Balancing Energy, Latency and Accuracy for Mobile Sensor Data Classification

David Chu[1] davidchu@microsoft.com
Nicholas D. Lane[2] niclane@microsoft.com
Ted Tsung-Te Lai[3†] tedlai@csie.ntu.edu.tw
Cong Pang[4†] pangcong@nus.edu.sg
Xiangying Meng[5†] xiangying.meng@pku.edu.cn
Qing Guo[6] guoqing@microsoft.com
Fan Li[2] fali@microsoft.com
Feng Zhao[2] zhao@microsoft.com

[1]Microsoft Research, Redmond, WA
[2]Microsoft Research Asia, Beijing, China
[3]National Taiwan University, Taipei, Taiwan
[4]National University of Singapore, Singapore
[5]Peking University, Beijing, China
[6]Microsoft, Shanghai, China

Abstract

Sensor convergence on the mobile phone is spawning a broad base of new and interesting mobile applications. As applications grow in sophistication, raw sensor readings often require classification into more useful application-specific high-level data. For example, GPS readings can be classified as running, walking or biking. Unfortunately, traditional classifiers are not built for the challenges of mobile systems: energy, latency, and the dynamics of mobile.

Kobe is a tool that aids mobile classifier development. With the help of a SQL-like programming interface, Kobe performs profiling and optimization of classifiers to achieve an optimal energy-latency-accuracy tradeoff. We show through experimentation on five real scenarios, classifiers on Kobe exhibit tight utilization of available resources. For comparable levels of accuracy traditional classifiers, which do not account for resources, suffer between 66% and 176% longer latencies and use between 31% and 330% more energy. From the experience of using Kobe to prototype two new applications, we observe that Kobe enables easier development of mobile sensing and classification apps.

Categories and Subject Descriptors

C.5.3 [**Computer System Implementation**]: Microcomputers—*Portable devices*

General Terms

Algorithms, Design, Performance

Keywords

sensors, classification, optimization, mobile devices, smartphones

1 Introduction

Mobile devices are increasingly capable of rich multi-modal sensing. Today, a wealth of sensors including camera, microphone, GPS, accelerometers, proximity sensors, ambient light sensors, and multi-touch panels are already standard on high- and mid-tier mobile devices. Mobile phone manufacturers are already integrating a host of additional sensors such as compass, health monitoring sensors, dual cameras and microphones, environmental monitoring sensors, and RFID readers into next generation phones. The convergence of rich sensing on the mobile phone is an important trend – it shows few signs of abating as mobile phones are increasingly the computing platform of choice for the majority of the world's population. As a result of this sensor-to-phone integration, we are beginning to see continuous sensing underpin many applications [22, 23, 18, 2].

Yet as sensing becomes richer and applications become more sophisticated, sensor readings alone are typically insufficient. Mobile applications rarely use raw sensor readings directly, since such readings do not cleanly map to meaningful user context, intent or application-level actions. Rather, mobile applications often employ sensor *classification* to extract useful high-level inferred data, or Application Data Units (ADUs). For example: a human activity inference application might sift through microphone and accelerometer data to understand when an individual is in a meeting, working alone, or exercising [22]; a transportation inference application might look at patterns in GPS and WiFi signals to determine when an individual takes a car, bike or subway [33]; or an augmented reality application might process the camera video feed to label interesting objects that the individual is viewing through the camera lens [2]. These examples span the gamut from rich multi-sensor capture to simple sensor streams. Yet, each application involves non-

[†]Work performed while at Microsoft Research

trivial conversion of raw data into ADUs.

The idea of mapping low-level sensor readings (collected on mobile phones or otherwise) to high-level ADUs has a long history. *Statistical Machine Learning* (SML) has been identified as offering good mathematical underpinnings and broad applicability across sensor modalities, while eschewing brittle rule-based engineering [9]. For example, activity recognition and image recognition commonly employ SML [7]. SML tools almost exclusively drive ubiquitous computing building blocks such as gesture recognition and wearable computing [19]. To assign correct ADU class labels to sensor readings, these scenarios all use *continuous classification* – classification that is repeated over some extended time window and is driven by the rate of sensor readings rather than explicit user requests.

Given this rich body of work, SML seems to offer an elegant solution for mobile sensing application development. The battery of mature SML classification algorithms directly address the problem of ADU construction from base sensor readings.

Unfortunately, SML classification algorithms are not purpose-built for mobile settings. Symptoms from early classifier-based mobile applications include swings of up to 55% in classification accuracy [22, 23], erratic user experience with response times fluctuating from user to user [11, 7, 17], and undesirable drop-offs of phone standby time from 20 hours to 4 hours due to unanticipated energy consumption [22]. Moreover, anecdotal evidence indicates that successful prototypes take tedious hand tuning and are often too brittle beyond the lab [22, 20].

In response to these symptoms, we have developed Kobe which is comprised of: a SML classifier programming interface, classifier optimizer, and adaptive runtime for mobile phones. Kobe offers no SML algorithmic contributions – rather, it extends systems support to app developers daunted by getting their mobile classifiers right. Specifically, Kobe addresses the following three challenges.

First, traditional classifiers designed for non-mobile environments target high accuracy but ignore latency and energy concerns pervasive in mobile systems. In contrast, mobile apps need classifiers that offer reasonable trade-offs among accuracy, latency and energy. With Kobe, developers supply accuracy, energy and latency constraints. Kobe identifies *configurations* that offer the best accuracy-to-cost tradeoff. Configurations can differ in the following ways.

- *Classifier-specific parameters.* As a simple example, an acoustic activity classifier may be configured to: take either longer or shorter microphone samples per time period; use either a greater or fewer number of Fourier transform sample points for spectral analysis, or; use higher or lower dimensional Gaussian distributions for Gaussian mixture model based classification. Parameter choices affect both accuracy and cost.
- *Classifier Partitioning.* Classifiers may be partitioned to either partially or entirely offload computation to the cloud. In the example above, the cloud may support offload of the Fourier computation, the modeling, or both. Previous work [5, 11] has looked closely at online cloud offload, and Kobe adopts similar techniques. Partitioning choices affects cost but not accuracy.

In contrast to previous work, Kobe's exclusive focus on classifiers permits it to perform extensive offline classifier profiling to determine Pareto optimal accuracy-to-cost tradeoffs. Offline profiling occurs on cluster servers, with only a small amount of configuration data stored on the phone. This shifts the overhead of optimal configuration search from tightly constrained online/mobile to the more favorable offline/cluster. As a result, Kobe classifiers are optimally-balanced for accuracy and cost, and operate within developer-defined constraints at low runtime overhead.

Second, traditional classifiers are not built to target the wide range of *environments* that mobile classifiers encounter: Networking and cloud availability fluctuates, user usage patterns vary, and devices are extremely heterogeneous and increasingly multitasking which cause dynamic changes in shared local resources of memory, computation and energy. In response, Kobe leverages the optimization techniques described above and identifies configurations under a range of different environments as characterized by network bandwidth and latency, processor load, device and user. For each environment, Kobe identifies the optimal configuration. During runtime, whenever an environment change is detected, the Kobe runtime reconfigures to the new optimal classifier.

Third, mobile application logic and the classifiers they employ are too tightly coupled. The two are inextricably intertwined because of the tedious joint application-and-classifier hand tuning that goes into getting good accuracy (not to mention latency and energy). Kobe provides a SQL-like interface to ease development and decouple application logic from SML algorithms. Moreover, we demonstrate that the decoupling allows two simple but effective *query optimizations*, namely, short-circuiting during pipeline evaluation and the substitution of costly N-way classifiers when simpler binary classifiers will suffice; as well as allowing classifier algorithm updates without application modification.

We evaluated Kobe by porting established classifiers for five distinct classification scenarios, and additionally used Kobe to prototype two new applications. The five scenarios were: user state detection, transportation mode inference, building image recognition, sound classification, and face recognition. From our in-house experience, we found applications straightforward to write, and SML classifiers easy to port into Kobe, with an average porting time of one week. Moreover, using established datasets, Kobe was able to adapt classification performance to tightly fit all tested environmental changes, whereas traditional classifiers, for similar accuracy levels, suffered between 66% and 176% longer latencies and used between 31% and 330% more energy. Furthermore, Kobe's query optimizer allowed additional energy and latency savings of 16% and 76% that traditional, isolated classifiers do not deliver. Lastly, from the experience of using Kobe to prototype two new applications, we observe that Kobe decreases the burden to build mobile continuous classification applications.

Our contributions are as follows.

- We present an approach to optimizing mobile classifiers for accuracy and cost. The optimization can run entirely offline, allowing it to scale to complex classifiers with many configuration options.

- We show our approach can also build adaptive classifiers that are optimal in a range of mobile environments.
- We show that our SQL-like classifier interface decouples app developers from SML experts. It also leads to support for two query optimizations.
- The Kobe system realizes these benefits, and is extensively evaluated on several classifiers and datasets.

