[preprint] Empirical Software Engineering manuscript No. EMSE-D-17-00197R2

What can Android mobile app developers do about the energy consumption of machine learning?

Andrea McIntosh · Safwat Hassan · Abram Hindle

Accepted on: 2018-05-10

Abstract Machine learning is a popular method of learning functions from data to represent and to classify sensor inputs, multimedia, emails, and calendar events. Smartphone applications have been integrating more and more intelligence in the form of machine learning. Machine learning functionality now appears on most smartphones as voice recognition, spell checking, word disambiguation, face recognition, translation, spatial reasoning, and even natural language summarization. Excited app developers who want to use machine learning on mobile devices face one serious constraint that they did not face on desktop computers or cloud virtual machines: the end-user's mobile device has limited battery life, thus computationally intensive tasks can harm end users' phone availability by draining batteries of their stored energy. Currently, there are few guidelines for developers who want to employ machine learning on mobile devices yet are concerned about software energy consumption of their applications. In this paper, we combine empirical measurements of different machine learning algorithm implementations with complexity theory to provide concrete and theoretically grounded recommendations to developers who want to employ machine learning on smartphones. We conclude that some

Andrea McIntosh
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
E-mail: akmcinto@ualberta.ca

Safwat Hassan
Software Analysis and Intelligence Lab (SAIL)
Queen's University
Kingston, Ontario, Canada
E-mail: shassan@cs.queensu.ca

Abram Hindle
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
E-mail: hindle1@ualberta.ca

implementations of algorithms, such as J48, MLP, and SMO, do generally perform better than others in terms of energy consumption and accuracy, and that energy consumption is well-correlated to algorithmic complexity. However, to achieve optimal results a developer must consider their specific application as many factors — dataset size, number of data attributes, whether the model will require updating, etc. — affect which machine learning algorithm and implementation will provide the best results.

1 Introduction

Imagine we are in a hot new start-up and your app, which will be deployed to millions of phones, needs to take advantage of machine learning. Which machine learning algorithms should we employ to avoid sapping the energy of your customers' phones? Should we use neural networks since they are so popular, or should we stick to simpler models to save energy? In this work we address the questions of "how energy efficient are these machine learning algorithms?" and "which algorithms should we use on a mobile device?" Others have asked these questions before [53].

Machine learning is growing in popularity. Google, in particular, has made the results of machine learning available to the general public in terms of speech recognition [72], translation [24], computer vision, and search. Many machine learning implementations have been deployed to servers in the cloud or data centers. But the popularity of mobile devices such as smartphones and tablets are causing a push toward mobile-apps that employ machine learning. One of the issues that mobile platforms face that servers and desktop computers do not, is that mobile platforms tend to rely on batteries for power and when the batteries are out of energy the mobile device is no longer available for use. This is different from data-centers that have machines on racks that face power limits and need constant cooling. Machine learning on mobile platforms is often outsourced to the cloud, but the bandwidth to the cloud is quite limited, so a lot of machine learning is pushed back to the mobile device itself. Additionally, apps are expected to work offline, as network availability cannot be guaranteed. Some apps engage in computer vision, others learn from the textual and event-based data on the phone to schedule appointments [25], and others link and organize documents [20].

If machine learning is pushed to mobile devices what should practitioners do about the software energy consumption of machine learning on their mobile devices? Surveys of developers and users have found that poor software energy consumption performance can lead to negative app-store reviews and poor user satisfaction [33, 58, 79]. In this work, we will empirically test, measure, and detail the costs and trade-offs between machine learning performance of several implementations and software energy consumption on Android smartphones. We will show that there is no best algorithm implementation but there are a wide range of trade-offs that one can make depending on the context that one is operating within. Furthermore, not all energy consumption is CPU bound as some implementations of algorithms cost more in terms of memory-use than others that in a memory constrained environment can induce more energy consumption.

All of this is tempered by the rapid pace of development in mobile devices, machine learning software, and specialized hardware development. This paper focuses on classic machine learning algorithms and implementations, executed in the context of the 1 version Android OS and on one brand and model of an Android phone. While we focus on classic machine learning algorithms and implementations, neural networks, deep learning, and stochastic gradient descent optimization have taken over machine learning [13, 24, 72]. New libraries and services are constantly being released [27, 36, 37, 72, 76]. Thus the pace of new libraries, their adoption, and their distribution is rapidly increasing. This is combined with the fact that newer versions of Android and iOS even come with machine learning libraries out of the box [27, 36]. This paper evaluates machine learning implementations on Android 4.2.2 but new versions of the Android OS have switched from the DalvikVM to ART which can change the performance profile of user-space code. Furthermore, the hardware of the Galaxy Nexus phones is different than the hardware used in newer modern devices, where changes in architectures, CPUs, buses, and memory, can all have an effect on performance. Furthermore, there is a push to put more machine learning related functionality on a chip to increase machine learning performance, such as Apple's A11 Bionic Chip with Neural Engine [10]. Thus our measurements should be viewed as a snapshot and an initial recommendation given a certain configuration as the state of machine learning rapidly changes. It will still be up to a developer to profile and benchmark their machine learning choices on their own, given their own context. All of this should be taken into account, but fundamentally developers should understand that mobile machine learning is becoming more and more important but it does not come for free in terms of energy consumption as different algorithms can behave differently.

The contributions of this paper are:

– an empirical evaluation of the trade-offs that machine learning algorithms and their implementations make between accuracy and software energy consumption;
– concrete recommendations for choosing machine learning algorithm implementations for use on mobile platforms;
– an investigation of machine learning imports in the Google Play App Store;
– empirical testing and measurement of multiple machine learning contexts that demonstrate "one size does not fit all".

1.1 Motivation: machine learning on mobile platforms

Why should we bother with machine learning on mobile devices? Are not machine learning algorithms expensive and unfit for resource limited devices? We believe in many cases the appropriate algorithms with the proper configuration and data can fit well on mobile platforms. There are many examples of frameworks and applications of machine learning for even the Android ecosystem.

Multiple frameworks exist that enable machine learning within mobile applications. As Android uses Java, any Java-based machine learning framework can easily be integrated into an Android app. For our tests, we used the Weka [31] and Neuroph [69] frameworks. Google Brain team's TensorFlow machine learning library [72] is also intended to be portable and work on mobile and embedded devices.

As a demo for an Android application, TensorFlow provides example code for an app that can classify what is being viewed in the phone's camera frame in real time. Similarly, the Google Translate mobile application can translate words being viewed through a phone's camera offline and in real-time using a trained convolutional neural net [24].

There are numerous cases of machine learning being used in apps. "Smart calendar" apps use machine learning to enhance calendar applications. Google Calendar Goals automatically schedules time for user-set personal goals, such as exercising three times a week, re-schedules these goals if a conflicting appointment is added, and learns the best times to schedule goals based on when the user completes or defers a goal [25]. The Tempo app could pull and bundle data related to calendar events from the user's accounts — such as participant contact information, directions to the location, associated documents — and present them together in one organized entry [20].

Triposo is an app that provides travel recommendations and booking options to users. It uses machine learning to process websites and reviews, and combines the results with user preferences to make personalized recommendations [74]. Weotta is an app that uses machine learning and natural language processing to provide event and activity recommendations to user queries [77].

1.1.1 Existing APIs and libraries

We employ Weka [31] and Neuroph [69] in this paper as they can be freely included into any Android project that needs machine learning training and evaluation. But there are currently numerous libraries and frameworks one could use. Some come with the mobile operating systems, some are provided by mobile vendors in SDKs, while others are remote machine learning services that one can access over the network.

For neural networks and deep learning, Google has released Mobile TensorFlow [72]. TensorFlow is a deep learning framework used on the desktop while Mobile TensorFlow relies on the Java native interface as it can generate platform specific and independent binaries for deep learning. Google recently (as of writing) released a wrapper for Android for neural network libraries like TensorFlow called Neural Networks API [26].

Apple has released CoreML [36] for iOS. It has support for decision trees, neural networks, matrix routines, GPU-accelerated operations, computer vision routines, and AI path-finding routines. It also comes with utilities that allow the conversion of 3rd party models into CoreML friendly models, enabling training on the desktop and deploying on mobile. CoreML enables both training and evaluation locally on the mobile device.

Google Android comes with computer vision classifiers and detectors for face detection, bar-code recognition, OCR/text detection out of the box [27].

OpenCV [56] is an iOS and Android compatible computer vision library that provides numerous image manipulation routines and classifiers. One common classifier is the face detector that employs Haar Wavelets to search for faces in an image in a hierarchical manner. OpenCV also comes with object detectors as well.

Some mobile machine learning libraries or APIs are web services meant to be remotely called from the device. The benefit is that the training and evaluation are not local and the energy cost is not local. The disadvantage is bandwidth, latency, and network traffic necessary to engage in a webservice—all of which can induce energy consumption. Whether or not it is more energy efficient to "outsource" the computation to a service in the cloud is not clear, as it is context dependent, and would require empirical evaluation.

Wit.ai [37] is a machine learning web service with SDKs for iOS and Android. Typically Wit.ai requests are sent to their web service and the evaluation via an HTTP request and the result is returned via the corresponding HTTP response. The SDKs provided wrap the HTTP requests for the native language used on the mobile platform. Thus the energy cost of machine learning is not born by the mobile device, but the network traffic of the request is.

Amazon AWS Machine Learning [76] provides SDKs for iOS and Android to enable calling Amazon AWS ML from a mobile device remotely. The intent is to train and evaluate completely in the cloud using Amazon AWS. This delegates any energy consumption to the cloud and to the network peripheral used by the mobile device.

Thus, machine learning is picking up in popularity on and off mobile devices in the service of machine learning through a mobile device.

1.1.2 Empirical evidence of the use of machine learning in the Google Play app Store

For a more thorough investigation on the use of machine learning in the Google Play app store, we studied how machine learning libraries are used in the most popular apps during 2016. We selected popular apps (using App Annie's report [8]) as these apps are used by a large number of users and we expect that developers of these apps may use the latest technologies (e.g., face detection and other machine learning technologies) to provide powerful features for their users.

We examined apps across different app categories to mitigate any bias in our results that may occur if we included only a single app category. App Annie's report contains top popular apps across 28 app categories (e.g., games

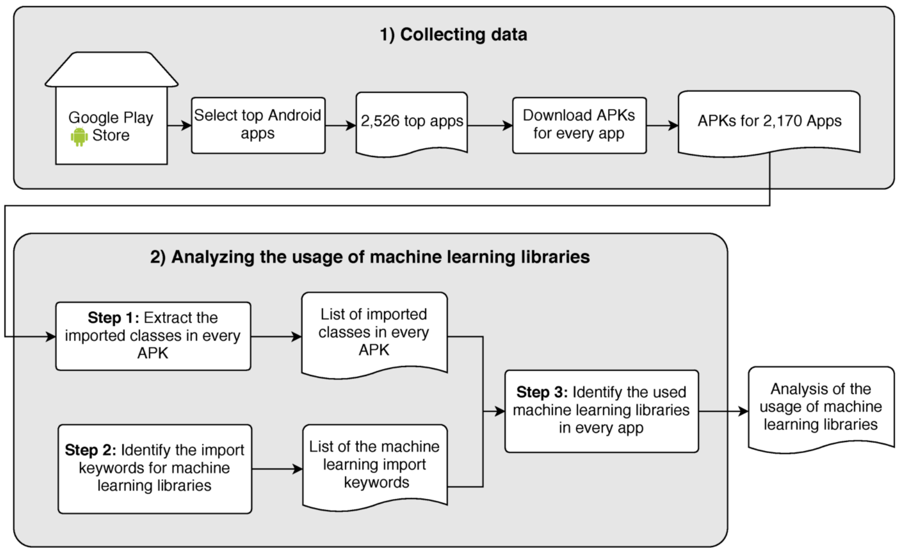

1) Collecting data
Google Play Store
Select top Android apps
2,526 top apps
Download APKs for every app
APKs for 2,170 Apps
2) Analyzing the usage of machine learning libraries
Step 1: Extract the imported classes in every APK
List of imported classes in every APK
Step 2: Identify the import keywords for machine learning libraries
List of the machine learning import keywords
Step 3: Identify the used machine learning libraries in every app
Analysis of the usage of machine learning libraries

Fig. 1 Keyword App mining methodology.

and social categories). In our study, we selected the top 100 apps in every app category (2,800 apps in total).

We used a Google Play crawler [7] to download the APKs for the studied apps. Our crawler connects to the Google Play Store using the Samsung S3 device model. We used the Samsung S3 device as it was one of the most popular devices when we started downloading the APKs [9].

We found that 214 apps were repeated across categories, 60 apps were already removed from the store (in total we studied 2,526 apps). We observed that the crawler was not able to download the APKs for 356 apps (out of the 2,526 apps) as these apps are not allowed for the used crawler device model (i.e., Samsung S3 device model and SDK version 19). This resulted in 2,170 APKs for the studied 2,526 apps.

With these 2,170 APKs, we examined which libraries that are used in every APK with the following approach that is also depicted in Figure 1

Step 1: Extract the imported classes in every APK. We identified the list of the imported classes as follows.

– We converted every APK to a jar file using Dex2jar tool [1,2]. We encountered four apps where the APK to jar failed.
– We decompiled the classes in the generated jar into Java source code using the Class File Reader (CFR) tool [12]. We encountered two apps where the jar decompilation failed. In total, we encountered six apps (out of the 2,170 apps) where either the APK to jar failed or the jar decompilation failed.

- For the 2,164 apps where apps decompilation succeeded, we examined classes within the app package name, by listing the import statements for these classes. For example, for the "Smart calendar" app with package name "com.google.android.calendar", we listed all import statements for all classes under the package 'com.google.android.calendar' and its subpackages. We examined the import statements for the classes under the app package name, as these classes are most probably written by the developers of the app themselves.
- We made a database of the import statements and related them back to the apps themselves.

Step 2: Identify the import keywords for machine learning libraries. We fashioned a list of machine learning related keywords that we would expect to see in import statements. These included:

- Weka and Weka relevant classes;
- Neuroph and Neuroph relevant classes;
- CoreML (Apple's IOS library) related namespaces;
- Wit.ai, a ML service, related namespaces;
- Amazon AWS Machine Learning related namespaces;
- Terminology from AI: gradient descent, heuristic search;
- Terminology from ML: classifiers, detectors, etc.;
- Terminology from relevant computer vision: face detection.

