Auditeur: A Mobile-Cloud Service Platform for Acoustic Event Detection on Smartphones

Shahriar Nirjon[1], Robert F. Dickerson[1], Philip Asare[1], Qiang Li[1], Dezhi Hong[1],
John A. Stankovic[1], Pan Hu[2], Guobin Shen[2], and Xiaofan Jiang[3]
[1]Department of Computer Science, University of Virginia, USA
[2]Microsoft Research Asia, Beijing, China
[3]Intel Labs China, Beijing, China
{smn8z, rfd7a, pka6qz, lq7c, dh5gm, stankovic}@virginia.edu, {v-pah, jacky.shen}@microsoft.com, fred.jiang@intel.com

ABSTRACT

Auditeur is a general-purpose, energy-efficient, and context-aware acoustic event detection platform for smartphones. It enables app developers to have their app register for and get notified on a wide variety of acoustic events. Auditeur is backed by a cloud service to store user contributed sound clips and to generate an energy-efficient and context-aware classification plan for the phone. When an acoustic event type has been registered, the smartphone instantiates the necessary acoustic processing modules and wires them together to execute the plan. The phone then captures, processes, and classifies acoustic events locally and efficiently. Our analysis on user-contributed empirical data shows that Auditeur's energy-aware acoustic feature selection algorithm is capable of increasing the device-lifetime by 33.4%, sacrificing less than 2% of the maximum achievable accuracy. We implement seven apps with Auditeur, and deploy them in real-world scenarios to demonstrate that Auditeur is versatile, $11.04\% - 441.42\%$ less power hungry, and $10.71\% - 13.86\%$ more accurate in detecting acoustic events, compared to state-of-the-art techniques. We present a user study to demonstrate that novice programmers can implement the core logic of interesting apps with Auditeur in less than 30 minutes, using only $15 - 20$ lines of Java code.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems

General Terms

Design, Experimentation

Keywords

Acoustics, Soundlets

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MobiSys'13, June 25-28, 2013, Taipei, Taiwan
Copyright 2013 ACM 978-1-4503-1672-9/13/06 ...$15.00.

1. INTRODUCTION

Being able to deliver smartphone apps quickly to customers is crucial in today's highly competitive app market. Developers depend heavily on APIs provided by their smartphone platform for common tasks, but when it lacks necessary functions for their application, it costs developers weeks of time for implementation. Beyond just time, many developers do not have the necessary technical background to implement these functions correctly or efficiently. Because of this, there is a growing trend where smartphone apps are paired with numerous web services by other providers to get access to specialized or computationally demanding tasks. Microsoft's Hawaii [1], for example, supports path prediction, key-value storage, translator, relay, rendezvous, OCR, and speech-to-text services. Google provides services such as web search, maps, play, YouTube, cloud drive, and Gmail. Other providers give access to data sources such as weather, financial data, airline information, and parcel tracking.

Smartphones have begun to use their microphones for various acoustic processing tasks, but most of the apps are special purpose acoustic event detectors. Examples of voice and music based systems include: speaker identification [16], speech recognition [31], emotion and stress detection [18, 28], conversation and human behavior inference [20, 21], music recognition [32], music search and discovery [3], and music genre classification. There are other types of apps that fall into the non-voice, non-music category, such as a cough detector [14], a heart beat counter [23], and logical location inference [4, 6]. These examples are limited in number and acoustic sensing is often one of multiple sensing modalities in these works. There are existing works [17, 19] that consider multiple types of acoustic events. SoundSense [17] considers speech, music and ambient sound, and provides a mechanism to label clusters of ambient sounds to extend the set of classes. Jigsaw [19] considers speech and sounds related to activities of daily living. These systems still lack in providing tools and mechanisms to easily extend detection capabilities and effectively support development process. They are limited for general purpose acoustic event detection service for smartphones and in exploiting the promising opportunities given by already prevailing smartphones equipped with microphones.

To address this need, we have built a general-purpose acoustic event detection platform called *Auditeur*. Compared to the previous works in the literature, we position Auditeur as a developer platform rather than just a system detecting a set of sounds. Auditeur provides APIs to developers to enable their app to register for and get notified on a wide variety of acoustic events such as speech, other types of man-made sounds, such as emotional or physiological sounds, music, environmental sounds, sounds observed in households, offices, or public places, sounds of vehicles, tools and so forth. Auditeur achieves this capability of classifying general-purpose sounds by utilizing its *tagged soundlets* concept which are crowd-contributed, short-duration audio clips recorded on a smartphone along with a list of user given tags and automatically generated contexts. The cloud hosts the collection of tagged soundlets and provides a set of services, which are used by the smartphones, to upload new soundlets and to obtain a detailed *classification plan* to recognize sounds specified by the list of tags as parameters. Typically a plan contains one or more acoustic processing *pipeline configurations*, corresponding to different energy-requirements and user-contexts, specifying the required acoustic processing units, their parameters, and how they are connected.

Several salient features when taken in combination make Auditeur unique. First, Auditeur provides a simple yet powerful API which novice developers with basic object-oriented programming skills can learn easily. Second, Auditeur supports both personalization and generalization. It is possible to manage personal sounds and classifiers as well as obtain sounds and classifiers that were created by other users of the system. Third, the acoustic processing pipeline inside the phone is flexible and extensible. The phone contains a wide range of acoustic processing primitives. However, only the ones that are required to recognize the desired sounds (as mentioned in the classification plan from the cloud) are dynamically instantiated and wired to form the pipeline. Addition of a new primitive is easy since the processing units have a generic structure and are dynamically wired. Fourth, the sound recognition service running on the phone is adaptive. Apps are notified on a change of context or at a specific battery level, and a new pipeline configuration is loaded to adapt to the change. Fifth, Auditeur is designed for efficiency in communication, computation, and energy consumption. The phone needs to connect to the cloud only once to obtain the classification plan. Hence an uninterrupted Internet connectivity is not a requirement. Acoustic processing in Auditeur is efficient as only the necessary components run within the phone. An energy-aware acoustic processing plan to be executed by the phone is generated at the cloud to increase the lifetime of the device while maintaining high accuracy.

Partitioning the planing and execution between the cloud and the phone let Auditeur have the best of both of an in-phone [17, 16, 18] sound recognizer and a cloud-assisted [28, 32, 14] one. In-phone recognizers perform all of the processing inside the phone. Typically, they are highly tuned to the application scenario, and use a fixed set of features and an offline-trained, fixed classifier. Thus, they are rigid and are not usable if either the application or its context changes. Cloud-assisted recognizers, on the other hand, send unprocessed or partially processed data to a server, and rely on web services to perform further processing and classification. This approach is more flexible, but has several limitations such as the requirement for an uninterrupted Internet connectivity, high bandwidth, and the expense of sending a large chunk of data over the cellular network. Our approach for Auditeur takes the best of these two strategies. Auditeur performs the signal processing and feature classification completely inside the phone, but uses the cloud to store user contributed sound clips, build new classifiers, and obtain an energy-aware classification plan. Once the smartphone receives the classification plan from the cloud, it dynamically instantiates the components required to execute the plan, and keeps detecting acoustic events efficiently and locally.

Auditeur and its energy and context-aware classification planning architecture are different from existing context monitoring and mobile sensing platforms [11, 12, 13, 7, 20]. SymPhoney [11] requires developers specify their hand-tuned processing pipelines, whereas the acoustic processing pipeline in Auditeur is generated automatically. SeeMon [12] maintains an essential set of sensors to optimize the sensing and transmission cost associated with the sensors. Orchestrator [13] provides a plan-based execution framework with policy-based system optimization which dynamically adapts to the changing system and application situations. It opportunistically changes its execution plan based on the current status of resources under a specified policy such as minimum accuracy or minimum overall energy cost. Unlike [12, 13], Auditeur maintains an informative set of acoustic features and takes a deterministic approach in maximizing its classification accuracy while satisfying the developer specified minimum lifetime goal. Certain aspects of Kobe [7] such as the API to create a classifier, the search for an optimum configuration, and runtime adaptation are conceptually similar to Auditeur's, however, they are quite different in design and technical details. For example, Kobe requires explicit declaration of the processing pipeline, performs an exhaustive search, and offloads code to server, whereas Auditeur generates pipelines automatically, applies dynamic programming instead of a search, and does not offload code to satisfy its energy constraint. DarwinPhones [20] proposes a collaborative approach of exchanging classifiers and classification results among phones. Auditeur's shows it collaborative nature at the data level via its crowd-contributed tagged soundlets.

We evaluate Auditeur with four types of experiments. First, we measure CPU and memory footprints, and energy consumptions of different processing units. Second, we perform an empirical study on different categories of sounds to demonstrate the efficacy of our energy-aware feature selection algorithm. Third, we describe case studies on seven apps, implemented with Auditeur, to demonstrate the versatility of Auditeur, and to quantify their energy efficiency, accuracy of event detection, and context awareness. Fourth, we describe a user study on developers evaluating the usability of the Auditeur API.

The contributions of this paper are the following:

- A versatile, general-purpose, flexible, extensible, context-aware, and efficient end-to-end acoustic event detection platform for smartphones, backed by the cloud.
- A sound recognition service on the smartphone, capable of dynamically wiring acoustic processing units, and running an app-tailored and context-aware classification plan on the phone, and an API to register and get notified on specific sound events.
- A collection of crowd-contributed, admission controlled, and contextually-tagged short sound clips in the cloud available to the developers, and an API to upload and manage them.
- A dynamic programming based acoustic feature selection algorithm that generates energy-aware acoustic event detection plans for smartphones. Our empirical evaluation shows that the algorithm is capable of increasing the device-lifetime by 33.4%, sacrificing less than 2% of its maximum achievable accuracy.
- We implement 7 apps with Auditeur, and deploy them in real-world scenarios to demonstrate that Auditeur is versatile, $11.04\% - 441.42\%$ less power hungry, and $10.71\% - 13.86\%$ more accurate in detecting acoustic events, compared to state-of-the-art techniques. A user study demonstrates that novice programmers can implement the core logic of interesting apps with Auditeur in less than 30 mins, using only $15 - 20$ lines of Java code.

