Estimating Fine-Grained Mobile Application Energy Use based on Run-Time Software Measured Features

by

Stephen Romansky

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

Master of Science

Department of Computing Science
University of Alberta

Abstract

Inefficient mobile software kills battery life. Yet, developers lack the tools necessary to detect and solve energy bugs in software. In addition, developers are usually tasked with the creation of software features and triaging existing bugs. This means that most developers do not have the time or resources to research, build, or employ energy debugging tools.

We present a new method for predicting software energy consumption to help debug software energy issues. Our approach enables developers to align traces of software behavior with traces of software energy consumption. This allows developers to match run-time energy hot spots to the corresponding execution. We accomplish this by applying state-of-the-art neural network models to predict the time series of energy consumption given a software's behavior. We compare our time series models to prior state-of-the-art models that only predict total software energy consumption. We found that machine learning based time series models, and LSTM based time series models, can often be more accurate at predicting instantaneous power use and total energy consumption. We also show that the machine learning time series models perform best in terms of learning the shape of the application power usage compared to the other investigated models. This means the time series models are better at modelling the hot spots in online energy consumption than the other tested models.

Acknowledgements

I would like to write and thank everyone I have got to interact with at the University of Alberta. Thank you to Dr. Hindle for supervising my studies, to peers and colleagues for providing discussion, and to my parents for support.

For funding thank you to: Nvidia for support through the NVIDIA TITAN X GPU grant that was used in this research; thank you University of Alberta for funding through the QE II award; thank you Alberta for the Graduate Student Scholarship; and thank you Engage and MITACs collaborators.

Collaboration

Thank you: Neil Borle; Shaiful Chowdhury; Dr. Abram Hindle; and Dr. Russ Greiner for collaboration and supervision in this work.

Thank you: Eddie Antonio Santos; Gregory Burlet; Hazel Campbell; Shaiful Chowdhury; and Dr. Abram Hindle for help editing and reviewing associated documentation.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Developers are relatively unaware of techniques and tools used to address software energy issues [53]. Energy bugs deplete mobile batteries and cost developers time to debug. Battery life is an important resource for end users because it dictates how long their applications can be used. To meet the end-users' demand for energy efficient applications, developers are more energy aware than ever, and ask more questions about their software's energy efficiency [53]. While answers to their energy related questions help developers to some extent, in order to build energy efficient applications, developers need actual feedback on their applications' energy consumption. Researchers have built tools for developers to model the energy consumption of mobile software and hardware components [61, 7]. However, these tools are not as easy to use as widely available performance metrics like: CPU load, network interface usage, or disk usage. We are interested in building tools that help developers to understand when and where their applications consume the most energy. Such tools would enable the software development processes to better track and maintain energy goals by finding hotspots and energy bugs [11, 42].

How would being able to see a timeseries of energy consumption from running software help developers? Previous energy estimation tools [7, 32, 44] offer a summary of the total energy consumption for a given run of the application. Although this gives an idea about an app's energy consumption, it does not give much about the location of the energy bugs and hotspots [11, 42]—which segment of source code is responsible for the most energy drain. Therefore, we hypothesize that developers provided with tools to align time series of software energy consumption to program behaviour can more easily debug energy bugs. But, this work takes the first few steps towards answering our hypothesis: we reproduce current state of the art total energy consumption prediction models; we construct models that can predict the time series of energy consumption; we compare the total and time series models against the observed software energy consumption to better understand how our models compare with the ground truth; and we compare the two types of total and time series models to determine if time series models have higher accuracy than the current total energy prediction models. Future work can evaluate the usage of our software-based energy prediction models in helping developers find energy bugs in their software, as we hypothesize.

Researchers have worked towards online energy estimation tools (also referred to as time series energy estimation tools) [43, 19]. However, current instantaneous energy prediction tools assume that developers have access to battery API information or that developers have hardware access to power measurement tools. In contrast, our work makes no hardware assumptions and provides developers with hardware-free energy usage predictions as if the developers had access to energy meters. As well, when refering to the online estimation tools in this work we mean the model that takes already-measured data points and outputs energy readings. Our work describes how to collect the data points from Android applications to use with these online estimation models.

In this work, we propose a series of models that use software performance measurements to predict instantaneous energy consumption. We can align performance measurements with our energy measurements to train neural networks and other machine learning models to predict the instantaneous energy consumption of a software application.

We have to assess the quality of our proposed energy consumption prediction method compared to existing solutions [16]. Our method uses instantaneous time measurements throughout a software test run whereas existing state-of-the-art models measure a summary of software features at the beginning and end of a software test run. We call the existing models *time-summarized*, or *1step* for short, as the software run is measured or aggregated in a single time step instead of many instantaneous time steps or windows.

In other words, the time-summarized models ignore the time information associated with software features used to predict the total energy consumption of a software test. Whereas, a time series model accounts for the many instantaneous energy measurements that would occur throughout a software test, from an energy measurement device, and creates windows/groups of software data based on the hardware sampling rate of the energy measurement device. This allows the time series model to take the time windows of software data and produce an energy consumption prediction for these observed software features.

To assess our models we raise the following questions:

- (RQ1) How well do time series models perform when predicting total energy consumption? Can time series models beat state-of-the-art?;
- (RQ2) Do time series models with built-in state or memory perform better than time series models without state when predicting energy consumption as a total or a time series?;
- (RQ3) Energy prediction models can be trained with different feature sets; how does performance change when using them individually versus together? What features affect the energy consumption models predictive power?;
- (RQ4) Do shallow multi-layer-perceptrons perform similarly to existing stateless models such as those based on linear regression?;
- (RQ5) How do the time series model predictions fit the shape of the observed energy consumption data?

The models in this work were created at a similar time to Nucci's [19] and used a different software feature measurement technique. Therefore, this thesis does not contain a comparison of models based on Green Miner software measurements and Nucci's.

1.1 Thesis Overview

In Chapter 3 a review of recent works is provided. Chapter 2 provides information on the features used in our work, as well as an introduction to the machine learning models and background that is needed to understand our methodology and evaluation. Chapter 4 details the methodology for collecting data and training models. Chapter 5 details the evaluation of the machine learning models trained in the previous chapter. Chapter 6 discusses the results and research questions. Chapter 7 concludes this theses.

Chapter 2

Background

Our work applies multiple machine learning models. Therefore, we provide background information on each of the models applied, what cost functions are, what features and labels we train the models with, and our process for training models. We cover the following models: linear and ridge regressors; support vector regressors; neural networks; multi-layer perceptrons; recurrent layer networks; and long-short term memory layer networks. Each model has a brief description of what makes them unique and a high level summary of how the models work. More detailed machine learning information can be supplemented with text books such as on how the training algorithms work for each model [26, 25]. The variety of models are mostly chosen for exploration while several like the RNN and LSTM have been picked for their performance on time series prediction tasks. When picking models, we picked evaluation functions that would allow everything to become comparable for assessing which model performed best on our learning task. We use the *Keras* and *scikit-learn* python modules throughout this work to implement the various machine learning models [13, 51].

2.1 Cost functions

To assess the quality of a model, a cost function is applied. A cost function provides a score for how well the model performed on its prediction task. There are multiple types of cost functions, which provide their own contexts for ranking models. For instance, a mean squared error (MSE) cost function will work well when it is important to detect negative errors. The result of the MSE function is more prone to becoming skewed when a large outlying error is present in the analyzed measurements. Whereas, the similar, yet different, mean absolute error (MAE) cost function will work well when negative outliers are not as important and we are interested in viewing the average error per prediction. The MAE can still become skewed to outlying errors but will have a smaller influence per outlier on the aggregated result.

It is also possible to assess how well prediction models fit the trends and shape of observed data by using a technique known as dynamic time warping (DTW). DTW is useful in addition to MSE and MAE because it can compare time series to assesses how well a predict time series fits a set of observations. There are several works that detail the DTW algorithm and improvements to it [38]. We describe DTW and its implementation in Section 4.1.2.3.

2.2 Linear and Ridge Regression

Linear and ridge regressors consist of a set of inputs, a set of weights corresponding to the input, and a constant value. Typically expressed as,

$$f(x_1, x_2, ..., x_n) = w_0 + w_1x_1 + w_2x_2 + ... + w_nx_n = y$$

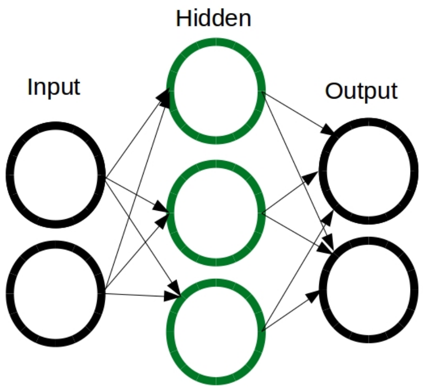
for some set of features x_i, some set of trained model weights w_i, and some label y – where i is an integer denoting an index of the total number of features the model has trained on.

Both models can be trained using similar cost functions such as the least squares method. However, ridge regressors have an additional constraint compared to linear regressors. Ridge regressor weights are limited by a chosen coefficient t such that $(\sum_{i=1}^{n} w_i^2) < t$ [25], which benefits the model by limiting the amount that the weights can overfit training data.

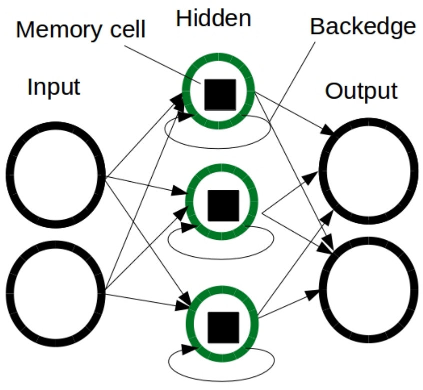
Linear and ridge regressors are simple models and are typically inexpensive to train as a result. This makes linear and ridge regression models convenient to test, because these can yield good results on datasets with linear relationships. The linear models are also explainable compared to those like deep networks due to reduced complexity.

2.3 Support Vector Regression

There are 3 types of SVR model kernels we include in our work: linear; radial basis; and polynomial function. The goal of each model is to create a line of best fit through the dataset. The best



(a) Feed forward neural network



(b) Recurrent neural network with memory cells

Figure 2.1: Feed forward and recurrent neural networks

fit is determined by minimizing a quadratic programing problem to achieve the best score on the ε-insensitive function:

$$V_\varepsilon(x) = \begin{cases} 0, & \text{if } |x| < \varepsilon \\ |x| - \varepsilon, & \text{otherwise} \end{cases}$$

with the observed and predicted values of the SVR [25].

SVRs also support parameters to configure the kernels to better fit the data. However, it is unknown which kernel configuration works best on a given dataset. Therefore, we select a subset of kernel parameters as shown in Table 4.4 and perform a parameter sweep to find the best. Our implementation of a hyper-parameter selection procedure for SVRs is discussed in Section 4.1.2.2. A similar hyper parameter selection process is discussed in Chapter 5 of Goodfellow *et al.* [26].

We learned about SVRs in CMPUT 551 and they appeared to be widely successful models so we applied them in our experiments. As well, SVRs also have reasonable explainability for their produced models, compared to linear and ridge regression models, which is beneficial for understanding the dataset if the models are found to perform well.

2.4 Neural Networks

Neural networks are currently quite successful models in machine learning capable of modelling linear and non-linear relationships. Successes have been demonstrated in fields such as speech recognition [31, 9], time series prediction [56], and even software engineering [62, 54, 63].

Deep neural networks are neural networks with multiple layers in contrast to shallow networks which have few layers. Deep learning is the process of training a model structure to represent a hierarchy of abstract relationships from a given data set [41]. It avoids the need for manual feature selection because the training algorithms for deep networks perform feature selection. Deep learning is also capable of exploiting non-linear relationships in the input data.

Common types of neural networks like feed forward neural networks (FNN) and recurrent neural networks (RNN) have nodes with specific edge configurations. FNNs are less complicated than RNNs, because FNN nodes and edges do not contain loops. Both FNN and RNN networks have 1 or more layers, each with 1 or more nodes. In a FNN, the edge are restricted such that their node outputs can only go to lower layers. Whereas, RNNs allow edges from node outputs to be connected backwards or forwards. Figure 2.1a shows an example of a multi-layer perceptron (MLP) which is a type of FNN. The direction of the edges in the MLP is forward from left to right in contrast to the RNN in Figure 2.1b which has edges going in both directions.

RNNs are capable of learning more relationships from input data than FNNs due to their back-edges. Back-edges refer to the looping edges of the network as shown in Figure 2.1b, because they go back from a layers output to somewhere higher in the network. Yet, RNNs have poor performance when used with time series of length 200 or more time steps [26]. This is due to the network nodes becoming saturated which leads to very large or very small parameters (edge weights and biases). However, this can be managed by adding memory cells to the RNN nodes. Popular networks that use memory cells are the long-short term memory layers (LSTM). LSTMs avoid the part of the saturation problem faced by RNNs. This makes LSTMs better at modelling the relationships that exist in sequential input data than RNNs and FNNs. Both RNN and LSTM networks have achieved strong results on tasks that require modelling sequences of inputs which motivates our exploratory usage of the models in this work [41, 58]. In this thesis we present the results of LSTM models because preliminary experimentation with the RNN models didn't yield meaningful models with respect to the amount of time required in training the RNN models.

Neural networks use activation functions in each neuron to transform the output of the network and to update the weights at each neuron. In our work, we apply: linear, sigmoid, hard sigmoid, and tanh activation functions. We use the linear activation function to model Gaussian distributions, and we apply the tanh and hard sigmoid activation functions in our LSTMs to model the sequence-based relationships of our input measurements [13]. We use the cross-entropy cost function with our networks as it has been found to yield good results in the training process due to training becoming a convex optimization problem [26].

2.5 System Calls and The Process File System

System calls are generated by Linux applications when requesting system resources [5]. For example, the *write* system call count explains how many times the processes has written to a file descriptor. Prior work has shown that energy models can be constructed from using system calls as input features [49, 16, 6, 20]. We collect *syscalls* with *strace*, a linux debugging utility for collecting system calls and signals [3]. We use the following system calls in our models: rename; lseek; pipe; bind; epoll_ctl; rt_sigtimedwait; mprotect; recvmsg; chmod; kill; close; prctl; pread; flock; futex; open; stat64; mmap2; unlink; sigprocmask; epoll_create; getpriority; nanosleep; mkdir; writev; write; set_tls; fchmod; exit; getpid; dup; _llseek; fdatasync; setpriority; epoll_wait; getdents64; fchown32; restart_syscall; getuid32; setsockopt; read; umask; clone; sched_yield; statfs64; getgid32; gettimeofday; munmap; mremap; socket; fcntl64; lstat64; clock_nanosleep; cacheflush; getegid32; gettid; pwrite; fsync; madvise; geteuid32; fstat64; access; ioctl; socketpair; recvfrom; sendto; clock_gettime. These are the system calls that we recorded from our *GreenMiner* based tests.

The process file system (*procfs*) provided by the Linux kernel provides process-specific run-time information [4]. For example, the process *utime* feature records how long a process has been scheduled in user land in clock ticks. The *procfs* provides additional resource-usage features associated with a running application that we can use to predict the energy consumption of a given application. It was also shown in prior work that using both system calls and process utilization features can create accurate models for predicting the total energy consumption of a software application test [20, 16]. We use the following procfs based features: pid_stat_kernel_mode_time; pid_statm_size; pid_stat_vsize; pid_stat_rss; stat_user; pid_stat_minor_faults; stat_iowait; pid_stat_num_threads; pid_stat_state; stat_forks_since_boot; stat_context_switches; pid_statm_resident; stat_system; date; pid_statm_data; pid_stat_user_mode_time; pid_statm_share; stat_idle; stat_total_interrupts.

2.6 Energy

Energy (E) is the capacity to do work measured in *joules*. It is represented by power (P), which is the rate work is done, multiplied by time (T): $E = P \cdot T$. Power is measured in *watts*. In our case, energy is consumed by the smart phones to execute software applications.

2.7 Energy measurement

We make use of the *GreenMiner* testbed: a set of 4 instrumented Android *4.2.2 Galaxy Nexus* devices that replay software test runs while recording device energy usage [35]. We apply the *GreenMiner* to mine information from multiple Android applications versions to build a model-training corpus. To pick an application for energy profile mining on the *GreenMiner* a researcher needs to check if the application and each of its versions run on the devices. As well, the researcher needs to write software tests which simulate how an end-user would interact with the application. This means a software test needs to cover important functionality from the application under test. It is possible to see how the energy consumption of tested functions change over time, because the software tests are repeated across multiple revisions of the same software application. The *GreenMiner* provides energy samples for each of the 4 mobile devices while tests are running at a rate of 50 Hz. The energy samples are provided continuously from each running mobile device. In our experiments we configure the *GreenMiner* to provide continous streams of software features in addition to the provided energy measurements. We wrote a program to associate the software features with the corresponding energy samples given the time the feature and energy samples occurred. The *GreenMiner* mobile devices do not use mobile batteries and instead use power supplies to maintain consistent energy sources for the running software tests; because, batteries degrade over time. The original *GreenMiner* work showed that running repeated tests of the same software can reduce error in the captured measurements from the non-deterministic environment of a linux-based mobile device [35]; e.g. the state of the operating system and hardware devices might be different between two given software tests running on the mobile phone causing the captured test results to differ between the two runs. The multiple test runs can aid in preventing overfitting between the captured results from adding multiple test runs.