Table 1 shows the list of the identified keywords along with reason and an example for each keyword.

Step 3: Using the list of import statements in Table 1, we searched for these import statements in every app. Then we manually removed all the imports that were not machine learning relevant according to our manual determination. This means the removal of references to enumerations, deep-links, gesture detection, WiFi detection, linear gradients, linear UI widgets, text direction heuristics, etc.

Results: We have summarized the manually filtered matches in Table 1.

As shown in Table 1, a total of 4.3% of the studied apps, 92 unique apps out of the 2,164 apps, have observable machine learning relevant imports. The keywords in Table 1 may overlap thus the table shows the total unique number of apps for which we observed evidence that developers use machine learning libraries. None of the studied apps used Weka, Wit.ai, or CoreML. TensorFlow was imported by 2 of the apps (0.01%). The most popular API seemed to be the Android face detector API. This and the popularity of OpenCV implies that images are the most common form of machine learning data being evaluated. From the imports, it is not clear if any training was being executed within the apps themselves. Internal validity is threatened by obfuscation—only 443 of 2,164 APKs (20%) are not obfuscated—and direct including of source code which could hide many of the keywords we searched for. Thus, we have clearly demonstrated that machine learning is present in the app store ecosystem.

Table 1 Investigation of Machine Learning related imports in a variety of Android apps from the Google Play Store. The keywords an not-exclusive, there is overlap between keywords.

Import Package Keyword	# Unique Apps	**R:** Reason **E:** Example
Total Unique Apps	92	Total Unique Apps with Machine Learning relevant imports
`detector`	57	**R:** Shake detectors (an accelerometer gesture) and Face Detectors **E:** `com.google.android.gms.vision.Detector`
`facedetect`	38	**R:** Face detector libraries **E:** `android.media.FaceDetector`
`opencv`	13	**R:** Computer Vision library including OpenCV's Object Detector **E:** `org.opencv.objdetect.CascadeClassifier`
`classif`	9	**R:** Objects, Photos, Image Classifiers, Handwriting. **E:** `com.google.android.libraries.handwriting.-classifiers.b`
`org.apache.commons.math3`	7	**R:** Uses of an apache commons library for KMeans, statistics, clustering, etc. used in Plenty of Fish and Snapchat **E:** `org.apache.commons.math3.stat.regression.-SimpleRegression`
`classifier`	6	**R:** Object, Photo, and Image Classifiers, etc. **E:** `com.google.android.libraries.vision.-semanticlift.CoarseClassifier`
`regression`	4	**R:** Linear Regression and Simple Regression used by Plenty of Fish and others **E:** `org.apache.commons.math3.stat.regression.-SimpleRegression`
`linear`	2	**R:** Linear regression **E:** `com.zzkko.bussiness.tinder.component.-LinearRegression`
`tensor`	2	**R:** TensorFlow deep learning framework (1 JNI) in Azarlive App and imo.im app **E:** `org.tensorflow.safematch.ImageClassifier`
`machine`	2	**R:** Machine Learned models and Machine Translation in apps like Etsy **E:** `com.etsy.android.lib.models.apiv3.-editable.MachineLearnedTaxonomySuggestion`
`kmeans`	1	**R:** K-means clustering **E:** `org.apache.commons.math3.ml.clustering.-KMeansPlusPlusClusterer`
`com.google.android.libraries.vision`	1	**R:** A library that contains recognizers, face-detectors, and barcode scanners **E:** `com.google.android.libraries.vision.-semanticlift.CoarseClassifier`
`tensorflow`	1	**R:** Tensorflow deep learning found in Azarlive **E:** `org.tensorflow.safematch.Classifier`

`adam ai.wit autoencoder`
`bagging bayes c45`
`com.amazonaws.services.machinelearning`
`coreml decisiontree deep`
`discriminat forest genetic glm`
`gls gradient heuristic ibk j48`
`learner linearmodel logistic`
`machinelearning mlmodel mlp`
`multiclass nadam naive neural`
`neuroph oner perceptron predictor`
`rbf reptrees ripper sdg svm`
`validation weka wit.ai zeror`
	0	**R:** potential matches for Weka, Neuroph, CoreML, wit.ai or Amazon AWS machine learning services. But none found.

2 Prior work

Prior work relevant to this paper include software energy measurement, modelling, developer knowledge, optimization, and recommenders.

2.1 Software energy measurement

Software energy consumption is an up and coming field in software engineering and computer engineering. With the popularity of mobile devices and apps, more and more software engineering research is targeted to energy constrained platforms.

Energy consumption recommendations and guidelines for developers are popular avenues of research. Hasan *et al.* [32] and Pereira *et al.* [61] investigated the energy profiles of Java collections to help developers manually choose the right collection. Linares-Vásquez *et al.* [45] produced a methodology of finding energy consuming libraries and APIs in Android applications. Li *et al.* [43] discussed causes of energy consumption on Android.

Measuring software energy consumption is another avenue of research. We used the GreenMiner [35] in this paper to measure software energy, but other researchers such as Banerjee *et al.* [11] have made their own measurement frameworks. We discuss GreenMiner's configuration in more detail in Section 4.1.

2.2 Software energy estimation and modelling

A very popular area of research is the modelling of software energy consumption. Pathak *et al.* [59, 60] and Aggarwal *et al.* [4] used system-call based models. Chowdhury *et al.* [19] used count based models. Nucci *et al.* [22] augment software based models with battery measurements provided by Android APIs to achieve low error software energy estimates of running processes. Some tools attempt to diagnose the actual cause of software energy consumption in terms of the code [5].

2.3 Effect of software development on energy consumption

Others have investigated the impacts of different kind of software development techniques, behaviours, and algorithms. Sahin *et al.* [66, 67] investigated the energy cost of obfuscation, finding it typically did not have a severe effect. While many researchers have studied the energy consumption costs of GUI colours on OLED displays, and how to optimize colour choice [6, 75]. Until Agolli *et al.* [6], prior work in colour optimization ignored the original aesthetics of the colour scheme of the app GUIs. Agolli et al. optimize GUI colours while keeping them perceptually similar, retaining the original aesthetics of the original colour choice.

Numerous empirical studies exist about different aspects of software development juxtaposed against software energy consumption. Researchers such as Rasmussen *et al.* [63], and Gui *et al.* [29, 30] have investigated the cost of advertisements on energy consumption. Chowdhury *et al.* [18] and Li *et al.* [44] benchmarked HTTP related energy concerns. Many researchers have suggested ranking and measuring apps by energy consumption [17, 39, 64].

2.4 Energy optimization

Many have investigated the impact of optimization and micro-optimization of code with respect to various resources such as energy, memory consumption, and CPU runtime [47, 65, 68]. Sahin *et al.* [65] found that many recommended optimizations are meaningful in terms of benchmarking, but meaningless when employed within real world apps as the optimization was simply not run enough to have an effect. Linares-Vásquez *et al.* [47] found that optimizations that freed unused resources did have an effect, but also found that developers were relatively unsure of the effect of micro-optimizations. Selby *et al.* [68] studied methods of having the compiler optimize code for energy-consumption.

2.5 Developer knowledge of software energy consumption

Many researchers investigated what developers know about software energy, that motivates this paper because most of the works conclude that developers are woefully ill-equipped to address software energy consumption concerns. Pinto *et al.* [62] and Malik *et al.* [50] sought questions developers were already asking. Pang *et al.* [58] surveyed developers to see what they understood about software energy consumption. Manotas *et al.* [51] went further and surveyed numerous industrial developers. Linares-Vásquez *et al.* [47] found that developers were relatively unsure of how to reduce energy consumption and were skeptical of the effect on energy consumption of micro-optimizations.

2.6 Recommenders

Recommenders quickly turn into optimizers that apply search techniques and find solutions to software energy consumption concerns. SEEDS from Manotas *et al.* [52] attempts to find the most energy efficient Java collections to use in a program particular context. GUI optimizations have also been approached using a search-based approach by Linares-Vásquez *et al.* [46]. Bruce *et al.* [16] explicitly applied search-based software engineering techniques to mutate existing source code into more energy efficient code. Saborido *et al.* [64] use multi-objective heuristics to find optimal apps where energy consumption is one dimension to optimize. Like Saborido et al., Jabbarvand *et al.* [39]

discussed EcoDroid, a method of ranking Android Apps based on energy consumption.

3 Machine learners and configuration

In this section, we describe the algorithms and the implementations of the algorithms we employ. We also describe our choices in terms of parameters used for machine learning algorithm implementations, as well as datasets chosen to exercise these implementations.

3.1 Algorithms and implementations used

We tested eight machine learning algorithm implementations: *Naïve Bayes* (NB), *J48* (Weka's implementation of C4.5), *Sequential Minimal Optimization* (SMO) which is a support vector machine, *Logistic Regression* (LogReg), *Random Forest* (RF), *k-Nearest Neighbour* (IBk), ZeroR, and *MultiLayer Perceptron* (MLP) which is a neural network. All algorithm implementations except for MLP were from the Weka Java codebase. The MLP implementation, a neural network, is from the Neuroph framework. We could not find a sustainable and reliable method of configuring and running a MLP within Weka that would terminate in a reasonable time. These algorithms were chosen because they are popular machine learners that implement a variety of classification strategies.

ZeroR is a very simple classifier, that disregards any attribute information and always predicts the majority class of the training set. As such, ZeroR can provide the baseline accuracy for a dataset [78]. For a dataset with n training instances, ZeroR will take $O(n)$ time to build a classifier as it needs to check the class value of each instance in order to find the most frequent class. However, it takes virtually no time, constant time $O(1)$, to classify.

Naïve Bayes is a type of Bayesian network that uses the simplifying assumptions that the predictive attributes are conditionally independent, and that there are no hidden attributes that influence predictions. With these simplifying assumptions, given a dataset with d attributes, n testing instances and m training instances, the Naïve Bayes classifier can perform training and testing in $O(dn)$ and $O(dm)$ time respectively [40]. The Weka Naïve Bayes algorithm implementation used for these tests is not updatable (online), although Weka also has an updateable implementation of Naïve Bayes.

J48 is Weka's implementation of the C4.5 decision tree algorithm [23]. For a dataset with d attributes and n testing instances, C4.5 training has an algorithmic time complexity of $O(nd^2)$ [71]. To evaluate the tree, the height of the tree is traversed, making the worst-case time complexity $O(d)$.

SMO is an algorithm implemented in Weka for training a Support Vector Machine (SVM) classifier, that breaks down the SVM quadratic programming optimization to simplify implementation, speed up computation, and save memory [38] [70]. Platt found empirically that the training time of SMO ranges from $O(n)$ up to $O(n^{2.2})$ for n training instances [38]. For a dataset with c classes, the classification time is $O(c)$ per instance [80]. In Weka's implementation, datasets are automatically processed to replace missing values, normalize all attributes, and convert nominal attributes to binary ones.

Logistic Regression is a statistical machine learning algorithm. Weka's implementation uses logistic regression with the Quasi-Newton method, thus a dataset with d attributes and n instances takes $O(d^2n + nd)$ time per iteration [54] to train. For a dataset with c classes, the classification time is $O(c)$ per instance [80] For our tests logistic regression was set to iterate until convergence. Weka's implementation of the algorithm is slightly modified from the original Logistic Regression to handle instance weights.

Random Forest is an advanced tree classifier that grows multiple trees and allows them to vote for the best class [15]. For a forest with L trees, n instances, and d attributes, theoretically the random forest will be constructed in $O(Ln^2d \cdot log(n))$ time, although practically the complexity is often closer to $O(Lnd \cdot log(n))$ [73]. It takes $O(L \cdot log(n))$ time to classify an instance [3].

IBk is an instance-based learner algorithm implemented in Weka, that is similar to the k-nearest neighbour algorithm [21]. For our tests, we classified instances based on the nearest three neighbours ($k = 3$). IBk is lazy when training, taking almost no time to create a model [57]. However, for a dataset with d attributes and n instances, it takes $O(nd)$ to classify an instance [21].

MLP (multi-layer-perceptron) is a neural network implementation. For our tests, MLP used back-propagation learning and had only one hidden layer of neurons. The number of hidden neurons was fixed at 15 and the number of training epochs was fixed at 100. In general, for a dataset with n instances and a neural network with a input neurons, b hidden neurons, and c output neurons, the network will take $O(nabc)$ time to train per epoch [55]. It takes $O(a + b + c)$ (the total number of nodes) time to classify an instance.

3.2 Parameters of implementations used

We describe the parameters in depth so that future researchers may replicate these results with similar implementations of the algorithms, most parameters were the default Weka parameters and were un-tuned. Neural network parameters were manually specified. Meta-heuristic search could be employed to tune parameters but that would add many dimensions of complexity to this study. We recognize that un-tuned parameters, and parameter tuning is not covered by this current study is a definitely an open area of research for sustainability of machine learning algorithms and their implementations.

Naive Bayes parameters as implemented by Weka were: `useKernelEstimator` *False* (do not use a kernel estimator); `useSupervisedDiscretization` *False* (do not convert numeric values to nominal values).

C4.5 parameters for Weka's J48 implementation were: `binarySplits` *False*; `confidenceFactor` 0.25 (for pruning); `minNumObj` 2 (minimum instances per leaf); `reducedErrorPruning` *False*; `subtreeRaising` *True*; `unpruned` *False* (enables pruning); `useLaplace` *False* (disable Laplacian smoothing).

SVM parameters for Weka's SMO algorithm were: `buildLogisticModels` *False*; `c` 1.0 (complexity); `epsilon` $1.0e-12$; `filterType` "Normalize training data"; `kernel` "PolyKernel" (polynomial kernel) with `cacheSize` 250007 and `exponent` 1.0; `toleranceParamter` 0.001.

Logistic Regression Weka parameters were: `maxIts` -1; `ridge` $1.0e-8$ (Ridge value in log-likelihood).

Random Forest Weka parameters were: `maxDepth` 0 (unlimited depth); `numFeatures` 0 (unlimited number of features per tree); `numTrees` 100 (number of trees generated); `seed` 1 (seed for random generator).

