Real-Time Classification of Everyday Fitness Activities on Windows Mobile

By

Alireza Bagheri Garakani

A senior thesis submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

With Departmental Honors

Computer Science & Engineering

University of Washington

June 2009

Presentation of work given on ______________________________

Thesis and presentation approved by ______________________________

Date ______________________________

Abstract

Human activity recognition enables ubiquitous computing applications to leverage knowledge of people's context. The ability to distinguish a small set of activities has already shown the capacity to enable a wide range of important applications that integrate with fitness/healthcare [1], *environmental sustainability* [2], *and entertainment [5]. However, the state-of-the-art in activity recognition is custom hardware that can be expensive to build, difficult to maintain, and an additional item for people to wear or carry around. Mobile phones provide an opportunity to augment an existing device that people already carry with them throughout their daily lives. In our work, we have implemented an activity classification system on the Windows Mobile OS paired with the HTC Touch Diamond mobile phone. Our system uses a three-axis accelerometer to classify human activities including running, walking, bicycling, and sitting. It achieves considerable accuracy and a day's battery life.*

Introduction / Motivations

Computing is growing ever more ubiquitous and context-aware. Systems falling into this category are those that are capable of communicating among each other and that are readily aware of their surroundings, including any individuals that may be present within these surroundings. One example is a system that interprets an individual's geographic location during call initiation and connects the line to the appropriate phone number (whether work, mobile, or landline) given personalized settings [3]. An important point to observe is that although we may have once considered phone communication and GPS as two distinct processes, this line of separation between these two processes is becoming increasingly blurred and information is becoming more readily accessible among many distributed devices.

Human activity classification is an important advancement in these efforts. Such systems provide the capability to interpret a user's activities with the use of sensors, which are commonly distributed as a set of several sensors of various types (accelerometer, pressure, GPS, RFID, etc). Undoubtedly, the future of intelligent computing will need to rely, at least partially, on a system's ability to correctly classify a user's activity and surroundings. Consider a system that is capable of adjusting room lighting based solely on a user's activities that it interprets – perhaps upon sitting at ones desk the light would center around the desk, upon sleeping the lights would turn off, upon waking up the lights would gradually turn on, and upon dancing the lights would begin to strobe with different colors. Once provided with the correct activity, such lighting operations could easily be set to perform in response to each of the activities just described. The *means* of detecting these activities (and ideally a much larger set of common activities), however, remains a difficult problem to tackle.

A preliminary step in advancing such systems is considering a limited domain of possible activities, perhaps composed of only a couple distinct values. This is an area that has thus far seen much more success and, despite its domain-limited capabilities, it can be a part of many important applications when integrated with areas such as: fitness/healthcare where individuals are encouraged to engage in fitness activities [1], environmental sustainability where individuals are encouraged to reduce use of public transportation [2], as well as for entertainment purposes such as the Mario Fit application that allows gameplay based on physical activity in lieu of a handheld controller [4]. This lattermost application of entertainment can also be viewed as encouragement of the same fitness/health benefits due to its interactivity [5].

Although certain custom hardware devices are made specifically for inferring activities, they are typically expensive to build, difficult to maintain, and an additional item for people to wear or carry around. It is thus no surprise that such efforts have not reached a widespread audience, despite their proven feasibility and obvious potential. Better hardware must be realized for activity inference that remedies the inadequacies of these custom-made devices. In the case of activity inference, and especially for long-term applications, it is further reasonable to consider a device that can at most times be assumed to be carried around by the user. As such, mobile phones appear to be a plausible solution for the choice of hardware. The internal accelerometer and GPS devices, which are increasingly common as built-in features of these phones, appropriately provide a plausible means to infer activity.

This paper discusses the implementation of real-time activity inference on the Windows Mobile platform using the HTC Touch Diamond mobile phone. Our system, which is based on similar findings and implementation on the iPhone [4], is capable of accurately classifying trained human activities, which can (as will be the case within this paper) consist of running, walking, sitting, and biking. Activity models are learned using a collection of features that are calculated based on samples from the 3-axis accelerometer. These models then serve as a means for recognizing new activities based on data that is collected in real-time.

I will begin by discussing some background work that was the basis of our motivation and by addressing the closely related solution developed for the iPhone. I will then briefly overview the three stages of our approach, which include logging data, creating models, and classifying activities. Afterwards, I will discuss in greater detail our design choices and architecture, as well as some challenges we faced. I will then provide results and a discussion of the accuracy and robustness of our solution based on several experiments. Finally, I will briefly discuss some possible paths for future work.

Background / Related Work

As previously mentioned, activity inference systems that distinguish between a few discrete values have already been successful in the past, although with specialized hardware. A subset of these systems include the MSP belt-worn sensing device [6], the eWatch wrist device [7], and the Nike+iPod Sport Kit [8] seen respectively in Figures 1-3. All three systems mentioned here can classify activities similar to those which I propose within this paper – running, walking, standing, etc.

Figure 1
An attachable belt-worn sensor used as part of the MSP system

Although all of these systems support activity inference, they have done so with a dependence on custom-made hardware. This is a problem because these devices are typically expensive to build and difficult to maintain. Perhaps more importantly, these are items that must be worn in *addition* to others that the user may be already wearing. In some cases this may also mean having to carry around a device that is awkward in appearance and/or placement.