2.8 Glossary

Here we introduce several terms that we use throughout the paper:

- a *time series* refers to a sequence of measurements or predictions grouped into time steps by time of measurement/prediction.
- We refer to models that have been constructed to consume *time series* and output *time series* predictions as *time series models.*
- If a model is unable to handle the multiple measurements across time, then we remove time by aggregating via summation every measurement together. We refer to the *time-summarized* data as a single time step or *1step.* Because, the time series only contains information aggregated into a single time step which covers from the beginning of a software test run to the end of its duration. Further into the thesis, we discuss which segments of collected software observations are used to train and test these *1step* models.
- We refer to software tests, where we observe and collect our data, as *test runs.*

Chapter 3

Literature Review

A summary of related work is provided in this chapter. In addition, this chapter also contains detailed summaries of the individual related works. This chapter aims to provide a record of the context in which our work and contribution exist.

Energy-aware software engineering poses many challenges in the development of energy-efficient software [45, 46, 44]. Researchers have investigated three separate approaches for profiling energy on given mobile devices: software, hardware, or hybrid models [7]. Software based energy profilers use recordable features like API calls, system calls, and resource utilization to model the energy consumption of a mobile device [7, 20, 20, 32]. Whereas, hardware based energy profilers can physically measure the energy consumption of specific hardware components [7, 55]. Hybrid approaches use both software and hardware based features to predict energy consumption [7].

Furthermore, a software based profiler might rely on an API to the battery to measure energy consumption on the device during model construction [7, 20]. The measurement of the energy consumption through an API is distinct from a hardware based profiler, which can sample the current used by the phone with an external measurement tool [7]. Here are two examples of energy profilers: *se-same* [20] is a smart-battery interface based application energy consumption predictor; therefore, it is a software based profiler [7]; and *Netw-trace* [55] which uses an external device to measure the energy used by the phone, and software to record the network traffic on the device's WiFi and radio devices [7]. *Netw-trace* is a hybrid profiler, because it uses software and hardware reading to predict energy consumption. Work by Banerjee et al. [11] and the *GreenMiner* [35] are examples of tools that measure both energy with hardware and software features to study software execution and energy consumption.

Researchers have also investigated energy draining hardware and software components that software applications use. For instance, researchers have developed energy-efficiency targeted operating systems like *ErdOS* [60] and *CondOS* [18, 61]. *ErdOS* [60] predicts how a user interacts with hardware components to schedule more sleep behavior. Whereas, *CondOS* provides more energy efficient APIs for sensing resources like GPS so that developers do not need to develop their own energy efficient heuristics. More energy efficient networking protocols for battery limited mobile devices have also been studied since WiFi and radio components can consume large amounts of device energy [61, 17] Furthermore, researchers have also investigated the process of sharing mobile computation with the cloud [61, 60]. This enables a phone to place energy-expensive computation on another computer to prevent the battery from being drained by a costly operation.

Work has been accomplished on identifying software system calls as a method to profile the energy consumption of software [6]. Tools that predict the total energy consumption of software applications exist which are based on system calls and additional software and hardware features [16, 15]. However, these tools often do not provide details of the software behavior that influences energy consumption. *GreenOracle* is a regression tool which can predict the total energy consumption of a software application test [16]. It uses system call and resource utilization metrics to predict the energy consumption of given software applications under test. In this paper we employ *GreenOracle* features and models to produce time series predicting energy models.

Prior work has created online energy profilers using hardware APIs and system calls to perform online energy prediction [19] similar to our goal. Furthermore, online energy models have been generated using hardware component models and a batteries API, or an external meter, for power measurement as in PowerTutor [64]. Other prior work has also assumed there is access to an energy measurement platform when predicting how much energy is consumed by software [43]. However, our work does not assume that the developers using our models have access to a battery API or that developers have access to energy measurement hardware. Therefore, developers can use our prediction models to generate the energy consumption information of their applications under test as if they had access to an energy meter of their own.

It is also notable that mobile lithium-ion battery hardware degrades over time and from charge cycles [1]. As a result of battery wear the hardware API can become less reliable over time. It is also notable that batteries can have varying voltage output due to variance in internal components [35]. Battery wear and variability motivate us to construct software based energy models that are resilient to prolonged device usage.

For each of the cited works in the thesis, we provide a brief summary of what the work did and categorize it into 1 of 6 categories: mining software repositories and green software engineering; software-based energy models; hardware-based energy models; hybrid energy models; software engineering and green conventions; plus deep learning and software engineering. The categories were picked by the author to help organize the reviewed work; literature surveys are included in this chapter which provide published categorizations of existing software energy related works too [7, 61]. The mining software repositories category was chosen because we collect data using techniques from multiple repositories, and revisions, and evaluate our model performance on multiple repositories and revisions; the 3 model type categories of software, hardware, and hybrid energy prediction models were chosen because we wanted to build a new energy prediction model and due to hardware-accessibility – hardware is not available to all developers and thus prevents researchers and developers from reproducing hardware and hybrid based models without fiscal investment; the software and green conventions category was created because understanding what features can be measured to predict energy consumption is important for constructing energy prediction models and this category discusses the rise, exploration, and development of what and how software development has been evolving to become more energy efficient; and we create a deep learning and software engineering category because we leverage deep learning in our model construction and because deep learning has had a recent successful resurrection in building prediction models.

3.1 Mining Software Repositories: Green Software Engineering

This section discusses works related to studying multiple revisions of software products; as well as, multiple software products. We are motivated to study multiple revisions of software because we want to build energy prediction tools that explain to developers how their software changes influence software energy consumption. For instance, our work aims to answer questions like: “did the last 5 commits to implement a new feature help or worsen the application’s energy consumption? As well, what part of the applications energy consumption changed over time due to the software changes?”

We did not find previous applications of time series models on this energy prediction task at the time of writing.

3.1.1 Ubuntu's Power Consumption Tested

Larabel investigates 6 releases of the Ubuntu operating system to study software energy consumption as it changes with respect to software updates [39]. This work demonstrates that hardware energy consumption can be influenced by the software updates it receives. The work is motivated by 1 release containing a new power management feature and curiosity to see how it actually changes energy consumption behavior. This shows that software users, and developers, are interested in energy performance alike.

3.1.2 Green Mining: A Methodology of Relating Software Change to Power Consumption

Known as Green Mining, Hindle provides a methodology for mining a software's history to understand the effect code change has on software energy use [33]. The work demonstrates Green Mining in 2 case studies that investigate the relation object oriented metrics have with energy consumption. This appears to be the first usage of mining software repositories to study software features and their relationship with energy consumption.

3.1.3 Green Mining: Investigating Power Consumption across Versions

Hindle applies Green Mining, the process of studying software history and energy consumption, to study 500 revisions of the Firefox software project and answers whether hardware component usage, or lines of code, in a changeset influences energy consumption [34]. It is shown that hardware components do influence energy usage, so future work will need to accomodate the various hardware components a software application can interact with, and that lines of code do not correlate with energy consumption. This work preludes Hindle's formalization of the Green Mining methodology [33].

3.1.4 GreenMiner: A Hardware Based Mining Software Repositories Software Energy Consumption Framework

Green Mining studies the history of software and energy consumption change. It is time consuming to run software tests and collect energy consumption from mobile devices. Therefore, the GreenMiner is constructed which consists of 4 instrumented mobile devices augmented to apply the Green Mining methodology [36]: a methodology to test and robustly collect information on multiple software revisions throughout the software's development in a given software repository. The GreenMiner automates running tests, collecting data, and it provides visualizations of the tests results to aid analysis. The GreenMiner is publicly available through a web service. This work reconfirms that screen contrast can influence energy consumption; it studies the affect UI changes have on energy consumption; and during development one of the researchers found an energy bug, and got it patched, in an Android application used for testing the GreenMiner.

3.1.5 Mining Energy-greedy API Usage Patterns in Android Apps: An Empirical Study

Linares-Vasquez *et al.* mine 55 Android applications to collect API calls and energy consumption traces from common use cases of the applications [45]. Based on analyzing the energy-hungry API calls the following recommendations are made for developers: avoid using a database management service if your data is not overly complicated; use the MVC pattern sparingly because it is costly to repeatedly update views; avoid displaying application state with energy-hungry widgets; design application views to connect to each other in the simplest way possible because it costs energy to switch views; and that developers should be critical when deciding between software design and battery savings.

3.1.6 Mining questions about software energy consumption

Pinto *et al.* survey 850 posts on the popular code discussion service StackOverflow with a specific focus on energy related discussion [53]. Pinto finds that energy questions appear to be more popular than the average programming topics on stackoverflow and it is hypothesized that energy questions are more challenging to answer given that the surveyed solutions suffer from varying quality. This work shows that energy questions appear to be a trending software development topic, at the time of writing, and the authors find 5 categories that the questions can be grouped into: measurement; general knowledge; code design; context specific; and noise [53].

3.2 Software-Based Energy Models

This section discusses prediction models that focus on using software based measurements to predict the energy consumption of running software.

3.2.1 Modeling Power Management for Hard Disks

Greenawalt builds and demonstrates an energy efficient statistical model for determining when to transition power states in a hard disk [28]. At the time of writing, hardware manufacturers have been contributing new more efficient components. But, little work has been done to create software that is also intended to be energy efficient. This paper is an early example of creating software models to approximate hardware usage for saving energy consumption of a given device.

3.2.2 Instruction Level Power Analysis and Optimization of Software

Tiwari *et al.* show that software is an important part of power consumption in computers and it can be studied from a CPU instruction point of view [59]. Tiwari *et al.* build and demonstrate instruction based power models for 3 different hardware models and provide an evaluation of their results. This work contributes a method for developing instruction based power models that do not rely on hardware to predict the energy consumption of running software, once trained.

3.2.3 Using Complete Machine Simulation for Software Power Estimation: The SoftWatt Approach

Gurumurthi *et al.* build a power consumption simulator that models the CPU, memory, and disk components of a computer to blame applications and operating system services for their contribution to device power consumption [30]. This allows users to understand what software tasks on their device are causing energy drain and helps developers prioritize what to improve to gain energy savings. The results show that power consumption simulators need to account for the whole system to better understand how running software applications are influenced by their hardware and software environments.

3.2.4 Run-Time Software Monitor of the Power Consumption of Wireless Network Interface Cards

Lattanzi *et al.* build a model for predicting the energy consumption of wireless network cards [40]. The model is trained using power consumption readings of a given network card. When in use, the model needs only software metrics to predict the energy consumption of the card. This provides developers with a view of how their software energy consumption is influenced by wireless network card usage.

3.2.5 Green Tracker: A Tool for Estimating the Energy Consumption of Software

Amsel *et al.* construct Green Tracker: a software model that uses software CPU usage to predict energy consumption [10]. The work contains several software case studies where benchmarks are ran against various softwares, that provide the same functionality, to figure out which software is more efficient at a given task. This provides researchers with a view of how to model CPU usage and methods for evaluating a model with software comparisons.

3.2.6 Detecting Energy Patterns in Software Development

Gupta *et al.* analyze mobile computing energy patterns to detect abnormal energy usage and potential software bugs [29]. The work builds models of software energy consumption to calculate per-module and total device energy consumption. It also provides a description of collecting power and data traces to detect energy related programming errors. From investigation, the work shows that most of the detected bugs were true-positives. This work shows a method for attributing energy consumption to software and an approach to energy debugging.

3.2.7 Fine-Grained Power Modeling for Smartphones Using System Call Tracing

Pathak *et al.* construct an online system call based energy profiling tool called `eprof` [50]. This work discusses and implements a method for building software-based energy models from system calls. It also discusses some of the complexities of modeling modern hardware components such as tail energy: the energy consumed from a components last use until it returns to a low-energy state. The work approximates the power states of hardware components by writing a sample application called CTester to interact with each of the components, while recording the system calls respectively. This work provides a method for building system call based energy prediction models with finite state machines and linear regression.

3.2.8 Where is the energy spent inside my app? Fine Grained Energy Accounting on SmartPhones with Eprof

Pathak *et al.* present a per-software routine energy profiler for Android and Windows phone device, named `eprof` [49]. The tool is able to account for tail energy, it also finds hot-spots and wake-lock bugs, in tested Android applications. The work also discusses case studies of 6 Android applications which are evaluated by the `eprof` tool. The case studies show how developers might use an energy profiler and explain how energy is consumed by the applications under test.

3.2.9 The power of system call traces: predicting the software energy consumption impact of changes

Aggarwal *et al.* apply a set of instrumented mobile phones for capturing software performance information, known as the GreenMiner, to show that software system call profiles correlate with software energy consumption usage [6]. This is interesting because it demonstrates that software measurable features are sufficient to predict change in hardware performance of mobile devices. It is also interesting because this work provides a methodology for testing software across multiple revisions to track software performance as it evolves over time. This allows developers to track software energy usage regressions over time and provides researchers with a method of extending this technique to build software energy models.

3.2.10 GreenOracle: Estimating Software Energy Consumption with Energy Measurement Corpora

Chowdhury *et al.* construct a linear model that can predict the energy consumption of mobile applications with, approximately, less than 10% error [16]. The work details training application selection; application test generation; model training; and model testing. A total of 24 applications with 984 revisions were used in creating the GreenOracle model. The work discusses future plans to build online energy models and to collect a larger training and testing corpus to improve model error rates. *GreenOracle* uses the following features: (system calls) recvfrom; fsync; setsockopt; mkdir; futex; write; sendto; unlink; open; and (procfs) user; ctxt; num_threads; intr; vsize; and duration [16].

3.2.11 GreenScaler: Automatically training software energy model with big data

Chowdhury *et al.* show that it is possible to construct representative energy models that can accurately predict the energy consumption of a mobile device [15]. The work explains: how hundreds of Android applications are downloaded; how manual tests can be generated for the downloaded applications; how automated tests can be generated to train representative energy models using the monkey technique; and how these energy prediction models can continuously be refined with the application of automated test generation and application. The GreenScaler mobile energy prediction model is generated based on automated tests and achieves a prediction error close to 10% error. GreenScaler is also capable of identifying energy inefficient revisions of a software application under test.

3.2.12 Estimating Mobile Application Energy Consumption Using Program Analysis

Hao *et al.* apply program analysis to create an instruction-level energy prediction model that achieve within 10% error [32]. The `eLens` tool is applied to 6 Android application from the play store. *eLens* estimate energy by recording the execution paths that are used in a developer's test for a given application. Then the overhead of each instruction on the executed path is estimated to determine energy consumption for the tested functionality in the application.

3.3 Hardware-Based Energy Models

This section discusses prediction models that focus on using hardware based measurements to predict the energy consumption of running software.

3.3.1 PowerScope: A Tool for Profiling the Energy Usage of Mobile Applications

Flinn *et al.* build PowerScope: a hardware based energy model which assigns energy consumption cost to specific software processes and methods [24]. A case study is performed with the tool that enables developers to fix hotspots in their code to reduce energy consumption by 46%. This work shows that giving developers access to the much needed energy profilers can improve software efficiency.

3.3.2 Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones

Zhang *et al.* create PowerBooter, a hardware based power model creation technique, and PowerTutor, an online power estimation tool that relies on the models created by PowerBooter [65]. PowerBooter uses a battery's API to measure voltage and approximates power consumption of various mobile hardware components, like the CPU and screen. PowerTutor combines the component estimates of PowerBooter into a single estimate for a given device. This work shows developers the energy consumption their software is causing and gives researchers an approach to modeling the energy consumption of multiple hardware components.

3.3.3 Sesame: Self-Constructive System Energy Modeling for Battery-Powered Mobile Systems

Dong *et al.* develop Sesame: a hardware based power modeling technique which takes advantage of battery interfaces to predict mobile device energy consumption [20]. The model is also capable of predicting online energy consumption of software, at a higher granularity, for an accuracy trade off in the predicted values. A case study is performed with 4 users who have the energy modeling software installed on their laptops. The case study shows that the modeling technique works well and that it is non-invasive to the user's computing experience. The work also discusses which features it collects from the Linux operating system to predict energy consumption. Some of the feature sources include the `proc` and `sys` file systems for system related information and the ACPI system (if available) to take advantage of hardware available power consumption information [20].

3.3.4 Decomposing power measurements for mobile devices

Rice *et al.* construct energy measurement toolage for the Android G1 and Magic mobile phones [55]. Rice details how to construct the measurement platforms and how to measure the energy consumption of software based questions. In particular, Rice experiment with the wireless message buffer size to demonstrate that buffer size can have a large influence over device energy consumption.

3.4 Hybrid Energy Models

This section discusses prediction models that focus on using software and hardware based measurements to predict the energy consumption of running software.

3.4.1 Software-based energy profiling of android apps: Simple, efficient and reliable?

Nucci *et al.* provide the `PETrA` tool that can provide instruction level feedback based on hardware and software observations of a running application [19]. Specifically, `PETrA` records information from the Android battery API, systrace, and dmtracedump. The model is evaluated on 54 android applications with software tests and is compared to the existing `Monsoon` hardware based model. The work shows that the `PETrA` tool performs within 5% error on each of the evaluated tests for the applications under test.