IBk parameters were: `KNN` 3 (3 neighbours); `crossValidate` *False* (disable cross validation to set k); `distanceWeighting` "No distance weighting"; `meanSquared` *False* (use mean absolute error); `nearestNeighborSearchAlgorithm` "LinearNNSearch" (linear scan); `windowSize` 0 (unlimited number of training examples).

MLP parameters used for the neuroph library were manually chosen: layers 3 (input, hidden, output); hidden layer neurons 15; learning rate 0.2 (fast and coarse); epochs 100 (constant number of epochs); activation function *sigmoid*. These parameters build an MLP with 1 hidden layer that can be trained relatively quickly on a mobile device, even on networks with 2000 inputs.

3.3 Datasets used

We used seven existing datasets to test the machine-learning algorithm implementations. The datasets chosen were of different sizes and datatypes, and represented different classification problems. We used a commit topic classification dataset (PGSQL) [34], the MNIST number classification dataset [42], and five datasets from the UCI archive [48] (Mushroom, Adult, Waveform, Spambase, and Pendigits). MNIST and Pendigits are image classification problems; PGSQL and Spambase are text classification problems; Adult and Waveform are numeric classification problems; and Mushroom is categorical classification.

We chose these datasets as they are standard workloads in machine-learning literature, and because we believe them to be representative of classification problems that may be posed to an app. For example, the MNIST and Pendigits datasets are similar to the image classification done by Google Translate [24] and mobile Tensorflow [72], and text classification problems, such as PGSQL and Spambase, are similar to common mobile classification problems such as spam filtering. Examples of datasets actually used by mobile applications are not clearly available, and would be difficult to test.

Weka is designed to work with the ARFF file format. A version of the MNIST dataset already converted to the ARFF format was obtained [49] and used for the tests. The other datasets were converted to ARFF files using the Weka Explorer's conversion capabilities. Due to memory limitations, for our tests the size of the MNIST dataset was reduced to 5000 randomly selected

Table 2 Size and type of datasets used in energy tests.

Dataset	Description	Attributes	Instances	Classes
MNIST	Image classifier – Integer attributes	785	5000	10
PGSQL	Text classification – Binary categorical attributes	2000	400	2
Mushroom	Classification – Categorical attributes	23	8124	2
Adult	Classification – Categorical, integer attributes	15	32561	2
Spambase	Text classification – Integer, real attributes	58	4601	2
Waveform	Numeric classification – Real attributes	22	5000	3
Pendigits	Image classifier – Integer attributes	17	10992	10

instances. The size of the PGSQL dataset was also reduced from 640 instances with 23008 attributes to 400 instances with 2000 attributes, one of which was the class. The datasets are summarized in Table 2.

The MLP implementation we used from the Neuroph framework required datasets in CSV format. It also requires that numeric attributes be normalized to values between 0 and 1, nominal attributes and classes be represented as one-hot binary inputs, and instances with missing attribute or class values be removed beforehand. This processing and conversion to CSV was done using the Weka Explorer. As a result of converting categorical attributes to one-hot binary attributes, the number of input neurons for the Mushroom dataset became 111, and 104 for the Adult dataset.

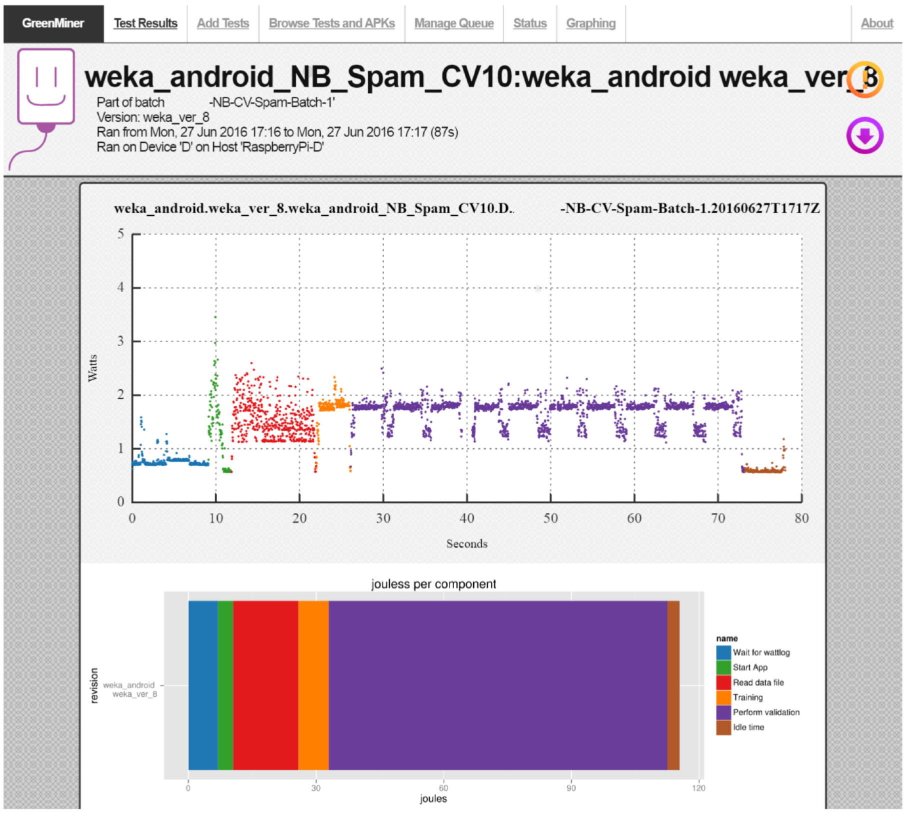
A mirror of our datasets can be found at this url: `https://archive.org/details/mnist_test_reduced_5k`.

4 Methodology and measurements

In this section, we describe how we setup benchmarks for the machine learning algorithm implementations and datasets. We also describe how we measured the energy consumption of the machine learning benchmarks.

4.1 Energy measurement with GreenMiner

Energy and power measurements were collected using the GreenMiner energy-measurement framework. This framework uses hardware-instrumented Android smartphones to physically measure the energy consumption and power use of apps running on the phones [35]. Each GreenMiner client includes a Raspberry Pi for data-collection; a Galaxy Nexus running Android 4.2.2; and a INA219 current sensor for energy measurement. The INA219 measures both voltage and current. It uses a shunt (a low resistance calibrated resistor) to measure the current via a mild voltage drop across the shunt. This system of external measurement of energy does not induce instrumentation overheads except in the script to run the tests and send input commands. In total there are four GreenMiner clients that test runs are distributed between them. This reduces the effects of device-specific abnormalities when averaging over a batch.



weka_android_NB_Spam_CV10:weka_android weka_ver_8
Part of batch -NB-CV-Spam-Batch-1'
Version: weka_ver_8
Ran from Mon, 27 Jun 2016 17:16 to Mon, 27 Jun 2016 17:17 (87s)
Ran on Device 'D' on Host 'RaspberryPi-D'

Fig. 2 Example of a GreenMiner profile for a test run of 10-fold cross validation on Naïve Bayes with the Spambase dataset.

The GreenMiner framework automatically runs submitted tests and uploads the results to a central web-service. Before each test is run, the application APK (Android package) is installed on the phone, required data is uploaded onto the SD card, and phone settings such as screen brightness, and screen timeout are set as required. After each test the application is uninstalled, the data is deleted from the SD card, settings are restored to previous values, and data generated during the tests such as log-files are pulled from the phones to be uploaded to the web service and then deleted from the phone, so that the next test can begin with a clean environment. Tests run for a set duration, and testers can split the test's energy measurements into partitions of varying duration to capture the energy and power use of different phases of app execution. Such a phase could be reading the data or training the model.

The GreenMiner measures and reports information about the test run including energy use, power use, and runtimes for both the entire test duration and over each tester-specified partition. An example of an energy profile for a cross-validated Naïve Bayes test displayed on GreenMiner's web interface is shown in Figure 2.

Newer versions of Android were not used because 4.2.2 was the last official release for the Galaxy Nexus phones. To use newer versions we would have to install non-standard firmware. Furthermore, to use multiple versions we would have to modify the Green Miner software to re-flash the phones between versions of tests. This could be quite valuable but is expensive in terms of time to develop and run. Support for multiple OS versions and OS re-flashing is currently unavailable within the Green Miner. Not using a newer version of Android could be an issue because of changes to the Android runtime. Android 4.2.2 uses DalvikVM and newer versions of Android use the Android Runtime (ART) [41]. Konradsson [41] compared DalvikVM and ART performance and found they differed somewhat in terms of memory use, garbage collection, start-up time, IO time, and other aspects. Typically ART was had better CPU performance but it depended on the test. ART often changes user-space app performance as it can pre-compile apps rather than just in time compilation like DalvikVM—ART can combine both in hybrid-on-device compilation. Other issues with OS versions can arise from the change in APIs meant to access resources, to the adoption of energy aware APIs. Thus performance of user-space apps on newer Android OSes could be somewhat different than on DalvikVM.

4.1.1 How do measurements of Energy and Power translate into battery life?

In this section, we explore how *energy*, measured as joules (J), or *power*, measured as watts (W), translate to to the end user: battery time. The factory battery that comes with a Galaxy Nexus is a 1800mah 3.7 V battery. Typically you get 4.2 to 4.3V from the battery when full and get above 3.6V near empty. A new battery promises 1800mah at 3.7V which is about $23976J$ (e.g., $3.7V \cdot 1.8Ah \cdot 3600s$). Older batteries decay, often rapidly and offer far less energy for use because their voltage dips below 3.7V sooner. Mobile devices typically cannot operate on lower voltages or drain the battery to 0V and 0J. Furthermore, the only way to actually know how much energy is in a battery is to use and consume it, which means that any measurement you see reported on your mobile phone is actually an estimate of energy left by the battery circuitry and your phone's power systems.

In this section, we make the following assumptions: your battery is new and has not been recharged numerous times; your battery is fully charged. Batteries change in charge profile over time and should generally not be used as sources of energy for energy measurement testing because they change over time, act differently at different temperatures and their behaviour changes on each recharge. We have also based our loads on observations from numerous runs on the GreenMiner. Typically these Galaxy Nexus phones are running at 0.1W idle with screen off, with screen on and idle and with screen brightness at the same conditions we tested at they are at 0.7 W, and finally when they are under heavy load (many cores at 100% CPU combined with network and disk I/O) they are at 1.8W—it can be more if you use the cellular network at the same time.

Table 3 Conversion of Joules to Galaxy Nexus Battery Time. Assuming a 1800mah battery (at 3.7V a maximum of $23976J = 3.7V \cdot 1.8Ah \cdot 3600s$).

Cost in Joules	δ Time with screen off (0.1W)	δ Time with screen on (0.7W)	δ Time under heavy use (1.8W)
Idle/Load Watts	0.1 W	0.7 W	1.8 W
1 J	2.70 s	0.39 s	0.15 s
10 J	27.03 s	3.86 s	1.50 s
50 J	135.14 s	19.31 s	7.51 s
100 J	270.27 s	38.61 s	15.02 s
500 J	1 351.35 s	193.05 s	75.08 s
1000 J	2 702.70 s	386.10 s	150.15 s
5000 J	13 513.51 s	1 930.50 s	750.75 s
10000 J	27 027.03 s	3 861.00 s	1 501.50 s

Before we convert joules or watts to battery time/life we must understand that the battery time/life depends on the load it is under. This means there is no single true measurement or estimate for cost in battery life. If your phone is mostly idle and you do some machine learning work at 0.1W for 100 seconds (10.0J) the change in battery life is more significant than if you do the same task when your phone is working heavily (1.8W). The difference in battery time between 1.8W and 1.9W (1.5 s) is not as stark as the difference between 0.1 W to 0.2W (27 s). Table 3 shows the difference in seconds of battery life time for different tasks under different conditions.

It is easier to characterize the difference in battery life for energy measurements because they are a fixed cost. For power measurements (watts W) we have to make some assumptions about how long the task runs. Typically we'll assume that you run this task constantly. There are use-cases where this makes sense such as mobile sensors, computer vision, or some kind of continuous job task. Table 4 assumes you have a fixed load (idle, or screen on, or some heavy background task) and you have added an additional load from a machine learner, it assumes you want to do as much work as you can before the battery runs down and thus the table estimates entire battery lifetime for the fixed load and your added workload.

Thus because battery life estimates are just estimates and depend on context (base load and battery and phone) we present energy consumption (J) and power (W) for the rest of the paper because those are the actual measurements. You may use these tables (Table 3 and Table 4) to convert from the reported values to Galaxy Nexus battery life.

4.2 Measurement process

Figure 3 shows the overall process for designing our empirical study and collecting energy consumption measurements and power measurements. To test machine learning algorithm implementations on the GreenMiner phones, two Android apps were created. An app was created to run Weka machine learn-

Table 4 Conversion of Watts (W) to Galaxy Nexus Battery Time under continuous load. In this scenario you never quit running the default load plus the induced load until the battery runs out. Assuming a 1800mah battery (at 3.7V a maximum of $23976J = 3.7V \cdot 1.8Ah \cdot 3600s$).

Cost in Watts	Total time with screen off (0.1W)	Total time with screen on (0.7W)	Total time under heavy use (1.8W)
Idle/Load Watts	0.1 W	0.7 W	1.8 W
0.00 W	66.60 h	9.51 h	3.70 h
0.10 W	33.30 h	8.33 h	3.51 h
0.25 W	19.03 h	7.01 h	3.25 h
0.50 W	11.10 h	5.55 h	2.90 h
0.75 W	7.84 h	4.59 h	2.61 h
1.00 W	6.05 h	3.92 h	2.38 h
1.25 W	4.93 h	3.42 h	2.18 h
1.50 W	4.16 h	3.03 h	2.02 h
1.75 W	3.60 h	2.72 h	1.88 h
2.00 W	3.17 h	2.47 h	1.75 h
2.25 W	2.83 h	2.26 h	1.64 h

ing algorithm implementations, based on an existing modification of the Weka codebase that can run on Android.[1] A second app was created to test a MultiLayer Perceptron neural net algorithm, using the Neuroph framework. Both apps ran the same datasets.