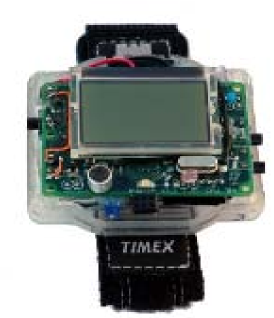


Figure 2
A sensor designed to appear like a common watch as used in the eWatch system [8]

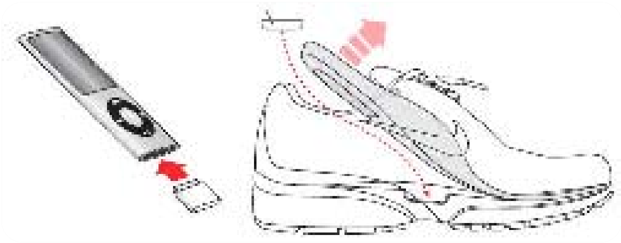


Figure 3
The Nike+ iPod sensor is placed in a person's shoe and communicates wireless to their iPod [9]

These limitations were recognized at the University of Washington and an effort to implement activity recognition on a mobile device was undertaken by T. Scott Saponas, Jonathan Lester, Jon Froehlich, James Fogarty, and James Landay. This effort was successfully implemented on the iPhone mobile device, making use of the device's three-axis accelerometer. Their goal was based on the idea that "utilizing commodity devices for activity inference provides researchers with access to robust, readily-available hardware and potentially large preexisting user bases" [4]. Furthermore, to reinforce this claim, they provided a set of open source tools for collecting, building, and running activity models that could be incorporated into future application development.

As accelerometers become an increasingly popular component of mobile devices, it will be necessary to expand this idea to other platforms in addition to the iPhone. Although work is already being done to expand this idea to the Android platform, the issue has not yet been addressed for the Windows Mobile platform, which is used by many mobile devices on the market and is capable of supporting the very same activity inference techniques. In order to further pursue the objective of providing activity inference capabilities based on readily-available hardware and to encourage a potential for even larger user bases for researchers, it will be necessary to port this capability onto the Windows Mobile platform.

Platform Overview

The activity classification system consists of three separate phases, each of which represents a major step in the entire process. For sake of compatibility and consistency, these three stages are based on the same design decisions made by the prior initiative for the iPhone. I will briefly discuss each of the three phases below and save discussion of specific design and architectural components for the next section. It should be noted that the interface designs are purely for functional purposes and only demonstrate a simple use of the API that it is built upon; this point is further emphasized below.

Logger Component

The logger component is used to collect data that is later used for training a model. This data is represented in the form of raw data and feature calculations, which are discussed in greater detail within the next section. This user interface, which is shown in Figure 4, consists of a set of simple buttons that represent each of the possible activities that the model will be trained on. It should be noted that, although this particular interface contains only four distinct activities, there is certainly no limitation in using additional activities. Accordingly, a different representation, such as drop-down menus, could be used in order to accommodate a larger set of activities.

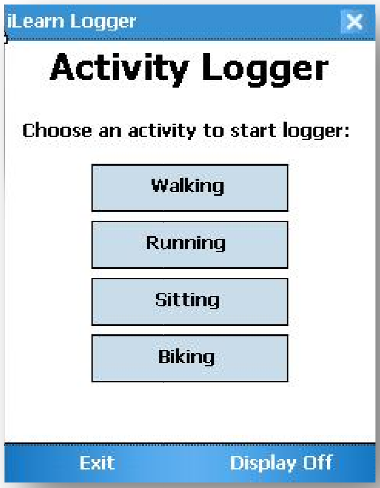


Figure 4
Our activity logger supports the few chosen activities shown above, however, support can easily be extended to other activities.

Modeling Component

The modeling component is used to transform the data collected from the logger into a model file that can be used for classification. More specifically, it uses a collection of 72 features that are calculated based on logger data to construct a Naïve Bayesian classifier, which was the machine learning algorithm of choice because it is computationally inexpensive and thus "allows a potentially large number of applications on a mobile phone to each classify among a different set of activities simultaneously in real-time" [4]. Unlike the other two phases (logger and classifier), the modeling phase is a desktop application and, thus, it does not run on the mobile device. Because a similar modeling tool was built from the iPhone initiative (called iModel) [4], there was no need to re-define this component as it runs on a desktop and is therefore compatible with our solution. This tool is a Java application built on the Weka machine learning toolkit. In addition to creating models of data, it contains other features such as hold-one-out tests, which "are useful for validating that models built with data from several people correctly classify data from a new person" [4]. This hold-one-out test will be used to evaluate the accuracy of our system within the "Evaluation and Discussion" section of this paper.

Classification Component

The real-time classification component is responsible for reading raw sensor values and using features calculated from these raw values in conjunction with the model constructed in the previous phase to produce an inferred activity. The API is constructed such that applications can request event notifications when new activities are classified. The user interface of the application, seen in Figure 5, shows two separate fields for classification updates – one for instant updates and one for smooth updates. The instant updates represent each individual classification that is made, whereas the smooth updates represent the most common individual classification over an adjustable interval of time. This smooth activity is optionally uploaded to a web server that displays a history of previous classifications by day and constructs a visual graph, as shown in Figure 6. Also, as was mentioned for the logger, the interface demonstrates the primary functionalities that the API offers but it is certainly not restricted to the format shown in the figure. Finally,

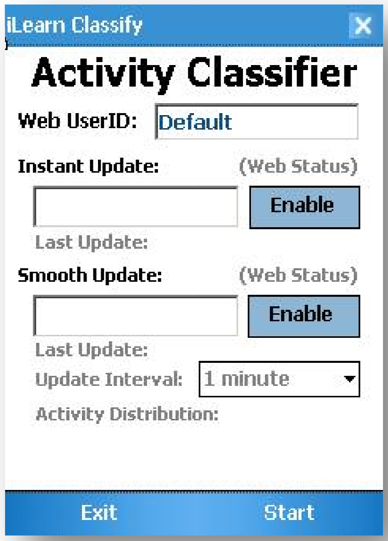

iLearn Classify
Activity Classifier
Web UserID: Default
Instant Update: (Web Status)
Last Update:
Smooth Update: (Web Status)
Last Update:
Update Interval: 1 minute
Activity Distribution:

Figure 5
Smooth activities can also be calculated using several different update intervals, as shown above.