The paper is organized as follows. §2 reviews several mobile classifiers. §3 delves into the challenges inhibiting mobile classifiers. §4 provides a brief system overview. §5 presents the Kobe programming interface. §6 discusses the system architecture. §7 details the implementation. §8 evaluates Kobe. §9 discusses related work, and §10 discusses usage experiences and draws conclusions.

2 Example Classifiers and Issues

SML classification is implemented as a data processing pipeline consisting of three main stages. The *Sensor Sampling (SS)* stage gets samples from sensors. The *Feature Extraction (FE)* stage converts samples into (a set of) *feature vectors*. Feature vectors attempt to compactly represent sensor samples while preserving aspects of the samples that best differentiate classes. The *Model Computation (MC)* stage executes the classification algorithm on each (set of) feature vector, emitting an ADU indicating the class of the corresponding sample. The MC stage employs a model, which is *trained* offline. Model training ingests a corpus of training data, and computes model parameters according to a model-specific algorithm. We focus on supervised learning scenarios with *labeled training data*, in which each feature vector of the training data is tagged with its correct class.

SML experts measure a classification pipeline's performance by its *accuracy*: the percentage of samples which it classifies correctly.[1] SML experts seek to maximize the generality of their model, measuring accuracy not only against training data, but also against previously unencountered test data. At best, energy and latency concerns are nascent and application-specific [20, 31].

To make things concrete, we walk through four representative mobile classifiers, and highlight the specific challenges of each.

Sound Classification (SC) Sound classifiers have been used to classify many topics including music genre, everyday sound sources, and social status among speakers. The bottleneck to efficient classification is data processing for the high audio data rate. Especially in the face of network variability and device heterogeneity, it can be unclear whether to prefer: local phone processing or remote server processing [11]; and sophisticated classifiers or simple ones [20].

Image Recognition (IR) Vision-based augmented reality systems continuously label objects in the phone camera's field of view [2]. These continuous image classifiers must inherently deal with limited latency budgets and treat energy parsimoniously. Vision pipelines tuned for traditional image recognition are poorly suited for augmented reality [13]. An approach that accounts for application constraints and adapts the pipeline accordingly is needed.

Motion Classification Sensor such as GPS, accelerometer and compass can be converted into user state for use in exercise tracking [3], mobile social networking [22], and personal environmental impact assessment [23]. *Acceleration Classifiers (AC)* can be used for detecting highly-specific ADUs. For example, some have used ACs for senior citizen fall detection, and our own application (§5) detects whether the user is slouching or not slouching while seated. Such app-specific ADUs mean developers must spend effort to ensure their custom classifiers meet energy and latency needs while remaining accurate. Another motion classifier example is using GPS to infer *Transportation Mode (TM)* such as walking, driving or biking. However, naïve sampling of energy-hungry sensors such as GPS is a significant issue [15, 22]. In addition, effective FE routines may be too costly on mobile devices [33]. As a result, practitioners often spend much effort to hand-tune such systems [22, 23].

3 Limitations of Existing Solutions

To address the challenges outlined above, many proposals have emerged. Unfortunately, our exploratory evaluation detailed below found that these options are not fully satisfying.

Energy and Latency To meet mobile energy and latency constraints, developers have sought to either engage in (1) isolated sample rate tuning, or (2) manual FE and MC manipulation [33]. In the former approach, downsampling sensors is appealing because it is straightforward to implement. However, downsampling alone leads to suboptimal configurations, since the remainder of the pipeline is not optimized for the lower sample rate [16, 14]. Conversely, upsampling when extra energy is available is likely to yield negligible performance improvement since the rest of the pipeline is unprepared to use the extra samples. Furthermore, as discussed in §2, the cost of sampling may be dominated by pipeline bottlenecks in FE and MC, and not by SS energy cost. For example, we found that sample rate adjustments in isolation resulted in missed accuracy gain opportunities of 3.6% for TM. Sample rate tuning is at most a partial solution to developing good classification pipelines.

In the latter approach, developers engage in extensive hand tuning of classification algorithms when porting to mobile devices [29]. Algorithm-specific hand tuning can offer good results for point cases, but does not scale. Optimizing FE and MC stages are non-trivial tasks, and it is not obvious which of the many FE- and MC-specific parameters are most suitable. As an example, we found that for a standard IR classifier [8], accuracy and latency were very weakly correlated (0.61), so simply choosing "more expensive settings" does not necessarily yield higher accuracy. As a result, developers may only perform very coarse tuning, such as swapping in and out entire pipeline stages as black boxes, easily

[1]Other success metrics such as true/false positives, true/false negatives, precision and recall, and model simplicity are also applicable, though we focus on accuracy as the chief metric in this work.

overlooking good configurations.

Adaptivity With the complications above, it may not be surprising that app developers are loathed to re-tune their pipelines. However, these fossilized pipelines operate in increasingly dynamic environments. Lack of adaptation creates two issues. Static pipelines tuned conservatively for cost give up possible accuracy gains when resources are abundant, suffering *underutilization*. Conversely, static pipelines tuned aggressively for accuracy often exceed constraints when resources are limited, suffering *overutilization*. As one example, in our tests with IR, we experienced overutilization leading to as much as 263% longer latencies for the nearly the same classification accuracy.

Tight Coupling Ideally, mobile applications and the classifier implementations they employ should be decoupled and able to evolve independently. Unfortunately, these two are currently entangled due to the brittle pipeline tuning that developers must undertake. The result is mobile applications, once built, cannot easily incorporate orthogonal advances in SML classification.

Furthermore, decoupling helps with pipeline reuse and concurrent pipeline execution. Multi-classifier applications are currently rare, since they suffer the single-classifier challenges mentioned above, as well as from the challenges of composition: how should developers construct multi-classifier applications, and what are efficient execution strategies? It turns out that Kobe's decoupling of application and pipeline leads to natural support for multi-classifier scheduling, facilitating multi-classifier apps.

Why Not Just Cloud and Thin Client? Mobile devices should leverage cloud resources when it makes sense. One approach taken by mobile applications is to architect a thin client to a resource rich cloud backend. However, this approach is not the best solution for all scenarios. First, cloud-only approaches ignore the fact that mobile phones are getting increasingly powerful processors. Second, with sensing being more continuous, it is very energy intensive to continuously power a wireless connection for constant uploading of samples and downloading of ADUs. Our experiments confirmed these two observations for our IR and SC pipelines: compared to the energy demands of local computation for phone-only, cloud-only classifiers used between 31-252% more energy for WiFi, 63-77% more for GSM, and 250-259% more for 3G, in line with findings in the literature [6].

4 Overview of Kobe

Kobe is structured as a SQL-like Interface, Optimizer and Runtime (see Fig. 1 and Fig. 2). It supports two workflows designed to cater to two distinct user types, app developers and SML experts. SML experts are able to port to Kobe their latest developed and tested algorithms by contributing brand new *modules* or extending existing ones. Each pipeline module is an implementation of a pipeline stage. Modules expose *parameters* that possibly affect the accuracy-cost balance.[2]

[2]Module parameters should not be confused with model training parameters.

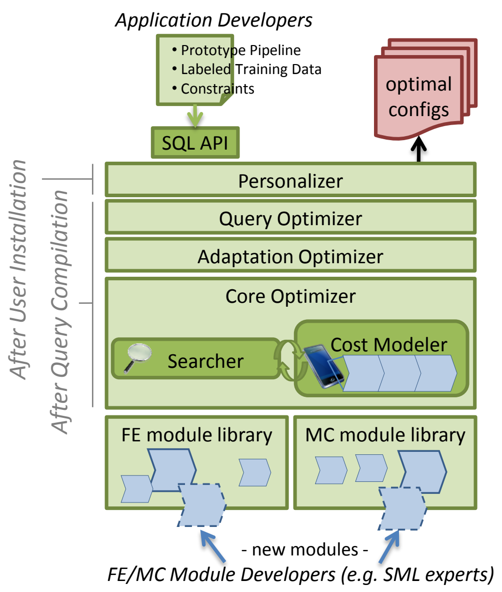


Figure 1. The Kobe Optimizer generates configuration files offline.

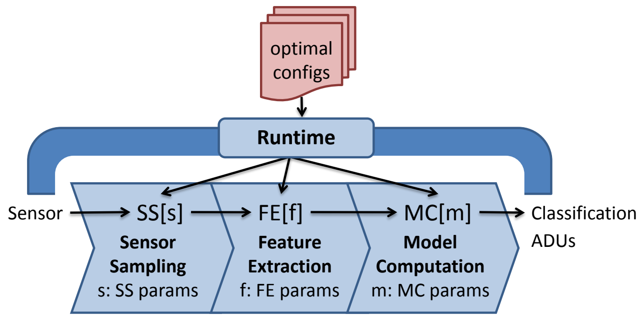


Figure 2. The Kobe Runtime. Note pipeline stages FE and MC may either run on the phone or in the cloud.

Module contributions join an existing library of Kobe modules that app developers can leverage.

App developers incorporate classification into their mobile applications by simply supplying: (1) one or more SQL-like queries, (2) training data, (3) app-specific performance constraints on accuracy, energy or latency, and (4) prototype pipelines consisting of desired modules.