Tests of the different algorithm implementations and datasets were written as Android `InstrumentationTestCases`, with the phases of evaluating an algorithm's implementation (reading data, training the model, validating the model) written as separate tests. The different tests were initiated by pressing buttons, and data was transferred between different test methods via a singleton object. To keep the screen energy consumption of the apps constant, the screens were almost completely black, with some small grey text on the buttons for debugging purposes. Both the Weka and the Neuroph apps had exactly the same user interface.

The tests were very simple and single threaded, they did no UI work, and had logging statements and assertions. The tests called Weka and Neuroph respectively and delegated all the work to Weka and Neuroph. Data was stored as an ARFF file for Weka and CSV for Neuroph. Each dataset was loaded using the respective libraries I/O routines via BufferedReader from the external storage. Reading datasets was timed such that once the dataset was in memory the reading energy would not be counted. Once the dataset was loaded, the training and evaluation or the cross-fold validation would begin. Weka handled 10-fold validation within the library while Neuroph was provided with a 10-fold validation implementation. An implementation of 50/50 split was created for both Neuroph and Weka applications. When either app trained or evaluated it was the respective library's responsibility. The test app lets one choose what implementation to test, then buttons 1) to read the data, 2) to train on 50% of the data, 3) to evaluate on 50% of the data, 4) train on all training data (for

[1] Weka for Android `https://github.com/rjmarsan/Weka-for-Android`

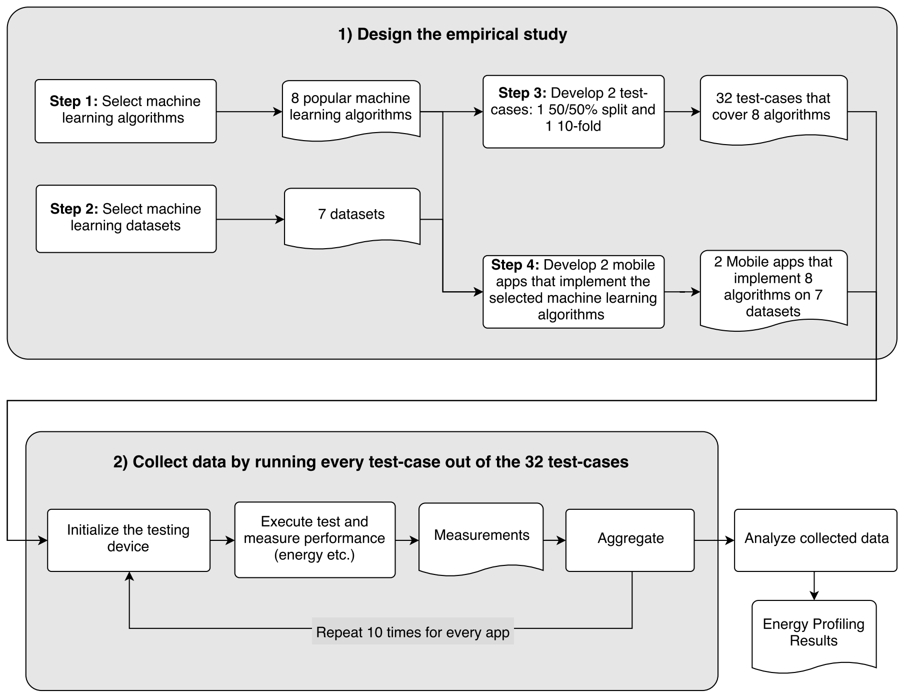

1) Design the empirical study
Step 1: Select machine learning algorithms
8 popular machine learning algorithms
Step 3: Develop 2 test-cases: 1 50/50% split and 1 10-fold
32 test-cases that cover 8 algorithms
Step 2: Select machine learning datasets
7 datasets
Step 4: Develop 2 mobile apps that implement the selected machine learning algorithms
2 Mobile apps that implement 8 algorithms on 7 datasets
2) Collect data by running every test-case out of the 32 test-cases
Initialize the testing device
Execute test and measure performance (energy etc.)
Measurements
Aggregate
Analyze collected data
Energy Profiling Results
Repeat 10 times for every app

Fig. 3 The overall process for designing our empirical study and collecting energy consumption measurements and power measurements.

timing and debugging) and 5) to engage in cross-fold validation. The Green Miner test was scripted to run 50/50 split training and testing separately by pressing UI buttons. Where as 10-fold cross validation was a single operation via 1 UI button. By using the UI to prompt testing it allows the Green Miner to segment the timeseries of energy measurements based on task.

Tests were created for eight different machine learning algorithm implementations to evaluate seven different datasets. Separate tests methods were written to perform two different types of evaluation. For each algorithm implementation two tests were written to train on 50% of the data and then test on the other 50%. Two more tests were written to train and test on the whole dataset using 10-fold cross validation. Each train/test evaluation pair ran separately on the GreenMiner.

The 50% evaluation was chosen so we could provide a balance between training and testing. This allows for tests to be partitioned/segmented and analyzed for their training and testing energy consumption. Furthermore, it means that train and test will be receive the same number of instances to make them comparable. Whereas 10-fold cross validation does far more training work than evaluation work. 10-fold cross validation will train 10 times on 90% of the data, where 50% evaluation will train 1 time on 50% of the data. This means that the total cross-folds validation includes a lot more training time and instances than evaluations, where as 50% evaluation is a balance between training and evaluation. 50% results should be interpreted as a balanced evaluation comparing training and testing performance. 10-fold cross validation tests model training with more subsets and larger amounts of training data and is less focused on evaluation performance.

We chose to test training as well because in many cases mobile devices will be used in scenarios where privacy is a concern, such as health information (EMG, sensors, fitness, and images of the body), this means that some models will have to be trained locally on information from the user themselves. Others [13, 28] have provided motivation for machine learning training and evaluation client-side to avoid privacy issues. Training on a mobile device is also useful because it does not need network access or a 3rd party service.

Each test method was invoked in turn by pressing a button on the app's interface once the previous method had completed. The GreenMiner framework cannot automatically detect when a test method has completed, because it runs uninstrumented, so in order to invoke the next method initial timing test runs were performed to determine appropriate delays to add to the GreenMiner scripts. Each algorithm-dataset-validation combination ran at least 10 times on the GreenMiner so that their results could be averaged and to allow for enough statistical power to determine an effect. Some combinations, such as random forest on the MNIST dataset with cross validation, ran out of memory when evaluating on the phones, and so are not included in our results. A classifier might not operate in certain contexts, but might be appropriate for others. The choice of a classifier might depend on the size of the instances and the number of instances so it is up to developers to investigate and balance these aspects.

The GreenMiner collects the energy consumption measurements and power measurements of each test method. The results of all successful test runs were compiled and compared. For comparisons, the training and testing phases of 50% split evaluation are combined, and are compared against the energy for cross-validating with 10-folds, that includes training and testing each fold. Energy consumption measurements are compared to determine which algorithm implementations will require the most or least energy to evaluate on each dataset. Power usages are compared to determine if some algorithm and their implementations are more energy-hungry, independent of how long it takes them to evaluate.

The machine learner performance, the accuracy, of the Weka algorithm implementations was gathered from the Weka 3.8 desktop application, based on performing 10-fold cross validation. The total root-mean-squared errors (RMSE) of the MLP algorithm were gathered from NeurophStudio. The average accuracies of an algorithm's implementation over all datasets were compared to determine which algorithm implementations were generally the most or least accurate. The accuracy for Logistic Regression could not be calculated for the MNIST dataset because both of the mobile Weka implementation and the desktop Weka application ran out of memory.

Statistical significance testing was executed using a Student's t-test as energy measurement data typically is normally distributed. Anders-Darling tests confirmed normality in the great majority of cases. Thus we opt to apply parametric statistics such as ANOVA and t-test due to efficient estimates and statistical power gained from using test appropriate for the measurements. For cases (batches of measurements) that were not normal according to Anders-Darling tests we applied the same parametric tests. We assume that if we had executed more measurements in these minority cases that Anders-Darling would report that there was a lack of evidence that the underlying distribution was not normal. We addressed multiple hypotheses and comparisons by applying Bonferroni correction with an initial alpha (α) of 0.05.

5 Energy profiling results

We profiled the energy and power use of eight machine learning algorithms, and compared how they varied with datasets of different sizes. We asked three research questions:

RQ1: Can we identify the best performing algorithm implementation in terms of energy?

RQ2: Can we identify the best performing algorithm implementation in terms of power?

RQ3: Can we identify the best performing algorithm implementation in terms of accuracy?

5.1 RQ1: Can we identify the best performing algorithm implementation in terms of energy?

This RQ deals with the efficiency of an entire run of machine learning from start to finish. This analysis is best for tasks that are limited in scope or work, such as classifying an email as spam or not. There is a start and end to the training or learning. Tasks that are continuous are more suited to analyzing their power use.

Which algorithm's implementations are more energy efficient? Figure 4 shows the energy used to train and test the algorithm implementation on a 50% split of each dataset. Figure 5 shows the energy used to perform 10-fold cross validation on the algorithm implementations for each dataset. Note that some algorithm implementations could not be evaluated on some datasets, and so not all algorithm-dataset combinations are shown in the figures.

Generally, energy consumption increases with increasing dataset size, however these increases typically do not strictly follow a clear trend. One reason for deviations could be related to memory cache; spikes in energy consumption could be due to the memory cache exhaustion for that particular dataset.

Figure 4 shows that other than ZeroR, Naïve Bayes and J48 tend to have the lowest energy consumption for 50%-split. SMO also has good energy performance for most datasets except for the Adult dataset. Figure 5 shows that

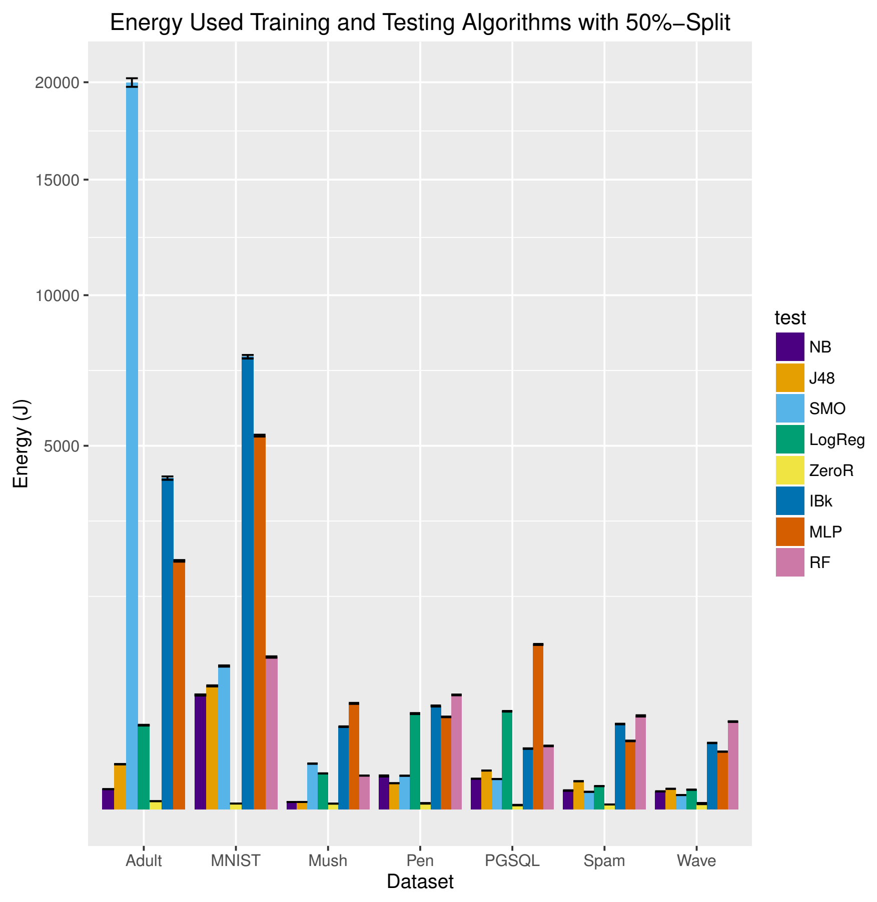


Fig. 4 Energy consumption to train and test on 50% split.

Naïve Bayes consistently consumes the least energy for cross validation, and J48 is one of the highest energy users for smaller dataset sizes, but one of the lower energy consumers for larger datasets.

The overall rankings of the algorithm implementations' energy use were determined by assigning a rank value to each algorithm implementation for each dataset, with 1 using the least energy and 8 using the most. The rankings for each dataset were then summed, and divided by the number of datasets. Table 5 shows that ZeroR always uses the least amount of energy, followed by Naïve Bayes and J48. There were some deviations in the rankings of each algorithm implementation on a dataset between cross-validation and 50% split. The order of average rankings for each evaluation method had high correlation of 0.93.