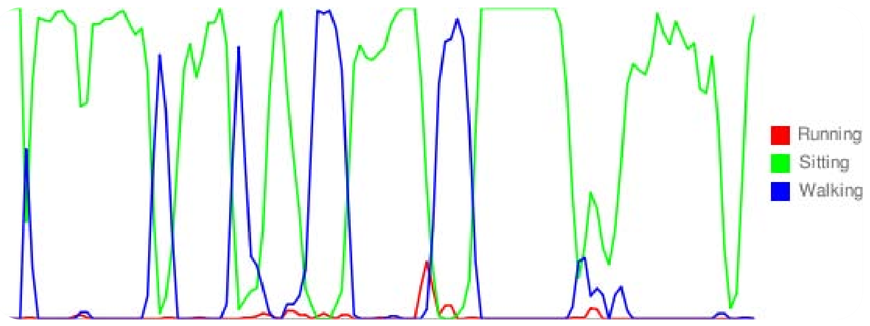


Figure 6
Data uploaded to the web server can be used in many ways, one of which includes creating a graph representation.

there are several additional parameters that can change the effects of activity classification; these variables (such as the smoothing factor, raw data re-use, etc.) are discussed in more depth within later sections.

Design Considerations and Architecture

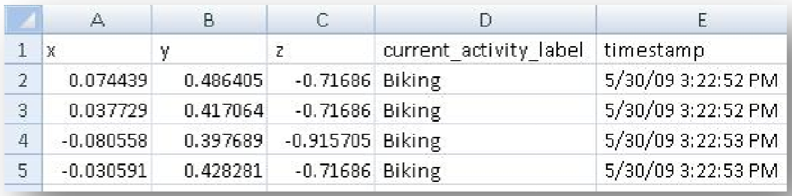
To better present the inner workings of our system, I will first discuss in greater detail the input, output, and processes that are involved within the logger and classification components. Secondly, I will discuss three significant challenges that arose in our transition to the Windows Mobile platform, including the (1) accelerometer usage and limitations, (2) feature calculation and activity extraction through P/Invoke methods, and (3) locking the screen and battery management.

Logger Component - Revisited

Our solution contains two individual loggers (the accelerometer and the GPS) and provides support for easy expansion to other sensors with the use of high-level abstract classes. These abstract classes are denoted as *SensorLogger* and *PollingSensorLogger*; the former is designed for implementations where an event notification from the device is expected when new data is available (as in the GPS device), while the latter includes a timer object to poll the device at some constant interval (as in the accelerometer device). Each logger operates on logger sensor events that are specific to that logger and that are used to require certain types of data that are specific to that sensor. This process will be made more concrete shortly. This sensor logger is abstracted from the high-level *SensorEvent* class and implemented by each sensor.

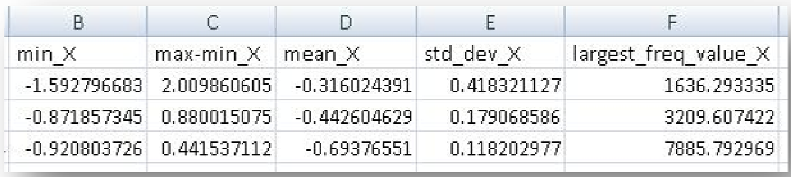
The accelerometer logging components are *AccelerometerLogger* and *AccelerometerEvent*, which conform to the description of the API just provided. With each new accelerometer sample obtained via a periodic timer, a new *AccelerometerEvent* is created and added to an output buffer using the high-level *AddSensorEvent* method. *AccelerometerEvent* contains: the raw data values for each axis, the activity label indicated by the client, and a timestamp – this is shown in Figure 7.

Because we want to also make available the features based on this raw data, the *AccelerometerLogger* contains functionality to output a second file designated for feature computation results. This process is implemented as a separate thread within the *AccelerometerLogger* class. The format of this data, whose derivation and individual components are discussed later, contains a fairly extensive set of values; a portion of this output is shown in Figure 8.



	A	B	C	D	E
1	x	y	z	current_activity_label	timestamp
2	0.074439	0.486405	-0.71686	Biking	5/30/09 3:22:52 PM
3	0.037729	0.417064	-0.71686	Biking	5/30/09 3:22:52 PM
4	-0.080558	0.397689	-0.915705	Biking	5/30/09 3:22:53 PM
5	-0.030591	0.428281	-0.71686	Biking	5/30/09 3:22:53 PM

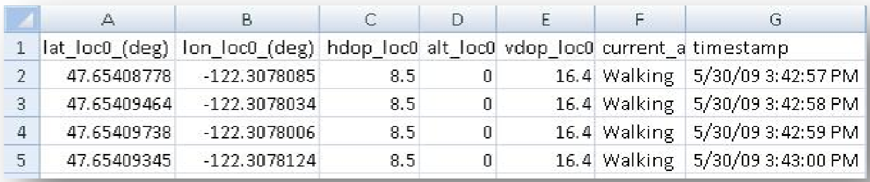
Figure 7
Raw values of the accelerometer are based on units of gravity (i.e., placing the device on a table top will show a magnitude of 1 for the up-and-down axis)



B	C	D	E	F
min_X	max-min_X	mean_X	std_dev_X	largest_freq_value_X
-1.592796683	2.009860605	-0.316024391	0.418321127	1636.293335
-0.871857345	0.880015075	-0.442604629	0.179068586	3209.607422
-0.920803726	0.441537112	-0.69376551	0.118202977	7885.792969

Figure 8
For each entry there is a total of 72 features, a subset of which are shown above.

The GPS logging components, *GpsLogger* and *GpsEvent*, function in much of the same way as the accelerometer. In particular, new data is sampled whenever an event is thrown (as opposed to using a periodic timer) and there is no feature computation that is used. The *GpsEvent* contains: latitudinal and longitudinal values, vertical and horizontal DOP values, the sea-level altitude, the user-specified activity, and a timestamp – this is shown in Figure 9.