The common Kobe workflow begins with the query submitted to the Optimizer. The Optimizer (Fig. 1) performs an offline search for pipeline *configurations* that meet app constraints. Configurations are modules with parameter values set. The Runtime selects and executes the best configuration at runtime, changing among configurations as the environment changes (Fig. 2).

The Optimizer consists of a series of nested optimizers. First, the Query Optimizer is responsible for multi-pipeline and query substitution optimizations. It invokes the Adaptation Optimizer for this purpose potentially multiple times to assess different query execution strategies. The responsibility of the Adaptation Optimizer is to explore environmental pipeline configuration variations. Each call to the Adaptation Optimizer typically will result in multiple calls to the

Core Optimizer. The role of the Core Optimizer is to profile the costs of all candidate pipeline configuration for a given environment. The Personalizer can further tune the pipeline based on an individual's observed data over time.

Prior to deployment the Optimizer completes its search of candidate pipeline configurations and selects the subset which are Pareto optimal. The output of this process is a mapping of the Pareto optimal pipelines to the appropriate discretized environments. Later, post deployment, the Kobe Runtime installs the appropriate pipeline configuration based on observed environmental conditions. A pipeline change can entail running lower or higher accuracy or cost configurations at any stage, as well as repartitioning stages between phone and cloud. §6 examines the Optimizer and Runtime in more depth.

5 Programming Interface

Kobe decouples the concerns of app developers from SML experts by interposing a straightforward SQL-like interface.

App Developer Interface. App developers construct classifiers with the *CLASSIFIER* keyword.

```
MyC = CLASSIFIER(TRAINING_DATA,
                 CONSTRAINTS,
                 PROTOTYPE_PIPELINE)
```

The *TRAINING_DATA* represents a set of pairs, each pair consisting of a sensor sample and a class label. While obtaining labeled data does impose work, it is a frequently used approach. Partially-automated and scalable approaches such as wardriving, crowdsourcing and mechanical turks can be of some assistance. The *CONSTRAINTS* supplied by the developer specify a latency cap per classification, an energy cap per classification, or a minimally acceptable expected classification accuracy. The *PROTOTYPE_PIPELINE* specifies an array of pipeline modules since it is not uncommon for the developer to have some preference for the pipeline's composition. For example, a pipeline for sound classification may consist of an audio sampling module, MFCC[3] module, followed by a GMM[4] module. Example parameters include the sampling rate and resolution for the audio module; MFCC's number of coefficients (similar to an FFT's number of sample points); and GMM's number of Gaussian model components used.

The SQL-like interface naturally supports standard SQL operations and composition of multiple pipelines. At the same time, this interface leaves certain execution decisions purposely unspecified. Consider the following two applications and their respective queries.

Example 1: Offict Fit Office workers may need occasional reminders to lead healthier lifestyles around the office. *Offict Fit* cues users on contextually relevant opportunities such as taking the stairs in lieu of the elevator, and sitting up straight rather than slouching. User states are detected via continuous sampling of the accelerometer. *Offict Fit* also identifies intense periods of *working* consisting of *sitting* (via accelerometer) and *typing* (via microphone). An AC differentiates among the user states mentioned. An SC checks whether sounds are typing or non-typing sounds.

Example 2: Cocktail Party Suppose that at a social gathering, we wish to automatically record names of other people with whom we've had conversations, but only if they are our coworkers. The application uses a continuous image stream and sound stream (*e.g.,* from a Bluetooth earpiece). A classifier is first constructed for classifying sounds as conversations. Second, images of faces are classified as people's names, and the person's name is emitted when the person is a coworker and a conversation is detected.

We first construct a query for *Offict Fit*. Classification can be combined with class filtering (SQL selection) to achieve detection. The following constructs a classifier that classifies ambient sound as one of several sounds ('typing', 'music', 'convo', 'traffic', or 'office-noise'),[5] and returns only the sounds that are typing sounds.

```
NoisePipe = CLASSIFIER(sndTrain, [maxEnergy
    = 500mJ], [AUDIO,MFCC,GMM])

SELECT SndStr.Raw EVERY 30s FROM SndStr
WHERE NoisePipe(SndStr.Raw) = 'typing'
```

Here, *SndStr* is a *data stream* [4]. Unlike static tables, data streams produce data indefinitely. Queries, once submitted, continuously process the stream. *EVERY 30s* indicates that the query should emit an ADU every 30 seconds. *SndStr* is a sound stream that emits sound samples as *SndStr.Raw*. Note that the frequency of accessing *SndStr* stream is not specified by design. We finish the *Offict Fit* example by showing the interface's support for two classifiers on two streams. The second stream, *AccelStr*, is an accelerometer stream that emits accelerometer readings and is classified by an Acceleration Classifier *MotionPipe*.

```
MotionPipe = CLASSIFIER(accelTrain,
    [minAccuracy = 95%],
    [ACC, FFT_SUITE,DTREE])

SELECT 'working' EVERY 30s
FROM AccelStr, SndStr
WHERE MotionPipe(AccelStr.Raw) = 'sitting'
AND NoisePipe(SndStr.Raw) = 'typing'
```

This multi-classifier query with logical conjunction is a simple to express in the SQL-like interface. Also, note that the query does not define the processing order of the two pipelines.

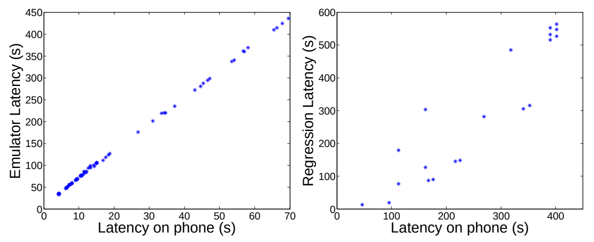
The *Cocktail Party* query similarly combines the results of a face classifier and sound classifier to achieve its objective of identifying conversations with coworkers.

```
FacePipe = CLASSIFIER(mugsTrain,
    [maxLatency = 10s],
    [ACC, FFT_SUITE,DTREE])
```

[3]Mel-frequency cepstral coefficients [12], frequency based features commonly used for sound classification and speaker and speech recognition.

[4]Classification by a mixture of Gaussian distributions [12].

[5]Note class labels are defined by the labeled training data.



(a) Emulation (b) Regression Model

Figure 3. Verifying accuracy of Cost Model techniques

```
SELECT Coworkers.Name EVERY 30s
FROM ImgStr, SndStr, Coworkers
WHERE FacePipe(ImgStr.Raw) = Coworkers.Name
AND NoisePipe(SndStr.Raw) = 'convo'
```

Note that *Coworkers* is a regular SQL table that just lists names of coworkers. It is consulted for a match whenever *FacePipe* classifies a person's face.

SML Expert Interface Each Kobe module corresponds to an implementation of a SS, FE or MC stage in a classification pipeline. While Kobe provides a standard library of modules, SML experts can contribute new algorithmic innovations or scenario-specific feature extractors through new module implementations. For example, even though face recognition is a subclass of image recognition, face-specific feature extraction is common. We implemented several face-specific modules [10] after implementing general purpose image modules [8]. The following describes the details of the Kobe API for connecting new SML modules.

Each module is tuned by the Optimizer via one or more parameters which affect the module's accuracy-cost tradeoff. The SML expert can expose as many parameters as desired, and exposes each parameter by an instance of the *SET_COMPLEXITY(CVAL)* call. The *CVAL* is a continuous [0,1] scalar. As a guideline, the module developer should map higher values of *CVAL* to more costly operating points. For example, a parameter of the image feature extractor is the number of bits used to represent the feature vector; setting this parameter at high complexity results in more feature vector bits. In addition, each parameter suggests a step size in the parameter space with the *GET_SUGGESTED_STEPSZ()* call. The Optimizer uses the step size as a guide when stepping through the parameter space.

6 Architecture

We first describe the Core Optimizer, which consists of the Cost Modeler and Searcher. Afterward, we describe the layers above the Core Optimizer: Adaptation Optimizer, Query Optimizer and Personalizer.

Cost Modeling The Cost Modeler maps pipeline configurations to accuracy and cost profiles that capture phone latency and energy consumption. Ideally, the cost profile of every configuration is profiled by performing measurement experiments using actual phone hardware. However, this is not scalable as manual measurement is required for every candidate combination of pipeline configuration and different phone hardware when searching for Pareto optimal pipelines. During the development of the Cost Modeler we investigated two alternatives for approximating the accuracy of making actual hardware measurements without requiring every configuration to be measured. The first of these uses software emulation, with the second using regression modeling.