The energy use of the algorithm implementations were compared using an ANOVA to determine if the energy differences are statistically significant for an alpha (α) of 0.05. For the combined training and testing energies of 50%

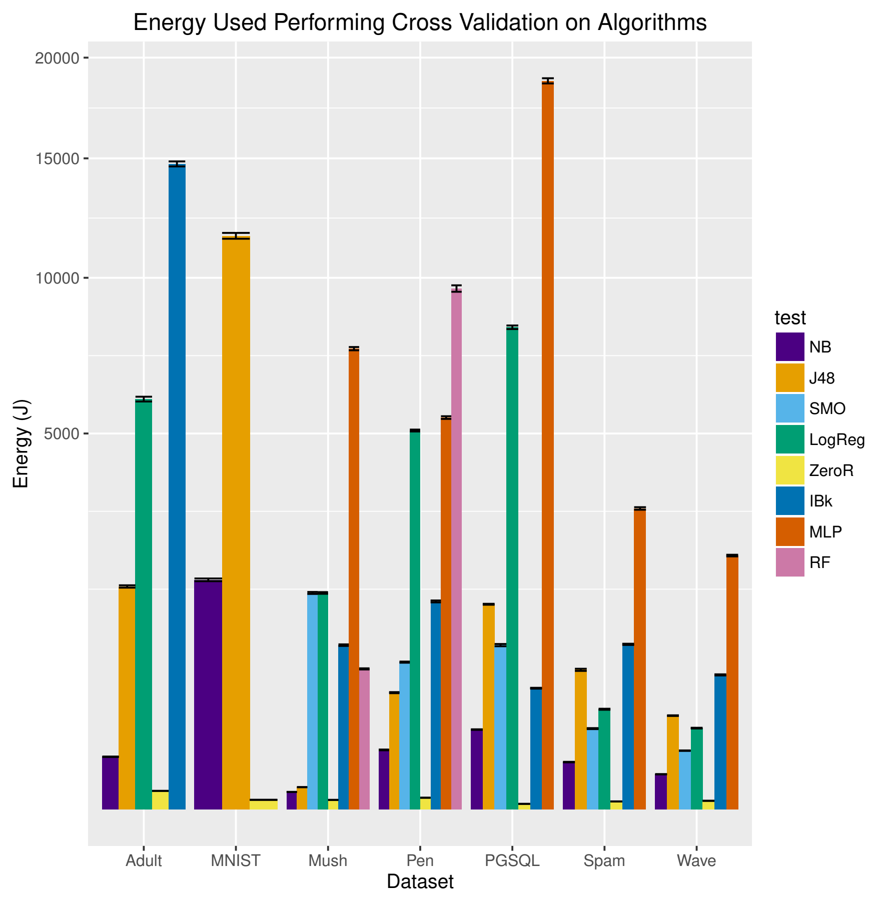


Fig. 5 Energy consumption to perform 10-fold cross validation.

split, ANOVA reported a p-value of $5.69e^{-15}$. For the combined training and testing energies of cross validation, ANOVA reported a p-value of less than $2e^{-16}$. It can be concluded that the energy use differences between algorithm implementations are significant.

Figure 6 compares the average energy used to train and the average energy used to evaluate each algorithm implementation over all datasets with 50% split. Lazy algorithms such as IBk were the most efficient for training, followed by Naïve Bayes. For evaluation, other than ZeroR J48 was quite efficient in terms of energy at classifying data. Naïve Bayes performed well for both training and evaluating.

Table 5 Average ranking of each algorithm implemention from lowest to highest energy consumption.

Sorted Algorithm	Rank 50%	Sorted Algorithm	Rank 10-CV
ZeroR	1	ZeroR	1
NB	2.57	NB	2
J48	3.57	J48	3.86
SMO	3.86	SMO	4.43
LogReg	5.43	LogReg	5
MLP	6.29	IBk	5.29
IBk	6.57	RF	7.14
RF	6.71	MLP	7.29

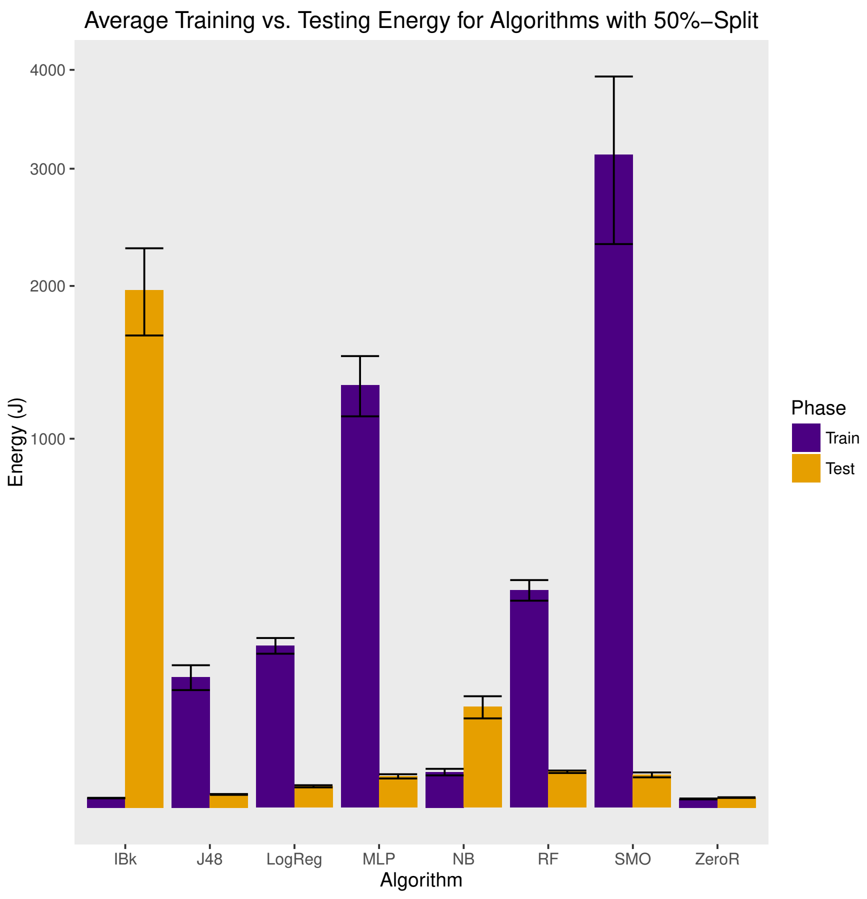


Fig. 6 Comparison of average energy use training and testing algorithms with 50% split.

Table 6 Average ranking of each algorithm implementation from lowest to highest power use.

Sorted Algorithm	Rank 50%	Sorted Algorithm	Rank 10-CV
ZeroR	1.43	ZeroR	1.14
NB	3.14	NB	2.86
MLP	3.57	LogReg	3.71
J48	4.43	J48	4.29
SMO	4.71	MLP	5
IBk	5.86	IBk	5.71
RF	6.14	SMO	6.29
LogReg	6.71	RF	7

5.2 RQ2: Can we identify the best performing algorithm implementation in terms of power?

This RQ focuses on the immediate performance of machine learning and not the entire task. Analysis via power is relevant because some algorithms could be parallelizable and deployed to multiple cores. These algorithms could use more power than non-parallel algorithms. Power can be important especially if you are running the task continuously, such as classifying images coming from the web-cam, or classifying audio snippets from the microphone. These tasks take as long as the user is using the app in question. Hence, developers of apps that run machine learning algorithms for a long time should consider selecting algorithms that are efficient in their power usage.

Figure 7 shows the average power use to train and test the algorithm implementation on a 50% split of each dataset. Figure 8 shows the average power use of each algorithm implementation performing 10-fold cross validation. Note that some algorithm implementations could not be evaluated on some datasets, and so not all algorithm-dataset combinations are shown in the figures.

Figures 7 and 8 show that the power use of all algorithm implementations are similar. Table 6 shows the average rankings for the algorithm implementations are less evenly-spread between 1 and 8, indicating that the rank of an algorithm implementation's power use varies more from dataset to dataset. Additionally, the rankings of algorithm implementation between 50% split and cross validation are not as well-correlated as the energy rankings, with a Spearman's rank correlation ρ value of 0.62. However, overall the algorithm implementations' power rankings are similar to the energy rankings, with ZeroR and Naïve Bayes consistently having the lowest power consumption.

The power use of the algorithm implementations were compared using ANOVA to determine if the power use differences are statistically significant for an alpha of 0.05. For the average training and testing power use of both 50% split and cross validation, ANOVA reported p-values of less than $2e^{-16}$. It can be concluded that the differences in power use between algorithm implementations are significant.

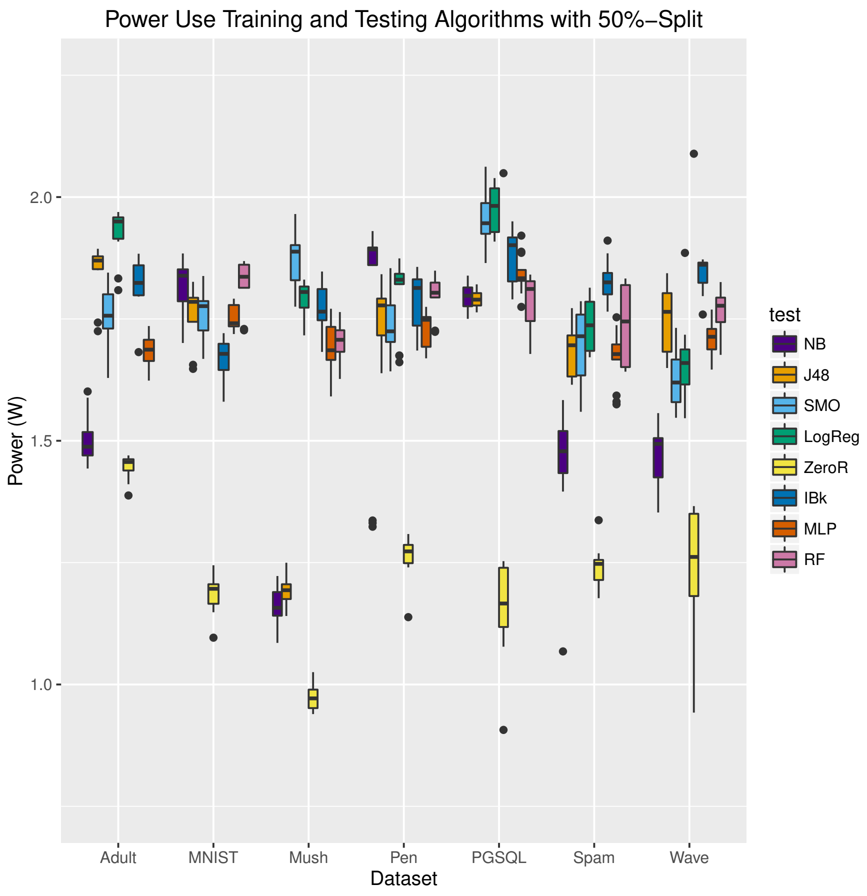


Fig. 7 Power consumption to train and test with 50% split.

5.3 RQ3: Can we identify the best performing algorithm implementation in terms of accuracy?

Algorithmic accuracy is determined based on the percentage of correctly classified instances and on the kappa statistic. Kappa measures agreement between prediction and the true class. As different algorithm implementations sometimes had the same accuracy for a dataset, rather than ranking algorithmic accuracy for each dataset — which would result in ties — the average accuracy of each dataset was calculated. As the accuracy for Logistic Regression could not be calculated for the MNIST dataset, the average for Logistic Regression was taken over only 6 values, while the other algorithm implementations were calculated over 7. Table 7 shows the algorithm implementations ordered in terms of both measures of accuracy.

Weka outputs predicted classes, and also provided a calculation of the root mean squared error (RMSE) of the predictions. Neuroph outputs the proba-

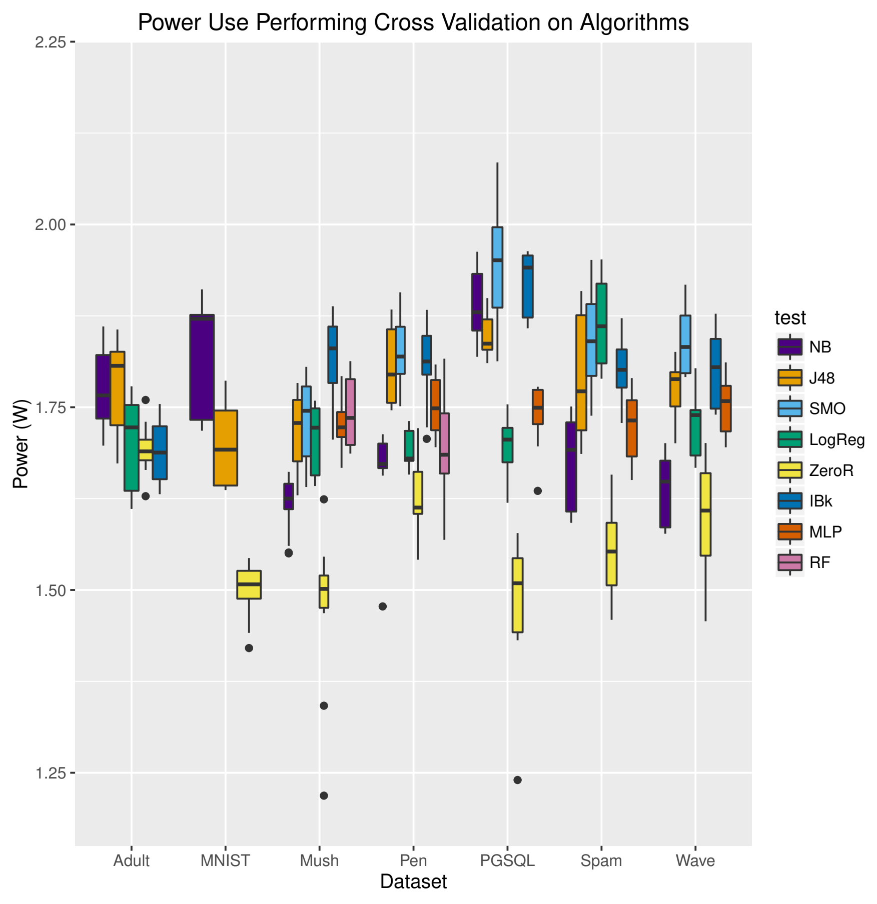


Fig. 8 Power consumption to perform 10-fold cross validation.

bilities of each class. These probabilities were normalized using softmax, and the highest normalized probability was taken as the predicted class. Then the accuracies and kappa statistics for MLP on each dataset were computed in R. The total RMSE of MLP on each dataset was obtained from NeurophStudio. The average RMSE of each algorithm implementation over all datasets is included in Table 7.

Table 7 shows the most accurate Weka algorithm implementations are Random Forest and SMO; their percentage of correctly classified instances are very close, with Random Forest being about 0.2% higher. Yet SMO had a slightly better kappa statistic implying its classifications are more balanced. Overall, MLP is clearly the most accurate algorithm implementation. It has significantly higher average classification accuracy and kappa statistic than the next-best algorithm implementation, and the lowest RMSE.

Table 7 Average algorithm implementation accuracies ordered based on percentage of correctly classified instances, kappa statistic, and Root Mean Squared Error using Cross-Validation.