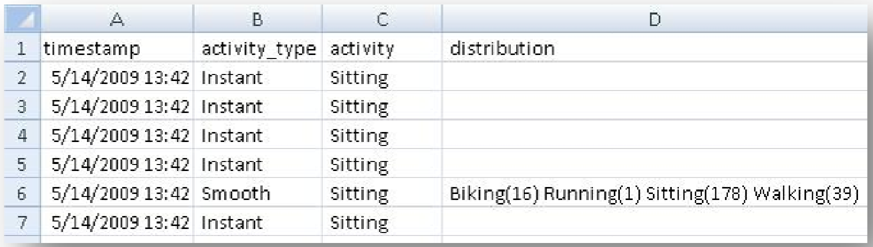


	A	B	C	D	E	F	G
1	lat_loc0_(deg)	lon_loc0_(deg)	hdop_loc0	alt_loc0	vdop_loc0	current_a	timestamp
2	47.65408778	-122.3078085	8.5	0	16.4	Walking	5/30/09 3:42:57 PM
3	47.65409464	-122.3078034	8.5	0	16.4	Walking	5/30/09 3:42:58 PM
4	47.65409738	-122.3078006	8.5	0	16.4	Walking	5/30/09 3:42:59 PM
5	47.65409345	-122.3078124	8.5	0	16.4	Walking	5/30/09 3:43:00 PM

Figure 9
Raw GPS values are collected directly from the device, similar to the accelerometer, and calculation were necessary.

Classification Component - Revisited

The classification procedure shares a similar hierarchical structure as the loggers, in that it implements the *SensorLogger* and *SensorEvent* abstract classes, except here the inferred activities are being logged to file. The events that carry this classification data contain: a timestamp of the classification, specification of either instant or smooth activity type, the classified activity label, and the distribution (which is non-empty only for smooth classification) – this is shown in Figure 10.



	A	B	C	D
1	timestamp	activity_type	activity	distribution
2	5/14/2009 13:42	Instant	Sitting	
3	5/14/2009 13:42	Instant	Sitting	
4	5/14/2009 13:42	Instant	Sitting	
5	5/14/2009 13:42	Instant	Sitting	
6	5/14/2009 13:42	Smooth	Sitting	Biking(16) Running(1) Sitting(178) Walking(39)
7	5/14/2009 13:42	Instant	Sitting	

Figure 10
The classification log file is very simple. For Smooth activities, as the one shown above, the distribution of instant activities is provided in the right-most field.

Unlike the logging tool, the classification tool contains several options that can alter how activities are classified. Firstly, it will be necessary to readdress the distinction between instant or smooth classification updates. Instant updates represent each individual classification that is made. Depending on the amount of data that is used per feature calculation (see later discussion within this section), instant activities are updated at different intervals. However, assuming the default settings of 128 data entries used per feature calculation and a 25 Hz accelerometer sampling rate, new instant activities are classified about every 5 seconds.

Smooth activities, on the other hand, consist of taking the activity with the majority count within a collection of consecutive instant activities. This idea is motivated by the fact that people do not change their activities every 5 seconds, but instead do so over larger intervals of time. Therefore, it would appear that smooth activities are more applicable for most practical purposes. A benefit of smooth classification is that it is not hurt by a few false classifications and can thus provide better overall accuracy. Our approach also provides a smooth transition of activities by using a half-blend technique where only the first half of the collection set is removed after each classification (as opposed to the entire collection). A configurable option for smooth classification is the window size, the number of consecutive instant activities to consider. To illustrate this concept, the smallest window size represents each instant activity itself; on the other hand, a very large window size, say the length of one day, represents the most frequently occurring activity within the timeframe of the last 24 hours.

One additional configuration option that pertains to both instant and smooth updates is the amount of accelerometer raw data re-usage through blending, which has the ability to encourage a higher rate of classification by some constant factor. Because the variable representing the amount of raw data per feature calculation must remain constant from the logging phase to the classification phase in order to maintain a consistent number of feature frequency values (as discussed later in this section), the only approach for increasing the classification rate is to re-use data in each FFT calculation. This allows the classification tool to compute features at a higher rate than the logger, if desired. To illustrate this point, assuming the default settings of 128 data entries used per feature calculation and a 25 Hz accelerometer sampling rate, new instant activities are updated every 5 seconds when using a data-reuse factor of 1 and every 2-3 seconds using a data-reuse factor of 2. Accordingly, smooth classifications contain many more instant activities to consider and this has the potential to improve classification results.

It should be mentioned that instant and smooth activities are both implemented in our approach by default. To make use of either (or both) activity types, it is only necessary to request notifications for each corresponding event.

Accelerometer Usage and Limitations

The built-in accelerometer of the mobile phone provided several challenges, including accessibility of the data and a limitation on the polling frequency. As is common for such devices in mobile environments, tools to access and control the accelerometer were not readily provided by the manufacturer. Luckily, as others too had encountered such restrictions, there were many resources and workarounds available online. Based on work started by Scott Seligman and later expanded by Koushik Dutta [10, 11], the raw data values from the accelerometer could be accessed in C# or C/C++ by calling the native functions of the mobile device. The second challenge was much more substantial. After experimenting with the device, it became clear that there was a 25 Hz limit on the frequency at which new values could be requested from the accelerometer. Experimentation showed that with higher requests values would begin to repeat for consecutive entries and at other times no entry would even exist. After several failed attempts to increase this limitation via different built-in C# polling techniques (e.g., using *System.Timers.Timer*, *System.Threading.Timer*, a combination of the *DateTime* structure & *Thread.Sleep* method, etc.), I considered the option of polling within unmanaged C++ code and then making some buffered collection of this data accessible to C# via P/Invoke methods. This was motivated by the fact that crossing the boundary between managed and unmanaged code can have negative consequences on performance speed. Thus, it was hoped that higher polling frequencies could be achieved by crossing this boundary once for each collection of 120 data values (instead of once for each data value). The results of this approach, however, proved closely comparable with the 25 Hz limit. A final attempt was made by modifying various registry values of the device to attempt to achieve greater frequency. Despite locating some promising registry keys, there was no meaningful increase to the frequency limit after changes were made.