3.4.2 Calculating source line level energy information for android applications

Li *et al.* provide the *vLens* tool [43]. *vLens* is a hybrid hardware/software based model that applies dynamic analysis to predict the energy consumption from byte code of Android mobile applications to provide instruction level energy feedback at a byte code granularity. After an application has been attributed for it's energy consumption, it is also possible to view the application in an IDE with energy annotation on each line of executed code to approximate the influence each line has on energy consumption. As well, the model does not always need source code to predict the energy consumption of instructions (you just wont get that pretty java-high-level feedback). The work also highlights run-time behaviors that model designers need to be aware of such as garbage collection, concurrency thread-swapping, and tail energy.

3.4.3 An Empirical Study of the Energy Consumption of Android Applications

Li *et al.* study 405 Android applications at source code line granularity to better understand how mobile applications consume energy on smart phones [42]. *vLens* and a Monsoon power meter are used to collect run time information from the Android applications under test. Monkey was then used to generate tests for the applications to enable energy runtime data collection. The investigation shows that: more than 60% of the energy consumed by the Android applications come from API invocations; that networking components on Android devices are used the most frequently compared to other components and draw the most energy; in 91.1% of the Android applications surveyed, only 10 of the used APIs were responsible for more than 50% of the applications non-idle energy consumption; and code in loops was found to be responsible for 41.1% of non-idle energy consumption.

3.5 Software Engineering: Green Conventions

This section discusses works on energy focused software engineering and how researchers have been developing methods and conventions to engineer energy efficient software. Understanding how more efficient software can be created provides intuition about which measureable software features can be measured to approximate whether software is running efficiently. These works helped us engineer features for our prediction task.

3.5.1 The Case for Higher-Level Power Management

This work advocates for more investment in high-level software engineering tools to enable developers to build more energy efficient software [22]. The author developers a tool to monitor the power consumption of software running in PalmOS and associates these power consumption values with OS states. The work shows that software action can influence energy consumption states and that if developers had access to this information during development then they would be able to configure their hardware usage more efficiently. Clearly, hardware usage patterns influence energy consumption and there is a call for awareness of hardware-usage patterns to save battery life on mobile devices. The work also explores using GPS efficiently on the device, which is still an operation mobile devices do today.

3.5.2 PC Energy Report 2007

This article studies how many PCs are left on over night, in the US, consuming energy and the environmental impact this has [8]. The work also explains how much this costs companies to not have a more efficient power management system setup for their office computers. This article provides grounded motivation for the impact energy saving software engineering techniques and tools could have.

3.5.3 Harnessing Green IT: Principles and Practices

This Murugesan write a call to arms for IT field members to develop and practice greener processes in both hardware and software development [48]. The work discusses several measures which IT workers can take in their development environments to reduce energy consumption as well as when planning and building new IT infrastructure. This work shows how to engage industry in energy efficienct hardware and software practices. It also discusses future steps that can be taken to further improve the state of eco-friendly IT development.

3.5.4 iOS App Programming Guide

Apple provides a development guide for their iOS platform that includes recommendations to create more energy efficient software applications [2]. Recommendations include minimizing the usage of hardware to minimize power consumption. For instance, the OS will shut off unused hardware and it is recommended that timers be used to interact with constrained resources instead of polling to enable hardware to idle for longer periods of time. As well, it is explained that wireless communications is the most hardware expensive operation that is frequently made on iOS devices. Therefore, developers are given several recommendations to minimize these events to save energy. This work show industrial interest in developing more energy efficient software. It also shows that industry members have developed and are practicing green IT conventions.

3.5.5 Climate Change: A Grand Software Challenge

Easterbrook points out tentative topics that will need investigating to address the growing carbon consumption conundrum created by modern IT [21]. Software practices need to be developed to make climate models more accessible; maintainable; and portable [21]. Data processing needs to become more transparent to allow end-users to understand how data was collected; what processed it; and assumptions that were made about it [21]. Easterbrook also mentions that it would be useful for climate science to be more open source to allow the community to view and better understand exactly how the science is being done to aid in the prevention of climate denial. The work also discusses more methods for educating the public on climate change as well as Green IT practices that can improve energy usage of computer users.

This work is important because it discusses creating a more robust methodology for performing climate science. It also provides a method for educating the public on the rising problem of CO2 emissions facing humanity. The method also provides a list of achievable tasks for approaching the introduced problems.

3.5.6 Performance Considerations for Windows 7 Phone

Gray discusses how to reduce memory usage and garbage collection done on the Windows 7 Phone development platform when developing games [27]. The work approaches this by discussing optimizations for interacting with textures; how to better use objects to avoid losing performance to the garbage collector at run-time; and to use development profiling tools to view how memory is utilized on a developer's device before releasing the game to customers. This work shows industrially motivated software efficiency conventions that developers can apply to get the most out of their mobile hardware to improve their user's experience.

3.5.7 Developing Green Software

This work discusses how to make software more energy efficient [57]. Getting work done quicker often means that the device can idle for a longer period of time resulting in more energy savings. Software should also minimize the amount of data accesses to minimize hardware usage and it should try to use the processor cache more efficiently to reduce memory and hard disk accesses. The paper also advocates for using device contextual awareness, the environment the device is operating in, to optimize the hardware's usage. An example of contextual awareness would be to dim the screen when the room is bright. The paper is meaningful because it provides techniques for improving energy efficiency of software and demonstrates the techniques with several examples.

3.5.8 Energy-Optimizing Source Code Transformations for Operating System-Driven Embedded Software

Fei *et al.* study the affect code transformations can have on energy consumption by improving software inter-process-communications (IPC) [23]. A case study, from the work, involves vectorizing IPC messages and adding data buffers to the processes resulting in IPC related energy savings. 3 other IPC optimizations are proposed and evaluated in the work that each show significant improvements to the energy consumption of IPC from processes. These optimization demonstrate that development tools can be improved to automatically help developers save energy consumption in their applications.

3.5.9 An Analysis of Power Consumption in a Smartphone

Carroll *et al.* provide a tutorial of measuring, and modeling, the individual hardware component's energy consumption in mobile devices, as well as the total consumption of the given devices [12]. This work provides a better understanding of which, and how, different components influence energy consumption so that developers will have a better grasp of hardware behavior when creating software. For instance, the screen and GSM components were found to contribute the most to energy consumption on the tested device. The screen can be accomodated by developers using contrast reduction when the device is idle, but the GSM energy usage was found to be a hardware issue that software developers cannot overcome with new programming techniques.

3.5.10 LessWatts.org – Saving Power on Intel Systems with Linux

Intel provides supporting evidence for making greener IT infrastructure in this work [37]. The work covers how, at the time of writing, energy efficiency features will be added to newer Intel processors to reduce the environmental cost of running servers and a new tickless idle time feature will be added to the Linux kernel to improve the total idle time of the CPU, similar to Larabel [39]. Intel also provides a variety of power saving software tools on their Less Watts web service, which are freely available and developers can take advantage of to improve energy savings.

This work provides context for needing more energy efficient servers. It then provides a list of future hardware changes that will be made as well as providing software tools that developers can use to improve their energy efficiency.

3.5.11 Energy management techniques in modern mobile handsets

Vallina-Rodriguez *et al.* survey software energy research works between 1999 and 2011 [61]. 6 categories are created and used to classify work on software energy consumption. These categories are: energy focused operating systems; efficient resource management; studying user interaction with mobile applications and devices; sensor management; and the study of mobile energy benefits from cloud computing.

3.5.12 ErdOS: achieving energy savings in mobile OS

Vallina-Rodriguez *et al.* develop an energy savings focused operating system for mobile devices [60]. The primary methods for acheiving energy savings, in this work, come from predicting user resource usage to optimize hardware component usage. This is done by predicting the demand of resources the device will use locally on the device; and, predicting when the device will access external resources based on sensor usage such as scheduling the connection to wireless access point based on how the user's applications periodically collect data and the user's location given gps measurements. The latter, of using device sensor information to schedule component usage, is further optimized by crowd sourcing the task across ErdOS devices in close proximity to one another; crowd sourcing allows a single device to share it's sensor readings with the group which saves energy by reducing duplicate work.

3.5.13 Optimizing Energy Consumption of GUIs in Android Apps: A Multi-objective Approach

Linares-Vasquez *et al.* propose a colour pallette generation technique that aims to create legible energy saving colour schemes for mobile devices [46]. In a case study 25 applications are updated with new colour schemes and the authors find that the users are okay with the changes. The most energy saving colour schemes are not always preferred by the users; but, users still pick a scheme with energy saving when offered.

3.5.14 Making Web Applications More Energy Efficient for OLED Smartphones

Li *et al.* study how to reduce the amount of energy used by the OLED screen used by most smart phone devices [44]. Li *et al.* develop a tool which can transform web applications to use more energy efficient colour schemes on mobile devices. Li's work shows that this approach saves 40% of energy usage from the screen's total energy consumption.

3.5.15 A Review on mobile application energy profiling: Taxonomy, state-of-the-art, and open research issues

Ahmad *et al.* provide a detailed review of recent software and hardware based energy models [7]. The work provides a review of 10 software models and 7 hardware models. This list includes: power-prof; power booter; se-same; hybrid-feedback; SEMO; elens; ARO; wattson; eprof; p-top; DuT; netw-trace; power memo; power scope. There is also discussion provided on measuring network; web-browser; and multi-core CPU energy performance. Based on the created taxonomy, the authors also discuss mobile energy profiling questions that still need to be addressed such as improving the ability of operating systems to provide energy usage feedback to software applications.

3.5.16 Client-side Energy Efficiency of HTTP/2 for Web and Mobile App Developers

Chowdhury *et al.* investigate the widely used `HTTP/2` protocol compared to the prior `HTTP/1.1` with respect to energy consumption and show that the newer protocol is more efficient for transmissions with longer round trip times [17]. The investigation is accomplished by using the GreenMiner testbed, a web server, and a test script to simulate the mobile phones of the testbed. In the test, 4 applications (world flags; gopher tiles; google; and twitter, each with 10 revisions) were used on the phones under the `HTTP/1.1` and `HTTP/2` protocols respectively. This testing process showed the influence each of the protocols had over energy consumption.

3.5.17 Detecting energy bugs and hotspots in mobile apps

Banerjee *et al.* provide an automated test generation framework for detecting 2 types of energy issues faced by mobile devices known as hot spots and energy bugs [11]. Hot spots are locations in an application which cause elevated energy consumption for their duration of use. Whereas, energy bugs prevent the mobile hardware, used by the application, from returning to an idle state. The work evaluates the test generation framework on 30 freely available mobile applications. The authors also discuss what further work needs to be completed on the framework and what limitations exist for the currently generatable software tests.

3.5.18 Mobile Apps: It's Time to Move Up to CondOS.

Chu *et al.* propose a new abstraction layer be added to operating systems that makes interpretting sensor data easier for software applications [18]. The work provides an example of having the operating system track the device user's current motion state, like walking or sitting, and providing this high-level detail in a *contextual data unit*, which is the name given to the new sensor-operating-system abstraction layers the authors' propose. Furthermore, Chu *et al.* illustrate potential energy savings that result from implementing CDUs throughout the operating system and reducing program complexity for developers as direct hardware device interaction would no longer be necessary.

3.6 Deep learning and software engineering

This section discusses several papers related to the resurgence of deep learning and deep learning's application to sequence prediction tasks. We use deep learning to predict sequences of energy consumption in this work.

3.6.1 Deep learning (nature paper)

Lecun *et al.* discuss the ability of deep learning to extract abstract representations of complex problems during training [41]. This discussion is extended to provide a review of convolutional networks with supervised learning. At the time of writing, CNNs provide ground-breaking results in image interprettation and it is hypothesized that future work will uncover similar ground-breaking results with recurrent neural networks and unsupervised learning. In particular, the author guesses that RNNs will have a strong influence on interpretting videos and natural language.

3.6.2 Deep learning in neural networks: An overview

Schmidhuber provides a survey of deep learning from it's origin upto 2014 [56]. The work is motivated by the recent success of deep learning models in machine learning contests.

3.6.3 Sequence to sequence learning with neural networks

Sutskever *et al.* investigate how to use deep neural networks to map a given sequence of inputs to a given sequence of outputs [58]. To accomplish the sequence to sequence mapping, long short-term memory layers are applied and experimented with in a deep network architecture. The model is applied on language translation, from English to French, and achieves state of the art results.

3.6.4 Toward Deep Learning Software Repositories

White *et al.* apply deep learning to code recommendation tasks and compare their new models against existing state-of-the-art non-deep learning models [62]. The findings show that simple recurrent neural networks significantly out perform existing n-gram based models.

3.6.5 Deep Speech: Scaling up end-to-end speech recognition

Hannun *et al.* apply deep learning to construct a robust speech recognition method [31]. Recurrent neural networks and GPU training are used to achieve state-of-the-art performance on a widely used data set called the `Switchboard Hub5'00`. This is related to our work; since, our energy prediction problem is also a time series and we approach it using RNN based layers. Although, Hannun *et al.* avoid using LSTM based layers due to the additional performance LSTM layers tax.

Chapter 4

Modelling time-series of software energy consumption

This section discusses motivation for predicting the timeseries of mobile device energy consumption, the research questions we would like to answer while building timeseries prediction models, the methodology to collect data to train and test models, and the model types and training methods used to build our predictors.

We would like to gain a better understanding of how to predict the impact software change has on energy consumption. However, it is not possible to predict the energy consumption of software using only software measurements. Developers need costly hardware devices and or instrumentation to be able to measure the energy consumption their software has. Therefore, we would like to work on developing more efficient means to predict energy consumption without requiring software developers to purchase extranuous hardware instruments.

To accomplish this, we develop a set of research questions that need to be addressed to construct more sophisticated software energy prediction models. Particularly, we investigate how to predict the time series of software energy consumption. This would allow developers to see how their software uses energy over time, which extends the current state of the art that allows developers to see the total energy consumption their software uses over a given time period [16].

We would like to answer the following research questions pertaining to the task of predicting mobile software application energy consumption and the corresponding feature sets we are using to construct our prediction models:

- (RQ1) How well do time series models perform when predicting total energy consumption?;
- (RQ2) Do time series models with built-in state or memory perform better than time series models without state when predicting energy consumption?;
- (RQ3) Energy prediction models can be trained with different feature sets; how does performance change when using them individually versus together? What features affect the energy consumption models predictive power?;
- (RQ4) Do shallow multi-layer-perceptrons perform similarly to existing stateless models such as those based on linear regression?;
- (RQ5) How do the time series model predictions fit the shape of the observed energy consumption data?

The rest of this chapter details the method used to answer these questions. The chapter details how data is collected to build models; as well as, how model types are selected for training on and predicting energy consumption.

4.1 Methodology

The steps of the methodology used to evaluate the energy models are:

1. Find applications with multiple versions for Android.
2. Write test cases to simulate user interaction with the applications.
3. Run application tests on the GreenMiner and collect data.
4. Create cross folds: 5 applications train; 1 application test.
5. Train models, or perform parameter selection, per fold.
6. Evaluate models on test folds.

Table 4.1: Mined Applications

Application	Description	Test Duration
Calculator	An Android calculator application	125s
Blockinger	A Tetris like Android game	150s
Dalvik Explorer	A phone properties viewer for Android	80s
2048	A number puzzle game	60s
Pinball	An Android pinball game	120s
Memopad	A freehand drawing application	95s

We want to make energy models that can predict the energy consumption of software as it runs. Such models would enable developers to associate their software behavior with corresponding changes in energy consumption. To train energy prediction models, however, we need examples of running software and their resource and energy usage.

We used the *GreenMiner* testbed, described in Section 4.1.1, to collect time series information from Android applications concerning *system calls* from *strace* and *process resource utilizations* from *procfs*. To use the GreenMiner we identified Android applications with multiple versions; we write test cases to simulate user interaction with the applications; and we record information from the tests while they run on the GreenMiner.

The collected *System calls* provide information about software resource requests over time. However, system calls do not provide any information on CPU usage and other relevant information like number of interrupts, number of context switches etc. This led us to periodically access and collect those measurements from the Linux *procfs* throughout a software test. We also used hardware instrumentation to collect the energy consumption of mobile devices as they run software tests. In Section 4.1.2.1, we discuss the various models that we evaluate in our work and the reasons why they were chosen. In Section 4.1.2.2 we discuss how the models are compared with one another to answer our research questions.

4.1.1 Data Collection

Section 4.1.1.1 discusses how behavior and energy consumption profiles are extracted from software applications. Section 4.1.1.2 explains how we collect behavior and energy measurements during the mining process of Section 4.1.1.1. Section 4.1.1.3 shows preprocessing steps taken prior to training.

4.1.1.1 Mining Applications

In order to train our models, we needed example applications so that we can run them with a test case, collect their resource usage (as model input) and energy consumption (as model output) periodically. We selected 6 Android applications from the *GreenOracle* dataset, as listed in Table 4.1 (step 1). Each application had multiple software revisions available and the applications are open source software. Each application had at least 30 unique software revisions, except `Dalvik Explorer` which only had 13 software revisions. 30 was chosen arbitrarily to provide enough statistical power for many of the distribution comparison tests and to keep measurement time reasonable. Having multiple versions of each application was good for models' accuracy, because we could collect different time series of software and energy features for each revision. We also needed to write only one test case for a particular application, but could enlarge our training data by running the same test case for the multiple versions (step 2). We only chose a subset of *GreenOracle* dataset because it was time consuming to collect time series of the system call and *procfs* features (step 3). In addition, with more number of applications, the training time of our deep networks is significantly longer.