	Rank by Accuracy		Rank by Kappa		Rank by RMSE	
	Algorithm	% Correct	Algorithm	Kappa	Algorithm	RMSE
Most	MLP	95.66%	MLP	0.9293	MLP	0.08
	Random Forest	90.32%	SMO	0.7488	Random Forest	0.21
	SMO	90.13%	Random Forest	0.7211	IBk	0.21
	IBk	88.32%	IBk	0.7194	LogReg	0.25
	LogReg	87.08%	LogReg	0.7087	J48	0.25
	J48	85.73%	J48	0.6911	SMO	0.29
	Naïve Bayes	81.97%	Naïve Bayes	0.6332	Naïve Bayes	0.32
Least	ZeroR	46.36%	ZeroR	0.0000	ZeroR	0.41

Table 8 Spearman rank correlation ρ value for 50% split energy use and CPU use between algorithms classifying a dataset.

Dataset	User Time	System Time	Idle Time	IO Wait Time	Number of Interrupts	Context Switches	Processes
Adult	1.00	0.57	1.00	0.07	0.96	0.79	0.85
MNIST	1.00	0.61	1.00	0.04	0.96	0.82	0.93
Mushroom	1.00	0.76	0.90	0.52	0.95	0.86	0.64
Pendigits	0.98	0.36	1.00	0.57	0.95	0.74	0.83
PGSQL	1.00	0.19	0.98	0.17	0.76	0.12	0.81
Spambase	1.00	0.00	0.98	0.45	0.79	0.07	0.50
Waveform	1.00	0.14	0.93	0.19	0.67	0.33	0.95

6 Causes of energy differences

6.1 Is energy use related to the CPU usage of an algorithm's implementation?

Before and after running a test, the phone's `/proc/stat` file is collected to gather information about the phone's CPU time and processes. The difference between the two measurements is used to determine the CPU time and resource usage of a test. These results are compared to determine how an algorithm implementation's CPU usage is related to its energy usage.

When comparing the results from 50%-split tests, energy use was strongly correlated to user time and idle time for all datasets. Table 8 shows that energy consumption was not strongly correlated to system time usage or IO wait time for most datasets. Energy was strongly correlated to the number of interrupts for most datasets, except for PGSQL and Waveform, where it was only moderately correlated. For other CPU use measurements, the strength of correlation to energy usage varied widely between datasets. The results were similar for cross-validation.

In general, the correlations between energy use and CPU use were stronger for cross validation. It should be noted that the Adult and MNIST datasets could not be evaluated by many algorithm implementations on the phones because they ran out of memory. Thus, there are fewer energy results to compare for these datasets.

For the 10-fold results, energy use was strongly correlated to user time, idle time, and number of processes. The number of interrupts was also well-correlated to energy use for all datasets. IO wait time was not strongly correlated to energy use, and, excluding the Adult and MNIST values, system time was generally not strongly correlated to energy use for any dataset.

The number of processes did not significantly increase between 50% split evaluation compared to cross validation. On average, over all datasets and algorithm implementations, only 1.2 times as many processes were created for cross validation as compared to 50% split. In contrast, on average, 10-fold evaluation used 7.0 times more idle time, and 10.5 times as much user time.

6.2 Is energy use related to the memory use of an algorithm's implementation?

Android's Dalvik VM automatically logs information about heap use and garbage collection (GC). These logs were collected for the algorithm implementations and datasets using Android's logcat tool. These logs have the number of kilobytes allocated for and used on the heap, the number of times the app's heap size was grown, the number of concurrent GCs performed when the heap grows too large, the number of GCs performed when the heap is too full to allocate required memory, and the total time taken to perform these GCs, could be parsed and compared.

Logistic Regression and Random Forest used the most memory on the heap and performed the most concurrent garbage collections. Overall, they are the most inefficient in terms of memory use. It should also be noted that Random Forest's performance was most affected by memory, as five datasets could not be evaluated with 10-fold cross validation on the phones as they ran out of memory or had a stack overflow occur. Excluding both MLP and ZeroR, Naïve Bayes, J48, and IBk performed the fewest garbage collections to make space for allocations, grew their heap the fewest number of times, and used the least amount of heap space. Random Forest and Logistic Regression were both large energy users, while Naïve Bayes and J48 were the lowest energy users, so for these algorithm implementations their memory use seems related to their energy use. However, IBk was one of the most memory-efficient in terms of space used for the instances and classifier, but the second-highest energy consumer, so memory use alone cannot account for energy efficiency. Additionally, MLP, which was implemented with the Neuroph framework rather than Weka, was very memory efficient (heap) despite being the highest energy user with cross validation. Excluding ZeroR, MLP used and allocated the least amount of heap space, and grew its heap the fewest number of times. However, it performed the third-most GCs, so it may be reducing its memory requirements by performing more frequent memory clean-ups.

Table 9 Spearman's rank correlation ρ value for 10-fold energy use and memory use between Weka-implemented algorithm classifying a dataset.

Dataset	GC Conc.	GC Conc. (ms)	GC Alloc	GC Alloc (ms)	Grow	Used	Alloc-ated
Adult	0.40	0.70	0.90	0.90	0.87	0.70	0.90
MNIST	0.50	0.50	1.00	1.00	1.00	1.00	1.00
Mush	0.75	0.75	0.64	0.64	0.26	0.96	0.96
Pen	0.68	0.68	0.79	0.82	0.71	0.86	0.86
PGSQL	0.71	0.71	0.77	0.83	0.06	0.66	0.66
Spam	0.49	0.49	0.49	0.60	0.60	0.60	0.60
Wave	0.14	0.31	0.60	0.60	0.60	0.60	0.66

The memory use of the Weka-implemented algorithms, not MLP, was compared to energy use, and the Spearman's correlation ρ estimates of this comparison are shown in Table 9. Table 9 shows that energy use is not consistently well-correlated to memory use. Generally energy use was most strongly correlated to the maximum heap space used in a test and the maximum heap space allocated in a test. Spambase and Waveform datasets generally showed weak correlations between their energy and memory use. MLP memory usage was similarly correlated. The relationship between memory and energy usage on mobile devices is likely complex. Our data shows that using large amounts of memory increases energy use, but strategies to decrease memory requirements, such as an aggressive garbage collection policy, may also cause increased strains and delays in the application which also increases energy use, as with Neuroph's MLP implementation. Additionally, developers must be concerned with their application's memory usage beyond its impact on energy consumption, as mobile devices often have limited storage compared to desktop computers or servers.

The lesson learned is that more or less memory use does not guarantee a particular behaviour in terms of energy consumption. Yet using memory in such a way that incurs garbage collection incurs overhead in terms of CPU use. The significance of GC overhead depends on the algorithm and implementation itself.

6.3 Is energy use related to the methods called by an algorithm's implementation?

Method traces for algorithm implementations with different datasets were generated using Android's Dalvik Debug Monitor Server (DDMS) and dmtracedump tools. The method traces were generated by sampling every millisecond. The methods called by each algorithm implementation are compared, and the total number of CPU cycles and total number of method calls made are correlated to energy use.

The total number of method calls is strongly correlated to the energy use of each algorithm implementation on a dataset, with algorithm implementations making more method calls using more energy. All datasets had ρ estimates of 0.9 or better. Similarly, the number of CPU cycles elapsed during execution also had a ρ estimate of 0.9 or better for all datasets when correlated to energy use.

Additionally, algorithm implementations that used more energy, such as MLP or Random Forest, called costly methods many times. For the applicable datasets Random Forest was able to perform cross validation to completion on, the method invoked the most number of times by the algorithm implementation was Weka's QuickSort. QuickSort is already $O(nlogn)$ in average time and $O(n^2)$ worst case. Naïve Bayes and J48 also invoked QuickSort, but significantly fewer times per dataset: Random Forest called QuickSort 9 to 41 times as often as often as J48 did, and 69 to 83 times as often as Naïve Bayes. QuickSort was never used on the Mushroom dataset with any algorithm implementation as it only has categorical attributes. MLP called methods to update weights with backpropagation calculations the most. Logistic regression, another high energy-user, frequently calls methods to evaluate the model's gradient vector and to perform exponentiation.

6.3.1 Time spent within API calls

If we analyze calls that were not made to Neuroph or Weka, potential API calls, we find that `java.lang.System.arraycopy` is quite popular among all test runs, using as much user time as `java.util.ArrayList.add`. `java.util.-ArrayList.get` was also a big consumer of time. Neuroph makes use of their own lists and many of those calls appear as well, but those are not 3rd party APIs. Through profiling we find `java.lang.Daemons$GCDaemon.requestGC` is called much but not directly from the source code. Streams and IO are used by calls such as `java.io.BufferedInputStream.read` but take up 1 order of magnitude less than time than ArrayList `add` calls and 3 orders of magnitude less time than ArrayList `get` calls. `dalvik.system.VMRuntime.concurrentGC` accounted for half as much time as ArrayList `get` calls. `requestGC` calls from the garbage collector used nearly as much time as ArrayList `add` calls.

Other popular Java API calls were: `java.lang.Math.log` for logarithms; `java.lang.Object.wait` which waited on threads (but didn't actually consume CPU as it was waiting on work to be done); `java.lang.Double.isNaN` which tested double values if there were not a number; `java.lang.String.-equals` for string equality; `java.lang.System.identityHashCode` the original Java `hashCode()` or an object; and `java.lang.reflect.Array.newInstance`. Most API calls were to collections, string calls, memory/GC, array copies, threading, mathematical utilities, and some file I/O routines. This makes sense as the machine learning tasks are primarily CPU and memory bound with some need for loading datasets from files. Potential optimizations could be in reducing GC use by memory optimizations such as free lists; could be reducing the need for NaN checks via provable NaN free double manipulation pipelines; optimizing or memoizing the logarithm function; and string interning to avoid expensive String `equals` calls.

Table 10 Spearman correlation ρ estimate between algorithmic complexity with constant factors and energy consumption when training models.

	PGSQL	MNIST	Mush	Adult	Wave	Spam	Pen
50%	0.81	0.96	0.83	0.96	0.90	0.93	0.93
10-CV	0.86	1.00	0.83	1.00	0.89	0.89	0.98

6.4 Is energy use related to algorithmic complexity?

To determine the correlation between algorithmic complexity and energy usage, the relevant statistics of each dataset, including number of attributes, and number of instances, were substituted into the algorithmic time complexity formulas for training each learner. For IBk, which has a constant time complexity of $O(1)$ for training, the cost was set to the constant 100000 for each dataset as its time should be independent of dataset size—this assumes it does not need another representation of the input data as Weka typically claims training took 0 seconds anyway for IBk. For SMO a time complexity of $O(n^2)$ was assumed, which was empirically determined to have a time complexity between $O(n)$ up to $O(n^{2.2})$ for n training instances [38]. The ρ values for the Spearman correlations between these computed numeric complexities and the energy required to train each algorithm implementation on a dataset are shown in Table 10. The curves of these complexity functions were tuned by a single coefficient for best fit.

Table 10 clearly shows that the big-O models of complexity, given the parameters provided, do mostly fit the real-world implementations. Thus in this study the complexity of an algorithm and the energy consumption of its implementation under test are rank correlated.

6.5 Analysis

Hasan *et al.* [32] found that the power use of different collection classes was similar, and that energy consumption seemed to increase at the same rate as program runtimes, indicating that programs that use more energy do so because they do more work in the extra time it takes them to run. Our results agree with this.

While the energy consumptions of different algorithm implementations could differ significantly, the algorithm implementations tended to have similar power use. This is likely because the processes are primarily CPU bound. We found that energy use was positively correlated to both runtime complexity, and the user and idle CPU time taken by an algorithm's implementation. Further, energy use was positively correlated to the number of methods called by an algorithm implementation during execution, indicating that algorithm implementations that use more energy to evaluate a dataset both take longer and call more methods, thus doing more work. Algorithm implementations and datasets that invoked garbage collection more typically took longer and consumed more energy.

7 Evaluating machine learning choices on mobile devices

In this section, we provide guidance to app developers who seek to use machine learning within their mobile-apps. Developers should decide if they need to train machine learners or if they can simply share a trained model with their mobile-app. Developers should also consider the effect that the number of attributes have on energy consumption. Furthermore, developers should consider how much energy consumption they are willing to allow for versus the accuracy or agreement they want to achieve.

7.1 What are the best algorithm implementations to use for models that do not need updating?

Some applications may only require a static, pre-trained model that does not require updating. This classifiers will be trained and shipped with the application when it is downloaded. For example, the Google Translate application uses a convolutional neural net that was trained on a carefully selected dataset, and then deployed in the application [24]. In such a situation, it may be best to select an algorithm implementation that has a high training energy cost since it only needs to be paid once, and not even necessarily on the phone, but has low classifying costs and errors.

J48, SMO, Logistic Regression, and MLP all have significantly higher training costs than classifying and evaluating costs (implementation energy consumption) and in terms of complexity. Thus, these algorithm implementations would be ideal for implementations where the model could be trained ahead of time, and not updated after release for classification in the application. J48, Logistic Regression and MLP are Pareto optimal choices, but SMO is close, based on our limited evaluation, depicted in Figure 9.

7.2 What are the best algorithm implementations to use for models that need updating?

We did not explicitly test algorithms for updating because most of the algorithms tested did not have updateable implementations. Updateable algorithms are referred to as online algorithms or “online-learning” [14]. These algorithms can be run piece by piece, updating themselves. The opposite, an offline algorithm is given all of its inputs at once and produces a result, whereas online algorithms can be continuous. In machine learning an online learner is a learner that can learn piece by piece, one or more examples at a time. Online algorithms are interesting in the mobile context because they can respond to change and can even defer training costs till later. Naïve Bayes is online capable because given a labelled document, token counts associated with the label are easy to increment and save. IBk’s lazy learning is online because one may just add training instances to a collection to be evaluated later. MLPs and neural networks can be online since the network can always be trained or updated with more instances at anytime [14]. Typically MLPs require an optimizer, such as stochastic gradient descent, to search for weights that match the new data, the optimizer can optimize the neural network's weights to label the new examples correctly. For the neural network libraries, iOS's CoreML [36] and Tensorflow-lite as of writing did not enable on-device training for neural networks, but Tensorflow Mobile did [72]—typically these libraries relied on pre-trained networks distributed to the apps themselves. If the model must be trained or re-trained on the phone, Naïve Bayes is the best algorithm to use to limit energy use, as it has the lowest energy use overall and has the same time complexity for training as for classifying [8]. The IBk classifier is trivial to update, making updating fast and low-energy, but it is slow and energy-intensive to classify and it is one of the worst energy consumers for classification. While we did not measure small updates to MLPs, gradient descent and back propagation are expensive so update costs to MLPs can be prohibitive in terms of runtime.