The desire to exceed this limit was based on the 120 Hz polling frequency that was used on the iPhone implementation. Thus, it was understood that this would provide better results. It would turn out, however, that sampling at 25 Hz was sufficient for our needs. This sufficiency can be assessed within the 'Evaluation and Discussion' section of this paper, where we also compare with results from the iPhone implementation.

Feature Calculation & Activity Extraction through P/Invoke

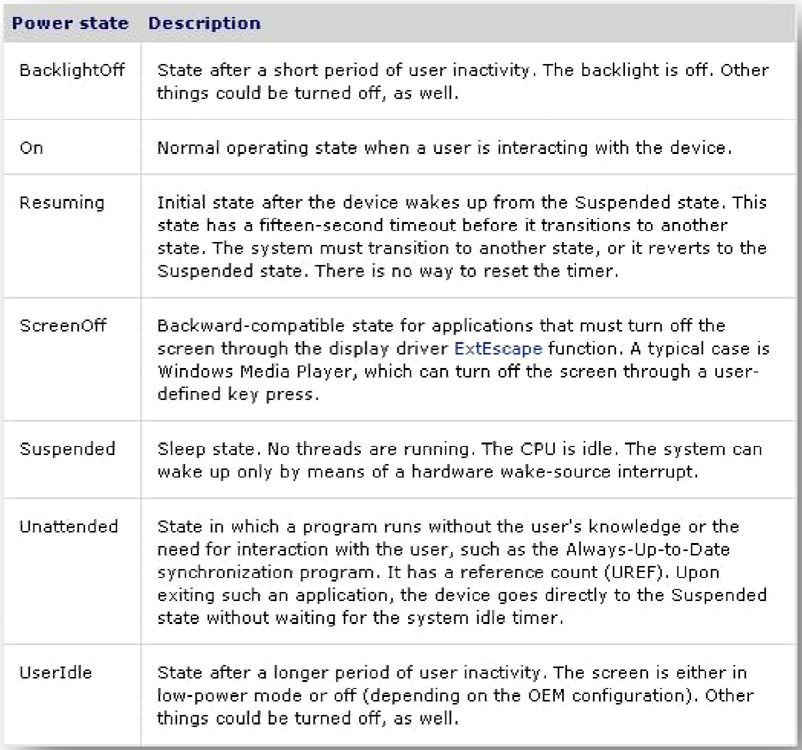
Models learned within the second phase of our system are based on the accelerometer feature calculations. It should be noted, however, that our implementation is compatible with any feature choice, whether related to the accelerometer or not. Each feature computation is based on 128 raw accelerometer entries, by default. For each calculation, we identified 24 distinct features per axis, for a total of 72 features. For each axis, features can be categorized into two categories – magnitude features and frequency features. The former consists of 5 features and includes the mean, standard deviation, minimum value, maximum value, and max-minus-min value. The latter consists of 19 features and is the result of computing a 256 point Discrete Fourier Transform (DFT), which includes "one feature for the energy in each of the first 10 frequency components, a feature for the energy in each band of 10 frequency components, the value of the largest frequency component, and the index of the largest frequency component" [4]. These DFT values are capable of showing the periodic sensor changes that occur when performing activities, and it makes this information explicit by converting its representation from the time domain to the frequency domain. Figure 8 shows a glimpse of this combination of both feature types together within a log file. Finally, activity extraction is based on the Naïve Bayes model that is created in the modeling phase in conjunction with the real-time feature calculations that are made during the classification phase.

Feature choice and methods for calculating them, as well as activity extraction techniques, are not platform-specific concepts and are thus taken directly from the solution for the iPhone. As such, it was not necessary to reconsider or re-implement these components. Because feature calculation and activity extraction are based on C/C++ code, however, it was necessary to export this functionality for use within our C# environment. As such, it was necessary to call native code (C/C++) from managed code (C#), which means using P/Invoke (Platform Invocation Services) techniques in C# to call unmanaged DLL files compiled in C/C++. This, however, is a lot easier said than done. While there are endless resources and examples online, there are many limitations to the types of data that can cross this boundary. Different workarounds were needed in each particular instance. For example, if we wanted to access a global variable within native code, these variables along with any needed functions had to be encapsulated within a DLL-exported C++ class. Additionally, this required DLL-exported C++ methods to return a pointer to a new instance of the class, to call specific class methods when given a class pointer, and to return the content of any global variables when given a class pointer. Essentially, within C# a pointer to a particular instantiation of the C++ class is used whenever referencing the native object's methods or variables – this technique was used for activity inference (see project *libClassify*). In another case, P/Invoke was necessary to access the C/C++ KissFFT library, which was used for calculating the frequency features (see project *libKissFFT*).

Locking Screen and Battery Management

It was known from the beginning that locking the screen would be necessary for these mobile-based applications – both for the logging tool and the classification tool. After experimentation it became clear that some period of time after locking the device (either explicitly via the pressing the power button or after some time of inactivity), Pocket PCs (which our particular device is) will transition into a system state that disables access to the accelerometer device and does not allow certain applications to run in the background. Based on previous work by Jon Froehlich and various other resources online, it became clear that the proper solution to keeping this device and our application available was to periodically call *SystemIdleTimerReset*, a windows native function to explicitly reset the timer that is used to notify the system when to transition into a suspended state. Furthermore, to insure that the internal accelerometer device is kept active for the duration it is needed, the native function *SetPowerRequirement* was used to explicitly indicate this device and to set the desired ON power setting associated with it.