2. USAGE SCENARIOS

We describe three scenarios to illustrate the use cases of Auditeur.

Auditeur is for Developers. Alice is a smartphone app developer who has an exciting idea for an app. Her envisioned app listens to a piece of music, determines its genre, and suggests similar music to the user. But the problem is, Alice does not know the technical details of how to classify music based on their acoustic features. She finds Auditeur helpful which has a collection of short music clips and their genre. She uses Auditeur's API to obtain a music genre classifier by specifying the music-types she wants to recognize, uses Auditeur's API to perform the classification on the phone, and uses Auditeur (or, any other online music databases) to get a list of songs of the same genre.

Auditeur is for Researchers. Bob is a behavioral researcher who wants to perform a quick study on how people interact socially. For his study, he gathers some volunteers and collects samples of their voice, laughter, and yelling sounds using a custom app powered by Auditeur. At first, he uses Auditeur's API to record the audio and add proper tags, e.g., voice, laughter, yell, context, and recording environment, to it. Then he uses Auditeur' API to upload the samples, and obtains a classifier from the cloud which detects speaker id, laughter, and yelling sounds in different environments and contexts. Finally, he uses Auditeur API to continuously listen to the microphone and log social interactions of his study subjects.

Auditeur is for End-users. Cathy is a busy housewife, a mother of a newborn, and a smartphone user. Recently she installed a smartphone app that monitors her sleeping baby when she is busy working downstairs. During the configuration of the app, the GUI asks her to record sound samples of her baby crying, whining, coughing, and doing baby-talks as well as other possible sounds that might occur in the room, e.g., the sound of the air conditioner, hanging toys, door squeaking, and noise made by the pet. She records all of these sounds using the app, protects them with a password, and selects the ones she wants to be notified of. Once the configuration is completed, she keeps the phone on a table within the suggested distance from the baby and starts the app. The app keeps monitoring the baby and whenever a sound of interest occurs, the phone alerts her by calling the phone in the kitchen.

3. THE TAGGED SOUNDLET CONCEPT

A *tagged soundlet* is a short-duration ($3 - 30s$) audio clip, recorded on a smartphone along with two types of contextual information: user given tags and phone generated context.

3.1 Tags

The logical tags associated with a soundlet are the user given identifiers that describe its content and the surrounding environmental context. Examples of tags are in Table 4 of Section 8.3.

Content and Container Tags. While the user chooses the tags that he wants to use to describe a soundlet, we require him to enlist two kinds of tags: *content*, and *container* tags. The content tags describe what the sound *is*, and the container tags describe the background which contains the sound. For example, a $15s$ recording of Alice's voice at her office should have `{voice, female, Alice}` in the list of content tags, and `{office}` in the list of container tags.

Lookfor and Within Tags. The content tags are used in two different ways during the training of a soundlet classifier. The *lookfor* tags describe a class of soundlets that the user wants to recognize, and the *within* tags describe the class of soundlets that represents the universe of sounds, i.e. the set of sounds that might be present in the environment along with the sound that he wants to recognize. For example, when creating a training set for a classifier that detects Alice's voice at her office: the lookfor tags should have `{voice, Alice}` on the list, the within tags should include `{voice, printer, phone}`, and the container should be `{office}` as before.

3.2 Phone Contexts

The phone context refers to additional information about the soundlets which may or may not be conveyed by the tags. These are similar to the tags in the manner they are used by the recognition algorithm, but they are different in the manner they are generated. The phone context is either automatically generated by an algorithm using information from the on-board sensors, or they are assumed. The developer of the app specifies whether an automatically generated context or an assumed context – whichever is more suitable to the app – should be used.

Aside from basic audio information such as the sampling rate, encoding, duration, and timestamp, Auditeur generates three other types of contexts, which are: (1) the location of the phone, (2) the position of the phone with respect to body, and (3) the environmental noise level. Section 6.2 provides an elaboration of phone contexts. Note that, the developer of an app can always override the context generation process. We keep this provision in Auditeur as context generation has its overhead and in many cases an app is used only in a few specific presumed contexts. For example, an app that detects vehicle sounds can assume that the location is outdoors instead of periodically getting it computed.

3.3 Public and Private Spaces

The space of soundlets is logically partitioned into two subspaces from a user's point of view – *public* and *private*. A *user* of Auditeur is anyone who has an account in Auditeur. Typically, the user is an app developer, who uses the public space by default, but may use a private space with proper permissions from an end-user.

3.3.1 Public Soundlets

The public space contains shared soundlets, contributed by all users of Auditeur. This is visible to everyone, and does not require any authentication to access. However, there are two constraints that must be met by each soundlet.

Tag and Sanity constraints. There is a predefined fixed set of tags that can be attached to public soundlets. The user cannot create, modify or delete any tag. The set of tags is maintained by the system admin of Auditeur. Although, at present, there are over 1000 different tags, we admit that no fixed set is ever sufficient to cover the enormous possibilities of tags. However, limiting the tags in the public space keeps the space manageable and provides us a way to ensure the confidence in sound recognition.

Every public soundlet goes through a sanity check to ensure that it is not an outlier with respect to other soundlets with the same tags. Thus, a public soundlet is not immediately made visible to everyone until it passes an outlier detection test. This is to ensure that no attacker can mess up the public space by filling it up with soundlets with non-representative tags. A service running on the cloud automatically detects non-conforming soundlets using distance based outlier detection technique [5] which is described in Section 7.2.

3.3.2 Private Soundlets

Private soundlets are user specific and are not shared. The user has to access his private space using proper authentications. Unlike public soundlets, the private ones are not limited by the tag and sanity constraints, i.e. a user can upload any sound he likes into his private space and attach whatever tag he wants to describe it. However, the list of content and container tags has to be the same (in number of tags and the spelling of tags) in order for two soundlets to be treated as belonging to the same class.

We keep provision for private space in our design in order to complement the public space which is limited by the choices of tags and has a high degree of sanity requirement. The private space is more flexible, tailored to the specific needs of a user, and if properly used, an infinite number of possible sounds can be stored and recognized by our system making it highly extensible.

4. AUDITEUR SYSTEM OVERVIEW

The design of Auditeur is two-tiered: the phone contains the mechanism for sound capture, sound processing, and sound recognition, while the cloud provides the logic for building classifiers and tuning the parameters for the sound processing modules on the phones.

4.1 In-Phone Processing

4.1.1 Tagging and Uploading Soundlets

Auditeur provides an API to record, add tags, and upload soundlets to the cloud. The API provides an interface for the default sound capture device, however the developer may choose a different one (such as a Bluetooth microphone) or an already recorded sound clip. After the sound is captured, the tagging process automatically generates the phone contexts, and the user can review and change it prior to uploading. The sound clip is then uploaded to the cloud. At this stage, the upload of the public soundlet could fail if it violates the tag or sanity constraints.

Uploading soundlets to the cloud is energy consuming. However, this is done only once and is used to create classification plans for several apps. An alternative to uploading the sound clips is to extract the features locally and then upload the acoustic features only. But we do not follow this approach to keep our design simple and extensible. By uploading only the features we lose the opportunity to explore the possibility of considering new acoustic features and creating more sophisticated classification plans.

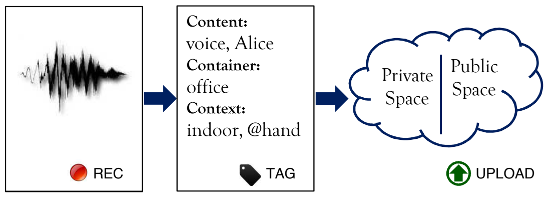

Figure 1: Recording, tagging, and uploading soundlets to the cloud.

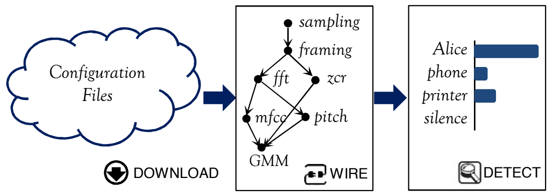

Figure 2: Downloading configuration, wiring components, and sound event detection.

4.1.2 Acoustic Event Detection

The cloud generates an app tailored acoustic event detection plan that directs the phone on which components should be instantiated on the phone and how they should be wired together to form an *acoustic processing pipeline*. The plan could contain more than one pipeline configuration, corresponding to different contexts or energy-requirements. The phone reads the plan from an XML file, instantiates and initializes the processing units with the specified parameters, and wires them together to form a pipeline. Once the pipeline is ready, an API call starts reading audio samples from the microphone, and pushes data down the pipeline. The app is issued a callback once the desired sound event is detected. A change in context, such as moving from indoors to outdoors, or putting the phone inside the pocket, automatically loads a different pipeline configuration by default, however, the developer can override this behavior.

4.2 In-Cloud Processing

An energy-aware acoustic event detection plan is generated in the cloud upon receiving a request from the phone. A request contains: sounds the app is looking for, other unwanted sounds that might occur in the environment, contextual information, and energy constraints. The tags and contexts are then used to create a comprehensive training set using a subset of the currently available soundlets in the desired space. To meet the energy constraint, the dimensionality of the training set is reduced by selecting a subset of features, considering the energy consumptions of different acoustic processing units. Finally, a number of classifiers are trained, and the one showing the highest accuracy during cross-validation tests while meeting the energy criteria is chosen. The sound processing pipeline, feature set, the classifiers, and their parameters altogether form the contents of a downloadable configuration file.