The selected applications are very different in nature. A variety of applications will employ a variety of device and hardware usage scenarios of various intensities. For example, a text editing application will use hardware differently compared to a video game application. Using different versions of the same application is also helpful for such variety. Such a process to build training data enables building robust energy models—models that are familiar to very different resource usage patterns and the corresponding energy consumption.

To measure energy consumption we wrote test runs which simulate common use cases of the applications (step 2). The common use case of an application was assessed by the authors to determine how the application is intended to be used. For example, test runs were written to do small calculations, like finding the GST of purchased goods for the Calculator application, because this is what we would expect an average user to do with a pocket calculator. Dalvik explorer displayed phone meta-data. Blockinger's test dropped and rotated tetris pieces randomly. 2048's test made random moves. Pinball's test randomly throws and paddles the pinball. Memopad's test draws a hexagon monster with legs. To reduce the generalization error of our measurements we repeat the test runs multiple times per revision to create multiple data sets for the same revision. In our study we repeated test runs 20 times each matching methodology of other works [16].

4.1.1.2 Mining Features and Labels

The software features we are interested in include: system calls, and device resource utilization (step 3). Similar to the *GreenOracle* [16], we used the *strace* program to collect all the system calls and retrieved measurements from `/proc/stat`, `/proc/pid/stat`, and `/proc/pid/statm` file systems to collect process related information. However, in contrast to *GreenOracle*, all of our features were collected periodically throughout the duration of a software test run so we can create new time series models.

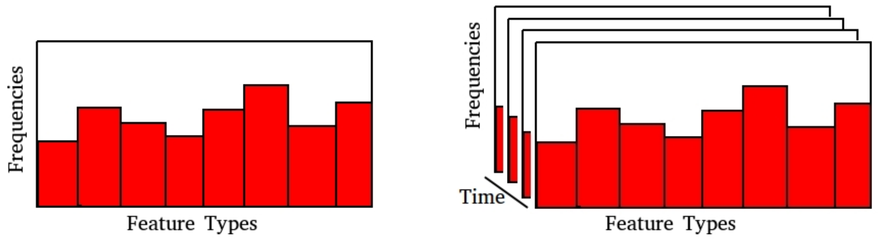
To measure resource usage for a given application over time, we define a time step to be a period of 0.02 seconds. The period of 0.02 is chosen because this is similar to the sampling rate of the *GreenMiner*. Figure 4.1a shows an example of what our model features look like in a given time step. In our energy models, after we have recorded an applications resource usage and system calls we partition the information into time steps. For each period of 0.02 seconds, we count the number of each system call that occurs in the time step, then we approximate additional applications resource utilization from *procfs* such as CPU load.

We also collect a power sample, which is a single watt measurement for the period, that is paired with the software-behavior measurements. We convert the watt measurement to energy usage using the duration between two watt samples which will be used as the dependent variable in our models.

We collect the measurements and energy usage periodically throughout a software application's usage. We then synchronize the data measurements with energy usage to create partitions, or time steps, of the data for training as in Figure 4.1b. This provides one time step for every 0.02 seconds, and the corresponding energy measurements as shown in Figure 4.2a. We preprocess these timeseries for training our energy models.

4.1.1.3 Preprocessing

We time align and bin our measurements for training. The software and energy measurement tools in our experiment are not aligned with respect to time (step 4). The software application under test begins first on the device and the GreenMiner creates a start time. The components that we wait on to start is the strace log. The sampling procfs occurs after the application has launched and even when we are waiting for the strace sampling to begin. We wait 3 seconds, or 150 time steps, after the strace has started to synchronize the captured strace and procfs components for our time series data.



(a) Features at a time step (b) Features over time steps

Figure 4.1: Figure (a) and (b): show feature set measurement groupings that are given to our *1step* and *time series* models respectively. The *1step* features are grouped together and ignore time. The *time series* features have the same features as the *1step*, but these features are grouped once every 0.02 seconds the software test runs.

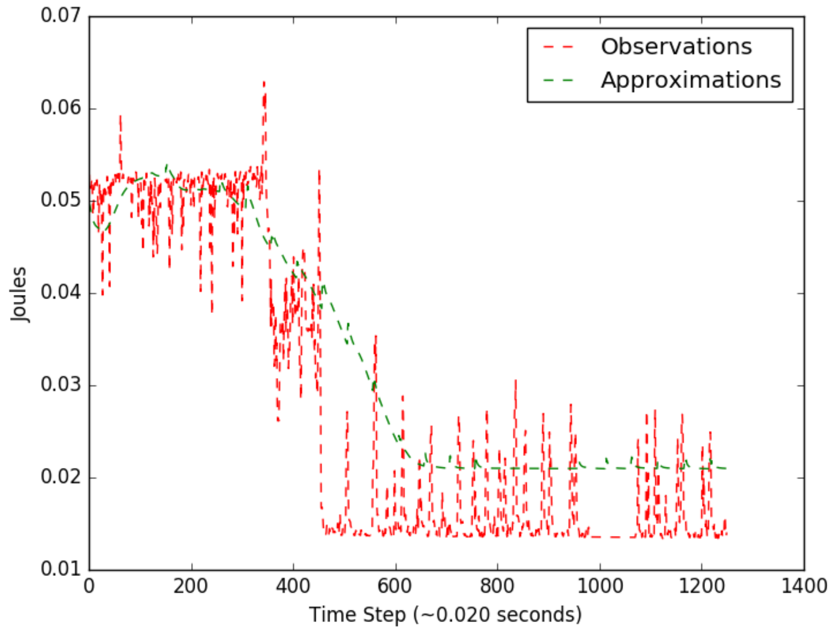
We use the next 1250 (25 seconds) time steps when training and evaluating our time series models. The number 150 was chosen to ensure that only procfs samples captured after strace logging has begun are used in upsampling and synchronization with the strace time series. The number 1250 was chosen due to hardware constraints at the time of writing. We initially tried 3200 time steps for generating our time series models but could not train in a reasonable period.

In terms of data points, after processing, we have 5 applications with 30 revisions each and 1 application with 13 revisions. Each revision is tested 20 times and from each test 1250 time steps of data with at most 80 features are used.

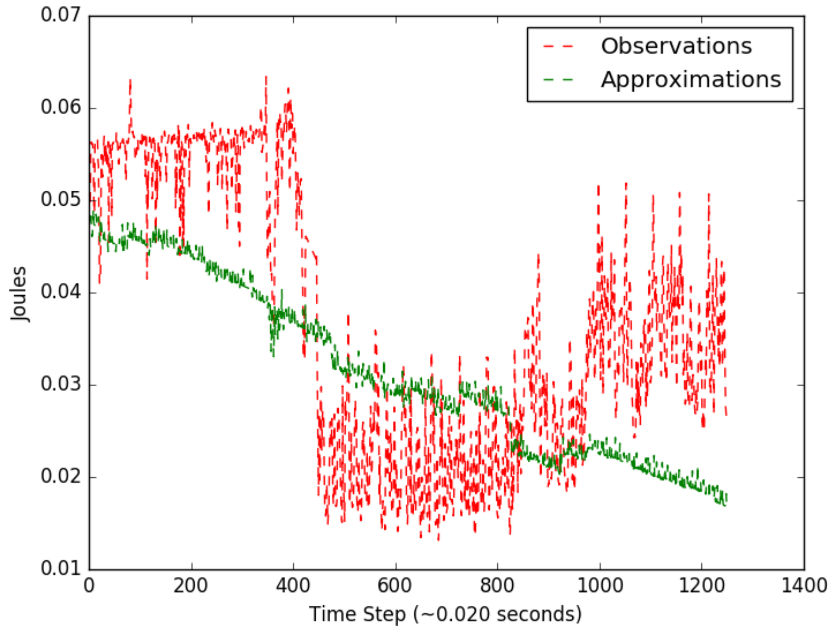
Therefore, we now describe how we time align the data. For the *procfs* and energy measurements that we collect, we apply linear interpolation to approximate measurements at fixed times. Fixed times allow us to partition our time series measurements into time steps which can be used by the model as inputs. The system call measurements which we collect are counted for each of the fixed time periods to create time steps. We collect every feature reported by *strace* and the selected *procfs* files.

We also take the difference of consecutive samples of *procfs* counters, because a single sample will only explain the current values of given resources used by a process. Whereas, the difference explains the rate of change in the resource that was caused by the running process. The *procfs* features that do not change throughout the duration of a test run are discarded.

With system calls, it is possible to have 2 semantically similar system calls occur in separate applications. We provide a list of the system call groups that we apply to our data set in Table 4.2. Duplicate names create complications when tuning the weights in machine learning models;



(a) Observed energy usage (red) over time steps with 2 layer LSTM with Sigmoid output layer predicting energy consumption (green) on Green Oracle sample



(b) Observed energy usage (red) over time steps with Linear kernel SVR model predicting energy consumption (green) from Green Oracle features

Figure 4.2: Figure (a), and (b) show measured energy over time compared with LSTM and SVR model predictions that are described later.

Table 4.2: System call feature groupings

Group	System calls
lseek	lseek, _llseek
write	write, pwrite
read	read, pread
stat	fstat64, lstat64, stat64
fsync	fdatasync, fsync

because, the training algorithm tries to attribute weight to multiple features causing the same effect in the models. This becomes problematic for us, because our system call features can perform similar operations and still have different names. Therefore, we group together similar system calls to make training the models simpler for the learning methods. We use a subset of the system call groupings as in the *GreenOracle* for comparison of our time series models against *1step* models [16]. An example of semantically similar system calls with different names are: *fstat64* and *stat64* which both get file status information.

To improve the training times of our models, and to try and avoid gradient problems in the neural networks models we train, input normalization is used in our preprocessing step. We chose to use min-max scaling for each approximated *procfs* measurement, system call count, and approximated energy usage values. For each software feature i in our data set, we calculate the scaled values $\boldsymbol{X}_{scaled_i}$ from the given software feature $\boldsymbol{X}_i$ using the following formula:

$$\boldsymbol{X}_{scaled_i} = \frac{\boldsymbol{X_i} - min(\boldsymbol{X}_i)}{max(\boldsymbol{X}_i) - min(\boldsymbol{X}_i)}$$

Feature normalization scales each features to the range of $[0, 1]$. Normalization is used to prevent the weights in models like neural networks from growing to infinity or shrinking to 0 known as the exploding or vanishing gradient problem. We also apply normalization to our energy measurements for model training. When training our models, input normalization tended to improve prediction performance.

4.1.2 Models

Section 4.1.2.1 provides the list of models we consider in our experiments. It also discusses the difference between models that consider a time series and models that are *1step* (step 5). The statefulness of LSTM models and whether or not that improves performance is also discussed. Section

4.1.2.2 explains how it is possible for us to compare the time series models with the *1step* models (step 5).

4.1.2.1 Model Selection

Tools for predicting the time series energy consumption of software are not widely available at the time of writing, although models do exist [49] or some models could be converted to do so [20], yet implementations are not available or rely on battery measurement like Di Nucci et al. [19]. Therefore, we do not have hardware-less time series models that we can compare our work against. However, there are models that predict the total energy usage of software components or hardware devices such as linear regression models of energy consumption [16]. We use the same performance metric as the prior work to make our models comparable to the existing state-of-the-art.

Because it is unknown which machine learning models could perform well, predicting the time series energy consumption, we select several. We evaluate ridge and lasso regression, and multiple support vector machine regressors based on the results of the prior work [16]. However, none of the models from the prior work are designed for taking advantage of the sequence in which data in a time series occur. Therefore, we also evaluate two different LSTM RNNs because we think that they will perform better on the time series energy prediction task. We also evaluate several MLPs because we think that they will perform similarly to the ridge and lasso regression models due to the number of weights in a single layer MLP being similar to the number of weights in a linear regression model.

We evaluate all of the models with 4 different feature sets to get an understanding of how the models perform. Some of our selected models like the LSTM and MLPs perform feature selection themselves, so it is not clear if manual feature selection would help these models perform more effectively. We use a full feature set which consists of all the features we collected from *system calls* and *procfs*, a *GreenOracle* feature set which consists of the best linear features as selected in the prior work [16], a *system call* feature set which only consists of system call features, and a *procfs* feature set which only consists of the process related features.

For the MLP and LSTM RNNs, we rely on stochastic gradient descent to perform model tuning. We select specific structures, in the case of the MLPs we look at: a sigmoid hidden layer with a linear output, two sigmoid hidden layers with a linear output, and three hidden sigmoid layers with a linear output. On the deep neural networks, we consider the use of one and two hidden LSTM layers with either a linear output layer or a sigmoidal output layer. Our neural network input and

Table 4.3: Time Series and Time-Summed *1step* Model Names and their Descriptions

Time Series Model	Description
MLP_1_layer	1 layer MLP
MLP_2_layer	2 layer MLP
MLP_3_layer	3 layer MLP
lstm3sigm	2 layer LSTM and sigmoid output
lstm2sigm	1 layer LSTM and sigmoid output
lstm3line	2 layer LSTM and linear output
lstm2line	1 layer LSTM and linear output
linear_regressor	lasso regression model
ridge_regressor	ridge regression model
svr_poly	polynomial SVR
svr_rbf	radial-basis-function SVR
svr_plain	linear SVR

1step Model	Description
1step_MLP_1_layer	1 layer MLP
1step_MLP_2_layer	2 layer MLP
1step_MLP_3_layer	3 layer MLP
1step_linear_regressor	linear regression model
1step_ridge_regressor	ridge regression model
1step_svr_poly	polynomial SVR
1step_svr_rbf	radial-basis-function SVR
1step_svr_plain	linear SVR

hidden layers contain one node per feature being evaluated by the model. For example, a 2 layer MLP evaluated on the GreenOracle feature set would have 14 input nodes, 14 nodes on the hidden layer, and 1 node on the output layer.

Furthermore, the selection of appropriate MLP and RNN LSTM structures is further complicated by neural network hyper parameters like number of nodes, number of layers, and activation functions. Capacity of a model is associated with how much information the model can learn and store from the training data. Capacity is associated with the number of edges, neurons, and whether or not the model neurons contain memory cells. It is not clear if more or less capacity would help with the learning task on the MLP and our deep learning LSTM models. Too much capacity in a model can result in over fitting, and too little capacity can lead to insufficient resources to capture the relationship from a data set. Similarly, it is not clear in the case of LSTMs whether a linear or sigmoidal output layer will produce better results. It has been found that linear outputs can model Gaussian distributions and our physical measurements of the mobile devices often have Gaussian distributions [26]. So, we evaluate several types of neural network structures to get a better understanding of the problem which we are approaching. A list of the models is provided in Table 4.3.

4.1.2.2 Model Comparison

We investigate how models perform when trained with each time step, and when we ignore time in the data set by aggregating our time series into a *single time step* (step 6). This means, we have a set of models which predict every instantaneous time step of a time series. The models use program resource usage to predict energy consumption. We train models to predict, from a whole time series, the total energy consumption of the application. Prior works evaluated performance using data sets similar to the *single time step* models [16]. We also use the mean relative error as the loss function. This is calculated by checking if the predicted total energy usage for an application is similar to the observed total energy usage for the application.

Since, we use the same loss function in the instantaneous time step and *single time step* formats of the data set, we are able to compare models trained on the separate contexts with one another because they are evaluating the total predicted joules. It would not be possible to compare the error of the per-time-window energy predictions of the time series models to the total energy predictions of the *1step* models. The loss function is given below, where $\boldsymbol{Y}$ denotes the energy observations per-time-window of the time series in joules and $\hat{\boldsymbol{Y}}$ denotes the energy predictions in joules for each time step in a series.

$$f(\hat{\boldsymbol{Y}}, \boldsymbol{Y}) = \left| \frac{\Sigma\hat{y} - \Sigma y}{\Sigma y} \right|$$

The models which predict each time step of a given series will produce a normalized energy prediction, we denormalize the prediction and denote it $\hat{y}$, for each step of the series. Whereas, in the case of the *1step* models which take the whole time series and predict the total energy consumption, in normalized joules for an application test run, there is only one predicted $\hat{y}$ value which we denormalize for evaluation. Furthermore, we test several different feature sets for comparison of our model performance. We test a full feature set (80 features), which consists of all system calls and process related features collected from the applications [5, 4]; a replica of the *GreenOracle* feature set (14 features), which consists of a subset of the full feature set; a feature set based only on the system calls (61 features); and a feature set based only on the process related features (19 features). The *GreenOracle* model used linear methods to perform feature selection, whereas our stateful models perform their own version of feature selection between features. So, if the full-feature set performs remarkably better than the *GreenOracle* feature set, we would expect a non-linear relationship to

Table 4.4: SVR Parameters

Parameter	Coefficients
Penalty	$2^{-5}, 2^{-3}, 2^{-1}, 2^{1}, 2^{3}, 2^{5}, 2^{7}, 2^{9}, 2^{11}, 2^{13}$
Gamma	$2^{-15}, 2^{-13}, 2^{-11}, 2^{-9}, 2^{-7}, 2^{-5}, 2^{-3}, 2^{-1}, 2^{1}, 2^{3}$
Degree	2

have been identified. We investigate the system call and process feature sets to determine if combining the two feature sets improves the models accuracy.