7.3 What are the best algorithm implementations to use to minimize energy consumption?

Excluding ZeroR, Naïve Bayes used the least amount of energy on average for training and testing. J48 was also energy efficient, being the next-lowest energy user on average, after Naïve Bayes. Thus, Naïve Bayes and J48 are the best algorithm implementations tested to use for applications trying to reduce energy use. For 50% split training and testing Naïve Bayes was the lowest energy consumer on average, but was the second-lowest energy consumer for some datasets. For cross-validation, Naïve Bayes was the lowest energy consumer across all datasets. This suggests that Naïve Bayes' energy performance will scale well.

Naïve Bayes is recommended over J48 in terms of energy use if the model must be trained as well as evaluated by the app. If the model can be pre-trained, J48 will likely use less energy and be faster to validate than Naïve Bayes, but Naïve Bayes can train models faster and with less energy than J48.

7.4 What are the best algorithms implementations to use to maximize accuracy?

Of the Weka implemented algorithms, Random Forest and SMO were the best classifiers overall, with Random Forest having the highest average accuracy and SMO having the highest average kappa statistic, making these the best algorithm implementations to use to obtain correct results. Random Forest was also the highest average energy user on 50% split datasets, and the second highest for 10-fold evaluation. SMO was less energy-hungry overall and dominated RF.

MLP had the highest average accuracy overall, with an average classification accuracy of over 95% and an average kappa of over 0.92. On some datasets it was able to achieve RMSEs smaller than 0.0001, suggesting potential overfitting. MLP could likely achieve even higher accuracies if optimized. To standardize the tests, all our MLP networks had the same number of hidden neurons (15), learning rate (0.2), and fixed number of training epochs (100) regardless of input size or type. Tuning these parameters for each dataset could improve prediction accuracies. For example, the Spambase dataset had the highest error, with a classification total mean square error of 0.37 with the test parameters, but using a learning rate of 0.1 and 1000 training epochs, the total mean square error could be reduced to 0.31. However, tuning these parameters could also affect energy consumption of the network.

7.5 What are the best algorithm implementations for datasets with many attributes?

Energy consumption is strongly-correlated to algorithmic time complexity. Thus, it is not surprising that the algorithm implementations with the lowest energy use on datasets with large numbers of attributes (PGSQL, MNIST, Spambase) also have algorithmic complexities that have a low dependence on the number of attributes. SMO had low energy use on the PGSQL and Spambase datasets, especially with 50% split evaluation. Naïve Bayes, which has a linear dependence on the number of attributes, also performs well on these datasets.

7.6 What algorithm implementations dominate in terms of energy versus accuracy?

Figure 9 shows a clear dominating Pareto front of machine learners that are “optimal” in classification/evaluation energy consumption or in accuracy measured in Kappa score. Clear dominators in order of Kappa score versus energy are ZeroR, J48, Logistic Regression and MLP. These candidates make sense because they are effectively conditional statements such as J48, small functions such as logistic regression, or numerous small functions combined such as the neurons of a feed-forward MLP, that are all quick to evaluate. For training, ZeroR, IBk and MLP dominate as IBk’s lazy training beats Naïve Bayes. Ignoring IBk, the training dominators are in order of Kappa are: ZeroR, Naïve Bayes, J48, logistic regression, RF, and MLP.

8 Threats to validity

Construct validity is threatened by our choice of experiments, machine learning algorithms, algorithm implementations, platforms tested, hardware, and data sets. We tried to control for attribution errors by having a constrained

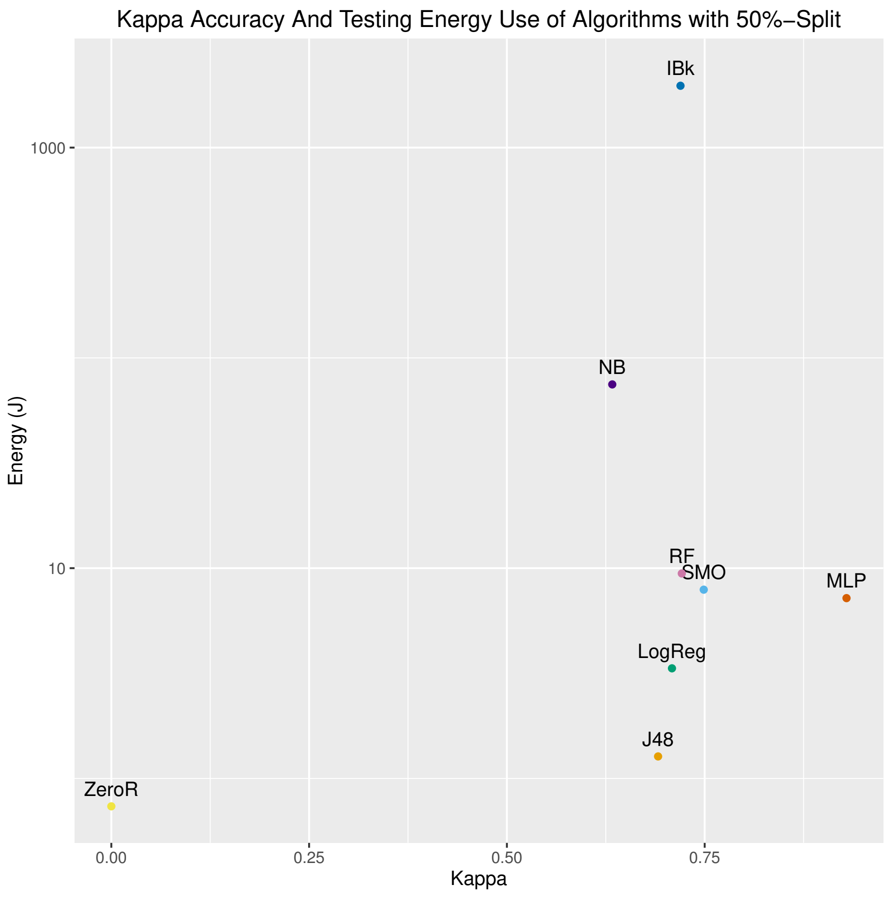


Fig. 9 Scatterplot of energy consumption during classification (not training) versus Kappa.

environment that was very similar for every run. Construct validity is also threatened in terms of the effect on end-users, without applications found in the wild, and benchmarked with different learners under different workloads it isn't clear if any choice in ML classifier will have an impact. Furthermore, construct validity is threatened by the perceived realism of choices made from libraries, datasets, training regime, since freely available observations about data and inputs were not available at the time of writing.

Internal validity is threatened by selection bias of datasets and algorithms, as well the use of two machine learning frameworks. The consistency of the measuring framework could affect internal validity.

External validity is threatened by the limited number of machine learning algorithms, libraries, and implementations evaluated. The limited number of datasets and their variation in terms of size and features is also an external threat—datasets were not sampled, they were manually chosen for their representativeness of actual mobile ML problems. Furthermore, some classifiers such as neural network MLPs and support vector machines such as SMO have numerous parameters that can be tuned—we did not explore parameter tuning thus some performance parameters in terms of both runtime and accuracy could be affected. In the case of SMO different kernels can have drastically different classification performance. Future study could focus on tuning specific classifiers in a multi-objective manner with energy, runtime, and accuracy. We could apply more and furthermore we are limiting ourselves to only two machine learning frameworks. Some frameworks could have better energy efficiency or run-times. External validity is also threatened by the lack of variation or exploration of different devices and different operating systems. Some platforms have hardware specialized for machine learning [10] and thus are not represented. We hope that a lot of the external validity can be addressed with the theoretical run-time estimates provided by complexity estimates but we recognize that different platforms and hardware will react somewhat differently. For instance later versions of Android transitioned from the DalvikVM to the Android Runtime (ART) which has different performance compared to DalvikVM [41]. ART also has different compilation models, such as ahead of time, just in time, and hybrid, which could result in different performance as well.

Thus the main limitations of this study are: the machine learning algorithms and implementations chosen, the realism of the tasks evaluated, the realism of the datasets and inputs used, the proportion of training to evaluation measured, the use of a single kind of device (without machine learning acceleration), the use of a single operating system (DalvikVM versus ART), and the use of only 2 ML libraries.

9 Conclusions

We conclude that developers can make choices, based on their applications, to reduce energy consumption effectively. It is important that developers recognize that machine learning is not a free lunch in terms of energy consumption. Their machine learning choices in terms of algorithm, implementation, OS, and hardware may affect the energy consumption of their applications. The rapid pace of machine learning hardware and software development is creating an ever changing landscape of performance trade-offs complicated by new hardware and new operating systems more capable of mobile code optimization (Android Run Time [41]). The results of our work should be viewed as a snapshot limited to 1 Android OS version (4.2.2) and the DalvikVM runtime, 1 make and model of phone (Galaxy Nexus), and 2 machine learning libraries (Weka and Neuroph) that implement classical machine learning algorithms. We did not engage in deep learning. Currently we would not recommend training Neural Nets (MLP) on mobile devices, however evaluation with neural networks on mobile devices is quite successful [24, 72].

We observed that many machine learning algorithm implementations cost more to train than to evaluate. Often this cost can be offloaded by training in the cloud — which we recommend for logistic regression, support vector machines, and neural networks.

Depending on the context and the need for updates, a lazy trainer, such as nearest neighbours, with expensive evaluation could make more sense than an algorithm implementation with relatively good performance balance between training and evaluation. One needs to balance how much evaluation versus how much training one needs to do. Constant evaluation implies one needs a cheap evaluator whereas constant updates and changing signals implies one need an algorithm and implementation that is cheap to train, such as Naïve Bayes or nearest neighbours.

Thus our results show that given the limited context the dominating algorithm implementations for only classification/evaluation include Neural Nets (ML), Logistic Regression and J48. Support Vector Machines, Random Forest, and Neural Nets (MLP) performed the best in terms of accuracy but with poor energy efficiency for training. Naïve Bayes was balanced and offered good accuracy compared with its training energy efficiency but suffers from high evaluation energy costs. Some algorithm implementations did not fare very well for training such as Logistic Regression, which requires lots of memory and CPU but had middle-ground accuracy and cannot be updated easily. Neural networks have superior accuracy but suffer from poor energy efficiency in terms of both training and evaluation. Thus, mobile app developers need to be aware of the trade-offs between different machine learning algorithms.

Future work would be to integrate smart search techniques to emulate the SEEDS approach [52] of choosing machine learning algorithms and implementations given domain context and constraints. Thus, recommender systems could be built that could analyze the problem and make the best suggestion based upon empirical and theoretical constraints and measurements. Future work can also include accounting for more neural-net architectures, different hardware, different platforms, different OS versions, more library versions, more learners, and more data-sets.

References

1. Dex2jar download - sourceforge.net. `http://sourceforge.net/projects/dex2jar/`. (Last accessed: May 2018)
2. Release v2.1-20171001-lanchon · dexpatcher/dex2jar · github. `https://github.com/DexPatcher/dex2jar/releases/tag/v2.1-20171001-lanchon`. (Last accessed: May 2018)
3. Abdulsalam, H., Skillicorn, D.B., Martin, P.: Classification using streaming random forests. IEEE Transactions on Knowledge and Data Engineering **23**(1), 22–36 (2011)
4. Aggarwal, K., Hindle, A., Stroulia., E.: Greenadvisor: A tool for analyzing the impact of software evolution on energy consumption. In: International Conference on Software Maintenance and Evolution (ICSME 2015), pp. 311–320 (2015). URL `http://softwareprocess.ca/pubs/aggarwal2015ICSME-greenadvisor.pdf`
5. Aggarwal, K., Hindle, A., Stroulia, E.: Greenadvisor: A tool for analyzing the impact of software evolution on energy consumption. In: 31st IEEE International Conference on Software Maintenance and Evolution. IEEE Computer Society (2015)

6. Agolli, T., Pollock, L., Clause, J.: Investigating decreasing energy usage in mobile apps via indistinguishable color changes. In: 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft), pp. 30–34 (2017). DOI 10.1109/MOBILESoft.2017.17
7. Akdeniz: Google Play Crawler. https://github.com/Akdeniz/google-play-crawler (Last accessed: May 2018) (2013)
8. App Annie: App Annie. https://www.appannie.com/ (Last accessed: May 2018)
9. AppBrain: Top Android phones. http://www.appbrain.com/stats/top-android-phones. (Last accessed: May 2018)
10. Apple Inc.: The future is here: iphone (2017). https://www.apple.com/newsroom/2017/09/the-future-is-here-iphone-x/ (Retrieved April 2018)
11. Banerjee, A., Chong, L.K., Chattopadhyay, S., Roychoudhury, A.: Detecting energy bugs and hotspots in mobile apps. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 588–598. ACM (2014)
12. Benfield, L.: Cfr - another java decompiler. http://www.benf.org/other/cfr/. (Last accessed: May 2018)
13. Bhattacharya, S., Lane, N.D.: Sparsification and separation of deep learning layers for constrained resource inference on wearables. In: Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM, SenSys '16, pp. 176–189. ACM, New York, NY, USA (2016). DOI 10.1145/2994551.2994564. URL http://doi.acm.org/10.1145/2994551.2994564
14. Bottou, L.: Online Algorithms and Stochastic Approximations. Cambridge University Press (1998)
15. Breiman, L.: Random forests. Machine Learning **45**(1), 5–32 (2001)
16. Bruce, B.R., Petke, J., Harman, M.: Reducing energy consumption using genetic improvement. In: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation, GECCO '15, pp. 1327–1334. ACM, New York, NY, USA (2015). DOI 10.1145/2739480.2754752. URL http://doi.acm.org/10.1145/2739480.2754752
17. Chenlei, Z., Hindle, A., , German, D.M.: The impact of user choice on energy consumption. IEEE Software pp. 69–75 (2014). URL http://softwareprocess.ca/pubs/zhang2014IEEESoftware-user-choice.pdf
18. Chowdhury, S., Sapra, V., Hindle, A.: Client-side energy efficiency of http/2 for web and mobile app developers. In: 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016), pp. 529–540 (2016). DOI 10.1109/SANER.2016.77. URL http://softwareprocess.ca/pubs/chowdhury2016SANER-http2.pdf
19. Chowdhury, S.A., Hindle, A.: Greenoracle: Estimating software energy consumption with energy measurement corpora. In: Proceedings of the 13th International Conference on Mining Software Repositories, MSR '16, pp. 49–60. ACM, New York, NY, USA (2016). DOI 10.1145/2901739.2901763. URL http://doi.acm.org/10.1145/2901739.2901763
20. Christina Bonnington: Your smartphone gains a mind of its own. Conde Nast http://www.wired.com/2013/07/ai-apps-trend/ (2013)
21. D. Aha and D. Kibler: Instance-based learning algorithms. Machine Learning **6**, 37–66 (1991)
22. Di Nucci, D., Palomba, F., Prota, A., Panichella, A., Zaidman, A., De Lucia, A.: Software-based energy profiling of android apps: Simple, efficient and reliable? In: Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on, pp. 103–114. IEEE (2017)
23. E. Frank: Class j48. http://weka.sourceforge.net/doc.dev/weka/classifiers/trees/J48.html (2016)
24. Good, O.: How google translate squeezes deep learning onto a phone. Google Research Blog https://research.googleblog.com/2015/07/how-google-translate-squeezes-deep.html (2015)
25. Google: Find time for your goals with google calendar. Google Blog https://googleblog.blogspot.ca/2016/04/find-time-goals-google-calendar.html (2016)
26. Google: Neural networks api: Android developers (2017). https://developer.android.com/ndk/guides/neuralnetworks/index.html (Last accessed: May 2018)