Our emulation-based approach executes pipeline configurations using software emulation of the phone platform running on a server. Emulation accurately reflects an actual classifier's performance because, provided faithful training data, classifiers are deterministic functions. We find that we can estimate energy and latency based on the latency observed in the emulator and the use of two simple calibration functions, one for energy and one for latency. To build this function we make around 45 measurements which compare the observed emulator latency with the energy and latency of one single phone model for different pipeline configurations. The calibration functions are simply linear regressions that model real phone latency or real phone energy consumption using the emulator latency as a dependent variable. This calibration function is independent of the pipeline configuration and is only specific to the hardware used, making the number required manageable. When this technique is used to model the cost of a particular configuration the emulator runs the configuration, producing the latency of the emulated pipeline. By applying the calibration functions to the emulation latency, an estimate of actual energy and latency is produced.

Our regression modeling approach trades emulation's fidelity for faster modeling speed. We again make a series of real latency and energy measurements for differing pipeline configurations. A separate multi-factor regression model is fitted for every pair of phone hardware and either FE or MC modules. Each regression model predicts: (1) energy, which incorporates the energy cost to the phone of both local data processing and any required wireless data transfers; and (2) latency, which considers the complete delay from module start (i.e., the sensor is sampled, or model inference begins) to completion (i.e., features extracted or model inference made) – including any delay incurred by data transfer and cloud computation. These predictions are based on a different set of dependent variables for each regression model. Specifically, predictions are based on the particular configuration options exposed by the FE or MC module being modeled. By summing the model estimates for the FE and MC modules defined by the pipeline configuration, we are able to estimate the cost of the complete pipeline configuration. Around 10 to 12 measurements are used for each regression model we use.

The classification accuracy of each pipeline is determined without the use of either of our cost modeling techniques. Instead, for each FE and MC configuration, we use five-fold cross validation over the training data to assign its accuracy.

So far our Cost Modeler has been used on two phones: the HTC Wizard and HTC Touch Pro. We experimentally validate the accuracy of our two Cost Modeling approaches for these two phones. Figure 3(a) and figure 3(b) summarize the results of these experiments for latency under the two approaches. Each data point represents a single validation experiment result in which a single pipeline configuration latency estimate was compared to an actual value. If the estimate and the actual values were perfect then all data points would sit along the diagonal of the graph. The goodness of fit values for these two figures are 99.98 and 92.01 respectively. Similar results supporting our Cost Modeling techniques were found for energy as well. Our final Cost Model is a hybrid of both approaches. We primarily use the regression based Cost Model, and then refine it by the emulator based Cost Model for configurations that appear close to optimal.

Searching Given all the possible configurations, Searcher finds the set of Pareto optimal configurations as judged by accuracy vs. cost. We employ an off-the-shelf search solution, Grid Search [30], because it is embarrassingly parallel and allows us to scale configuration evaluation to an arbitrarily-sized machine cluster. Grid Search determines which configurations to explore, and calls out to the Cost Modeler to retrieve their accuracy and cost profiles. It is during this time that the Searcher explores various sample rates and other module-specific parameters. Grid search is simply one search algorithm, and in principle, we can substitute more efficient (but less parallel) search algorithms [30]. Searcher precomputes all Pareto optimal configurations offline prior to runtime.

Searching also benefits from the fact that pipelines consist of sequential stages. This means that the number of configurations is polynomial, not exponential, in the number of module parameters. In §8, we show that Grid Search scales to support this number of configurations.

Runtime Adaptation and Cloud Offloading The Adaptation Optimizer and Runtime cooperate to support runtime adaptation for both multi-programming, and cloud offload. Kobe tackles the two cases together. Offline, Kobe first discretizes the possible environments into fast/medium/slow networking and heavily/moderately/lightly loaded device processor. Kobe also enumerates the possible mobile and cloud *pipeline placement* options: FE and MC on the mobile; FE on the mobile and MC on the cloud; FE on the cloud and MC on the mobile; FE and MC on the cloud.[6] For each environment discretization, Adaptation Optimizer calls Core Optimizer to compute a Pareto optimal configuration set. The Core Optimizer considers all pipeline placement options in determining the Pareto optimal. The Cost Modeler maintains a collection of regression models for each discrete environment – if cloud servers are involved, the Cost Modeler actually runs the pipeline across an emulated network and server. The collection of all Pareto optimal configuration sets is first pruned of those that do not meet developer constraints. The remainder is passed to the Runtime.

Online, Runtime detects changes to networking latency, bandwidth or processor utilization, and reconfigures to the optimal configuration corresponding to the new environment as well as the application's accuracy and cost constraints. Reconfiguration is simply a matter of setting parameters of each pipeline module. In the case that the new pipeline placement spans the cloud, Runtime also initializes connections with the remote server and sets parameter values on its modules.

[6]SS must occur on the phone since it interfaces to the sensors.

The advantages of this approach are that heavy-weight optimization is entirely precomputed, and reconfiguration is just a matter of parameter setting and (possibly) buffer shipment. The modules need not handle any remote invocation issues. It does require additional configuration storage space on the phone, which we show in §8 is not significant. It also requires FE and MC modules to provide both phone and server implementations. Both are installed on their respective platforms before runtime.

Query Optimizer The Query Optimizer performs two optimizations. First, it performs *multi-classifier scheduling* to optimally schedule multiple classifiers from the same query. To accomplish this offline, it first calls Adaptation Optimizer for each pipeline independently. All of these optimal configurations are stored on the device. During runtime, the Runtime selects a configuration for each pipeline such that (1) the sum processor utilization is equal to the actual processor availability, and (2) the application constraints for each pipeline are satisfied.

Commonly, applications are only interested in logical conjunctions of specific ADUs. For example, in *Cocktail Party*, only faces of coworkers AND conversation sounds are of interest. Query Optimizer short circuits evaluation of latter pipelines if former pipelines do not pass interest conditions. This results in significant cost savings by skipping entire pipeline executions. While short-circuiting logical conjunction evaluation is well-understood, Kobe can confidently calculate both criteria for good ordering: the labeled training data tells us the likelihood of passing filter conditions, and the Cost Modeler tells us the pipeline cost. In practice, this makes our approach robust to estimation errors. This scheduling optimization is in the spirit of earlier work on sensor sampling order [21]. However, it differs in that we deal explicitly with filtering ADUs, not raw sensor data which is more difficult to interpret.

Second, the Query Optimizer performs *binary classifier substitution* for *N-way classifiers* when appropriate. N-way classifiers (all of our examples thus far) label each sample as a specific class out of N labels, whereas a binary classifier simply decides whether a sample belongs to a single class (*e.g.,* "in conversation" or "not in conversation" for a Sound Classifier). By inspecting the query, Query Optimizer identifies opportunities where N-way classifiers may be replaced with a binary classifiers. An example is *NoisePipe* in *Offict Fit*, which involves an equality test with a fixed class label. Upon identifying this opportunity, Query Optimizer calls Adaptation Optimizer *n* times to builds *n* classifiers each of which is binary, one for each class. Specifically, for one class with label *i*, Kobe trains a binary classifier for it by merging all training data outside this class and labeling them as $\bar{i}$. During online classification, when binary classification for class *i* is encountered, the Runtime substitutes in the binary classifier for class *i* to perform the inference. Besides equality tests, the binary classifier substitution also applies to class change detection over continuous streams; when an

N-way classifier detects i at time t, a binary classifier may be used to detect changes to i at subsequent time steps. If a change to i is detected, then the full N-way classifier is invoked to determine the new class. The benefit of the binary classifier is that it can be more cost-effective than the N-way classifier, as we show in §8.

Personalization The Personalizer adapts the classifier(s) to a user's usage patterns. It does this by calling Query Optimizer with training data reduced to only that of the end user for whom the application is being deployed. Training data is often sourced from many end users, and the resulting classifier may be poorly suited for classifying particular individual end users. The advantage of personalization is that models trained on an end user's data should intuitively perform well when run against test data for the same individual. The disadvantage is a practical one: any one individual may not have sufficient training data, and hence constructed pipelines may not accurately classify long-tail classes. Therefore, Kobe runs the Personalizer as an offline reoptimization after an initial deployment period collects enough personal data. Initially, the regular classifier is used.

7 Implementation

The Optimizer is implemented in C# on a cluster of 26 machines. One machine is designated the master, and the others are slaves. A user invokes the master with a new *request* with the elements discussed in §5. The master first copies all the training data and required classifier modules to each slave. Next, the classifier's parameter space is subdivided among the slave machines. Each slave is responsible for estimating the accuracy and cost of all configurations in its parameter subspace across all environments, and returning these estimates to the master. The master sorts all configurations by accuracy and cost to determine the Pareto optimal set per environment. The Query Optimizer invokes the master-slave infrastucture multiple times for the purpose of profiling binary classifiers, and for each classifier that is part of a multi-classifier query. The Personalizer is simply a request with user-specific training data. The master outputs configuration files that are used by the Runtime. These files follow a JSON-like format. Since our cluster is small, any node failures are restarted manually. Slaves can be restarted without impacting correctness since each configuration profile is independent of others. Master failure causes us to restart the entire request.

