gaming when the phone has lost battery drastically).

The ultimate step, after a classifier can perform reasonably well on most users, is to integrate a Tree-Diagram-like structure into the operating system. Such a component would likely sit between the application and the application framework layer, or be integrated directly into the OS task manager. Applications can provide the OS with a notion of priority, indicating events that can be delayed until more suitable conditions are met. The strength of this approach in wireless health would allow a smart-phone to send data only when real-time data is requested or in emergency situations. Implementing machine learning to gain a better awareness of how the user interacts with their phone is truly the next step in improving optimization and efficiency.

7. CONCLUSION

In culmination, the goals of this research were partially met. We have succeeded in discovering a practical way of performing classification on phones. Our application, Tree-Diagram, solves the barriers of machine learning’s large data and resource requirement by utilizing a server. Data storage and model creation is completely delegated to the server, while less intensive tasks such as the actual classification step is done by the phone. The compression of large models into the JSON format has been found to increase classification speeds and makes use decision tree models more feasible to send and store on the device.

Classification done using the Random Tree model performs well across several users, but more recent analysis suggests exploration into other classifiers. Nevertheless, the performance of other classifiers has been strong enough to indicate that there is indeed a pattern amongst the average charging behavior of users. While there are many other features that could provide a more accurate inference, we’ve found that the time of day, battery level, and various statistics about previous charging events are strong indicators of future behavior.

8. ACKNOWLEDGMENTS

We are grateful to the members of the Networked & Embedded Systems Laboratory who provided us guidance we needed to complete this project: Chenguang Shen and Kevin Ting. Prof. Mani Srivastava provided us with the vision and goals for the duration of our research. The Wireless Health Institute contributed the funds to this program, without which we would not have had this wonderful opportunity. We sincerely thank them all.

9. REFERENCES