We use 6-fold leave-one-out analysis on our 6 applications to train multiple instances of our models (step 5). The data collected from each application is used to build a holdout fold. The models are evaluated on the respective holdouts (step 6). We calculate the effect-size of a representative sample from the training set and use it to train each SVM model. We perform 6-fold leave-one-out analysis 4 times for the following feature sets mentioned above: full feature set, *GreenOracle* feature set, process-only feature set, and the system-call-only feature set. This evaluation format was chosen because it will explain if a particular model performs well for a given feature set and if a particular model performs well for every feature set. As a result, in our evaluation we do not distinguish folds from one another, we compare model performance by combining all of the folds for comparison of each model.

For the SVR models [51] which we train, we perform parameter selection using 5-fold cross validation on each fold. Table 4.4 shows the list of hyper parameters which we use for parameter selection. The *Gamma* coefficients are used for radial basis and linear kernels, and the *degree* coefficient is used for a polynomial kernel. SVRs are applied to the full and *GreenOracle* feature sets only, as these are combined feature sets of the system-call-only and processor-only feature sets.

4.1.2.3 Time Series Model Comparison with Dynamic Time Warping

Dynamic Time Warping (DTW) is a common technique for determining how to align time-dependent sequences with one another [47]. In other words, it is the minimum cost to transform one sequence into the other. The alignment cost can also be viewed as the similarity between given time-dependent sequences. We apply a Python implementation of the DTW algorithm from GitHub to calculate the cost of aligning our approximated energy consumption costs, as predicted by our models, against the observed energy consumption costs of running android applications [52]. The Python source code of the algorithm is provided in Listing 4.1. We apply the absolute distance function as $dist(a, b) = |a - b|$ in the program listing.

4.1.2.4 DTW implementation and visualization

The algorithm for calculating the optimal mutation sequence from one time series to another uses dynamic programming. DTW uses a matrix to store the minimum cost mutation between two given sequences, up to given time steps in the respective sequences, at each entry $[x, y]$. If we assume that $[x, y]$ is a valid entry in the matrix, this position will store the minimum cost to mutate the 1st time series $t1[1..x]$ into the 2nd time series $t2[1..y]$ – where the time series indices denote time steps up from the start of the sequences up to the x and y time steps of the respective sequences. The next paragraph goes through the code implementation in Python and discusses how the minimum cost paths are calculated inside of the matrix.

In Listing 4.1, line 20 can be seen as calculating the distance between every time-based measurement in the sequences being compared. Line 25 updates this matrix *D1*, of point-to-point costs, to now store the minimum cumulative path cost of aligning the sequences up to points *i* and *j*, in the respective sequences. Once the loop on line 23 has run to completion, the minimum cost path to align the two sequences is stored in the matrix position *D1[n, m]* where n is the length of the input sequence x and m is the length of the input sequence y. Line 25 can be interpretted as mutating the x input sequence into y by considering the points at $x[i]$ and $y[j]$ to be a minimal distance from one another, incurring no mutation; $x[i]$ and $y[j+1]$ are closer than the formentioned path so $y[j]$ becomes ignored; or $x[i+1]$ and $y[j]$ are the closest points in the iteration of the algorithm and $x[i]$ becomes ignored/deleted from the sequence x. Note, line 18 makes a copy by reference from D0 to D1, the variables refer to the same matrix just different parts; this works because x and y are numpy arrays and D0 and D1 are numpy matrices. Visualizing how the DTW algorithm calculates the minimum cost alignment seems simplest to understand by drawing/viewing the cumulative-alignment-cost matrix (initialized on line 15 of Listing 4.1) and viewing how the algorithm compares nodes between the sequences to figure out the best alignment cost (on line 25 of Listing 4.1).

Figure 4.3 shows the minimum cost path to mutate a predicted energy time series into an observed energy time series. If the line blue line minimum cost path moves diagonally then the sequences being aligned were similar at the given time steps and did not need to be modified. However, if the blue line moves vertically, this means that the aligned sequence that is being generated used values from the observed data whereas horizontal movement of the blue line implies that the predicted data was used to generate the mutated sequence. From this understanding, we can note that the predictions and observations are differ interchangable until the coordinate region $(750, 800)$ where the predicted data begins to differ from the observations more dramatically with a long horizontal strech with mild incline and a long vertical strech with mild incline. The horizontal and veritcal streches imply that neither time series was similar across these regions.

```python
from numpy import array, zeros, argmin, inf, equal, ndim
from scipy.spatial.distance import cdist

def dtw(x, y, dist):
    """
    Computes Dynamic Time Warping (DTW) of two sequences.
    :param array x: N1*M array
    :param array y: N2*M array
    :param func dist: distance used as cost measure
    Returns the minimum distance, the cost matrix, the accumulated cost matrix, and
    the wrap path.
    """
    assert len(x)
    assert len(y)
    r, c = len(x), len(y)
    D0 = zeros((r + 1, c + 1))
    D0[0, 1:] = inf
    D0[1:, 0] = inf
    D1 = D0[1:, 1:] # view
    for i in range(r):
        for j in range(c):
            D1[i, j] = dist(x[i], y[j])
    C = D1.copy()
    for i in range(r):
        for j in range(c):
            D1[i, j] += min(D0[i, j], D0[i, j+1], D0[i+1, j])
    if len(x)==1:
        path = zeros(len(y)), range(len(y))
    elif len(y) == 1:
        path = range(len(x)), zeros(len(x))
    else:
        path = _traceback(D0)
    return D1[-1, -1] / sum(D1.shape), C, D1, path


def _traceback(D):
    i, j = array(D.shape) - 2
    p, q = [i], [j]
    while ((i > 0) or (j > 0)):
        tb = argmin((D[i, j], D[i, j+1], D[i+1, j]))
        if (tb == 0):
            i -= 1
            j -= 1
        elif (tb == 1):
            i -= 1
        else: # (tb == 2):
            j -= 1
        p.insert(0, i)
        q.insert(0, j)
    return array(p), array(q)
```

Listing 4.1: DTW implementation from the Python pip2 module `dtw` under the GPLv3 license [52].

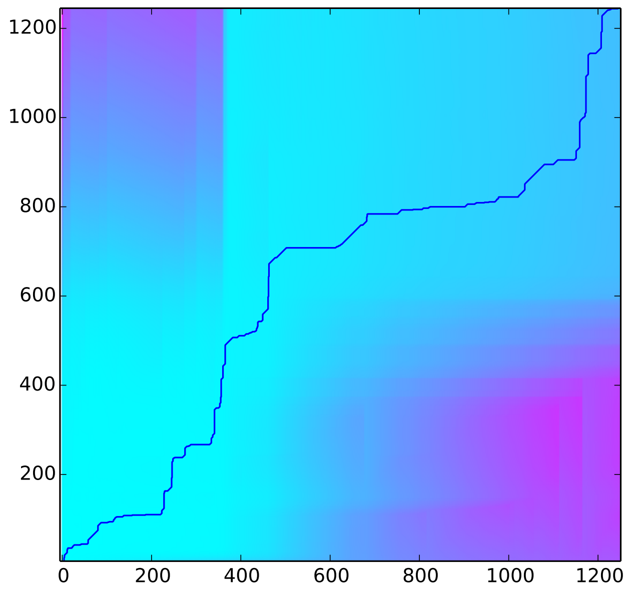


Figure 4.3: Visualization of DTW path calculation as applied to an Android applications measured energy consumption time series (on the X axis) and an LSTM's prediction of the same application's energy consumption time series (on the Y axis.) The numeric values on the X and Y axis denote the segment of time in the respective time series. e.g. 4 on the X axis is the 4th time slot of the observed energy consumption representing the amount of energy consumed since the 4rd time slot was measured. A magenta colour in the figure denotes a more expensive cost to mutate the predicted (Y axis) time series into the observed (X axis) time series. The cyan colour denotes a cheaper mutation between the given sequences. The dark line connecting the bottom-left to the top-right denotes the minimum cost mutation path found with DTW.

Chapter 5

Evaluation

In this section, we discuss our evaluation of the models, listed in Table 4.3, that were created in the previous chapter. We investigate if the models perform differently or similarly to one another when predicting the total energy consumption of mobile applications. We perform a similar investigation on the time series models to determine if they predict sequences that fit (follow the trends in) the observed energy sequence values.

We provide our models with the time series of recorded software behavior for given software test runs. The time series for the *1step* models are summarized because the *1step* models ignore the time dimension of the series. Each of the multistep models will predict a series of energy predictions for each of the input time steps. The models predict the energy consumption of each step of the time series or cumulatively the energy consumption for the whole series.

Before it is possible to compare all of the models, we have to perform hyper parameter selection to evaluate our SVR models. The SVR training parameter selections are shown in Table 5.1. We generate SVR models for each of our feature sets because there might be non-linear relationships which the RBF or polynomial kernel SVR could identify.

We want to know how our models compare against one another on the task of predicting normalized joules from software behavior. We evaluate our models by denormalizing the joule predictions such that we can compare energy usage across the 1*step* and time series models. With joules, we can compare our prediction accuracy to the prior works [16].

We had numerous models and thus we wanted to see if models were significantly different in their performance. To check whether or not our models produced normal distributions for the data

Table 5.1: Selected SVR Parameters and Feature sets

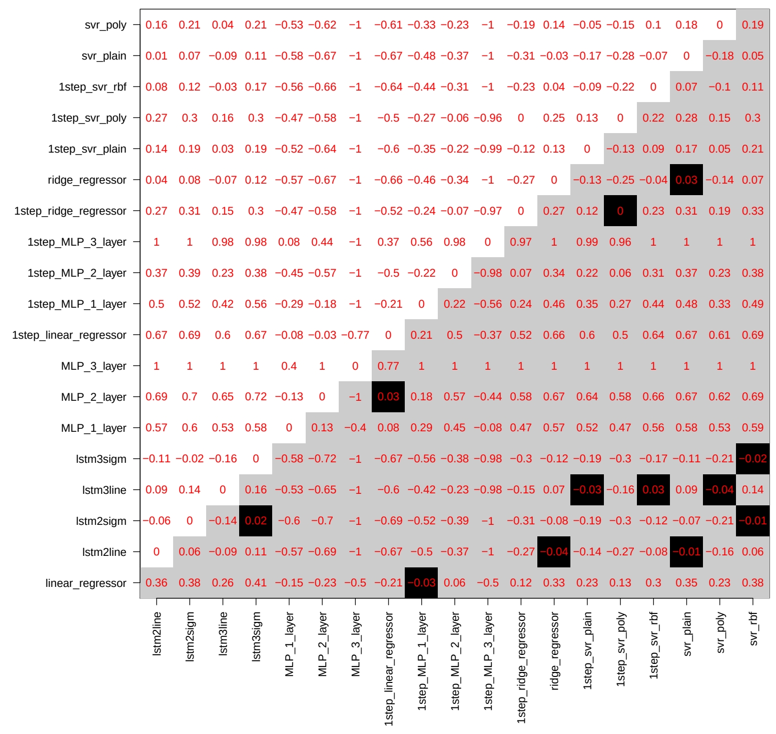
Model	Feature set	Penalty	Gamma	Degree
linear	Full	0.125	0.0125	-
rbf	Full	2	8	-
polynomial	Full	2048	-	2
linear	GreenOracle	0.03125	0.0714	-
rbf	GreenOracle	0.5	8	-
polynomial	GreenOracle	8	-	2
linear	System call	0.125	0.0125	-
rbf	System call	32	0.125	-
polynomial	System call	2048	-	2
linear	Proc	2	0.0125	-
rbf	Proc	128	0.5	-
polynomial	Proc	128	-	2
*1step*_linear	Full	0.125	0.0125	-
*1step*_rbf	Full	2	8	-
*1step*_polynomial	Full	2048	-	2
*1step*_linear	GreenOracle	0.125	0.0714	-
*1step*_rbf	GreenOracle	0.5	8	-
*1step*_polynomial	GreenOracle	32	-	2
*1step*_linear	System call	2	0.0125	-
*1step*_rbf	System call	2	0.125	-
*1step*_polynomial	System call	2048	-	2
*1step*_linear	Proc	8	0.0125	-
*1step*_rbf	Proc	8192	0.0005	-
*1step*_polynomial	Proc	512	-	2

being predicted, we used the Anderson-Darling test. However, we found p-values below $2e{-}16$ which confirmed that none of the distributions under consideration are normal. Thus, non-parametric tools are used to assess the relationships between the models. A Kruskal-Wallis test on each feature set shows a p-value $< 2e{-}16$ which implies that the model type has a significant influence on the output. Thus the models are mostly different from each other and we can show this with the pairwise Wilcoxon test with Holm p-value correction for multiple comparisons. The Wilcoxon and Cliff's Delta methods do not make assumptions about the data distributions being evaluated and the two tools can inform us whether or not two models perform significantly different from one another.

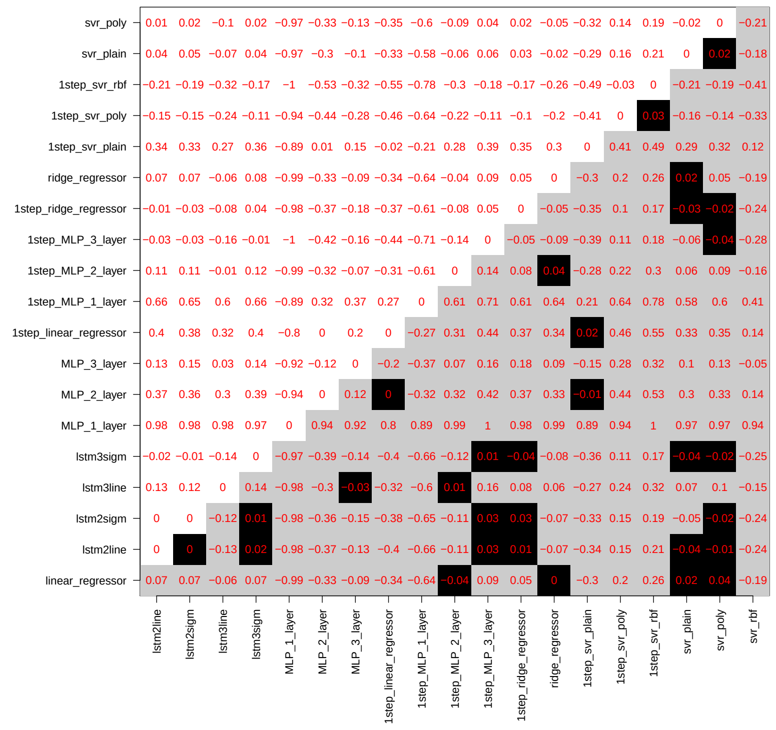
In the Figures 5.1a, 5.1b, 5.0c, 5.0d, 5.1a, 5.1b, 5.1c, and 5.1d, the text *1step* denotes a model which takes the total counts of different syscalls and other resource usage as the input and predicts the total energy consumption for the test run. Models that do not include the *1step* prefix predict energy at each time step for the given time series of a given test.

Figure 5.1a, Figure 5.1b, Figure 5.0c, and Figure 5.0d show the evaluation of pairwise Wilcoxon tests on each group of models as well as the evaluation of Cliff's Delta. The color of a square shows the Wilcoxon result where: *white* means the two models were not tested against one another for their Wilcoxon p-values (this is due to the compared models being the same model or the compared models are already compared on the other side of the grid), *black* (■) means a p-value above or equal to 0.05, and *grey* (▨) means a p-value less than 0.05. A p-value less than 0.05 means that two given models are significantly different from one another as shown by the grey squares. Most models were different from each other, as there were few black squares. The number on the square identifies the Cliff's Delta estimate of how different the two models under comparison are. A number close to 0 means that the models are very similar, whereas -1 or 1 mean that the models predict different intervals of joules from one another. Models which predict different intervals from each other can be compared to see if one creates stronger results than the other. We compare the models performance against each other using their total energy prediction accuracy [16]. Figures 5.4a, 5.4b, 5.3c, and 5.3d show analogous pairwise Wilcoxon tests and Cliff's Delta evaluation of the time series model DTW alignment costs using absolute distance; these figures allow us to assess which models fit the trends of the data the best.