Since communication to the cloud and training classifiers on-the-fly is costly, the cloud packs a set of pipeline configurations, corresponding to different contexts and energy requirements, into one classification plan.

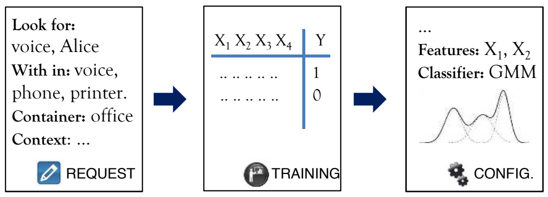

Figure 3: Request for a classifier, training the classifier, and generating configuration file for download.

4.3 API Example

Figure 4 shows a code snippet which demonstrates the usage of Auditeur API for recording, tagging, and uploading soundlets to the cloud, and obtaining and using a classification plan from the cloud. Lines $2-3$ create a `SoundletRecorder` to record a $16s$ audio and obtain a `Soundlet`. Lines $4-5$ create the content and container tags, and lines $6-7$ set the tags and make the soundlet private. Lines $8-10$ grab the `SoundletManager`, and upload the soundlet to the cloud. Lines $12-13$ create the lookfor and within tags. Line 14 creates a `SoundletDetector` and registers the application as a listener. Line 15 obtains a classification plan and initializes the detector, and line 17 starts the continuous sound event detection process.

```java
//Record and upload a soundlet.
SoundletRecorder recorder = new SoundletRecorder();
Soundlet soundlet = recorder.recordSoundlet(16);
String[] content = {"voice", "female", "Alice"};
String[] container = {"office"};
soundlet.setTags(content, container);
soundlet.setSharing(false);
SoundletManager manager = new SoundletManager(
    getSystemService("SoundletService"));
manager.uploadSoundlet(soundlet);
//Get a classifier and register for a match event.
String[] lookfor = {"voice", "Alice"};
String[] within = {"voice", "printer", "phone"};
SoundletDetector detector = new SoundletDetector(this);
detector.setPipeline(manager.getConfig(
        lookfor, within, container));
detector.start(SoundletDetector.CONTINUOUS);
```

Figure 4: An example code snippet to create, tag, upload, get classification plan, and detect soundlets.

5. KEY FEATURES OF AUDITEUR

5.1 People in the Loop

The person who contributes a soundlet knows it the best. Auditeur provides an API to let people record their own sounds, and tag them appropriately. An alternate approach, such as automated tagging, might help the user with some suggestions, but is not sufficient since a whole bunch of different sounds is often indistinguishable. For example, aerosol spray, steam, waterfall, and white noise sound almost the same. The tagging process in Auditeur is guided by the concept of the content and container tags, which keeps the background explicit from the sound of interest. The sound recognition process is guided by the *lookfor* and *within* tags, which specify the sound of interest as well as other possible sounds in the environment – the best way to identify these is to engage the end-user.

5.2 Personalization as well as Generalization

Auditeur has provision for both personalization and generalization in sound recognition. This is achieved by the two logical spaces: public and private spaces, which are used to store soundlets and create classifiers. The public space contains soundlets that are shared by everyone, and hence this collection is enormous, and the classifiers created using these soundlets have more generalization ability in sound recognition tasks. The public space is for developers who want to build and test an app quickly, and for apps that do not require personalization. The private space, on the other hand, ensures privacy, and the classifiers created using the private soundlets are tailored to meet the end-user's need.

5.3 Cloud-Directed On-Device Processing

Sound recognition on the phone is not a one-size-fits-all problem. Every problem is unique and involves different sets of tasks along the pipeline. This is why, in Auditeur, we keep provision for dynamically creating a chain of tasks for a specific problem. An algorithm running on the cloud decides which set of tasks are appropriate for a given problem, and creates a configuration file describing the tasks and parameters required to solve it. The phone downloads the configuration file, constructs the chain of tasks, and executes them locally. Communication with the cloud is not continuous, rather the phone is required to connect to the cloud only once to get the execution plan.

5.4 Designed for Efficiency

Mobile phones are limited by their processing capability and battery life. A novel feature of Auditeur is that, it provides the smartphone with an API to obtain energy-aware classification plans from the cloud. For this to work, we profile the energy consumption of all the tasks that are involved in the sound recognition process on the phone. The cloud uses this profile to formulate a dynamic programming problem that selects the set of acoustic features required to recognize desired soundlets, under a given energy constraint. That means, a smartphone having 30% remaining battery life can use a classifier which is aware of its current condition and is different from the one being used since the battery was full.

5.5 Context Awareness

Auditeur considers the contexts in which the sound is recorded and is matched. Keeping contextual information such as the location, body position, and environmental noise level makes the sound matching adaptive to change. This is where we exploit the mobility of the device to make our solution elegant when compared to standard sound matching algorithms. Contexts increase the adaptability of our system in different ways. For example, the physical location of the device greatly eliminates a large number of unlikely sounds from consideration during the sound matching process. Body position of the phone is used to normalize the received signal strength, and the environmental noise level guides the noise reduction process. We describe a set of experiments to demonstrate the benefits of using contextual information.

6. IN-PHONE COMPONENTS

The implementation of in-phone processing is modular and layered as shown in Figure 5. At the bottom, we have three services that are running in the background. The *Sound Engine* service is the one that performs the actual sound recognition task, while the other two manage contexts and communications to the cloud. These services, along with some internal APIs, support the public API layer, which interacts with the applications.

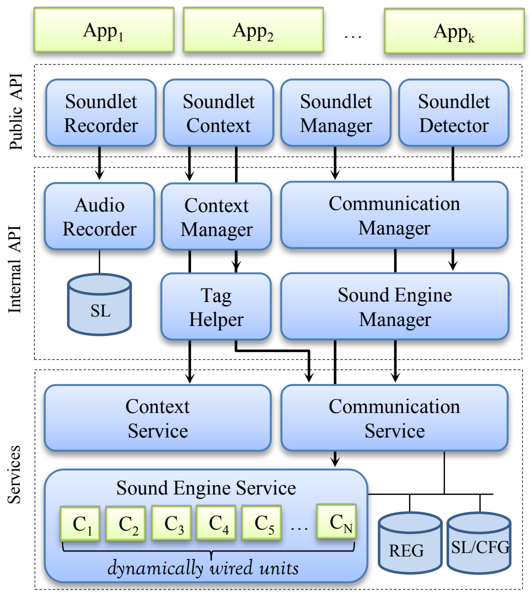

Figure 5: In-phone processing components.

6.1 The Sound Engine

The `SoundEngine` is a singleton, persistent service that is responsible for instantiating and initializing the acoustic processing units, forming a processing pipeline that detects sound events, and providing registration and notification services to the running applications.

6.1.1 Acoustic Processing Pipeline

Acoustic processing in Auditeur is divided into a five stage pipeline: (1) preprocessing, (2) frame level feature extraction, (3) frame level classification (frame admission), (4) window level feature extraction, and (5) window level classification. Each stage involves multiple tasks, and each task is performed at an *acoustic processing unit* (APU).

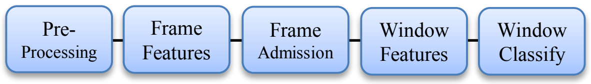

Figure 6: The acoustic processing pipeline in Auditeur.

Figure 6 shows the high level structure of the pipeline. The process starts with a *preprocessing* stage which captures audio from microphone, converts the byte stream into a stream of fixed sized frames, and applies filtering, windowing, and noise compensation as needed by the application. Each frame then passes through a *frame level feature extraction* stage. We have implemented a total of 25 time and frequency domain features, but not all of them might be used in single sound recognition task. These frames are then classified using a *frame level classifier* which acts as an admission controller, and decides whether or not to process a frame any further. If a frame is admitted to the *window level feature extraction* stage, a fixed number of consecutive frames are gathered to form a window, and up to 12 statistics are computed per feature per window, resulting in a maximum of 121-element feature vector. This number is less than 25 $\times$ 12 since not all statistical functions are applicable to all features. Each window is then classified by a *window level classifier*.

Table 1 shows the list of APUs that are implemented on the phone. We have implemented 5 preprocessors, 25 feature extractors, 12 statistical units, and 7 classifiers. We have put more emphasis on the frequency domain features as they are more robust to noise. We have implemented the classifiers that are commonly known to solve sound recognition problems, and implemented only their classification logic inside the phone as their training happens in the cloud.

APU	List
Preprocessor	Sampling, Framing, Filters, Windowing, Noise Compensation.
Features	FFT, ZCR, RMS, 13-MFCCs, Low Energy (Weak) Frame Rate Spectral {Entropy, Energy, Flux, Rolloff, Centroid}, Bandwidth, Phase Deviation, Pitch.
Statistics	Mean, Stddev, Geometric Mean, Harmonic Mean, Range, Moment, Zscore, Skewness, Kurtosis, Median, Mode, Quartile.
Classifiers	Naive Bayes, Decision Tree, GMM, MLP, SVM, kNN, HMM.

Table 1: Acoustic processing units in Auditeur.