Although the accelerometer device was now accessible to the application the entire time, a new problem had emerged; the device was constantly in a state that kept the device screen powered on (though it should be noted that this is different from the back light, which would automatically turn off). This had two negative consequences; firstly, this would mean the device screen would also be unlocked and thus the client could not safely put the device within their pockets; secondly, this system state would certainly drain the battery power quickly and reduce the applicability of this solution as a whole. To solve this problem, native function *SetSystemPowerState* was used to instruct the device to transition into the more appropriate UNATTENDED state (as opposed to SUSPENDED state), which allows applications and devices to run in the background with the screen off and locked. Figure 11 gives a more detailed description of each system state [11]. This system power state is forced when either it is instructed to do so by the UI via the "Display Off" button or some specified time after transitioning to the BACKLIGHTOFF state.



Power state	Description
BacklightOff	State after a short period of user inactivity. The backlight is off. Other things could be turned off, as well.
On	Normal operating state when a user is interacting with the device.
Resuming	Initial state after the device wakes up from the Suspended state. This state has a fifteen-second timeout before it transitions to another state. The system must transition to another state, or it reverts to the Suspended state. There is no way to reset the timer.
ScreenOff	Backward-compatible state for applications that must turn off the screen through the display driver ExtEscape function. A typical case is Windows Media Player, which can turn off the screen through a user-defined key press.
Suspended	Sleep state. No threads are running. The CPU is idle. The system can wake up only by means of a hardware wake-source interrupt.
Unattended	State in which a program runs without the user's knowledge or the need for interaction with the user, such as the Always-Up-to-Date synchronization program. It has a reference count (UREF). Upon exiting such an application, the device goes directly to the Suspended state without waiting for the system idle timer.
UserIdle	State after a longer period of user inactivity. The screen is either in low-power mode or off (depending on the OEM configuration). Other things could be turned off, as well.

Figure 11
Pocket PCs (which our device was) transition between various device power states. In our solution, we had to avoid transitioning to the 'Suspended' state, as shown above.

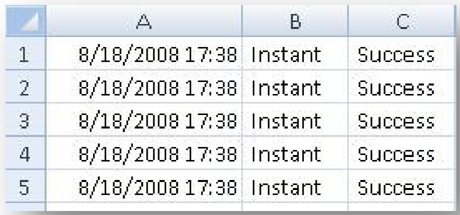
The outcome of having both these solutions simultaneously is as desired. Calling *SystemIdleTimerReset* and *SetPowerRequirement* will ensure the accelerometer device is active for the entire duration of the application. When the screen is inactive, the back light will automatically turn off and within the next 10 seconds (by default) this will trigger a force of the device into UNATTENDED state – this situation can be closely compared to the normal situation where the device enters SUSPENDED state after some time of inactivity, except here it instead transitions to the more useful UNATTENDED state. Another trigger for this can be when the client has explicitly requested the device screen to be turned off, which essentially speeds up this very same process. The 'Display Off' icon can be seen at the bottom of Figure 4, for instance.

Evaluation and Discussion

To assess whether this solution was applicable to the initial goals we set for activity classification, we conducted two experiments – one pertaining to the solution's robustness and the other pertaining to its accuracy and performance.

Robustness Experiment

For the robustness experiment several trials were conducted using different activity classification update intervals in an attempt to evaluate the battery life and the connectivity to the web server. These components were measured by referring to two log files at the end of each trial – the classification log file (shown in Figure 10) and the connectivity log file (shown in Figure 12). This latter file was specifically added to measure the number of connection attempts that were successful over the total number attempted. The format of this file is simple; it contains: a timestamp, the type of activity that is uploaded (instant or smooth), and an indication of either success or failure.



	A	B	C
1	8/18/2008 17:38	Instant	Success
2	8/18/2008 17:38	Instant	Success
3	8/18/2008 17:38	Instant	Success
4	8/18/2008 17:38	Instant	Success
5	8/18/2008 17:38	Instant	Success

Figure 12
The connectivity log file indicates whether the application could upload to the web server at the specified timestamp.

Each trial was conducted with the battery being completely charged. The battery life was measured by looking at the start and end times of the classification log file, and the connectivity was measure by considering the percentage of successful transmissions to the web server. The connectivity of the system was perfect in three of the four trials that had server updates enabled; one trial with 5 minute smoothing updates experienced about 92% connectivity (14 failures among 168 attempts), however, this was due to walking within an area with little or no cellular reception. As such, it appears connectivity is reliable as long as cellular service is available. Within the 'Future Work' section we evaluate some easy extensions to improve this percentage within these limited service areas.

As for the issue of battery life, Figure 13 illustrates the effect on battery life using the following smooth upload intervals: 3 seconds, 1 minute, 2 minutes, and 5 minutes. This range of values provides a good lower and upper bound on what is practical for most activity inference applications. Table 1 shows a slightly different aspect of battery life; it illustrates the effect on battery life under different connectivity environments. These experiments were done to analyze whether successful or failed connections change the overall usage of battery, where upload rate was kept the same at 5 minutes. Finally, results are also shown from an experiment where uploads were disabled as a whole.