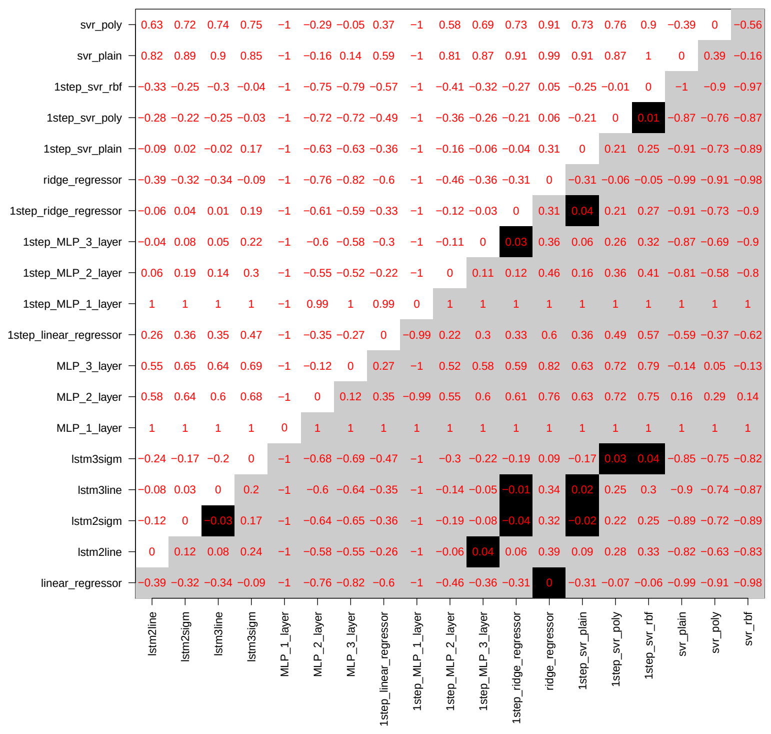
Figures demonstrate that the models can predict software energy use and are mostly different from each other we want to evaluate the performance of each model. We show box plots, ordered by the mean total joule prediction error for each model, to illustrate the difference in performance between



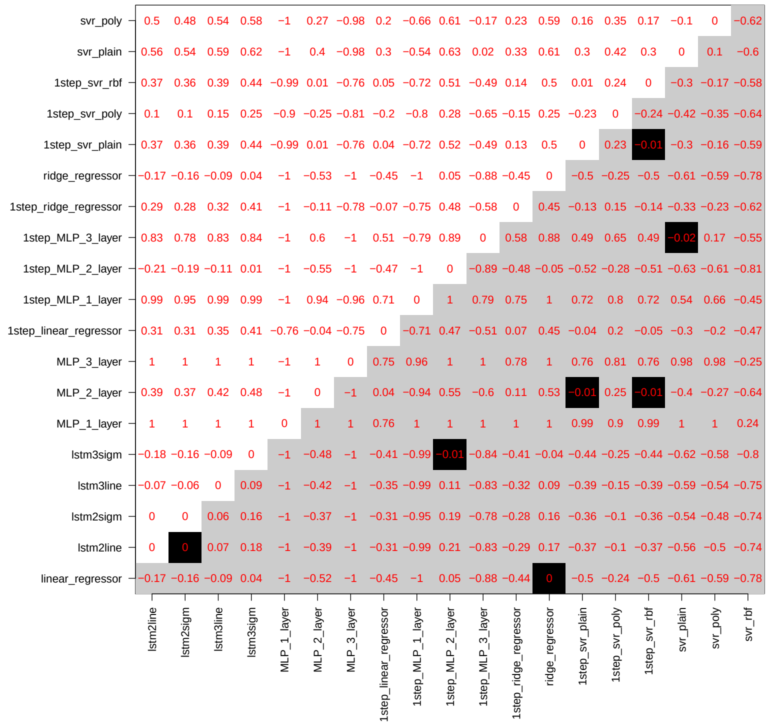
(a) Full feature set



(b) GreenOracle feature set

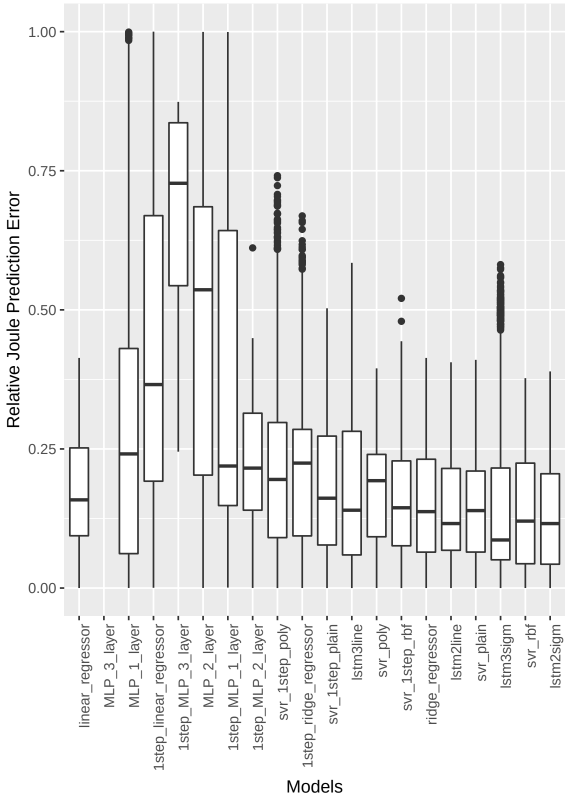


(c) System call feature set

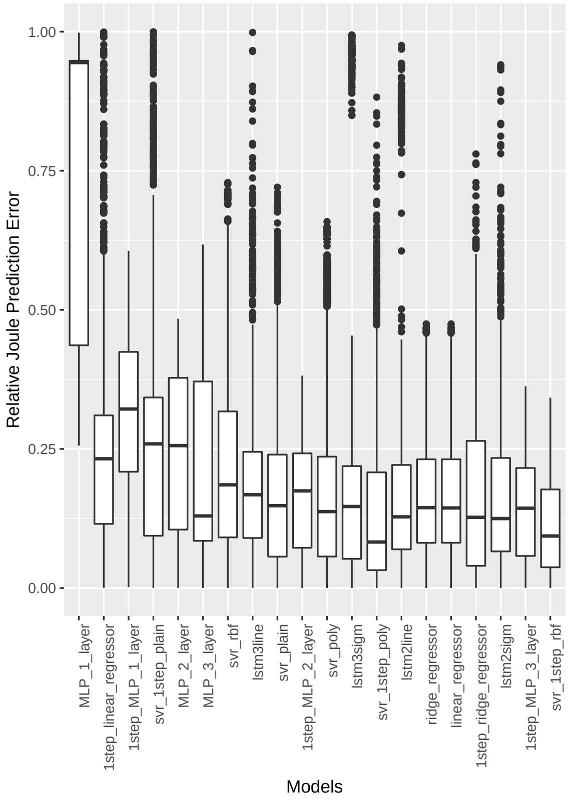


(d) Processor feature set

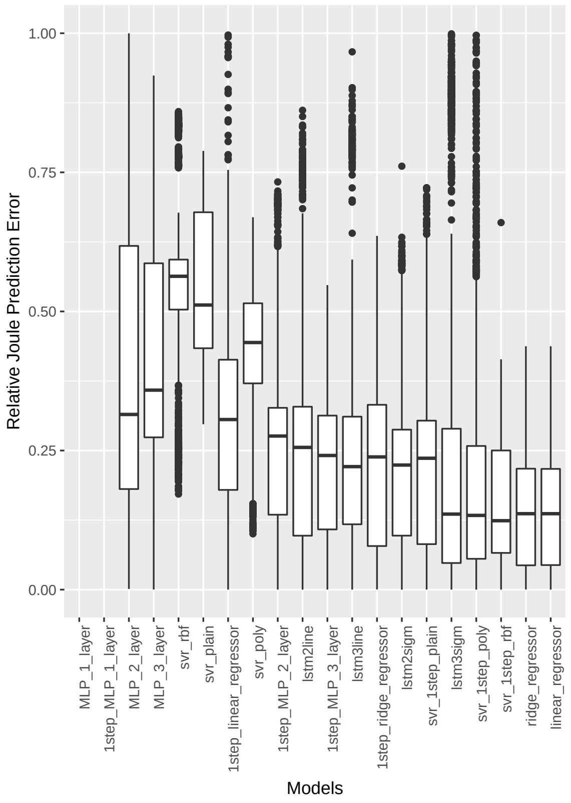
Figure 5.0: Pairwise Wilcoxon and Cliff's Delta of energy models under 4 feature sets. A ▨ denotes a significant difference and ■ denotes no-significant difference using Wilcoxon t-test with Holm correction. Note these are the lower triangle of comparisons.



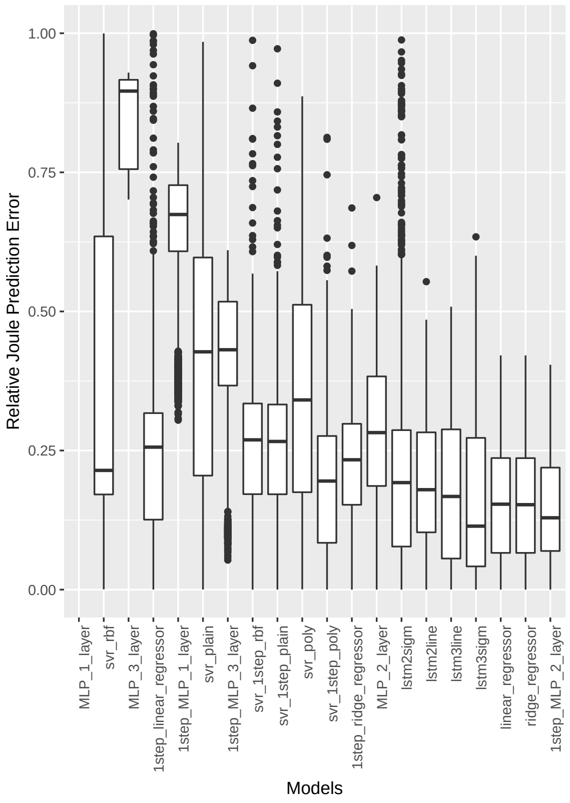
(a) Full feature set evaluated on *1step* and discrete contexts



(b) GreenOracle feature set evaluated on *1step* and discrete contexts



(c) System call feature set evaluated on *1step* and discrete contexts



(d) Processor feature set evaluated on *1step* and discrete contexts

Figure 5.1: Model relative prediction error box plots under various feature sets

models. Figure 5.1a, Figure 5.1b, Figure 5.1c, and Figure 5.1d show box plots of model predictions using different feature sets. To generate the values for each box plot, the model is evaluated on how well it predicts the total energy consumption for each test run. We show the combined results of the *1step* models and the time series models, in terms of relative error, of each model trained on a given fold predicting the respective holdout set. Note in some of figures that the models boxplots are missing, such as lasso regression or 3 layer MLP, because the models have huge cumulative error and would not fit within the domain of the plot. We calculate the mean relative joule prediction error for the total energy consumption using the following formula for the time series models:

$$\frac{|\sum_{i=1}^{n} a_i - \sum_{i=1}^{n} b_i|}{\sum_{i=1}^{n} b_i}$$

Where n is the number of time steps, a_i is the predicted energy consumption at timestep i, and b_i is the observed energy consumption at time i. For the *1step* models we use: $\frac{|a-b|}{b}$ where a is the predicted energy consumption for a model, and b is the total observed energy consumption of a software test.

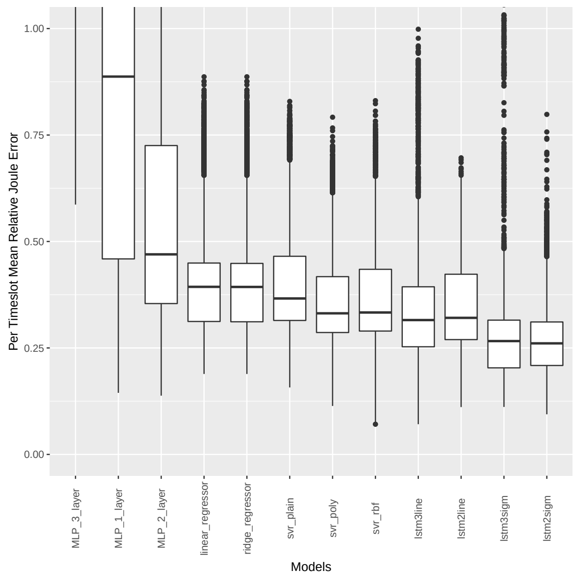
We also present box plots ordered by the mean relative joule prediction error for each time series model. The mean relative joule prediction error shows how well the time series models performed when predicting each time slot of the time series energy consumption. Figure 5.2a, Figure 5.2b, Figure 5.2c, and Figure 5.2d show the timeslot relative error box plots for each of the feature sets under evaluation. We use the following formula to calculate the mean relative joule prediction error per time step:

$$(\sum_{i=1}^{n} \frac{|a_i - b_i|}{b_i})/n$$

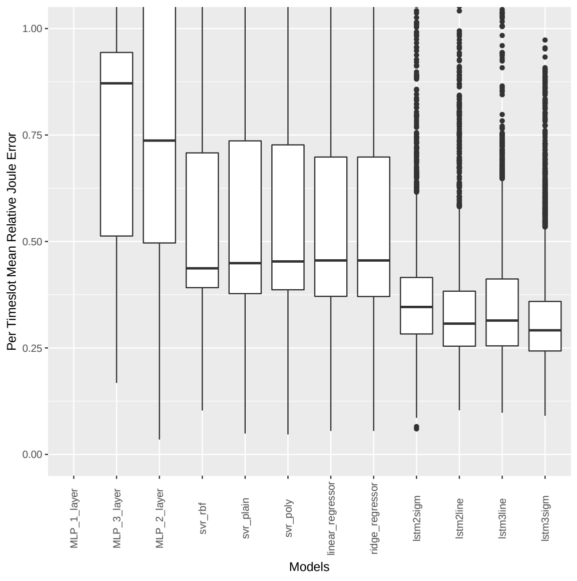
Where n is the number of time steps, a_i is the predicted energy consumption at timestep i, and b_i is the observed energy consumption at timestep i.

The subfigures of Figure 5.1 explain the total energy prediction error and the subfigures of Figure 5.2 explain the energy prediction error for each time slot of the time series. However, neither figure explains how well the time series models predict the trends of the software test runs such as when energy consumption is increasing over a period of time or decreasing. Therefore, we evaluate the time series models using Dynamic Time Warping. We calculate the warping distance between the observed energy usage of a time series and the predicted energy consumption. Figure 5.3a, Figure

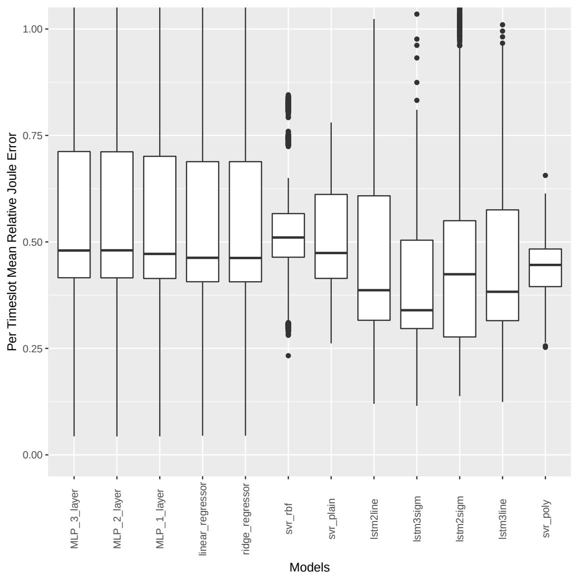
5.3b, Figure 5.3c, and Figure 5.3d show boxplots of the time series model performance under dynamic time warping. Figures 5.4a, 5.4b, 5.3c, and 5.3d show comparisons of the time series models under a Pairwise Wilcoxon comparison and the Pairwise Cliff's Delta values.



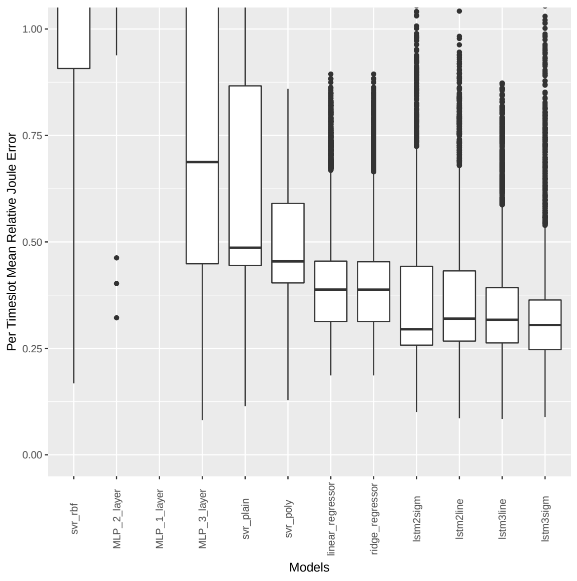
(a) Full feature set



(b) GreenOracle feature set

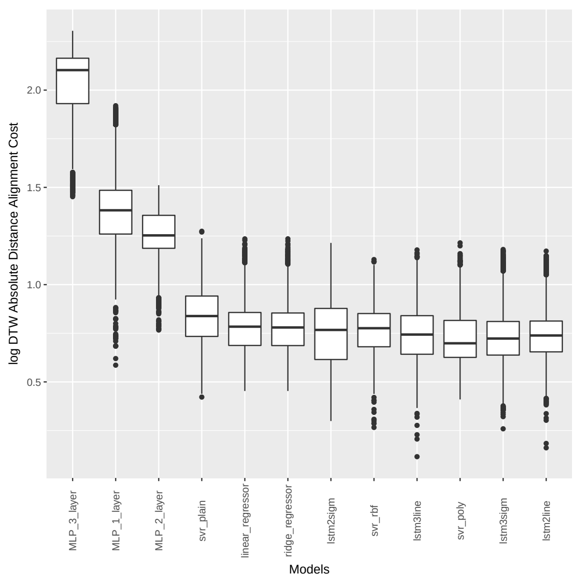


(c) System call feature set

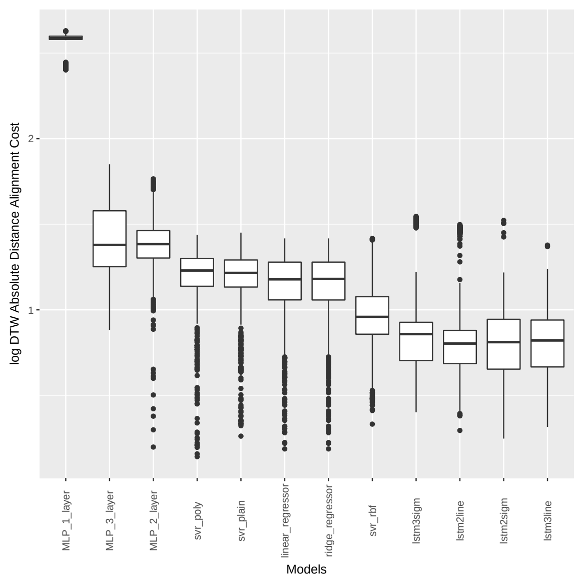


(d) Processor feature set

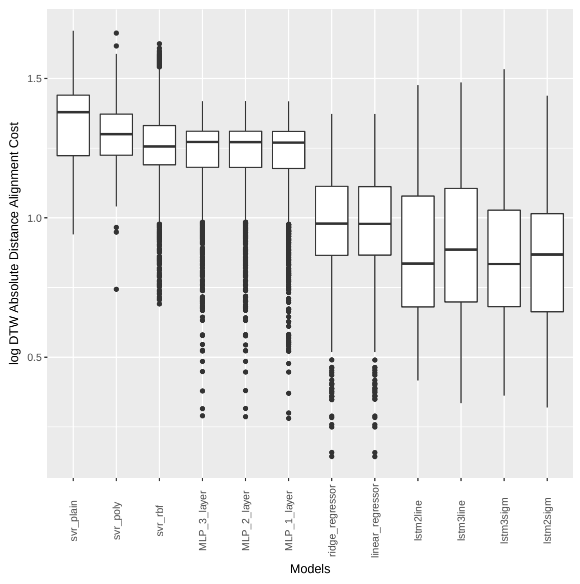
Figure 5.2: Model relative prediction error box plots under various feature sets per timeslot