6.1.2 Forming the Pipeline

The acoustic processing units (APUs) are the building blocks of the acoustic processing pipeline. APUs are dynamically instantiated and wired at runtime to form the pipeline. Each APU keeps a list of its immediate successor APUs, and implements two methods: `process` and `forward`. The `process` method performs its intended work, and the `forward` method pushes its output to its children. Instantiation of the APUs and the wiring process is guided by an XML-complaint configuration file that is obtained from the cloud. The configuration file is essentially the description of a directed acyclic graph where the nodes are the APUs and the directed edges denote the wiring among them.

```
<?xml version="1.0" encoding="utf-8"?>
<nodes>
     <node id="1" label="frame" />
     <node id="2" label="rms" />
     <node id="3" label="fft" />
     <node id="4" label="zcr" />
     <node id="5" label="entropy" />
     <node id="6" label="flux" />
     <node id="7" label="DT" parents="2" >
            <params> ... </params> </node>
     <node id="8" label="admitter" window="32" />
     <node id="9" label="stat.mean" window="32" />
     <node id="10" label="stat.mean" window="32" />
     <node id="11" label="aggregator" parents="3" />
     <node id="12" label="GMM" >
            <params> ... </params> </node>
     <node id="13" label="HMM" >
            <params> ... </params> </node>
</nodes>
<edges>
     <edge source="1" target="2" />
       ...
</edges>
```

Figure 7: Example of an XML file, describing the APUs and wiring to create the processing pipeline.

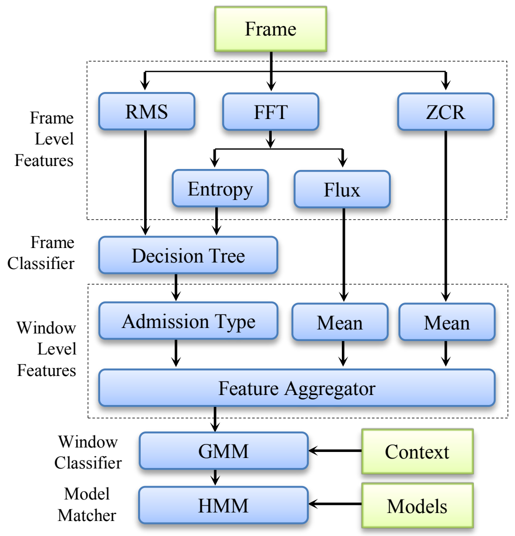

Figure 8: An instance of a dynamically wired pipeline corresponding to the XML file in Figure 7.

Example: Figure 8 shows an example of dynamically created pipeline based on the configuration file in Figure 7. This is similar to the sound processing as in [17], except that we show only a subset of features due to space limitation. In this example, 2 frame level features (RMS and spectral entropy) are used to admit frames using a decision tree classifier, and 2 window level features (spectral flux and zero crossing rate) are used to classify the window using a GMM. Finally, a HMM smoothens the classification results.

6.1.3 Registration and Notification

The sound engine internally maintains a register which keeps records of applications, desired sound events, and notification logic. Whenever an event, registered by a running application is detected, it broadcasts a notification according to the notification logic. Currently we support three types of notification: continuous (always notify), counter-valued (up to a number of events), and timed (up to a time duration after the registration or the first event).

6.2 Phone Context Generation

Auditeur provides three types of contextual information: location, position, and noise level. The location context refers to the GPS coordinates and whether the location is indoors or outdoors. While uploading a soundlet, we store the GPS coordinate if it is available. The availability means – we are getting a location fix and also the app has permission to use GPS. However, the current version of Auditeur only uses indoors or outdoors information. Yet we kept the GPS coordinates to investigate more fine grained location aware sound matching which we leave as a future work.

We adopt a simplified version of [34] to detect indoors or outdoors using only the accelerometer and cellular signals. The position context refers to the position of the phone with respect to human body which could be either direct (i.e. the sound source is close to the microphone), in pocket, or at a distance. This is similar to the phone contexts used in [17], and is estimated by the same principle. The noise level refers to the environmental noise level in dB scale.

Auditeur provides APIs to specify an action to be taken whenever there is a change in context. For example, when a person comes home from outside, the context engine would detect the change, and take actions that the developer programmed for, which is by default – loading a different pipeline configuration from the classification plan that is appropriate for home environment.

Auditeur provides APIs to enable and disable context checking, and to control the frequency of monitoring the change in each context. As context checking is costly, the developer of the app should choose an appropriate duty cycle to check and balance its benefit over cost.

6.3 Communication to the Cloud

Applications communicate to the cloud via the communication service, which submits new soundlets to the cloud through a Web HTTP interface. We implement a basic web application using JAVA and the Spring 3 Framework for this purpose. For hosting, we use the Amazon web services infrastructure by running our application in a Tomcat 7 web container on an Amazon EC2 instance. Soundlets, along with their tags and contexts, are serialized in JSON format and are submitted to the Web. Afterwards, the data is committed to the database as described in Section 7.3.

When an app requests a classification plan, the communication service issues a GET request to the web interface on its behalf with search parameters such as the tags, contexts, and energy constraints. In response, the cloud creates the classification plan, serializes it as a JSON object, and sends it to the phone.

6.4 Public and Internal API

Four components constitute the public API layer. The `Soundlet Recorder` records audio from the microphone, converts it to a Soundlet, and stores them locally. The `SoundletContext` uses the `ContextManager` to get the current context, and the `TagHelper` to get a list of matching tags from the cloud. The `SoundletManager`, with the help of `CommunicationManager`, manages the communication between the phone and the cloud to upload soundlets and to download classification plans. The `SoundletDetector` registers the running application as a listener for desired sound events, initializes the `SoundEngine`, and notifies the application at desired sound events.

7. IN-CLOUD COMPONENTS

The primary job of the cloud service is to generate an energy-aware acoustic event detection plan which we describe in detail. Other services such as tag suggestion, storage and caching are described briefly.

7.1 Energy Aware Classification Plan

Given a set of tags and phone context, a service in the cloud creates an energy aware acoustic event detection plan.

7.1.1 *Training Set Generation*

The cloud creates a training set by choosing a subset of the soundlets in the cloud. First, it selects the soundlets having all of the *container* tags – indicating that the training is environment specific. It then filters out soundlets that do not contain any of the *lookfor* or *within* tags assuming that those are not present in the acoustic environment. Among the remaining soundlets, the ones having all of the *content* tags are marked as positive examples, and the rest are marked as negative examples. Finally, all 121 acoustic features are extracted to form an extended training set, which is then reduced by a feature selection algorithm.

7.1.2 *Feature Selection Algorithm*

Unlike feature selection [9] in the machine learning literature, where the criterion is to minimize the *number* of features, Auditeur's goal is to minimize the total *energy consumption* of feature extraction on the phone, while maximizing their ability to retain enough information for an accurate classification. A brute force approach that enumerates all 2^{121} subsets is not feasible; hence we take a dynamic programming approach to this problem. A formal description of the problem and its solution are described next.

Given a training set T of size $n \times m$, where n is the number of samples and m is the dimension of the feature space, the target class labels C of size $n \times 1$, the energy cost of computing each feature on the phone $E = \{e_1, e_2, \ldots e_m\}$, and an energy budget B, the goal is to select a subset of features that uses no more than B units of energy while maximizing their ability to retain enough information for an accurate classification.

Since we are selecting a subset of the features, we need a metric to measure the goodness of a feature. A feature is *relevant* if it is correlated to the target class, and is *non-redundant* if it is not highly correlated to other features. In Auditeur, we take an information theoretic approach in measuring the correlation, which is based on the concept of *entropy*. The entropy of a feature X is defined as

$$H(X) = -\sum_i P(x_i)\log(P(x_i)), \tag{1}$$

and the entropy of X, after observing Y is defined as

$$H(X|Y) = -\sum_j P(y_j)\sum_i P(x_i|y_j)\log(P(x_i|y_j)), \tag{2}$$

where $P(x_i)$ and $P(x_i|y_j)$ are the prior and posterior probabilities of X, respectively. The difference between Equation (1) and (2) denotes the *information gain* [26], indicating how much is the added value of taking feature X, given that Y is already selected. In our algorithm, we use the normalized information gain, which is called the *symmetrical uncertainty* [24], to limit its value in the range [0, 1].

$$SU(X|Y) = 2\left[\frac{H(X) - H(X|Y)}{H(X) + H(Y)}\right], \tag{3}$$

We now formulate a recurrence relation which is solved using

Processing Unit	Parameter	Choices
Sampling	rate (KHz)	8, 22.05, 44.1
Framing	length (ms)	32, 64
Window Size	length (sec)	1, 3, 5
Feature Extraction	discretization (level)	8, 16

Table 2: Choices of parameters for APUs.

dynamic programming technique. Let $f_j(b)$ be the optimal subset of features within an energy bound of b, considering the first j features, for $(0 < j \leq m)$. Based on the decision on X_j, there can be two cases:

$$f_j(b) = \begin{cases} f_{j-1}(b) & \text{if } X_j \text{ is not taken} \\ f_{j-1}(b - e_j) \cup \{X_j\} & \text{otherwise} \end{cases}$$

This is because, if X_j is taken, the bound b is reduced by e_j, the amount of energy required to compute X_j. The decision on X_j depends on the symmetrical uncertainties. X_j is taken only if $SU(X_j|C, f_{j-1}(b - e_j)) < SU(X_j|C, f_{j-1}(b))$, and not taken otherwise. The recurrence is solvable for $f_m(B)$ by beginning with the knowledge of $f_0(b) = \phi$ for all $b > 0$.

There are two implementation issues that require further explanation. First, our algorithm assumes that energy is integer valued; The definition of entropy also requires nominal values. To handle this, we quantize both of them to fit into our algorithm. Second, the energy budget B is for feature extraction only. It is derived by subtracting the energy cost of other APUs and services, e.g., pipeline overhead and context generation service, that run on the phone from the total budget for acoustic processing.

7.1.3 APU and Parameter Selection

The energy consumption and the classification accuracy depend on the choices of parameters for other APUs as well. Table 2 describes the choices of parameters considered in Auditeur for the sampling rate, frame size, window size, and number discretization levels for feature quantization. The higher the sampling rate the more energy it consumes, but provides better classification results. Frame size, window size, and discretization levels are problem dependent, and have effect on accuracy. The choice of classifier is also problem dependent, and their parameters are computed by analyzing the training set using standard practices. Considering all 36 combinations of these units and 7 classifiers, the cloud generates 252 different execution plans (all within the total energy budget), and selects the one that results in the highest accuracy in 10-fold cross validation test.