The Runtime spans the phone client and the cloud server. The modules are implemented in C# and C++ on the phone's Windows Mobile .NET Compact Framework and on the server's .NET Framework. Cloud offload invocations use .NET Web Services for RPC. Modules can either be stateless or stateful. Stateless modules generate output based only on the current input. Therefore, they can switch between phone and cloud execution without any further initialization. Most FE and MC modules are stateless. Stateful modules generate output based on a history of past inputs. Two stateful modules are HMM[7] which smooths a window of the most recent feature vectors to produce an ADU, and the Transportation Mode FE which batches up many samples before emitting a features vector. Stateful modules expose their state to the Runtime so that the Runtime may initialize the state at the phone or server whenever the place of execution changes. To obtain state storage, modules make the upcall *GET_BUFFER(BUFFERSZ)*. The input *BUFFERSZ*, the maximum possible history of inputs that the module should ever need. It returns a buffer which can be populated by the module, but is otherwise managed by the Runtime.

Cloud offload does introduce the possibility of server failure independent of phone failure. Server fail-stop faults are handled by reverting to phone-only execution after timeout. Since most FEs and MCs are stateless, restarting execution is straightforward. For those that are stateful, the fault does not impact correctness, and only degrades accuracy temporarily *e.g.,* while an HMM repopulates its window's worth of data.

To monitor environmental change at runtime, we adopt standard tools `ping` and [1] for measuring latency and bandwidth respectively. Active probing is only necessary when phone-only execution is underway. Otherwise, probing metadata is piggybacked on to phone-server communication, as in [11]. Currently, active probing is initiated at coarse time scales – upon change of cellular base station ID. Idle processor time is periodically checked for processor utilization.

Modules Implemented Kobe's SS modules are very simple: all SS modules expose parameters for sampling rate, and some, such as image and audio, also expose sample size or bit rate. Kobe currently implements all of the FE and MC modules listed in Table 1. Module implementations were ported from Matlab and other publicly available libraries. Our pipeline porting times – typically one week – suggest that it is not difficult to convert existing code into Kobe modules. There was an outlier time of less than 1 day when we modified a preexisting pipeline. The types of parameters that FE and MC modules expose to Kobe are varied and overwhelmingly module-specific. Examples include an FFT's number of sample points, Gaussian Mixture Model's number of components, and a HMM's window size. We refer the interested reader to the citations in Table 1 for discussion of the semantics of these parameters. Some parameters are general. For example, all FE modules expose a parameter that controls the feature vector size through a generic vector size reduction algorithm, Principle Component Analysis [9]. Parameters that are set-valued and enumerations may not yield natural mappings to continuous scalar ranges as required by *SET_COMPLEXITY()*. To address this, modules use the statistical tool rMBR which performs feature selection to map sets and enumerations to continuous ranges [26].

8 Evaluation

In this section, we evaluate Kobe with respect to each of the three challenges it addresses, and we find that: (1) Kobe successfully balances accuracy, energy and latency demands. (2) Kobe adapts to tightly utilize available resources, for practically equivalent accuracy nonadaptive approaches suffer between 66% and 176% longer latencies and use between 31% and 330% more energy. (3) Kobe's interface enables query optimizations that can save between 16% and 76% of latency and energy. (4) Kobe optimization is scal-

[7]Hidden Markov Model

Scenario	Pipeline Name	Classifier Pipeline Modules *SS*	*FE*	*MC*	Port Time
Transportation Mode (TM)	`TransPipe`	GPS	TransModeFeatures [33]	Hidden Markov Model [9]	5 days
Image Recognition (IR)	`ImgPipe`	Image	SURF [8]	K-Nearest Neighbor [9]	7 days
Sound Classification (SC)	`SoundPipe`	Sound	MFCC [12]	Gausian Mixture Model [9]	8 days
Acceleration Classification (AC)	`AccelPipe`	Accel	MotionFeatures [28]	Decision Trees, HMM [28]	7 days
Face Recognition (FR)	`FacePipe`	Image	FaceFeatures [10]	K-Nearest Neighbor [9]	<1 day

Table 1. Classifiers Ported to Kobe

able and introduces minimal runtime overhead.

Methodology

We evaluate Kobe based primarily on the four scenarios of §3: Transportation Mode Inference (TM), Image Recognition (IR), Sound Classification (SC) and Acceleration Classification (AC). We also evaluated a Face Recognition (FR) scenario.

Data Sets. Pipeline performance is highly dependent on the input sensor data sets used. For this reason we evaluate Kobe using established, large and real-world data sets when possible. The TM data set is provided by the authors of [33]. It consists of 200 GPS traces of people in Beijing labeled with the transportation mode they used: {*subway, drive, walk, bike*}. The IR data set is of 5,000 building images taken at Oxford University [27]. Each image is labeled with the building visible in the image. The SC data set comes from a previous sound scene classification study [20] and consists of 200 sound clips of 10 different everyday classes of sounds: {*driving, vacuuming, clapping, classical music, showering, a fan, street noise, conversation, crowd noise, pop music*}. The FR data set is provided by the authors of [10] and consists of 3700 face images of 130 individuals collected from photo albums. The AC data is collected over a period of two weeks from a developer using Kobe to build his application for the first time.

Classifiers. For each scenario, we port established classifiers developed by SML experts to the Kobe system. Refer to Table 1 and §7 for discussion of the modules.

Measurements. Energy and latency measurements are made with a Monsoon Solutions Power Monitor FTA22D. We employ two Windows Mobile phones, the HTC Touch Pro and HTC Wizard. The HTC Touch Pro has 200 MB of memory, a 528 MHz processor, and GPRS and 3G radios. The HTC Wizard, a less powerful device, has 50 MB of memory, a 200 MHz processor, and a GPRS radio. Both have WiFi 802.11b/802.11g. Accuracy measurements are performed with five-fold cross validation to ensure high confidence.

Baseline Comparisons. We compare Kobe against five baselines. Four are manually tuned static solutions. `C-Aggr` and `C-Safe` are cloud-only configurations, and `P-Aggr` and `P-Safe` are phone-only configurations. `C-Aggr` and `P-Aggr` are set aggressively to the max possible accuracy, and `C-Safe` and `P-Safe` are set to a safe conservative latency. The fifth baseline, `P-Samp`, is representative of the class of solutions that only tunes sensor sample rate. These baselines all correspond to different heuristics commonly used by system designers when manually tuning a system prior to deployment. When comparing these baselines to Kobe, we define *underutilization* to mean the % additional unrealized accuracy gain when compared to the closest Pareto optimal configuration while still preserving user constraints. We define *overutilization* to mean the % latency or energy exceeding the user-specified constraints.

Balancing Accuracy, Energy & Latency

We first show that Kobe finds good balances of accuracy and cost for the phone-only setting, and that finding such solutions manually is non-trivial. We applied Core Optimizer to the Wizard phone and show results for optimization of latency in the case of IR and SC, and energy in the case of TM and AC.

Fig. 4(a) illustrates IR's Pareto curve for the accuracy-latency trade-off as found by Kobe on the Touch Pro. Sub-optimal configurations encountered by Kobe are also shown. The costs of all configurations are estimated using Core Optimizer's Cost Model, with the addition that Pareto configurations are verified by actual phone measurements. The figure shows that there are many suboptimal points; a randomly chosen configuration suffers an accuracy loss of 4% and latency increase of 263% more than an optimal configuration with the same constraints. The correlation between cost and accuracy is only 0.61, indicating that a simple heuristic of choosing more costly configurations does not guarantee good accuracy.

Fig. 4(b) shows a similar result for SC: the Pareto curve is found, with many suboptimal points. A randomly chosen configuration loses 14% accuracy and suffers 176% additional latency versus the closest optimal configuration.

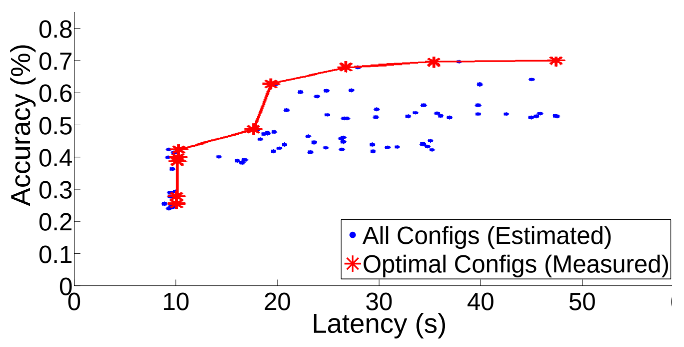
Fig. 4(c) illustrates the accuracy vs. energy Pareto optimal configurations for TM. It is compared to `P-Samp`. Kobe tunes sampling rate along with other classifier parameters, whereas `P-Samp` only tunes sampling rate. We can see that while `P-Samp` does manage to scale energy use by adjusting sampling rate, it is unable to reach accuracy gains as good Kobe because other pipeline parameters are not tuned. Consequently, `P-Samp` suffers a 3.6% accuracy underutilization penalty on average.

Fig. 4(d) shows the Pareto optimal configurations for AC. As with the previous pipelines, a range of accuracies (86% - 98%) and energy usage (0.9-4.3mJ) tradeoffs are possible.