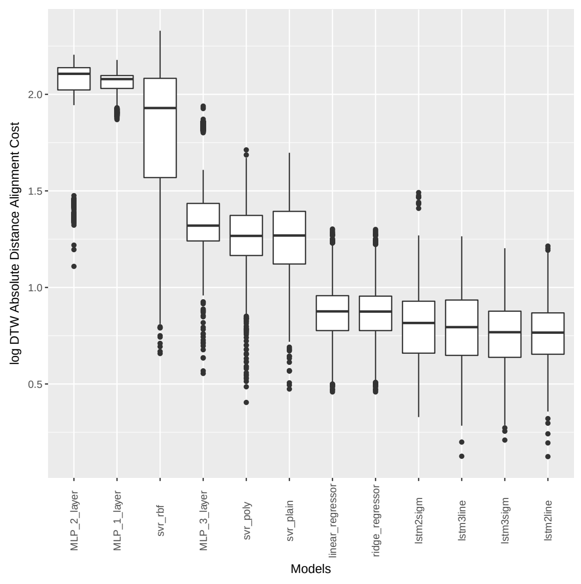
(a) Full feature set



(b) GreenOracle feature set

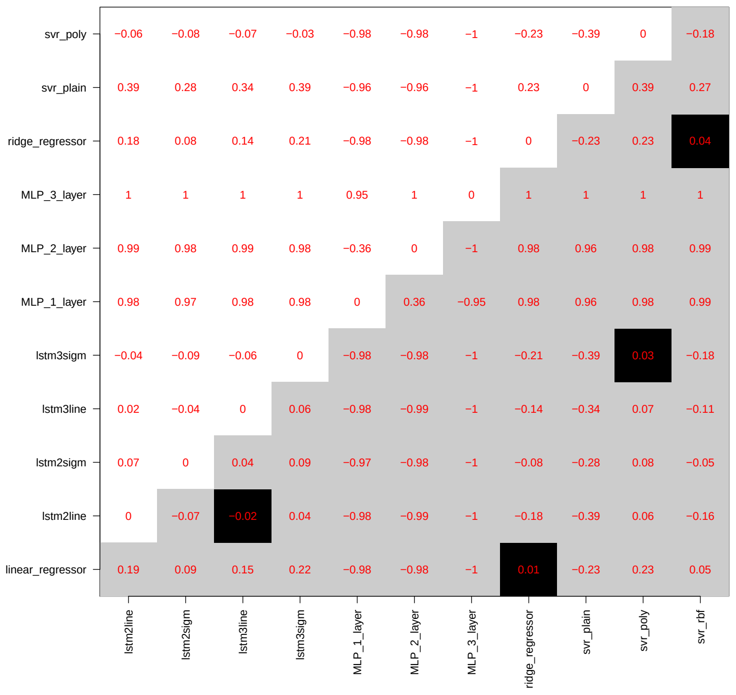


(c) System call feature set

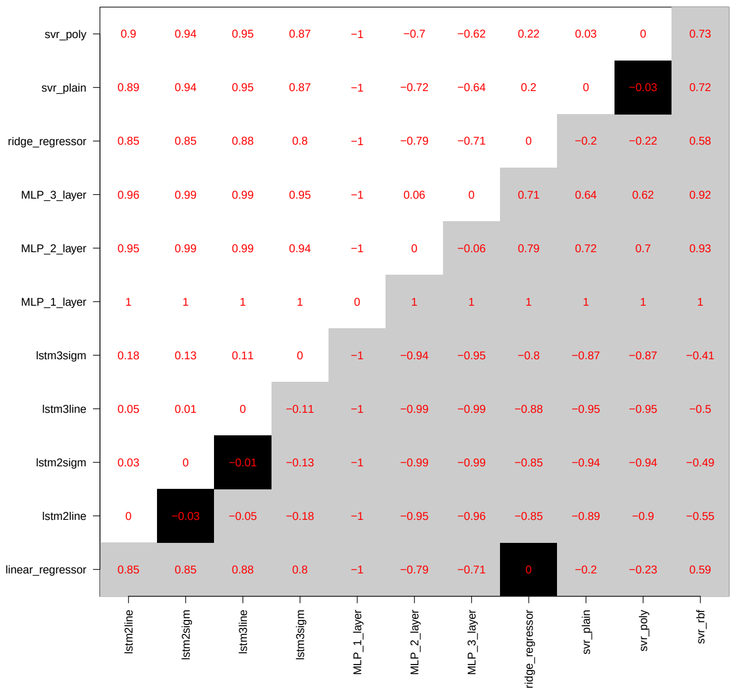


(d) Processor feature set

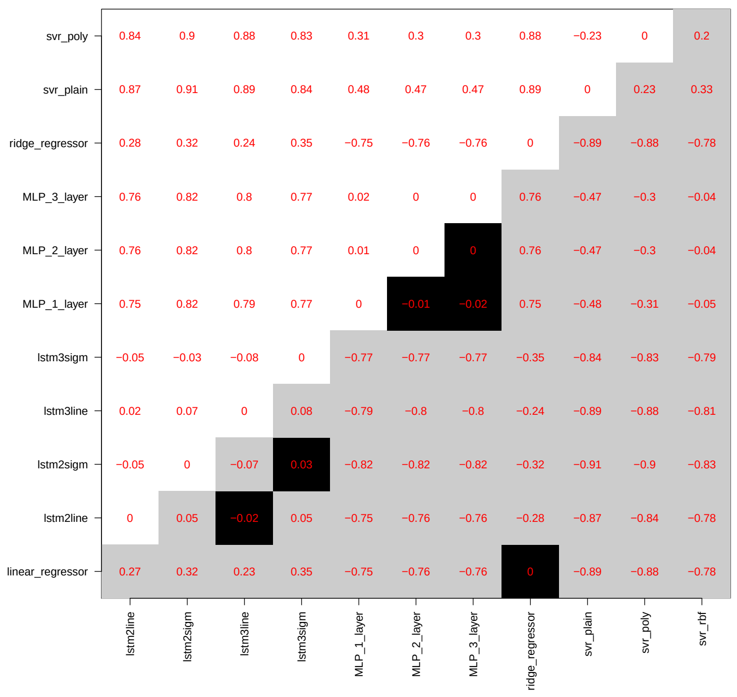
Figure 5.3: How well does our sequence of predicted energy values fit the curve of the observered energy values? We provide the alignment costs of our predictions against the observations with box plots under various feature sets. The box plots compare timeseries sequence alignment between predicted and observed values using the dynamic time warping cost of mutating our predict sequence into the observed sequence, based on the absolute distance between sequence values.



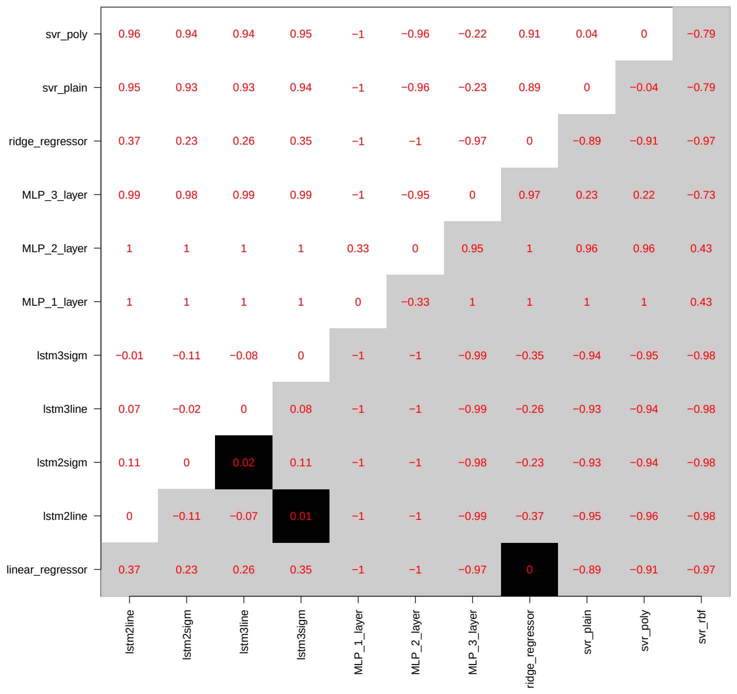
(a) Full feature set



(b) GreenOracle feature set



(c) System call feature set



(d) Processor feature set

Figure 5.3: Timeseries Models evaluated with DTW with absolute distance alignment cost under various feature sets with Pairwise Wilcoxon and Cliff's Delta. A ▨ denotes a significant difference and ■ denotes no-significant difference using Wilcoxon t-test with Holm correction. Note these are the lower triangle of comparisons.

Chapter 6

Research Questions and Discussion

In this section we answer our 5 research questions:

- RQ1: How well do time series models perform when predicting total energy consumption?
- RQ2: Do time series models with built-in state perform better than time series models without state when predicting energy consumption?
- RQ3: Energy prediction models can be trained with different feature sets. How does performance change when using them individually versus together?
- RQ4: Do shallow multi-layer-perceptrons perform similarly to existing stateless models such as those based on linear regression?
- RQ5: How do the time series model predictions fit the shape of the observed energy consumption data?

and our findings:

- RQ1: Model performance depends on the feature set but timeseries models do slightly better than models built specifically for predicting total energy consumption.
- RQ2: We find that stateful models are better than stateless models when predicting time series of energy consumption.
- RQ3: We found that using a combination of CPU and *procfs* features worked better than using the CPU or *procfs* features individually.

- RQ4: No. MLP models perform worse than existing linear models.
- RQ5: The stateful time series models fit the shape of observed energy consumption much better than the stateless models. Further work could be invested developing these models to achieve better fits.

6.1 RQ1: How well do time series models perform when predicting total energy consumption?

Time series models often perform just as well or better than time-summarized/1step models. We are motivated to answer *RQ1* partially due to the idea that the time series models could have a large cumulative error from miss-predicting many data points in a time series compared to the total energy prediction model that only has to train to predict a single data point. We evaluated whether or not any of the models perform similarly by checking their relative prediction error of total energy consumption for given software test runs. Our Wilcoxon and Kruskal-Wallis results suggest that each of our models perform differently. We plotted the relative errors of evaluation for each model predicting energy consumption sorted by mean performance in Figure 5.1a, Figure 5.1b, Figure 5.1c, and Figure 5.1d. These figures show that the time series and *1step* models perform similarly to each other. For example, as shown in Figure 5.1a there is no well defined partition which divides the models into time series or *1step* models. Furthermore, Figure 5.1b shows that the time series and time-summarized models are competitive with one another. As, the *1step ridge regressor* model can outperform several time series models, but it is outperformed by the time series *lstm2sigm* model. The *1step ridge regressor* model is our baseline for comparison because it represents the state-of-the-art software energy prediction model *GreenOracle* [16]. It is surprising that 4 of the time-summarized models outperform both the *1step ridge regressor* and the *lstm2sigm* models as outlined in Figure 5.1b. The Wilcoxon rank sum test shows that there is a significant difference ($p < 2.2e-16$), one shot models have slightly more mean relative error than timeseries models (95% CI $[0.0342, 0.0400]$), but Cliff's delta (0.119) suggests this 3% to 4% difference is negligible. Model performance depends on the feature set but timeseries models do slightly better.

6.2 RQ2: Do time series models with built-in state perform better than time series models without state when predicting energy consumption?

The answer is Yes. Figure 5.1a and Figure 5.1b show that the median error of the LSTM models is very low for most of the models. In the Figure 5.1a, 4 of the top 5 models are LSTMs. In Figure 5.1b, with *GreenOracle* feature set, 1 of the top 5 models is an LSTM. The best LSTM-based model in Figure 5.1b has a median performance of 12% error. Prior work showed that a median performance of roughly 14% was expected for 6 mobile applications [16]. Therefore, we conclude that the LSTM models are competitive with the state-of-the-art. The stateful models are outperformed by *1step* models, or by lasso and ridge regression when considering fewer features since the LSTMs are unable to do their own feature selection. On the full feature set, the Wilcoxon rank sum test shows that there is a significant difference ($p < 2.2e - 16$), stateless models have more mean relative error than stateful LSTM models (95% CI $[0.101, 0.108]$), and a medium effect size (Cliff's delta 0.376).

We also select two random examples to show how our models perform in predicting the actual energy time series. Do they maintain similar shape and scale as the ground truths? Shown in Figure 4.2b is a stateless time series SVR model predicting the denormalized energy consumption of a given test run. This model's predictions do not fit the observed data well. On the other hand Figure 4.2a shows a stateful-time series LSTM model predicting the energy consumption of a given test run. The LSTM is capable of achieving a better fit to the shape of the observed data with its denormalized energy predictions. The SVR model does well in the aggregate rather than shape. Whereas, the LSTM model is capable of predicting the fit in some runs, but can miss-predict when the energy consumption of an application is going to change significantly. Figure 4.2a demonstrates that a developer using the *lstm2sigm* model could predict their energy consumption for the given test run and attribute the energy consumption to a part of that test run.

Figure 5.2a, Figure 5.2b, Figure 5.2c, and Figure 5.2d show the mean relative energy prediction error per time slot. In all the feature sets, except the syscall feature set, the stateful models perform better when predicting the energy consumption of each time slot than stateless models. Figure 5.3a, Figure 5.3b, Figure 5.3c, and Figure 5.3d show the dynamic time warping distances of the predicted and observed time series models. These figures show that the LSTM models have better relative performance when predicting the shape of the time series compared to the stateless models. The per time slot relative error figures and the dynamic time warping figures support the idea that the stateful time series models perform better than the stateless time series models when predicting energy consumption. Both sets of figures provide evidence that the stateful models are better at predicting the shapes of the timeseries under test compared to the stateless models.

6.3 RQ3: Energy prediction models can be trained with different feature sets. How does performance change when using them individually versus together?

Models with multiple kinds of features tend to perform better. Two sources of software behaviour are collected from `strace` and process summary information from *procfs* throughout the software tests. Figure 5.1c shows the performance of models trained only using the collected system call features and Figure 5.1d shows the performance of models only trained on the *procfs* features. In these figures it is clear that most of the models have a median error near 20%. Whereas, in Figure 5.1b and Figure 5.1a most of the models have median errors below 20%. The Wilcoxon rank sum test shows that there is a significant difference ($p < 2.2e - 16$), models trained on smaller feature sets tend to have more mean relative error than the full feature set models (95% CI $[0.00712, 0.0103]$) with negligible effect size (Cliff's delta 0.0289). When we compare *procfs* features to a full feature set the model behave significantly differently (95% CI $[0.0438, 0.0489]$) with nearly a 4.5% difference, although the effect size is negligible (Cliff's Delta 0.131). This suggests there is a difference but across all models it is a 1% difference or less. The full set of features perform better than any subset. CPU/*procfs* features alone are not enough. own.

6.4 RQ4: Do shallow multi-layer-perceptrons perform similarly to existing stateless models such as those based on linear regression?

We expected that the single and shallow layer MLP models would perform similarly to the linear or ridge regression models. However, in most cases the MLP models performed very poorly on the learning task. Lasso and ridge regression models performed well on the system-call only and process-only feature sets. The Wilcoxon rank sum test shows that there is a significant difference ($p < 2.2e - 16$), MLP models have more mean relative error than linear models (95% CI $[0.377, 0.397]$) with a medium effect size (Cliff's δ 0.387).

6.5 RQ5: How do the time series model predictions fit the shape of the observed energy consumption data?

An additional metric used to measure the quality of the time series model predictions, beyond total energy prediction, is whether the predictions follow the same trends as the observed values over time. To measure how well our predictions fit the trends of the energy consumption in the time series, we calculate how much it costs to align our predictions with the observed values. Calculating the alignment cost of our predictions with the observations is accomplished by using the dynamic time warping algorithm with an absolute distance measure.