7.2 Tag Matching Service

The tag matching service is used in two scenarios: (1) admission control of a public soundlet, and (2) finding matching tags for an untagged soundlet. Both of these are done using a distance based outlier detection technique [5], i.e., extracting the feature vector of the soundlet, computing similarity scores of the feature vector against a subset of the existing soundlets, and returning the top k tags in the order of decreasing similarity. For admission control, one or more of the user given tags must be in the list of k tags, and for tag-suggestions, the list of k tags is sent to the phone.

The collection of soundlets against which the similarity is measured is selected using the phone contexts. For example, location (indoors or outdoors) and position contexts are used to eliminate any unlikely soundlets from considerations. If GPS coordinates are available, soundlets that are recorded within the vicinity of the untagged soundlet are always considered for a similarity match.

7.3 Storage and Caching

The cloud stores thousands of tagged soundlets and their acoustic features, therefore fast search and retrieval of items are paramount. We choose to use MongoDB which is a NOSQL database over a relational database for many reasons. First, the number of features for an instance can vary considerably depending on the processing configuration used at the time therefore a schema would be too rigid for our purposes. Secondly, NOSQL databases are optimized for write once, read many times situations and tend to scale linearly using replicas in the cloud infrastructure. For efficient queries into the training set, we create an index on the tag and context attribute set.

When a classification plan is requested from a client, the local cache of prepared classifiers is first searched. If a classifier for the particular tag combination and context are not found, a new classifier is generated from the set of matching instances in the training set and committed to the cache. If a new sound instance is added with the same tag combination as existing classifiers, the cache entry is marked dirty allowing an updated classifier to be created. Our default strategy uses on-demand creation of classifiers since many queries are very specific, however we also allow the cloud to preemptively create new classifiers on a daily basis from popular search queries.

8. EVALUATION

We describe four types of evaluations. First, we measure CPU and memory footprints, and energy consumptions of different processing units of Auditeur (Section 8.2). Second, we perform an empirical study on different categories of sounds to demonstrate the efficacy of our energy-aware feature selection algorithm (Section 8.3). Third, we implement seven apps, each in three ways: in-phone, in-cloud, and using Auditeur, and compare their energy consumption (Section 8.4.1), accuracy (Section 8.4.2) and context awareness (Section 8.4.3). Fourth, we perform a usability study of Auditeur API (Section 8.5).

8.1 Experimental Setup

We collect data from two sources. The first one is our own collection, obtained from a total of 35 participants from two different regions (Virginia and Beijing), collected over nine months, resulting in a database of about 5000 tagged soundlets with contextual information. The group of participants is comprised of undergraduate and graduate students, researchers, professionals, and their family members. Their ages are in the range of $10 - 60$, and they have diversities in speaking-style, life-style and ethnicity. The smartphones we have used during the data collection are Galaxy Nexus phones running Android 4.0 OS, each having a 1.2 GHz TI OMAP4460 dual core processor, 1 GB RAM, 32 GB internal storage, and 28 GB USB storage.

The other source of data is mainly websites including findsounds.com and grsites.com, yielding another 2000 tagged soundlets. The web crawled data is used only in the empirical evaluation, not in case studies. We use only those sound clips whose file format, number of channels, bit resolution, and sampling rates are the same as our phone recordings. We do not mix up web crawled data with data from phones in most cases. For example, sounds that are tagged 'monkey' are all from the web, we do not have any phone recorded monkey sounds. For some rare sounds in the empirical study, we re-recorded some of the web-audios with a smartphone. For example, we play the 'siren' sounds loudly on a laptop and then record that on a smartphone.

	CPU	Memory
Auditeur (silence)	6%	6.5 MB
Auditeur (active)	16% (42%)	11.8 MB (22 MB)

Table 3: CPU and memory footprints.

8.2 System Measurements

8.2.1 *CPU and Memory Footprint*

We measure the CPU and memory footprints of Auditeur when used in an app with a simple GUI. The GUI is used to select different sound recognition tasks that are run during the experiment. We use two utility apps (Norton and OS Monitor) to measure the CPU and memory usages. Table 3 shows the average (and the maximum in brackets) CPU and memory usages of the apps. The CPU usage is only 6% during silence when the system is duty cycling, 16% on average, and could reach up to 42% in the worst case if all of the processing units are used in an app (which is unlikely). The memory usage is about 6.5 MB during silence, but reaches 11.8 MB for an average app, as heap space is allocated for the dynamically instantiated units. However, this is not so high when compared to services such as Maps (62.7 MB), Music (54.9 MB), and Calendar (18.9 MB) on Android. The total size of the binary is only 596 KB.

8.2.2 *Energy Measurements*

We measure the energy consumption of each acoustic processing unit on the phone with a high precision power monitor [2]. Prior to the experiment, we kill all background services and running apps, disable wireless connectivity, and set the screen brightness to its minimum. We use two minute long audio files recorded at 44.1 KHz, and measure the total energy consumption.

Figure 9 shows the average energy consumption to process a 32 ms frame for each unit. We observe that, the most expensive units are the frequency domain feature extractors. However, all of these components (marked with an asterisk (*)) use the same FFT which is computed only once per frame to save energy. Despite this, the extraction of frame level features, together with the computation of window level statistics, account for 98.48% of the total energy consumption of the processing pipeline. This signifies why Auditeur emphasizes carefully choosing the features to increase the lifetime of the device. The sum of individual energy consumptions however is not an accurate estimate of the actual energy consumption; but from our experience with Auditeur we have seen that, this is a conservative estimate, and the phone lasts longer than the estimated lifetime in practice.

8.3 Empirical Evaluation

We analyze the trade-off between the accuracy and energy savings, compare the feature selection algorithm with a greedy heuristic, illustrate the energy efficiency, and measure the processing delay.

We use our empirical dataset in these experiments, which is segmented into 10 categories of sounds as described in Table 4. In this study, we are showing intra-category sound recognition accuracy within each dataset. Note that, these are very demanding datasets with many similar sounds in different categories within the same dataset to stress the accuracy metric, and even for humans these are hard to distinguish. However, later in our case studies (Section 8.4), we consider real-world scenarios where we have sounds from different classes.

8.3.1 *Lifetime and Accuracy Trade-off*

Auditeur's feature selection algorithm trades off less informative features for a longer lifetime of the device. The goal of this experiment is to quantify how much accuracy Auditeur compromises to achieve this.

ID	Dataset	Count	Subcategories
D1	Alert	334	alarm, bell, horn, siren, whistle.
D2	Animal	145	cat, cow, dog, horse, monkey, tiger.
D3	Household	453	air conditioner, doorbell, doorslam, fan, spray, vacuum, water tap, and more.
D4	Instruments	309	drum, guitar, piano, violin, flute.
D5	Music	1253	country, folk, indian, jazz, rap, rhymes, rock, spiritual, and more.
D6	Non-speech	783	asthma whizz, chew, clap, cough, cry, foot-steps, laughter, whistle, yell.
D7	Office	653	deskbell, keyboard typing, mouse clicks, phone, printer, sharpener, stapler.
D8	Speech	1725	female, male, child, speaker ID.
D9	Tools	439	broom, chain saw, drill, grinder, hammer, lawnmower.
D10	Vehicles	815	airplane, ambulance, bus, car, subway.

Table 4: Description of the empirical dataset.

Figure 10 shows the classification accuracy of Auditeur for different datasets. For each dataset, the first bar corresponds to the highest achievable accuracy considering no energy bound, and the other three bars correspond to minimum lifetime requirements of 300, 400, and 500 mins, respectively. The number on top of each bar denotes the number of selected features. We see that, for an unbounded energy, Auditeur would use all 221 features and achieve the highest average 10-fold cross validation accuracy of $65.4\% - 97.2\%$. The accuracy is below 70% for some datasets, such as D7, D3, and D9, which contain varieties of similar sounds. However, in an actual app, not all of these might be used or they might be representing the same class; hence the accuracy will be higher in practice as evident from Section 8.4.2. The tighter the energy bound becomes, i.e. the larger the minimum lifetime requirement is, Auditeur selects less number of features to maintain the lifetime goal. For a 500 min (8.3 hours) lifetime, Auditeur chooses only 18 features on average to keep the system running, sacrificing about 8.17% accuracy. However, for a moderate lifetime of 400 mins (6.67 hours), the difference in accuracy is $< 2\%$.

8.3.2 *Evaluating Feature Selection Algorithm*

We compare Auditeur's energy-aware feature selection algorithm with Weka's [10] symmetrical uncertainly attribute evaluator. Since Weka does not consider energy, we choose the first few highest ranked attributes as long as the sum of their energy costs remains within the bound. Figure 11 compares the average accuracy of a classifier for different minimum lifetime bounds, when the classifier uses these two feature selection methods. When the energy bound is lower, i.e. the minimum lifetime is as high as $500 - 600$ mins, less number of features are chosen by both of the algorithms. But Auditeur selects the subset that is optimal within the bound, whereas the greedy algorithm performs poorly. The gap between these two however gets closer as the bound becomes loose since both algorithms then select enough number of features to classify the sounds properly. Therefore, applications that require long term $(8 - 10$ hours) continuous sound recognition, Auditeur would provide $8\% - 14.9\%$ higher accuracy than the greedy algorithm.