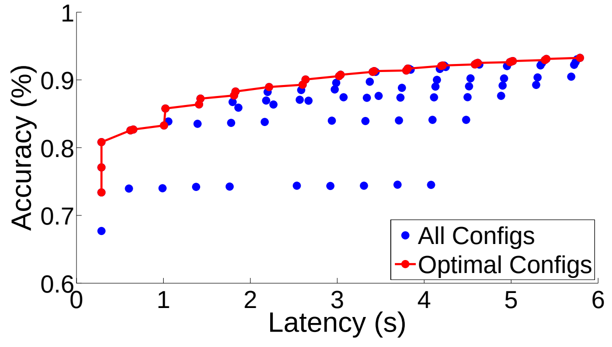
Adapting to Changes

We next evaluate Kobe's and existing solutions' abilities to adapt to device heterogeneity, networking and cloud availability, multi-programming and user usage behavior with Adaptation Optimizer.

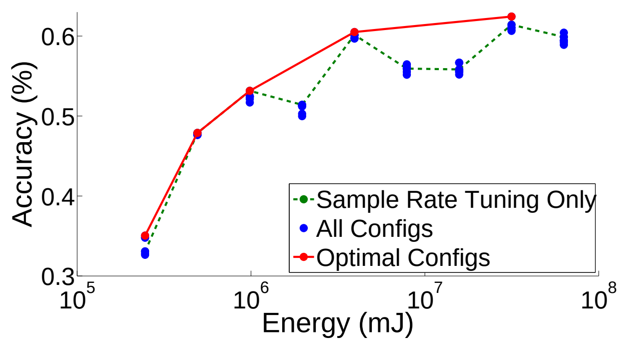
Device Adaptation. In this experiment, we are interested in the performance penalty of not tightly optimizing the pipeline for the device. Using the IR scenario, we first deter-



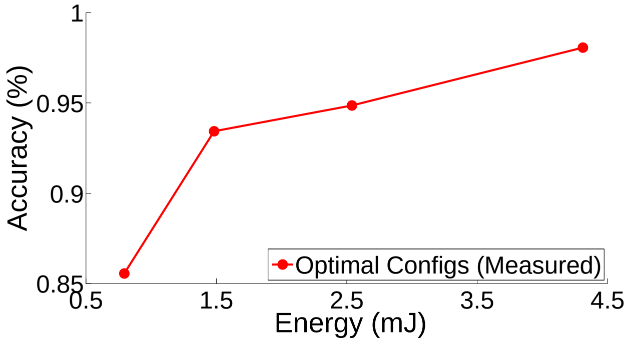
(a) Building Image Recognition



(b) Sound Classification



(c) Transportation Mode Inference



(d) Accelerometer-based Classification

Figure 4. Optimal and suboptimal configurations found for various classification scenarios

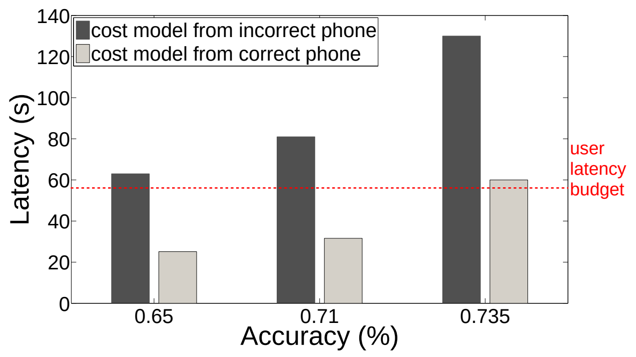


Figure 5. Penalty incurred for not accounting for device differences in Image Recognition

mine Pareto configurations taking the high-end Touch Pro as the a ‘reference’ platform. We then evaluate the performance of these configurations against the less capable low-end Wizard. In Fig. 5, we see that the penalty for using the wrong reference platform is an average latency penalty of 176%.

Network and Cloud Adaption. In this experiment Kobe has the option of placing FE and MC modules on a remote server or local phone. Using the IR scenario, we compare Kobe under a latency constraint to `C-Safe` and `C-Aggr`. These two manually-tuned configurations represent the conservative and optimistic extremes in use. The top pane of Fig. 6 shows the independent variable, network bandwidth, which changes at time t_1 and t_2. We purposely choose a long duration between t_1 and t_2 to make the accuracy trend clear. The middle pane of Fig. 6 shows that despite network changes, Kobe remains slightly below the latency constraint, whereas `C-Aggr` suffers from overutilization, exceeding the latency constraint by 36% on average. The bottom pane of Fig. 6 shows that Kobe appropriately improves its accuracy as the network gets better (t_1) and scales back its accuracy as the network gets worse (t_2), whereas `C-Safe` consistently experiences underutilization, resulting in 7% lower accuracy on average.

We also test the IR scenario’s energy usage when running Kobe vs. cloud-only executions `C-Safe` and `C-Aggr`. Kobe is able to find lower energy phone-only executions whereas `C-Safe` and `C-Aggr` are handicapped with high-energy network access because they must power on/off their network connections at every periodic classification (keeping the network connection on continuously performed significantly worse). On 3G, WiFi and GSM networks, `C-Safe` and `C-Aggr` use between 31-252%, 250-259% and 63-77% more energy than Kobe, depending on the accuracy desired.

Multi-programming Adaptation. We test Kobe’s ability to adapt to multi-programming relative to `P-Safe` and `P-Aggr` for a fixed latency constraint. Due to space, we omit the graphs. As a summary, we find that in the IR scenario, `P-Safe` experiences underutilization leading to accuracy loss of 13%, `P-Aggr` experiences overutilization leading to excess latency by 440%, whereas Kobe successfully varies its accuracy while remaining within the latency constraint.

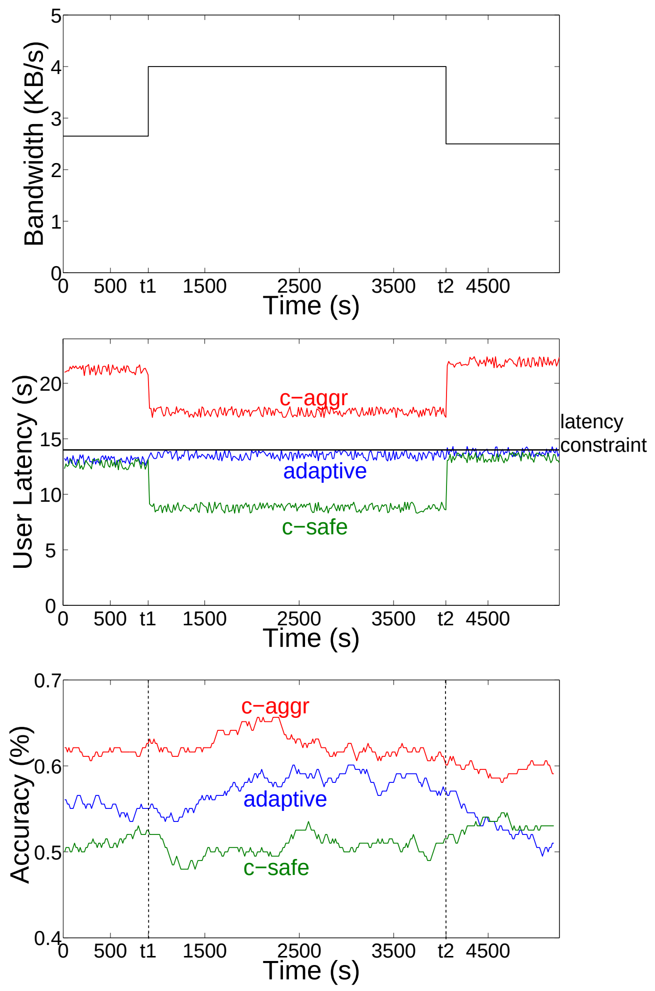
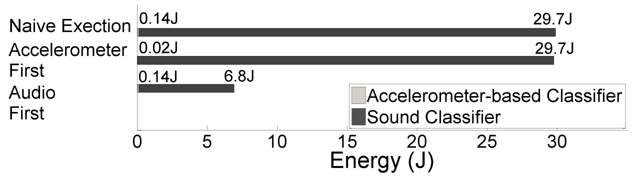
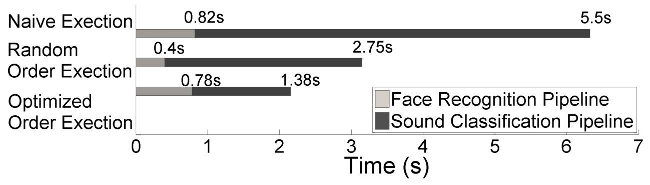


Figure 6. Dynamic network adaptation achieves higher accuracy than static safe configs, and lower latency than static aggressive configs.



(a) *Offict Fit* Classifier Scheduling.