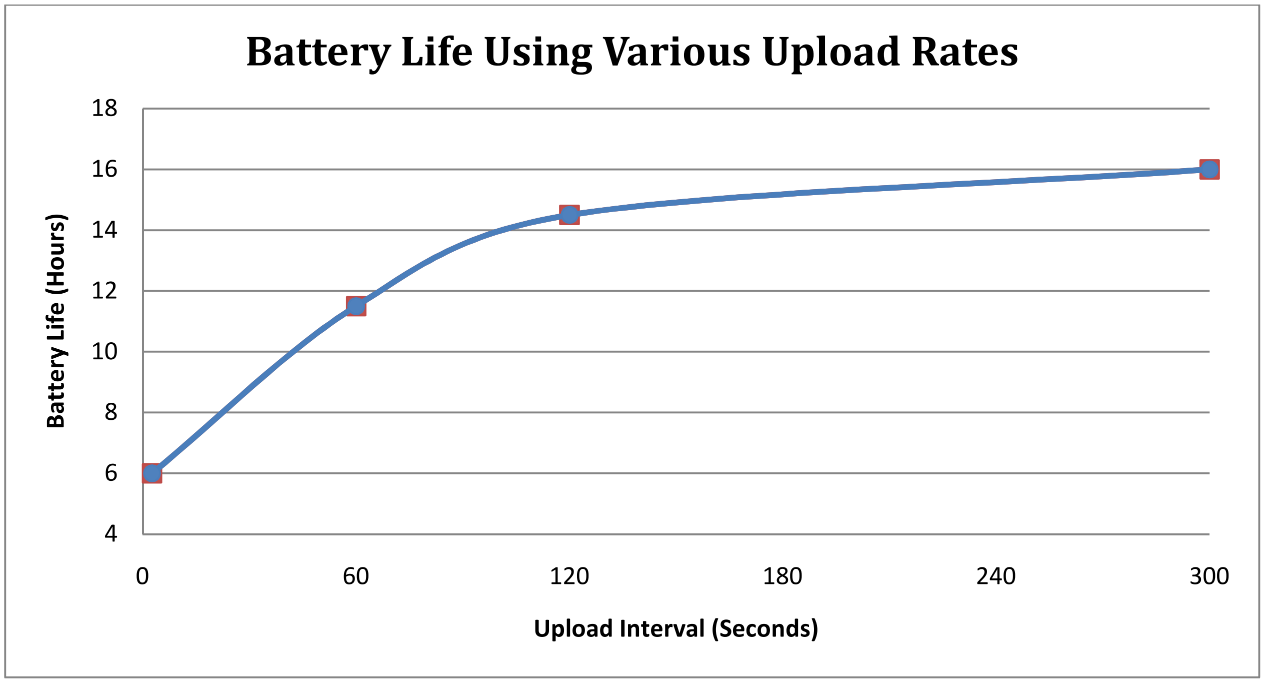


Figure 13

Illustrates results from four experiments to evaluate battery life using various upload rates.

Battery Life Under Various Connectivity Environments

Experiment	Connectivity Environment	Battery Life (Hours)
1	Perfectly Successful Uploads	16
2	Perfectly Failed Uploads (airplane mode)	20.75
3	Disabled Uploads	23.5

Table 1

When avoiding connection attempts, either due to internet unavailability or disabling updates to the web server, there is a considerable improvement in battery life, as the data above illustrates.

The data from Figure 11 shows that with a small upload interval (and therefore a large number of attempted uploads), battery life is depleted relatively quickly. In fact, it is pretty safe to say that 6 hours is too short of a battery life for most purposes. Updating every 5 minutes, on the other hand, results in almost three times the battery performance. Data from Table 1 reinforces this hypothesis as it shows that when the phone modem is disabled (airplane mode), causing all updates to result in a failure, there is a 30% increase in battery life. Better yet, when uploads are disabled completely (and thus never attempted in the application) there is a 47% increase from the original 16 hours to nearly 24 hours. The current performance of the battery life appears practical for most applications. Without focusing too greatly on battery types and phone-specific details, these results currently outperform those of the iPhone and Android mobile platforms, which run at 6 hours and 10 hours under normal conditions, respectively.

As reference, in one experiment with the screen lock and the battery management settings in effect but *without* the classification application running on the phone, battery life expired after 7 days. This result, being nearly 6 days longer than when the application was running with uploads disabled, shows the enormous effect on battery life from repeatedly sampling the accelerometer at 25 Hz. In the 'Future Work', however, I will discuss some ways to optimize battery use even further.

Classification Accuracy Experiment

To evaluate the accuracy of our system, data was collected from 5 male and female students between the ages of 18 – 22. Data was collected for each individual while the participant performed approximately 3 minutes each of the following activities: walking, jogging, bicycling on a stationary bike, and sitting. Cross-validation tests were performed where for each participant we conducted a hold-one-out test. Table 2 shows a confusion matrix that combines the hold-one-out tests of all participants.

Confusion Matrix from Hold-One-Out Test Results

		Classified As			
		Biking	**Running**	**Sitting**	**Walking**
Ground Truth	**Biking**	122	0	2	14
	Running	0	152	0	1
	Sitting	4	0	121	0
	Walking	30	17	0	103

Table 2

The diagonal with the largest values illustrate that classification accuracy was relatively high. Misclassifications did occur within our data, however, most commonly between Biking and Walking.

This data makes transparent the frequency at which two activities were 'confused' during the cross-validation testing. In particular, the activity of Biking and Walking were often misclassified as one when it really was the other – in fact, this occurred in nearly 15% of our data. On the other hand, correct classifications (represented by the diagonal where each activity meets itself) were made in about 88% of our data. Table 3 illustrates the percentage of correct classifications for each individual participant when validated with the model built from data gathered among all other participants.

Percentage of Correct Classifications for Each Participant

	Percentage (%)
Participant 1	99.2
Participant 2	93.0
Participant 3	65.6
Participant 4	97.5
Participant 5	79.5

Table 3

Hold-one-out tests conclude that data from Participant 3 was substantially different from the other participants, while Participant 1 experienced nearly perfect validation.

This data reveals some important information from our experiment. In particular, it appears that much of our misclassification was related to the data from the third participant. Without the ability to extensively review this circumstance, one would predict that this is because the data was either erroneously logged during this trial or simply that the model based on the other four individuals was not a complete-enough description of the activities. Consequently, in order to improve overall accuracy, one would be encouraged to re-evaluate these percentages after incorporating an additional set of participants and/or performing a more controlled experiment that requires participants to follow designated routes and wear particular types of clothing to regulate how the device is held in the pocket. The accuracy results for the iPhone solution, which implemented stricter control in their experiments similar to those just described, achieved nearly perfect classification accuracy and, thus, serves as a good model to follow for subsequent rounds of experimentation.