8.3.3 *Illustration of Energy Efficiency*

We illustrate the energy efficiency of Auditeur with a simple example scenario. In this scenario, we run the same workload on two

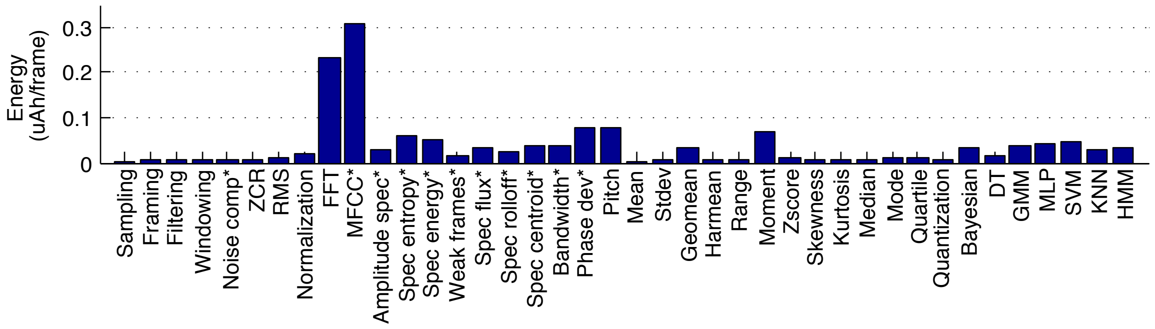

Figure 9: Feature extraction accounts for 98.48% of the total energy consumption of the processing pipeline.

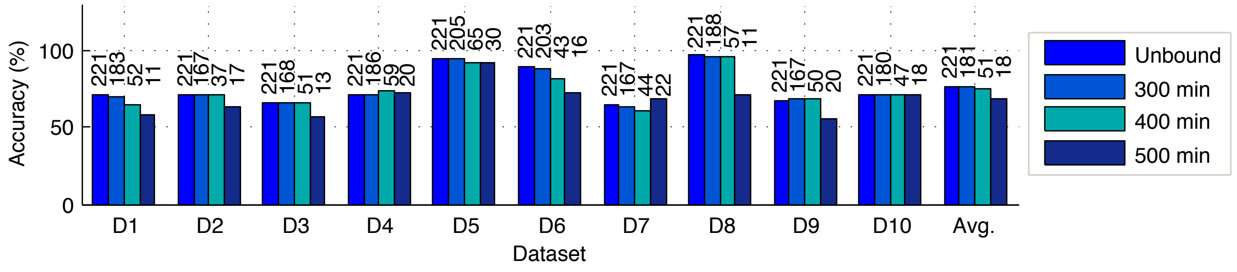

Figure 10: Auditeur increases the lifetime by $33.4\% - 66.7\%$, sacrificing $2\% - 8.17\%$ accuracy.

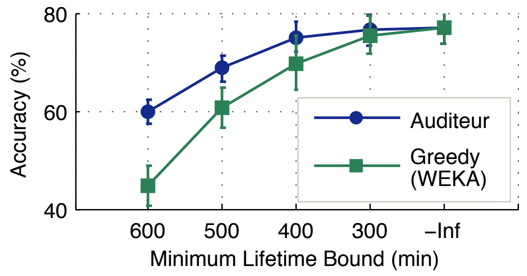

Figure 11: Auditeur achieves $8\% - 14.9\%$ higher accuracy than WEKA for long running apps.

identical Android phones for about 9 hours. Both of the phones run Auditeur, but one assumes an infinite energy bound, and the other one changes its classification plan after two hours to last longer.

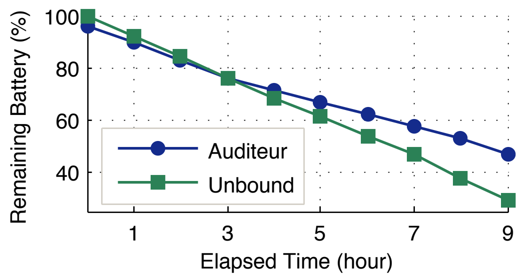

Figure 12: Auditeur's remaining battery is 18.12% higher than the unbounded one after 9 hours.

Figure 12 plots their remaining battery life at each hour mark. We see that, the phone with unbounded energy drains battery at the rate of 7.89% per hour, and after 9 hours, it's remaining charge is only 29%. The other one (denoted by Auditeur) initially drains battery at the rate of 6.67% per hour, but this rate is reduced to 4.81% after reducing the energy bound by 15% at hour 2. After 9 hours, Auditeur's remaining battery is 47.12%, which is 18.12% higher than the unbounded one. That means, Auditeur would last for 31 hours, whereas the unbounded one will die after 19 hours.

8.3.4 Processing Delay

We measure the processing delay, i.e., the average duration of a $1s$ long frame inside the pipeline. To obtain this, we capture $60s$ audio, and process it with 300 different pipeline configurations, each having a different minimum lifetime bound. The total time to process the frames is normalized to compute the processing delay of a frame.

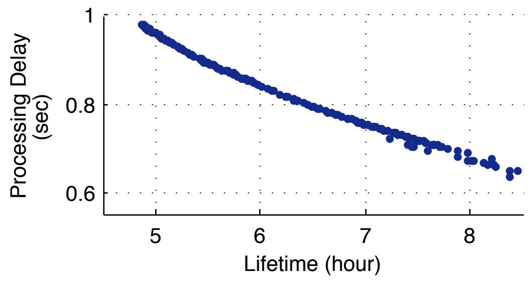

Figure 13: Processing delay for $1s$ frames is always $< 1s$.

Figure 13 shows processing delays of $1s$ long frames for different lifetime bounds, ranging approximately $5 - 8$ hours. We observe that, the delay is higher ($0.98s$) at the beginning, and drops below $0.7s$ as the bound crosses the 8-hour mark. This is because, at tighter bounds, Auditeur extracts a small number of features to meet its lifetime goal, which results in shorter processing delays. However, overall the delay is always $< 1s$ for a $1s$ frame.

8.4 Case Studies

We implement and evaluate seven apps with Auditeur and compare their energy efficiency, accuracy, and context awareness against two baselines: the in-phone and the in-cloud implementations of the apps. The features and parameters for the baselines are taken from corresponding existing works. Both baselines extract features on the phone, but the in-cloud version sends them to the cloud in real-time to get the classification results, while the in-phone version does everything on the device. The definitions of in-phone and in-cloud implementations thus consider any hybrid approach (such as – partitioning or offloading computation to the cloud at runtime [7]) as an in-cloud implementation. For communication, WiFi is used indoors and 3G outdoors. Auditeur however downloads all of its classifier configurations, corresponding to different location and position contexts, at the beginning and does not use wireless connectivity afterwards.

Five Android phones and a tablet are used at the same time during these experiments. Two phones run the baselines, one runs Auditeur, one runs Auditeur with only location context enabled, and one runs Auditeur with only position context enabled. The tablet is used for recording the ground truth and bookkeeping purposes. Each phone stores the timestamp, detected events, and the battery level.

App	**Detected Events**
Sound Sense	male, female, music.
Speaker Sense	person identification.
Musical Heart	heart beats.
Music Match	music genre recognition.
Vehicle Sense	car, bus, subway, trolley.
Kitchen Sense	door, blender, pots, stove, microwave, tap.
Sleep Monitor	talk, cough, steps, bathroom door, fan.

Table 5: Apps in case studies.

Table 5 shows the list of apps that we study. We replicate the first three apps from existing literature [17, 16, 23], and add four more to demonstrate Auditeur's performance in different real-world scenarios. Ground truths for Speaker Sense and Sound Sense are obtained from six volunteers, by logging who is talking and what music is played with a tablet. These experiments are done in ten $15 - 120$ minutes long sessions at indoors and outdoors. Musical Heart uses the dataset of [23] which is read from files. Music Match is trained on 200 English songs of different genre, and then the tablet randomly chooses and plays them at regular intervals, while the phones listen and classify them. Vehicle Sense and Kitchen Sense are trained on samples collected from members of two two-person families, and are tested on them separately in $3 - 5$ hours long experiments. Two volunteers participate in the Sleep Monitor experiment separately for two consecutive days where each session lasts for about $6-8$ hours. We record the entire sleep duration with a tablet, and perform a post-facto analysis, with visualization and manual classification, to identify the events.

8.4.1 Power Consumption

We compare the power consumptions of Auditeur with in-phone and in-cloud implementations. The power consumption is obtained from the total energy consumption, which we calculate using the values that are logged periodically into the phone, i.e. the remaining battery life, voltage, running time, and the battery capacity.

Figure 14 shows that, power consumptions of in-cloud implementations are $3.3 - 6.8$ times higher than the other two, as they continuously send and receive data over the Internet. Especially, for outdoor experiments where 3G is used (Vehicle, Sound, and Speaker), the mean is 4093 mW. Power consumption of in-phone version is comparatively closer to Auditeur. For long running experiments (Sound, Speaker, Vehicle, Kitchen, and Sleep), in-phone ones consume 19.56% more power than Auditeur, as unlike Auditeur, they do not have energy bounds. For short duration experiments (Heart and Music), in-phone versions are slightly more energy efficient than Auditeur. However, this is not a problem for Auditeur because of two reasons: (1) when sufficient energy is available, Auditeur uses slightly more power to achieve a higher detection accuracy, and (2) even in such cases, developers can specify a tighter energy bound to achieve a lower power consumption. Overall, the power consumption of Auditeur is on average 11.04% less than the in-phone version.

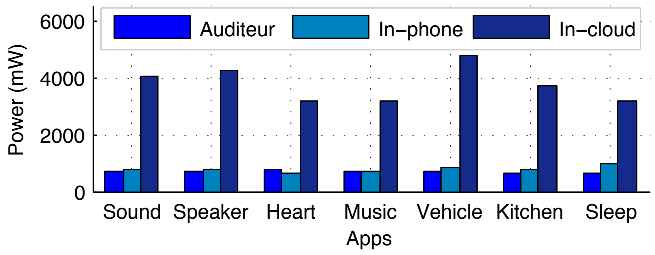

Figure 14: On average, Auditeur is 11.04% and 441.42% less power hungry than in-phone and in-cloud versions.