27. Google: Mobile vision (2018). https://developers.google.com/vision/ (Last accessed: May 2018)
28. Greene, T.: Google brings on-device machine learning to mobile with tensorflow lite (2017). https://thenextweb.com/artificial-intelligence/2017/11/15/google-brings-on-device-machine-learning-to-mobile-with-tensorflow-lite/ (Retrieved January 2018)
29. Gui, J., Li, D., Wan, M., Halfond, W.G.: Lightweight measurement and estimation of mobile ad energy consumption. In: Green and Sustainable Software (GREENS), 2016 IEEE/ACM 5th International Workshop on, pp. 1–7. IEEE (2016)
30. Gui, J., Mcilroy, S., Nagappan, M., Halfond, W.G.J.: Truth in advertising: The hidden cost of mobile ads for software developers. In: 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1, pp. 100–110. IEEE (2015). DOI 10.1109/ICSE.2015.32. URL http://dx.doi.org/10.1109/ICSE.2015.32
31. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The weka data mining software: An update. SIGKDD Explor. Newsl. **11**(1), 10–18 (2009). DOI 10.1145/1656274.1656278. URL http://doi.acm.org/10.1145/1656274.1656278
32. Hasan, S., King, Z., Hafiz, M., Sayagh, M., Adams, B., Hindle, A.: Energy profiles of java collections classes. In: International Conference on Software Engineering (ICSE 2016), pp. 225–236 (2016). DOI 10.1145/2884781.2884869. URL http://softwareprocess.ca/pubs/hasan2016ICSE-Energy-Profiles-of-Java-Collections-Classes.pdf
33. Hern, Alex, a.: Smartphone now most popular way to browse internet – ofcom report. https://www.theguardian.com/technology/2015/aug/06/smartphones-most-popular-way-to-browse-internet-ofcom/ (2015). (last accessed: May 2018)
34. Hindle, A., Ernst, N.A., Godfrey, M.W., Mylopoulos, J.: Automated topic naming supporting cross-project analysis of software maintenance activities. Journal of Empirical Software Engineering **18(6)**, 1125–1155 (2013). URL http://softwareprocess.ca/pubs/hindle2011EMSE-automated-topic-naming.pdf
35. Hindle, A., Wilson, A., Rasmussen, K., Barlow, E.J., Campbell, J., Romansky, S.: Greenminer: a hardware based mining software repositories software energy consumption framework. In: International Working Conference on Mining Software Repositories (MSR 2014), pp. 12–21 (2014). URL http://softwareprocess.ca/pubs/hindle2014MSR-greenminer.pdf
36. Inc., A.: Core ml: Apple developer documentation (2017). https://developer.apple.com/documentation/coreml (Last accessed: May 2018)
37. Inc., W.: Wit.ai: Natural language for developers (2018). https://wit.ai/ (Last accessed: May 2018)
38. J. Platt: Fast training of support vector machines using sequential minimal optimization. In: B. Schoelkopf, C. Burges, A. Smola (eds.) Advances in Kernel Methods - Support Vector Learning. MIT Press (1998). URL http://research.microsoft.com/~jplatt/smo.html
39. Jabbarvand, R., Sadeghi, A., Garcia, J., Malek, S., Ammann, P.: Ecodroid: an approach for energy-based ranking of android apps. In: Proceedings of the Fourth International Workshop on Green and Sustainable Software, pp. 8–14. IEEE Press (2015)
40. John, G.H., Langley, P.: Estimating continuous distributions in bayesian classifiers. In: Eleventh Conference on Uncertainty in Artificial Intelligence, pp. 338–345. Morgan Kaufmann (1995)
41. Konradsson, T.: Art and dalvik performance compared. Master's thesis, UmeåUniversity (2015). http://www8.cs.umu.se/education/examina/Rapporter/TobiasKonradsson.pdf
42. LeCun, Y., Cortes, C., Burges, C.J.: The mnist database of handwritten digits. http://yann.lecun.com/exdb/mnist/ (1998)
43. Li, D., Hao, S., Gui, J., Halfond, W.G.J.: An empirical study of the energy consumption of android applications. In: 30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014, pp. 121–130. IEEE Computer Society (2014). DOI 10.1109/ICSME.2014.34. URL http://dx.doi.org/10.1109/ICSME.2014.34

44. Li, D., Lyu, Y., Gui, J., Halfond, W.G.J.: Automated energy optimization of http requests for mobile applications. In: Proceedings of the 38th International Conference on Software Engineering, ICSE '16, pp. 249–260. ACM, New York, NY, USA (2016). DOI 10.1145/2884781.2884867. URL `http://doi.acm.org/10.1145/2884781.2884867`
45. Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Oliveto, R., Di Penta, M., Poshyvanyk, D.: Mining energy-greedy api usage patterns in android apps: An empirical study. In: Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014, pp. 2–11. ACM, New York, NY, USA (2014). DOI 10.1145/2597073.2597085. URL `http://doi.acm.org/10.1145/2597073.2597085`
46. Linares-Vásquez, M., Bavota, G., Cárdenas, C.E.B., Oliveto, R., Di Penta, M., Poshyvanyk, D.: Optimizing energy consumption of guis in android apps: A multi-objective approach. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, pp. 143–154. ACM, New York, NY, USA (2015). DOI 10.1145/2786805.2786847. URL `http://doi.acm.org/10.1145/2786805.2786847`
47. Linares-Vásquez, M., Vendome, C., Tufano, M., Poshyvanyk, D.: How developers micro-optimize android apps. Journal of Systems and Software **130**, 1–23 (2017)
48. "M. Lichman": "UCI machine learning repository" ("2013"). URL `"http://archive.ics.uci.edu/ml"`
49. Machine Learning Laboratory: Mnist arff files. http://axon.cs.byu.edu/data/mnist/ (2015)
50. Malik, H., Zhao, P., Godfrey, M.: Going green: An exploratory analysis of energy-related questions. In: Proceedings of the 12th Working Conference on Mining Software Repositories, MSR '15, pp. 418–421. IEEE Press, Piscataway, NJ, USA (2015). URL `http://dl.acm.org/citation.cfm?id=2820518.2820576`
51. Manotas, I., Bird, C., Zhang, R., Shepherd, D., Jaspan, C., Sadowski, C., Pollock, L., Clause, J.: An empirical study of practitioners' perspectives on green software engineering. In: Proceedings of the 38th International Conference on Software Engineering, ICSE '16, pp. 237–248. ACM, New York, NY, USA (2016). DOI 10.1145/2884781.2884810. URL `http://doi.acm.org.login.ezproxy.library.ualberta.ca/10.1145/2884781.2884810`
52. Manotas, I., Pollock, L., Clause, J.: Seeds: A software engineer's energy-optimization decision support framework. In: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, pp. 503–514. ACM, New York, NY, USA (2014). DOI 10.1145/2568225.2568297. URL `http://doi.acm.org/10.1145/2568225.2568297`
53. Matyunina, J.: How do i apply machine learning in an android app? (2017). `https://www.quora.com/How-do-I-apply-machine-learning-in-an-android-app` (Last accessed: May 2018)
54. Minka, Thomas P: A comparison of numerical optimizers for logistic regression. Unpublished paper available at http://research.microsoft.com/en-us/um/people/minka/papers/logreg/minka-logreg.pdf (2007)
55. Mizutani, Eiji and Dreyfus, Stuart E: On complexity analysis of supervised mlp-learning for algorithmic comparisons. In: Neural Networks, vol. 1, pp. 347–352. IEEE (2001)
56. OpenCV Team: Android - opencv library. `https://opencv.org/platforms/android/` (Last accessed: May 2018)
57. Padraig Cunningham and Sarah Jane Delaney: k-nearest neighbour classifiers. Tech. Rep. UCD-CSI-2007-4, University College Dublin (2007). URL `https://csiweb.ucd.ie/files/UCD-CSI-2007-4.pdf`
58. Pang, C., Hindle, A., Adams, B., Hassan, A.E.: What do programmers know about the energy consumption of software? IEEE Software pp. 83–89 (2015). URL `http://softwareprocess.ca/pubs/pang2015IEEESoftware.pdf`
59. Pathak, A., Hu, Y.C., Zhang, M.: Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices. In: Proceedings of the 10th ACM Workshop on Hot Topics in Networks, HotNets-X, pp. 5:1–5:6 (2011)
60. Pathak, A., Hu, Y.C., Zhang, M., Bahl, P., Wang, Y.M.: Fine-grained Power Modeling for Smartphones Using System Call Tracing. In: EuroSys '11, pp. 153–168. Salzburg, Austria (2011). DOI 10.1145/1966445.1966460. URL `http://doi.acm.org/10.1145/1966445.1966460`

61. Pereira, R., Couto, M., Saraiva, J.a., Cunha, J., Fernandes, J.a.P.: The influence of the java collection framework on overall energy consumption. In: Proceedings of the 5th International Workshop on Green and Sustainable Software, GREENS '16, pp. 15–21 (2016)
62. Pinto, G., Castor, F., Liu, Y.D.: Mining Questions About Software Energy Consumption. In: MSR 2014, pp. 22–31 (2014). DOI 10.1145/2597073.2597110. URL `http://doi.acm.org/10.1145/2597073.2597110`
63. Rasmussen, K., Wilson, A., , Hindle, A.: Green mining: energy consumption of advertisement blocking methods. In: Proceedings of the 3rd International Workshop on Green and Sustainable Software (GREENS 2014), pp. 38–45 (2014). URL `http://softwareprocess.ca/pubs/rasmussen2014GREENS-adblock.pdf`
64. Saborido, R., Beltrame, G., Khomh, F., Alba, E., Antoniol, G.: Optimizing user experience in choosing android applications. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol. 1, pp. 438–448 (2016). DOI 10.1109/SANER.2016.64
65. Sahin, C., Pollock, L., Clause, J.: From benchmarks to real apps: Exploring the energy impacts of performance-directed changes. Journal of Systems and Software **117**, 307–316 (2016)
66. Sahin, C., Tornquist, P., Mckenna, R., Pearson, Z., Clause, J.: How does code obfuscation impact energy usage? In: ICSME, pp. 131–140. IEEE Computer Society (2014)
67. Sahin, C., Wan, M., Tornquist, P., McKenna, R., Pearson, Z., Halfond, W.G., Clause, J.: How does code obfuscation impact energy usage? Journal of Software: Evolution and Process **28**(7), 565–588 (2016)
68. Selby, J.W.A.: Unconventional applications of compiler analysis. Ph.D. thesis, University of Waterloo (2011)
69. Sevarac, Z., Goloskokovic, I., Tait, J., Carter-Greaves, L., Morgan, A., Steinhauer, V.: Neuroph: Java neural network framework. `http://neuroph.sourceforge.net/` (2016)
70. S.S. Keerthi and S.K. Shevade and C. Bhattacharyya and K.R.K. Murthy: Improvements to platt's smo algorithm for svm classifier design. Neural Computation **13**(3), 637–649 (2001)
71. Su, Jiang and Zhang, Harry: A fast decision tree learning algorithm. In: American Association for Artificial Intelligence, vol. 6, pp. 500–505 (2006)
72. TensorFlow: Mobile tensorflow. `https://www.tensorflow.org/mobile.html` (2016)
73. Tomita, T.M., Maggioni, M., Vogelstein, J.T.: Randomer forests. arXiv preprint arXiv:1506.03410 (2015)
74. Triposo: Triposo. `https://www.triposo.com/` (2016)
75. Wan, M., Jin, Y., Li, D., Gui, J., Mahajan, S., Halfond, W.G.: Detecting display energy hotspots in android apps. Software Testing, Verification and Reliability **27**(6) (2017)
76. Webservices, A.: Amazon aws machine learning (2018). `https://aws.amazon.com/machine-learning/` (Last accessed: May 2018)
77. Weotta: About weotta. `http://www.weotta.com/about` (2016)
78. Witten, I.H., Frank, E.: Data Mining: Practical machine learning tools and techniques, 3 edn. Morgan Kaufmann (2011)
79. Woollaston, V.: Customers really want better battery life. http://www.dailymail.co.uk/sciencetech/article-2715860/Mobile-phone-customers-really-want-better-battery-life-waterproof-screens-poll-reveals.html (2015). (last accessed: May 2018)
80. Yang, Y., Zhang, J., Kisiel, B.: A scalability analysis of classifiers in text categorization. In: Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval, pp. 96–103. ACM (2003)