(b) *Cocktail Party* Classifier Scheduling

Figure 7. Multi-classifier scheduling with logical conjunction shortcutting optimization

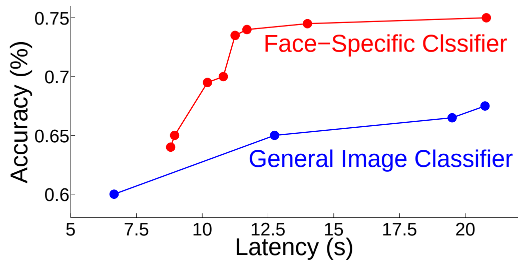


Figure 8. Switching pipelines w/o app modification

User Usage Behavior Adaptation. We test the Personalizer on the TM scenario, the only dataset which was tagged with user id. Personalizer builds better models for 36% of the users – users whose traces were too short did not benefit as much. Of those that benefited, the average classification accuracy rises from 0.598 to 0.726 – a 21% improvement.

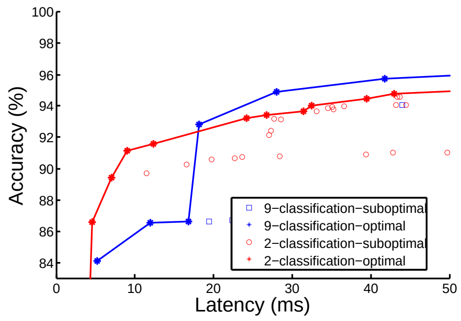
Programming Interface Benefits

We evaluated several optimizations enabled by the Kobe SQL-like interface.

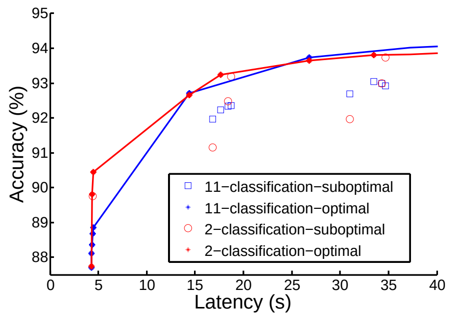
Switching Pipelines. We evaluate the developer's (1) effort and (2) app performance improvement when switching from one pipeline to another, and use the scenario of Face Recognition, a subclass of Image Recognition. For this experiment, we use the face data set. We assume the developer had already built their application with `ImgPipe` but wants to switch to the purportedly superior `FacePipe` which implements a face-specific FE module [10]. Our findings are that the effort to switch is very minimal: `FacePipe` took less than 1 day to port from the original implementation in [10] and required a change only to the *PROTOTYPE_PIPELINE* of the classifier declaration. Moreover, Fig. 8 shows that given a latency budget, accuracy improved in accuracy by 0.27 (64%). It is also interesting to note that `FacePipe` did not completely dominate `ImgPipe`; if the app developer desires latency below 7.5s, it is actually preferable to use `ImgPipe` since `FacePipe` cannot meet such latencies.

Scheduling Multiple Classifiers We use Kobe to implement two apps that each uses multiple classifiers. These are *Offict Fit* and *Cocktail Party*, discussed in §5. Logical conjunction shortcircuiting for *Offict Fit* saves 76% in energy simply because AC is such an inexpensive pipeline relative to SC (Fig. 7(a)). Optimized scheduling for *Cocktail Party* saves 66% latency by selecting FR ahead of SC. This result may at first seem counterintuitive when considering the larger latencies of IR in Fig. 4(a) over SC in Fig. 4(b). However, the probability of detecting a conversation is much higher than that of detecting a coworker's face. Therefore, despite FR's longer latency, it is the better pipeline to run first. Random scheduling for *Cocktail Party* only saves 31% (Fig. 7(b)).

Binary Classifier Substitution We test the binary classifier substitution optimization on SC and IR. N-way SC consists of 9 classes, so Kobe built 9 binary classifiers. A sequence of binary classification queries is generated, with the prob-



(a) Sound



(b) Image

Figure 9. Binary classifiers can substitute for N-way classifiers for accuracy and latency benefits.

ability of each binary classification query matching that of the prior probability of the class label in the training data. Fig. 9(a) shows the accuracy vs. latency tradeoff for N-way and Binary Classifiers respectively. The N-way Classifier cannot achieve latencies lower than 19ms without sacrificing substantial accuracy, whereas Binary classifiers offer more gradual accuracy degradation. Given an accuracy bound lower than 92%, Binary Classifiers can shorten the latency up to 3ms – an 16% improvement. It should be noted that to achieve higher accuracy, N-way Classifiers are preferrable. We performed similar experiments on IR, with the `ImgPipe`. We used a subset of the dataset, choosing 11 classes, which led to the generation of 11 Binary Classifiers. Figure 9(b) shows that in most cases, the Binary Classifier outperforms or is comparable to the N-way Classifier. Given a latency bound below 10s, Binary Classifiers can improve the accuracy by up to 1.8%.

Scalability and Overhead

The cost of Kobe is primarily offline Optimizer computation time, and online Runtime overhead. We find neither offers significant cause for concern.

Optimizer Scalability The offline Optimizer running time is dependent upon the number of configurations profiled. Approximately 13,000 configurations are evaluated per each invocation of the Adaptation Optimizer. This is a product several factors whose typical values follow in parentheses. These are the parameters per module (3), settings per parameter (20), number of modules (3), network discretizations (3), module placement options (4), processor discretization (3), and phones supported (2). Scalability with respect to the number of phones can be made more manageable if it is assumed that similar phone chipsets exhibit similar performance, which is often the case. The Query Optimizer invokes Adaptation Optimizer once per class for the purpose of binary classifier substitution. The entire Optimizer only needs to be executed once per application, or once per user if personalization is desired. Typical running times for the Optimizer on a 10 machine cluster are shown in Table 2.

Runtime Overhead The Runtime incurs four potential overheads. First, the Runtime stores each of the optimal configuration files generated by Optimizer. From the number of profiled configurations, Kobe selects only those that are Pareto optimal, meet the developer constraints, and are for the specific user's phone. The number of stored configurations per classifier is a product of the number of network discretizations (typically 3) and processor discretization (typically 3). Each configuration file is small; the largest is only 2KB. Therefore, configuration file storage is not a significant overhead. Second, the Runtime must store the FE and MC module binaries on the phone, which can be avoided by cloud-only solutions. The max and min pipelines we encountered were respectively 12MB for IR and 5KB for TM. The discrepancy stems from the fact that IR's MC stage, KNN, scales linearly with the training data size, whereas TM's does not. Third, Runtime incurs an overhead for monitoring the network and estimating bandwidth and latency changes when currently executing phone-only classifier configurations. To estimate the significance of this overhead, we randomly sampled traces from our TM dataset, and estimated the frequency with which users switched cellular base stations. Fig. 10 shows that this overhead is under 80 milliseconds even when user is switching regions rapidly. Anecdotally, our estimation heuristic was reasonably accu-

Pipeline Name	Time
`TransPipe`	2 hours
`ImgPipe`	15 - 20 mins
`SoundPipe`	30 mins - 3 hours†

Table 2. Optimization running time
†Longer times reflect Binary Classification Substitution Optimization running time.

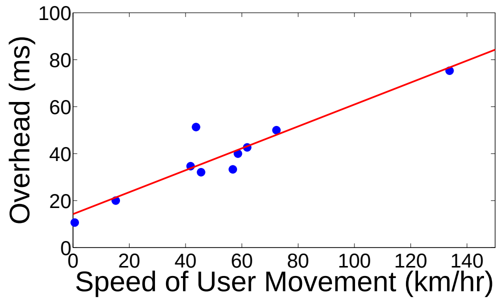


Figure 10. Bandwidth estimation overhead for phone-only classifier executions. Bandwidth estimation is piggy-backed on to existing traffic for phone-server executions.

rate. Lastly, Runtime incurs a CPU cost when switching between configurations. Since it is simply setting configuration parameters and possibly passing a buffer, the cost was negligible and did not noticeably affect performance (see Fig. 6).

9 Related Work

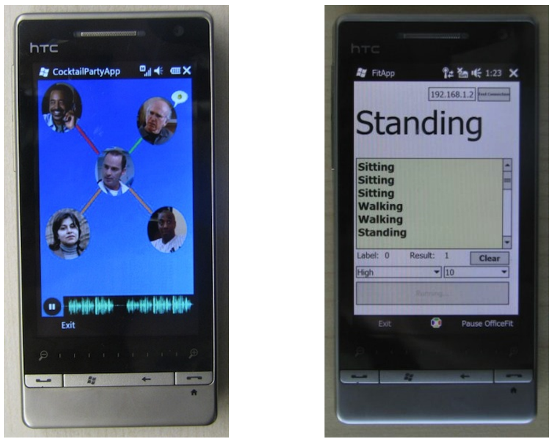
The early work of [25] describes programming interfaces for application-aware adaptation where developers specify functions to execute for each given resource range. [13] pioneered systematic mobile energy adaptation by jointly turning down hardware components and degrading application fidelity. [11] dynamically decided remote vs. local execution based on online measurements of application resource usage. [24] uses prediction models to assess resource availability and adjust app fidelity correspondingly. From this line of work, Kobe is most similar to that of Chroma [5], which proposes interfaces to expose server offload opportunities, as well as variable fidelity execution options.