An Auditeur-powered app's lifetime can be further extended by duty cycling. However, this has to be done by the app developers by explicitly specifying the cycling interval for their app using the API (see line 17 of the code snippet in Figure 4). Since none of the baseline apps in this experiment implement duty cycling, for a fair comparison of energy consumption, we do not perform duty cycling in Auditeur as well.

8.4.2 Detection Accuracy

The accuracy of event detection is the percentage of events that are detected and classified correctly. All three implementations use the same silence vs. non-silence detection method whose accuracy is almost perfect (98.72%). Hence, we consider the classification of non-silent, fixed length time windows as the overall accuracy.

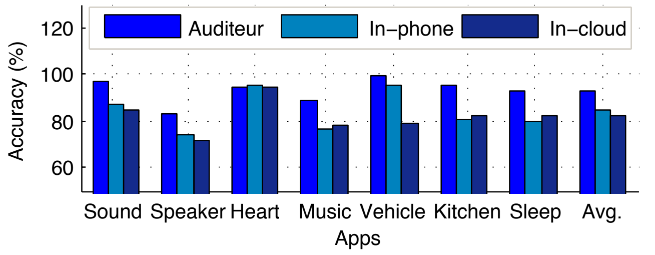

Figure 15: On average, Auditeur is 10.71% and 13.86% more accurate than in-phone and in-cloud versions.

Figure 15 shows the accuracy of event detection for three different implementations of the seven apps. Auditeur shows higher accuracies than in-phone and in-cloud versions in 6 out of seven apps. This is because of Auditeur's adaptiveness to changing contexts, and the limitations of the other two. Both in-phone and in-cloud versions use offline-trained classifiers, which are trained on all soundlets disregarding their contexts. For example, the accuracy of voice related apps, e.g., the Sound and Speaker, depends on the location (indoors vs. outdoors) due to the presence of reverberation. Auditeur being aware of such contextual information are capable of handling them separately, which the other two cannot. Furthermore, the in-cloud versions lose $14.3\% - 26\%$ accuracy in outdoor scenarios (Vehicle, Sound and Speaker), due to long communication delays in a 3G network. The accuracy of Auditeur in case of Heart app however is similar to the other two versions. This is because, the Heart app, which is one of our previous works [23], uses a highly sophisticated algorithm for heart rate detection and has a very high accuracy. It is hard to beat such an algorithm using Auditeur especially when the accuracy is close to 100%. Overall, Auditeur shows $10.71\% - 13.86\%$ higher accuracy than the other two versions.

8.4.3 Context Awareness

Auditeur uses different pipeline configurations for different location (indoors vs outdoors) and body position (direct, pocket, or distant) contexts to boost its accuracy. In this experiment, we measure their individual effects on accuracy.

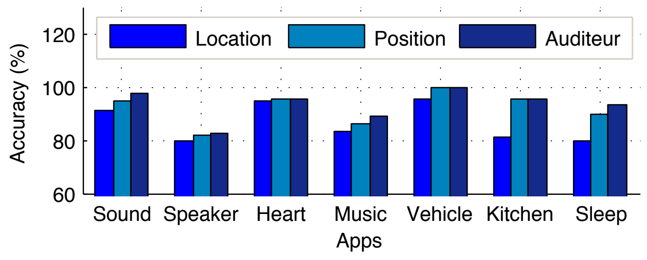

Figure 16: Position context is in general more effective than location context.

Figure 16 compares Auditeur with two other versions of it: (1) only location, or (2) only position context enabled. A combination of the two always results in a higher accuracy, but their individual contribution varies with apps. Location context is ineffective in Heart, Vehicle, Kitchen, and Sleep, since these are performed entirely indoors or outdoors. For other apps, location alone improves accuracy over baselines (of Figure 15) by $4.08\% - 7.93\%$. Position contexts are more effective than location contexts in environments where the phone may be in any of the three position contexts, such as in Sound, Speaker, Music, Kitchen (pocket and table), and Sleep (table and under pillow), contributing 6.45% more accuracy over the location only version.

8.5 User Study

We perform a user study to evaluate the usability of Auditeur API, i.e. how easy or hard it is to learn and use the API. Another motivation is to observe what application ideas burgeon when such an API is made available to the public.

A total of 15 undergraduate students from 3 universities participated in this study. We recruited them by posting an online advertisement and their participation was voluntary. Their experience-levels in Java and Android programming are $0 - 5$ years (avg. 16.6 months), and $0 - 12$ months (avg. 4.73 months), respectively. We provided them with a documentation explaining the API and an example code snippet. Each of the participants was asked to read the documentation to learn the API, think of an app scenario, code it, and then comment on their overall experience.

Since developing a fully functioning application involves other non-sound recognition tasks, such as, programming the GUI and handling events, which are not part of our study, we ask the participants to code only the portion of the app that is Auditeur-specific, while the rest of the app has already been coded for them. This is done to remove any biases that originate from sources unrelated to learning or coding with the Auditeur API. Figure 17 shows the learning time and coding time as they have reported in their answers. About 75% of the participants take less than 15 mins to learn the API, and about 60% of them program the core logic in just 10 mins using only $15 - 20$ lines of Java code. The worst case learning time is as high as 30 mins, while the highest of the coding times is about an hour. This indicates that, Auditeur is very easy to learn and code.

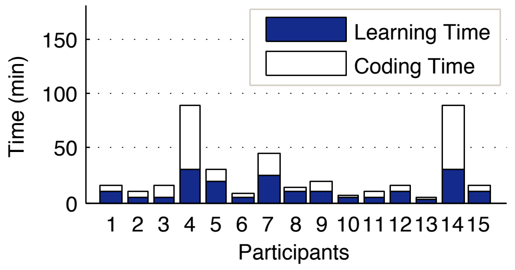

Figure 17: Learning and coding time.

It is interesting to observe how our participants have come up with a number of fascinating app ideas and have been able to realize them with Auditeur. Table 6 describes some of the ideas that we find more interesting than others.

App	Short Description
Snore	Detect snoring and vibrate the phone until he stops.
Honk	Detect car honks on road and alert inattentive user.
Asthma	Detect asthmatic wheezing sound and alert the caregiver.
Subway	Detect the arrival of a train in subway.
Dog	Detect pet dog barking while the user is away from home.

Table 6: Interesting app ideas found in our study.

We also ask our participants to comment on the Auditeur API. Some of those comments turn out to be helpful and are actually incorporated in the system. For example, the participant who implemented the Dog app, asked if we could provide an API to get the last few seconds of audio, so that she knew what caused her dog to bark (e.g. an intruder came). Another participant asked if we could provide a confidence of a match instead of just the detected event. These comments have led us to expose some internal API to obtain last few minutes of audio and the probability distribution of the detected events from the application layer.

9. DISCUSSION

Auditeur is capable of detecting sound events that are recognizable from short-term time and frequency domain features. Recognizing sounds that depend heavily on temporal variations, such as recognizing long spoken sentences, are not suitable for it. Especially, speech recognition is a complex, well-studied problem having several well-known services, and hence we leave it out of the scope of Auditeur.

To make sure that people do not upload soundlets with wrong labels, Auditeur performs sanity checks and discards the sound if there is a lack of similarity. This was not an issue in our experiments since the data were uploaded by our trusted users and we only had to deal with unintentional human errors. However, this could be a big issue when the system is open to the public. A more sophisticated sanity checking mechanism will be required at that moment. This is a limitation of our current implementation and we leave it as future work.

Lack of sufficient training examples, specially for private spaces, is another issue. To cope with this, we suggest developers to utilize sounds from public spaces as long as the private space is small, while keep adding new soundlets to the private space. Auditeur provides an API to retrieve last few minutes of audio; developers should use this facility and upload new soundlets via user-feedback.

The features and classifiers used in Auditeur are shown to recognize varieties of sounds, but yet no such list is ever exhaustive. However, Auditeur is an extensible system where addition of a new feature extractor or a new classifier unit is easy since APUs have a generic structure and are dynamically wired. Hence, such extensions do not require any change to the framework.

The energy consumption and lifetime depend on the phone model and battery capacity. The relative energy costs of the processing units are, however, implementation dependent, and remain the same. This is, therefore, a scaling problem, and the lifetime bound for such cases should be considered as power levels instead of an absolute value. We leave it as a future work to remotely profile energy consumption of the device to provide device tailored lifetime values.

Scalability of the services running on the cloud is not addressed in this paper which is by itself a well-formed problem. Our current implementation is a proof of concept which runs on an Amazon EC2 instance. However in future we plan to move our server to Google App Engine which provides automatic scaling of apps without requiring us to manage the machines by ourselves.

10. RELATED WORK

Acoustic sensing on smartphones has been used in several works. Examples of voice and music sensing applications are, speaker identification [16], speech recognition [31], emotion and stress detection [18, 28], conversation and human behavior inference [20, 21], music recognition, search and discovery [32, 3]. There are some limited number of applications that consider other types of sounds. MusicalHeart [23] uses a special-purpose microphone to count heart beats, [14] counts coughs, Nericel [22] detects horns, Ear-phone [29] monitors noise pollution, SurroundSense [4] and CSP [6] infer logical location. All these systems detect only one specific type of sound. SoundSense [17] distinguishes voice from music and noise, and cluster other sounds. But unlike Auditeur, none of these systems aim at solving the general purpose acoustic event detection problem.

There are several web services available for smartphone developers. Hawaii [1] provides social mobile sharing (SMASH) service for rapid prototyping of social computing apps, service to predict user's destination using current route, key-value storage for app-wide state information, text translation service, providing relay point in the cloud for apps' communication, optical character recognition service (OCR), and English speech-to-text services. But Hawaii does not have an acoustic event detection service. Google provides services such as: web search, Google maps, app store (Google play), video streaming (YouTube), Google cloud drive, and email service (Gmail). Other providers give access to data sources such as weather, financial data, airline information, and parcel location tracking. Unlike Auditeur, none of these provide a sound recognition service and API to the developers.