Because we could not overcome the 25 Hz accelerometer polling frequency in our design and were thus concerned with how this would affect our results, our accuracy results were quiet pleasing to discover. Although we will make some suggestions for the accelerometer in the 'Future Work' section, it does not appear necessary to modify use of the accelerometer to achieve better classification. There remain several of options to tweak and to consider: better machine learning, better choice of features, etc. Therefore, based on the results of our experiment, it does not appear that the accelerometer polling frequency is necessarily the sole difficulty that needs to be solved in order to increase accuracy results. Our results in general were quite pleasing given that this was only our first implementation. We discuss some future work for our solution in the next section.

Future Work

Though we have successfully implemented activity classification on our mobile phone, there is certainly much more work that can be done. Some of these ideas arose throughout the course of our design and could not be addressed due to lack of time, while others arose after considering our evaluation results as discussed above.

One idea is to sleep the device during time when activity can be inferred as constant. This is based on the idea that people do not change activities every 5 seconds, but takes this idea even further to say that sometimes (though not always) people will not change their activity for hours. With this said, consider when a person has been sitting for a few hours; if we can detect that there is no change taking place, we can gradually reduce the frequency at which computations are being made (i.e., reduce accelerometer sampling rate, etc.). This solution seems feasible since we can use the smoothing window to determine the accuracy at which an activity is classified each time. If an activity is classified with a high percentage (perhaps even on a few successive iterations of smooth updates), then we could possible detract from regular sampling. We could expect perhaps a small tradeoff in accuracy for a large increase in battery life.

Another important task for future work is considering how our solution performs on other Windows Mobile phones, since we only tested with the HTC Touch Diamond. On a similar note, it would be also worthwhile to experiment with the upcoming Windows Mobile 7 operating system. These two items will be necessary to assure we have addressed the issue for a range of potential Windows Mobile devices.

Finally, there are a number of improvements that were brought to our attention after our evaluation experiments. One idea is re-tackle the effort to improve the accelerometer sampling rate. While, as we concluded before, this is not necessary an essential step needed to improve performance, it is nonetheless worthwhile to evaluate changes in performance using higher frequencies. Connectivity too could be expanded such that it does not suffer within areas that do not have strong cell service. Using the wireless capabilities of the phone, if present, we can ensure a better overall connectivity percentage.

Conclusion

Activity inference has many important applications and is already being integrated in many different fields. For the class of problems that distinguish between a few different activities, systems often suffer from various downfalls that accompany use of custom-made hardware. Using mobile phones as the answer to this hardware question allows developers interested in this technology, including researchers at universities or elsewhere, to readily have access to activity inference functionality. Taking a step towards achieving this goal, our solution contains a carefully structured and clearly commented API that allows for flexibility throughout each of the three steps of our system – collecting data, creating models, and classifying activities. Our results were favorable, especially for being our first implementation. Still, as the next step there are certainly many improvements that can be made as well as extensions (perhaps to additional sensors) to further increase functionality and make our system applicable to an even larger scope of applications.

Acknowledgements

This research could not have been possible without the constant assistance and suggestions of a few notable individuals, including Professor James Fogarty, Professor James A. Landay, T. Scott Saponas, Jon E. Froehlich, Jonathan Lester, and Microsoft Corporation for providing the mobile phone.

References

1. Consolvo, S., McDonald, D., Toscos, T., Chen, M., Froehlich, J., Harrison, B., Klasnja, P., LaMarca, A., LeGrand, L., Libby, R., Smith, I., & Landay, J.. Activity Sensing in the Wild: A Field Trial of UbiFit Garden, In Proceedings of CHI 2008, ACM Press.
2. Froehlich, J., Dillahunt, T., Klasnja, P., Mankoff, J., Consolvo, S., Harrison, B., Landay, J. UbiGreen: Investigating a Mobile Tool for Tracking and Supporting Green Transportation Habits, Conference on Human Factors in Computing Systems, 2009.
3. Cisco Systems. Redefining Mobile Applications: Mobility Services and the Unified Network. Farpoint Group White Paper, May 2008.
4. Saponas, S., Lester, J., Froehlich, J, Fogarty, J., Landay, J. iLearn on the iPhone: Real-Time Human Activity Classification on Commodity Mobile Phones. CSE Technical Report , University of Washington, 2008.

5. Campbell, T., Ngo, B., and Fogarty, J. (2008). Game Design Principles in Everyday Fitness Applications. Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW 2008), pp. 249-252.
6. Choudhury, T., Borriello, G., Consolvo, S., Haehnel, D., Harrison, B., Hemingway, B., Hightower, J., Klasnja, P., Koscher, K., LaMarca, A., Lester, J., Landay, J., Legrand, L., Rahimi, A., Rea, A., & Wyatt, D., The Mobile Sensing Platform: An Embedded System for Capturing and Recognizing Human Activities, In IEEE Pervasive Computing, vol. 7, no. 2.
7. Rowe, A., Smailagic, A., & Siewiorek, D., Location and activity recognition using eWatch: A wearable sensor platform, In Ambient Intelligence in Every Day Life. Springer (2006).
8. Apple. Nike + iPod Kit. http://www.apple.com/ipod/nike/
9. Seligman, Scott. Fun with the Diamond's Tilt Sensor. http://scottandmichelle.net/scott/comments.html?entry=784
10. Dutta, Koushik. My Brain Hurts Using HTC Touch Diamond's Accelerometer-Sensor SDK from Managed Code. 2008. http://www.koushikdutta.com/2008/07/using-htc-touch-diamond-sensor-sdk-from.html
11. MSDN. System Power States. Microsoft Corporation, 2008. http://msdn.microsoft.com/en-us/library/aa930499.aspx