Figures 5.3a, 5.3b, 5.3c, and 5.3d show the log-scaled DTW alignment costs for our time series models under each of our studied feature sets. In each of the figures, the stateful time series models have the best scores. It is notable that in Figure 5.3a that 10 of the 12 models have similar medians and overlapping first and third quartiles. Whereas, in the other 3 Figures the LSTM based models have lower alignment costs and do not have overlapping quartiles with 6 or more of the 12 models. Therefore, the other 3 Figures show that the LSTM models perform significantly better than the half of the tested timeseries model.

Figures 5.4a, 5.4b, 5.3c, and 5.3d show that the LSTM models achieve DTW alignment costs similar to each other under the Cliff's Delta evaluation. The figures also show that there is at least

1 pair of models in each set of LSTMs that are not significantly different from one another under the Wilcoxon test. The LSTM models appear to be significantly different than the nonLSTM based models included in the evaluation. To confirm this we calculate 95% confidence intervals on the mean alignment cost differences of the LSTM and non-LSTM models for each feature set: Full feature set (95% CI [-23.87574, -15.25068]); GreenOracle feature set (95% CI[-67.14663, -42.83967]); Syscall feature set (95% CI[-9.070578, -7.247535]); Proc feature set (95% CI[-49.13488, -38.52599]).

Figure 4.2a shows an example of a stateful timeseries model predicting the energy consumption of a software test based on the GreenOracle feature set. Figure 4.2b shows another timeseries prediction based on a stateless model trained on the GreenOracle feature set. The predictions in Figure 4.2a are more similar to the observation, and would achieve a better DTW cost, than the predictions in Figure 4.2b.

6.6 Threats to Validity

Construct validity is threatened by energy and software behaviour measurements. Internal validity is threatened by timeseries alignment, synchronization, and interpolation. External validity is threatened by the small number of applications used to produce the models (5 apps per fold, 6 in total), this is balanced by using multiple versions of each application.

The external validity is also threatened by the software tests used to exercise the applications for machine learning feature and label collection. The threat is caused from researchers generating tests that would represent how an end user would interact with the application. Because, the tests did not study or validate how a user would interact with the application. Therefore, these *representative* tests are not gauranteed to represent a users behaviour and may introduce bias into the model training. Chowdhury *et al.* have addressed this problem in work completed after our experiments with `GreenScaler` [14]. Chowdhury's work provides an unbiased method for generating exploratory Android software tests that exercise energy consuming components of the applications under test.

The *GreenMiner* does not suffer from the threat of battery degredation, because it uses power supplies and does not use batteries in its mobile devices.

Chapter 7

Conclusion

In summary, we explored software energy consumption prediction using time series regression models. Our work evaluated and compared several existing state of the art machine learning based energy prediction methods. Deep learning based models based on LSTMs did well but often simpler linear regression models performed just as well. LSTM based models had best median relative error on the full feature set.

Time series based models were accurate per step and cumulatively across the whole test run. We found that machine learning based time series models and *1step* models perform similarly when predicting the total energy consumption of a software test run. We found evidence that stateful time series models like LSTMs who maintained state/memory about the past, predicted energy consumption better than stateless models, like SVR. We found that shallow MLPs performed poorly in comparison to the lasso and ridge regression models when predicting energy consumption for a given test run. We also found that LSTM model predictions fit the shape of observations better than other models based on model comparison using the alignment cost of dynamic time warping with an absolute distance metric; this shows that the LSTM models predicted more representative time series than the other models we investigated.

Thus if a trained model is distributed to developers [1], tools need only to record output *strace* and *procfs* output in order to estimate the energy consumption of their application as it runs without the need for external hardware measurement. Depending on the complexity of the application, the developer can apply the trained models to predict their applications energy consumption over time.

[1]Download our models here https://archive.org/details/deep_green.tar

The developer can then look up what code ran in each predicted energy time slot with the associated strace log that was collected to run the model. This enables the developer to associate the energy consumption of their software with the code that ran.

7.1 Future Work

Android developers have access to Android device emulators. Accessibility to emulators allows developers to test and develop their code. Because of the accessibility to Android emulators, it would be interesting to determine whether or not our generated time series models can run on emulators to predict Android application energy consumption. A researcher would have to check if Android emulators impose limitations on the *syscalls* that an encapsulated application can generate, as well the emulator would have to be checked if it simulates *procfs* or imposes limitations on the process file system. The ability to predict energy consumption with a device emulator would permit developers to set software energy consumption development goals and to continuously measure their software's energy consumption as it is developed with little to no extra development cost. Further motivation exists to use emulators to predict software energy consumption due to the costly nature of hardware mobile devices; hardware energy measurement devices; and energy measurement expertise to configure the mobile devices with measurement devices.

New Android devices are introduced annually and device users can use their device for multiple years or purchase a new one as needed. Developers also have to account for multiple types of devices as they churn out new software updates to accomodate new screens, faster processors, and other hardware upgrades. Therefore, constructing energy models that work across multiple hardware devices would be interesting. Since, developers need to ensure that their new software developed features do not cause energy bugs on the different platforms that their software is going to run. The theoretical benefit to this work would be that we could model the energy consumption various hardware components across different devices. We could also emulate hardware components with component-specific energy models to determine how much energy may be spent with given Android programs.

The timeseries models constructed in our work were limited by the number of times the testbed could sample the energy consumption of the software running on devices. The sampling rate of energy consumption limits how accurately our trained models can predict the energy consumption of the running software. If we were able to upgrade the testbed used to sample energy consumption we could construct software models that are more accurate at predicting software energy consumption. For example, if we can measure energy consumption at a 30 Hz, then we can only predict energy consumption for 0.03 seconds time steps – therefore, if software method ran in 0.01 seconds we could not determine the energy consumption of this specific routine. Instead, we would only be able to determine the total energy consumption of all software that ran within that 0.03 second period. From the example, we can see that having a high precision energy consumption sampling device would allow use to determine the energy consumption of software methods. e.g. There is currently a 20GHz oscilliscope that is designed to measure computer processors and it supports 4 input channels. Such a device would enable us to construct software models that can predict the energy consumption of high-level Android functions as well as low-level Android byte-code. Being able to see what code is using how much energy would help developers plan how to better optimize their code to meet energy consumption constraints.

The Android RunTime (ART) environment adapts itself to how a user is interacting with their software. For instance, functions can be optimized at run time to improve the runtime speed of code that is run more frequently. However, this heuristic is unable to account for the runtime energy consumption of the functions that are running on the device. Therefore, energy expensive functions that are run less frequently may not be optimized to save battery life on the user's device. It would be interesting to determine if integrating runtime energy prediction LSTM models with ART allows device users to save energy while interacting with their devices. If integrating energy consumption awareness into ART enable software to be adapted/optimized more efficiently to users then this would give device users better energy savings and cost developers little.

Bibliography

[1] Why your gadgets' batteries degrade over time. https://www.popularmechanics.com/technology/gadgets/how-to/a7432/why-your-gadgets-batteries-degrade-over-time-6705747/. Accessed: 2018-06-01.

[2] iOS Application Programming Guide: Tuning for Performance and Responsiveness. http://ur1.ca/696vh, 2010.

[3] *strace(1) General Commands Manual*, 03 2010.

[4] *proc(2) Linux Programmer's Manual*, 4.04 edition, January 2015.

[5] *syscalls(2) Linux Programmer's Manual*, 4.04 edition, January 2015.

[6] Karan Aggarwal, Chenlei Zhang, Joshua Charles Campbell, Abram Hindle, and Eleni Stroulia. The power of system call traces: Predicting the software energy consumption impact of changes. In *Proceedings of 24th Annual International Conference on Computer Science and Software Engineering*, pages 219–233. IBM Corp., 2014.

[7] Raja Wasim Ahmad, Abdullah Gani, Siti Hafizah Ab Hamid, Feng Xia, and Muhammad Shiraz. A review on mobile application energy profiling: Taxonomy, state-of-the-art, and open research issues. *Journal of Network and Computer Applications*, 58:42–59, 2015.

[8] Alliance to Save Energy. PC Energy Report 2007: United States, 2007.

[9] Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Erich Elsen, Jesse Engel, Linxi Fan, Christopher Fougner, Tony Han, Awni Y. Hannun, Billy Jun, Patrick LeGresley, Libby Lin, Sharan Narang, Andrew Y. Ng, Sherjil Ozair, Ryan Prenger, Jonathan Raiman, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Yi Wang, Zhiqian Wang, Chong Wang, Bo Xiao, Dani Yogatama, Jun Zhan, and Zhenyao Zhu. Deep speech 2: End-to-end speech recognition in english and mandarin. *arXiv e-prints*, abs/1512.02595, 2015. Preprint.

[10] Nadine Amsel and Bill Tomlinson. Green tracker: a tool for estimating the energy consumption of software. In *Proceedings of the 28th of the international conference extended abstracts on Human factors in computing systems*, pages 3337–3342. ACM, 2010.

[11] Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay, and Abhik Roychoudhury. Detecting energy bugs and hotspots in mobile apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 588–598. ACM, 2014.

[12] Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, pages 21–21. USENIX Association, 2010.

[13] François Chollet. Keras. https://github.com/fchollet/keras, 2015.

[14] Shaiful Chowdhury, Stephanie Borle, Stephen Romansky, and Abram Hindle. Greenscaler: training software energy models with automatic test generation. *Empirical Software Engineering*, pages 1–44, 2018.

[15] Shaiful Alam Chowdhury, Stephanie Gil, Stephen Romansky, and Abram Hindle. Greenscaler: Automatically training software energy model with big data. Technical report, PeerJ Preprints, 2016.

[16] Shaiful Alam Chowdhury and Abram Hindle. Greenoracle: Estimating software energy consumption with energy measurement corpora. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 49–60, New York, NY, USA, 2016. ACM.

[17] Shaiful Alam Chowdhury, Varun Sapra, and Abram Hindle. Client-side energy efficiency of http/2 for web and mobile app developers. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 529–540. IEEE, 2016.

[18] David Chu, Aman Kansal, Jie Liu, and Feng Zhao. Mobile apps: It's time to move up to condos. In *HotOS*, 2011.

[19] Dario Di Nucci, Fabio Palomba, Antonio Prota, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. Software-based energy profiling of android apps: Simple, efficient and reliable? In *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*, pages 103–114. IEEE, 2017.

[20] Mian Dong and Lin Zhong. Self-constructive high-rate system energy modeling for battery-powered mobile systems. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 335–348. ACM, 2011.

[21] Steve M Easterbrook. Climate change: a grand software challenge. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 99–104. ACM, 2010.

[22] Carla Schlatter Ellis. The case for higher-level power management. In *Hot Topics in Operating Systems, 1999. Proceedings of the Seventh Workshop on*, pages 162–167. IEEE, 1999.

[23] Yunsi Fei, Srivaths Ravi, Anand Raghunathan, and Niraj K Jha. Energy-optimizing source code transformations for operating system-driven embedded software. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(1):2, 2007.

[24] Jason Flinn and M Satyanarayanan. Powerscope: A tool for profiling the energy usage of mobile applications. In *Mobile Computing Systems and Applications, 1999. Proceedings. WMCSA'99. Second IEEE Workshop on*, pages 2–10. IEEE, 1999.

[25] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2001.

[26] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2016.

[27] Kris Gray. Performance Considerations for Windows Phone 7. 2010.

[28] Paul M Greenawalt. Modeling power management for hard disks. In *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 1994., MASCOTS'94., Proceedings of the Second International Workshop on*, pages 62–66. IEEE, 1994.

[29] Ashish Gupta, Thomas Zimmermann, Christian Bird, Nachiappan Nagappan, Thirumalesh Bhat, and Syed Emran. Detecting Energy Patterns in Software Development. Technical Report MSR-TR-2011-106, Technical report, Microsoft Research, 2011.

[30] Sudhanva Gurumurthi, Anand Sivasubramaniam, Mary Jane Irwin, Narayanan Vijaykrishnan, and Mahmut Kandemir. Using complete machine simulation for software power estimation: The softwatt approach. In *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*, pages 141–150. IEEE, 2002.

[31] Awni Y. Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and Andrew Y. Ng. Deep Speech: Scaling up end-to-end speech recognition. Preprint, 2014.

[32] Shuai Hao, Ding Li, William G. J. Halfond, and Ramesh Govindan. Estimating Mobile Application Energy Consumption Using Program Analysis. In *ICSE '13*, pages 92–101, 2013.

[33] Abram Hindle. Green mining: A methodology of relating software change to power consumption. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, pages 78–87. IEEE, 2012.

[34] Abram Hindle. Green mining: Investigating power consumption across versions. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 1301–1304. IEEE, 2012.

[35] Abram Hindle, Alex Wilson, Kent Rasmussen, E. Jed Barlow, Joshua Charles Campbell, and Stephen Romansky. Greenminer: A hardware based mining software repositories software energy consumption framework. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 12–21, New York, NY, USA, 2014. ACM.

[36] Abram Hindle, Alex Wilson, Kent Rasmussen, E. Jed Barlow, Joshua Charles Campbell, and Stephen Romansky. Greenminer: A hardware based mining software repositories software energy consumption framework. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 12–21, New York, NY, USA, 2014. ACM.

[37] Intel. LessWatts.org - Saving Power on Intel systems with Linux. http://www.lesswatts.org, 2011.

[38] Eamonn J Keogh and Michael J Pazzani. Derivative dynamic time warping. In *Proceedings of the 2001 SIAM International Conference on Data Mining*, pages 1–11. SIAM, 2001.

[39] M Larabel. Ubuntu's power consumption tested. http://www.phoronix.com/scan.php?page=article&item=878, 2007.

[40] Emanuele Lattanzi, Andrea Acquaviva, and Alessandro Bogliolo. Run-time software monitor of the power consumption of wireless network interface cards. *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*, pages 352–361, 2004.

[41] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[42] D. Li, S. Hao, J. Gui, and W. G. J. Halfond. An empirical study of the energy consumption of android applications. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 121–130, Sept 2014.

[43] Ding Li, Shuai Hao, William GJ Halfond, and Ramesh Govindan. Calculating source line level energy information for android applications. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 78–89. ACM, 2013.

[44] Ding Li, Angelica Huyen Tran, and William G. J. Halfond. Making Web Applications More Energy Efficient for OLED Smartphones. In *ICSE 2014*, pages 527–538, Hyderabad, India, June 2014.

[45] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. 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, pages 2–11, New York, NY, USA, 2014. ACM.

[46] Mario Linares-Vásquez, Gabriele Bavota, Carlos Eduardo Bernal Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. 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, pages 143–154, New York, NY, USA, 2015. ACM.

[47] Meinard Müller. Dynamic time warping. *Information retrieval for music and motion*, pages 69–84, 2007.

[48] San Murugesan. Harnessing green IT: Principles and practices. *IT professional*, 10(1):24–33, 2008.

[49] Abhinav Pathak, Y Charlie Hu, and Ming Zhang. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 29–42. ACM, 2012.

[50] Abhinav Pathak, Y Charlie Hu, Ming Zhang, Paramvir Bahl, and Yi-Min Wang. Fine-grained power modeling for smartphones using system call tracing. In *Proceedings of the sixth conference on Computer systems*, pages 153–168. ACM, 2011.

[51] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[52] pierre rouanet. Dynamic time warping python module version 1.2. https://github.com/pierre-rouanet/dtw, 2015.

[53] Gustavo Pinto, Fernando Castor, and Yu David Liu. Mining questions about software energy consumption. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 22–31. ACM, 2014.

[54] Veselin Raychev, Martin Vechev, and Eran Yahav. Code Completion with Statistical Language Models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'14, pages 419–428. ACM, 2014.

[55] Andrew Colin Rice and Simon Hay. Decomposing power measurements for mobile devices. In *PerCom*, volume 10, pages 70–78, 2010.

[56] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[57] B Steigerwald and ABHISHEK Agrawal. Developing Green Software. Technical report, Technical report, Intel Corporation, 2011.

[58] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[59] Vivek Tiwari, Sharad Malik, Andrew Wolfe, and Mike Tien-Chien Lee. Instruction level power analysis and optimization of software. *The Journal of VLSI Signal Processing*, 13(2):223–238, 1996.

[60] Narseo Vallina-Rodriguez and Jon Crowcroft. Erdos: achieving energy savings in mobile os. In *Proceedings of the sixth international workshop on MobiArch*, pages 37–42. ACM, 2011.

[61] Narseo Vallina-Rodriguez and Jon Crowcroft. Energy management techniques in modern mobile handsets. *IEEE Communications Surveys & Tutorials*, 15(1):179–198, 2013.

[62] M. White, C. Vendome, M. Linares-Vasquez, and D. Poshyvanyk. Toward Deep Learning Software Repositories. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 334–345, 2015.

[63] Xin Ye, Hui Shen, Xiao Ma, Razvan Bunescu, and Chang Liu. From word embeddings to document similarities for improved information retrieval in software engineering. In *Proceedings of the 38th International Conference on Software Engineering*, pages 404–415. ACM, 2016.

[64] Lide Zhang, Birjodh Tiwana, Robert P Dick, Zhiyun Qian, Z Morley Mao, Zhaoguang Wang, and Lei Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP International Conference on*, pages 105–114. IEEE, 2010.

[65] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P Dick, Zhuoqing Morley Mao, and Lei Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 105–114. ACM, 2010.