Feature selection is a problem studied by many; [9] provides a survey. WEKA [10] implements several of these attribute selection methods. But our problem is different as our selection criterion is to minimize energy consumption, not to minimize the number of features. Like Auditeur, symmetrical uncertainty has been used in [33] as a goodness measure, but their approach is greedy whereas we apply dynamic programming principle.

There are several works that deal with resource aware sensor data classification on smartphones. Kobe [7] is tool to generate energy and latency aware classifiers, but unlike Auditeur, it requires uninterrupted mobile-cloud communication for adaptive code-offloading (so is required for MAUI [8] and Odessa [27]), it is not sensitive to user context, and their set of features is predetermined and fixed by the developer. Orchestrator [13] enables multiple applications to effectively share resources (e.g. sensors) whose availability changes dynamically. The optimum plan selection problem in Orchestrator is similar to the acoustic feature selection problem in Auditeur. However, unlike Auditeur, their selection algorithm is not scalable as they enumerate an exponential number of plans while Auditeur uses a polynomial time dynamic programming algorithm to eliminate the need for enumeration of exponential number of subsets of features and thus limits the total number of plans to consider. [30] proposes a heuristic algorithm to select a subset of sensors and their tolerance levels so that they can infer multiple contexts in an energy efficient manner. Their solution is efficient, but is not optimum, whereas feature selection in Auditeur is both optimum and efficient.

Some works are technically similar to Auditeur, but solve different problems. TagSense [25] senses people, activity and context in a picture, and creates tags on-the-fly, whereas Auditeur involves people in the sound tagging process, but provides automated tag suggestions. Sphinx [31] uses generic processing units which are similar to the APUs in Auditeur, but they have a simpler pipeline, highly tuned to solve speech recognition problem. Pickle [15] creates privacy preserving classifiers on the cloud whereas we preserve privacy by providing the user with private spaces, and classifiers in Auditeur are trained for energy efficiency.

11. CONCLUSION

This paper presents the design, implementation, and evaluation of Auditeur which is a general-purpose, energy-efficient, and context-aware acoustic event detection platform for smartphones. This is a useful platform and would save months of development time for developers who want to build apps that act upon acoustic events. Aside from its ability to recognize a wide variety of sounds, the platform is shown to be energy efficient and accurate. We provide empirical evidence that Auditeur's energy-aware algorithm is capable of increasing the device-lifetime by 33.4%, sacrificing less than 2% of the maximum achievable accuracy. Seven apps have been implemented with the Auditeur API to demonstrate its versatility and to show that apps developed with Auditeur are $11.04\% - 441.42\%$ less power hungry, and $10.71\% - 13.86\%$ more accurate in detecting acoustic events compared to state-of-the-art techniques. A user study on 15 undergrads shows that even novice programmers can implement the core logic of interesting apps with Auditeur in less than 30 minutes, using only $15 - 20$ lines of Java code.

Acknowledgments

This paper was supported, in part, by NSF grants CNS-1035771 and EECS-901686, and by DGIST via funding from the Ministry of Education, Science and Technology of the Republic of Korea.

12. REFERENCES

[1] Microsoft's Project Hawaii. http://research.microsoft.com/en-us/projects/hawaii/.
[2] Power Monitor. http://msoon.com/LabEquipment/PowerMonitor/.
[3] Sound Hound. http://www.soundhound.com/.
[4] M. Azizyan, I. Constandache, and R. Roy Choudhury. SurroundSense: mobile phone localization via ambience fingerprinting. In *MobiCom '09*, Beijing, China.
[5] S. D. Bay and M. Schwabacher. Mining distance-based outliers in near linear time with randomization and a simple pruning rule. In *KDD '03*, Washington, D.C.

[6] Y. Chon, N. D. Lane, F. Li, H. Cha, and F. Zhao. Automatically characterizing places with opportunistic crowdsensing using smartphones. In *UbiComp '12*, Pittsburgh, PA.

[7] D. Chu, N. D. Lane, T. T.-T. Lai, C. Pang, X. Meng, Q. Guo, F. Li, and F. Zhao. Balancing energy, latency and accuracy for mobile sensor data classification. In *SenSys '11*, Seattle, Washington.

[8] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: making smartphones last longer with code offload. In *MobiSys '10*, San Francisco, CA.

[9] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182, Mar. 2003.

[10] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update. *SIGKDD Explorations*, 11(1), 2009.

[11] Y. Ju, Y. Lee, J. Yu, C. Min, I. Shin, and J. Song. SymPhoney: a coordinated sensing flow execution engine for concurrent mobile sensing applications. In *SenSys '12*, Toronto, Canada.

[12] S. Kang, J. Lee, H. Jang, H. Lee, Y. Lee, S. Park, T. Park, and J. Song. SeeMon: scalable and energy-efficient context monitoring framework for sensor-rich mobile environments. MobiSys '08, Breckenridge, CO, USA.

[13] S. Kang, Y. Lee, C. Min, Y. Ju, T. Park, J. Lee, Y. Rhee, and J. Song. Orchestrator: An active resource orchestration framework for mobile context monitoring in sensor-rich mobile environments. In *PerCom '10*, Mannheim, Germany.

[14] E. C. Larson, T. Lee, S. Liu, M. Rosenfeld, and S. N. Patel. Accurate and privacy preserving cough sensing using a low-cost microphone. In *UbiComp '11*, Beijing, China.

[15] B. Liu, Y. Jiang, F. Sha, and R. Govindan. Cloud-enabled privacy-preserving collaborative learning for mobile sensing. In *SenSys '12*, Toronto, Canada.

[16] H. Lu, A. J. B. Brush, B. Priyantha, A. K. Karlson, and J. Liu. SpeakerSense: energy efficient unobtrusive speaker identification on mobile phones. In *Pervasive '11*, San Francisco, CA.

[17] H. Lu, W. Pan, N. D. Lane, T. Choudhury, and A. T. Campbell. SoundSense: scalable sound sensing for people-centric applications on mobile phones. In *MobiSys '09*, Poland.

[18] H. Lu, M. Rabbi, G. T. Chittaranjan, D. Frauendorfer, M. S. Mast, A. T. Campbell, D. Gatica-Perez, and T. Choudhury. StressSense: Detecting stress in unconstrained acoustic environments using smartphones. In *UbiComp '12*, Pittsburgh, PA.

[19] H. Lu, J. Yang, Z. Liu, N. D. Lane, T. Choudhury, and A. T. Campbell. The jigsaw continuous sensing engine for mobile phone applications. In *SenSys '10*, Zurich, Switzerland.

[20] E. Miluzzo, C. T. Cornelius, A. Ramaswamy, T. Choudhury, Z. Liu, and A. T. Campbell. Darwin Phones: the evolution of sensing and inference on mobile phones. In *MobiSys '10*, San Francisco, CA.

[21] E. Miluzzo, N. D. Lane, K. Fodor, R. Peterson, H. Lu, M. Musolesi, S. B. Eisenman, X. Zheng, and A. T. Campbell. Sensing meets mobile social networks: the design, implementation and evaluation of the CenceMe application. In *SenSys '08*, Raleigh, NC.

[22] P. Mohan, V. N. Padmanabhan, and R. Ramjee. Nericell: Using mobile smartphones for rich monitoring of road and traffic conditions. In *SenSys '08*, Raleigh, NC, 2008.

[23] S. Nirjon, R. Dickerson, Q. Li, P. Asare, J. Stankovic, D. Hong, B. Zhang, G. Shen, X. Jiang, and F. Zhao. MusicalHeart: A hearty way of listening to music. In *SenSys '12*, Toronto, Canada.

[24] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical recipes in C (2nd ed.): the art of scientific computing*. Cambridge University Press, New York, NY, USA, 1992.

[25] C. Qin, X. Bao, R. Roy Choudhury, and S. Nelakuditi. TagSense: a smartphone-based approach to automatic image tagging. In *MobiSys '11*, Bethesda, MD.

[26] J. Quinlan. *C4.5: Programs for machine learning*. Morgan Kaufmann, 2002.

[27] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan. Odessa: enabling interactive perception applications on mobile devices. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, Bethesda, Maryland, USA.

[28] K. K. Rachuri, M. Musolesi, C. Mascolo, P. J. Rentfrow, C. Longworth, and A. Aucinas. EmotionSense: a mobile phones based adaptive platform for experimental social psychology research. In *Ubicomp '10*, Copenhagen, Denmark.

[29] R. K. Rana, C. T. Chou, S. S. Kanhere, N. Bulusu, and W. Hu. Ear-phone: An end-to-end participatory urban noise mapping system. In *IPSN '10*, Stockholm, Sweden.

[30] N. Roy, A. Misra, C. Julien, S. K. Das, and J. Biswas. An energy-efficient quality adaptive framework for multi-modal sensor context recognition. In *PerCom '11*, Washington, DC, USA.

[31] W. Walker, P. Lamere, P. Kwok, B. Raj, R. Singh, E. Gouvea, P. Wolf, and J. Woelfel. Sphinx-4: a flexible open source framework for speech recognition. Technical report, Mountain View, CA, 2004.

[32] A. Wang. An industrial-strength audio search algorithm. In *ISMIR '03*.

[33] L. Yu and H. Liu. Feature selection for high-dimensional data: A fast correlation-based filter solution. In *International Conference on Machine Learning*, pages 856–863, 2003.

[34] P. Zhou, Y. Zheng, Z. Li, M. Li, and G. Shen. IODetector: A generic service for indoor outdoor detection. In *SenSys' 12*, Toronto, Canada.