Unlike Kobe, these pioneering systems attempt to address adaptivity broadly for all mobile applications. Kobe takes a strict focus to mobile SML classification, an emerging and important domain. This has three benefits. First, we leverage properties of this domain (such as offline labeled training data availability and linear pipelines) to aggressively precompute all of the optimal configurations ahead of time, thereby incurring very little runtime overhead while simultaneously offering extremely simple programming interfaces and a highly adaptive runtime. In contrast, a system like Chroma uses runtime trial-and-error to measure performance because it can make few assumptions about general mobile programs. This leads to significant performance overhead and scalability limitations. In fact, the authors of Chroma assume that the user enumerates only a handful of configurations (<10) for exploration. Second, Kobe decouples app developers from SML experts, whereas general systems have no notion of distinct domains of expertise. Lastly, Kobe builds in classification-specific optimizations, such as Binary Classifier Substitution, Multi-Classifier Scheduling, and the Personalizer for additional gains, which general systems do not attempt to do.

Recognizing the need for mobile classifier optimization, some promising work has recently begun. Many proposals have surfaced for abstracting and dynamically adapting sampling rate [16, 14, 32]. Optimizations that rely on sampling rate adaptation have commonalities in their techniques with work investigating energy efficient mobile localization (e.g., [15]). As we've illustrated in this work, sampling rate tuning alone is myopic and only addresses energy, not latency concerns. Several works have looked at either application-specific [20], application-scenario specific classifier optimization [31]. While Kobe does not offer application- nor scenario-specific tuning, Kobe otherwise subsumes this earlier work and is applicable to all classification problems.

10 Experiences, Discussion and Conclusion

Thus far we have prototyped two applications with Kobe: *Cocktail Party* and *Offict Fit*. Their screenshots are shown in Fig. 11. Our in-house experiences with Kobe suggest that the SQL-like interface is good for quickly assembling classifiers and embedding them into the overall application.



(a) *Cocktail Party* (b) *Offict Fit*

Figure 11. Application Screenshots

Kobe's ability to tune classifiers to different latency and energy budgets proved quite useful. Low classification latencies allowed *Cocktail Party* to provide timely reminders relevant to the current conversation partner *e.g.,* detecting "conversation" and "Alice" triggers a personal note reminder such as "Talk to Alice about the homework assignment." Low energy usage allowed *Offict Fit* to run for long periods during the day, an important criteria for a background fitness app. On the whole, we are sanguine about Kobe's ability to build classifiers for mobile apps. However, Kobe has several limitations discussed below.

Kobe currently only supports three stage linear classifiers. There are domain-specific classifiers with more sophisticated processing stages that have evolved over time, such as in hierarchical models used in speech recognition and speech-to-text processing. The focus for Kobe is to complement these established techniques. SML expert hand-tuning will continue to be the norm for widely-used classifiers. However, app-specific classifiers with highly specialized ADUs may not receive the same expert treatment, and prebuilt classifiers for specific ADUs may not be available. We suggest that Kobe is an appropriate solution for non-experts that wish to classify such ADUs.

Kobe supports only one sensor input at the SS stage. As demonstrated by the multi-pipeline examples, Kobe accomplishes sensor fusion through execution of disjoint pipelines. Alternatively, a classifier that directly ingests multiple sensors may provide better accuracy (at higher sampling cost). Supporting these classifiers is part of future work.

Some apps, such as archival personal sensing [22], may not need ADUs until long after the signal collection time. *Delay-tolerant classifiers* ought to defer processing until resources are abundant, such as when the phone is docked for charging. Currently, Kobe does not support delayed classification, but it would be natural to consider extensions to support delayed classification because latency constraints are already part of the Kobe interface.

Kobe offers a basic but effective query optimizer. Further optimizations should be possible as the types of queries posed expands. As more applications subscribe to continuous ADUs, there may be opportunities for sharing the computation subprocedures among different classifiers. For example, two SCs could share their FE stages if they both are MFCC FEs. We intend to use Kobe as a starting point for further investigation.

Many compelling applications are now emerging as mobile devices integrate an increasing array of sensors. However, employing raw sensor data directly is only the beginning – meaningful application data units will drive interesting applications. We developed Kobe as an automated approach to constructing classification pipelines. Kobe lets developers supply what they already know: which sensors to use, and what labeled training data to target. It lets SML experts easily contribute their new algorithms. In exchange, Kobe builds adaptive classifiers that balance accuracy, latency and energy.

11 References

[1] Available bandwidth measurement in ip networks. http://www.ietf.org/proceedings/56/I-D/draft-anjali-ippm-avail-band-measurement-00.txt.

[2] Layar. `http://layar.com/`.

[3] Nike+. http://nikerunning.nike.com/nikeos/p/nikeplus/en_US/.

[4] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: Semantic foundations and query execution. Technical Report 2003-67, Stanford InfoLab, 2003.

[5] R. K. Balan, M. Satyanarayanan, S. Park, and T. Okoshi. Tactics-based remote execution for mobile computing. In *MobiSys*, 2003.

[6] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy consumption in mobile phones: a measurement study and implications for network applications. In *IMC '09*.

[7] L. Bao and S. S. Intille. Activity recognition from user-annotated acceleration data. In *Pervasive '04*.

[8] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool. Speeded-up robust features (surf). *Comput. Vis. Image Underst.*, 2008.

[9] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer-Verlag New York, Inc., 2006.

[10] Z. Cao, Q. Yin, X. Tang, and J. Sun. Face recognition with learning-based descriptor. In *CVPR '10*.

[11] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: Making smartphones last longer with code offload. In *MobiSys '10*.

[12] S. B. Davis and P. Mermelstein. Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. *IEEE Transactions on Acoutstics, Speech, and Signal processing (1980)*, 28(4), 1990.

[13] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. *SIGOPS Oper. Syst. Rev.*, 2000.

[14] H. Junker, P. Lukowicz, and G. Trster. Sampling frequency, signal resolution and the accuracy of wearable context recognition systems. *Wearable Computers*, 2004.

[15] M. B. Kaergaard, J. Langdal, T. Godsk, and T. Toftkaer. Entracked: energy-efficient robust position tracking for mobile devices. In *MobiSys '09*.

[16] A. Krause, M. Ihmig, E. Rankin, D. Leong, S. Gupta, D. Siewiorek, A. Smailagic, M. Deisher, and U. Sengupta. Trading off prediction accuracy and power consumption for context-aware wearable computing. In *ISWC '05*.

[17] N. D. Lane, H. Lu, S. B. Eisenman, and A. T. Campbell. Cooperative techniques supporting sensor-based people-centric inferencing. In *Pervasive '08*.

[18] N. D. Lane, E. Miluzzo, H. Lu, D. Peebles, T. Choudhury, and A. T. Campbell. A survey of mobile phone sensing. *Comm. Mag.*, 48(9):140–150, 2010.

[19] J. Liu, Z. Wang, L. Zhong, J. Wickramasuriya, and V. Vasudevan. uwave: Accelerometer-based personalized gesture recognition and its applications. In *PERCOM '09*.

[20] 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*.

[21] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *SIGMOD '03*.

[22] 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*.

[23] M. Mun, S. Reddy, K. Shilton, N. Yau, J. Burke, D. Estrin, M. H. Hansen, E. Howard, R. West, and P. Boda. Peir, the personal environmental impact report, as a platform for participatory sensing systems research. In *MobiSys '09*.

[24] D. Narayanan and M. Satyanarayanan. Predictive resource management for wearable computing. In *MobiSys '03*.

[25] B. Noble, M. Satyanarayanan, and M. Price. A programming interface for application-aware adaptation in mobile computing. In *MLICS '95*.

[26] H. Peng, F. Long, and C. Ding. Feature selection based on mutual information: criteria of max-dependency, max-relevance, and min-redundancy. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2005.

[27] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman. Object retrieval with large vocabularies and fast spatial matching. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2007.

[28] N. Ravi, N. D, P. Mysore, and M. L. Littman. Activity recognition from accelerometer data. In *IAAI '05*.

[29] S. Reddy, J. Burke, D. Estrin, M. Hansen, and M. Srivastava. Determining transportation mode on mobile phones. *Wearable Computers, IEEE International Symposium*, 2008.

[30] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, 2009.

[31] S. Srinivasan, Z. Fang, R. Iyer, S. Zhang, M. Espig, D. Newell, D. Cermak, Y. Wu, I. Kozintsev, and H. Haussecker. Performance characterization and optimization of mobile augmented reality on handheld platforms. 2009.

[32] Y. Wang, J. Lin, M. Annavaram, Q. A. Jacobson, J. Hong, B. Krishnamachari, and N. Sadeh. A framework of energy efficient mobile sensing for automatic user state recognition. In *MobiSys '09*.

[33] Y. Zheng, Y. Chen, Q. Li, X. Xie, and W.-Y. Ma. Understanding transportation mode based on gps data for web applications. *TWEB*, 2009.