An approach to Machine Learning with Big Data

Ella Peltonen

Master's Thesis
University of Helsinki
Department of Computer Science

Helsinki, September 19, 2013

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Ella Peltonen

Työn nimi — Arbetets titel — Title

An approach to Machine Learning with Big Data

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's Thesis	September 19, 2013	65

Tiivistelmä — Referat — Abstract

Cloud computing offers important resources, performance, and services nowadays when it has became popular to collect, store and analyze large data sets. This thesis builds on Berkeley Data Analysis Stack (BDAS) as a cloud computing environment designed for Big Data handling and analysis. Especially two parts of the BDAS, the cluster resource manager Mesos and the distribution manager Spark will be introduced. They offer important features, such as efficiency, multi-tenancy, and fault tolerance, for cloud computing. The Spark system expands MapReduce, the well-known cloud computing paradigm.

Machine learning algorithms can predict trends and anomalies of large data sets. This thesis will present one of them, a distributed decision tree algorithm, implemented on the Spark system. As an example case, the decision tree will be used on the versatile energy consumption data from mobile devices, such as smart phones and tablets, of the Carat project. The data consists of information about the usage of the device, such as which applications have been running, network connections, battery temperatures, and screen brightness, for example.

The decision tree aims to find chains of data features that might lead to energy consumption anomalies. Results of the analysis can be used to advise users on how to improve their battery life. This thesis will present selected analysis results together with advantages and disadvantages of the decision tree analysis.

ACM Computing Classification System (CCS):
Networks → Cloud computing
Theory of computation → MapReduce algorithms
Information systems → Mobile information processing systems

Avainsanat — Nyckelord — Keywords
Data Analysis, Machine Learning, Cloud Computing, Big Data

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Contents

1 Introduction

Nowadays, many of corporations, companies and organizations can gather gigabytes or even terabytes of data from their customers and applications. Data can include various information, for example, which products have been searched and purchased from online stores, how location or battery state of mobile phones has changed, or which pictures have been uploaded to the Internet by customers. These masses of data need to be analyzed and processed to information and towards new applications. Despite the contents of each data set, many analyzing frameworks and softwares can be very general in purpose; their common challenge is how to handle large amounts of data safely, reliably, and with sufficient performance.

Some super computers can load large amounts of data to their memory, but in many cases distribution offers better solutions to handle large data sets. Shared computing load enables scheduled and structured models: computers can specialize and relocate operations among themselves, and take responsibility for fault tolerance in common.

In virtual cluster or cloud based computing, called simply cluster computing in this thesis, the cluster itself manages things such as security, reliability, scalability, and performance [12]. Then the analysis part is possible to separate to its own abstraction layer. Figure 1 presents three layers over a cluster operating system.

The first layer, called *cluster resources*, is a platform for a virtual cluster architecture. It manages connections and communication between computing nodes, administration operations, possible joins and removals of nodes, file system accesses, and other resource allocations [27].

Over the cluster resource layer there is a middle layer, *computing manager*, which is the actual distribution layer. It is responsible for different computation jobs and it allocates these jobs for the nodes via the cluster resource layer below [27, 37]. One reason to separate these two layers is to separate abstract distribution logic and an architecture-related clustering system. In this way, it is possible to use the same distribution frameworks in different cluster architectures and, vice versa, a cluster can offer its services to different kinds of distribution frameworks.

Analysis software has also been separated to its own layer. There lies all the data and application-specific operations. Data analysis software can

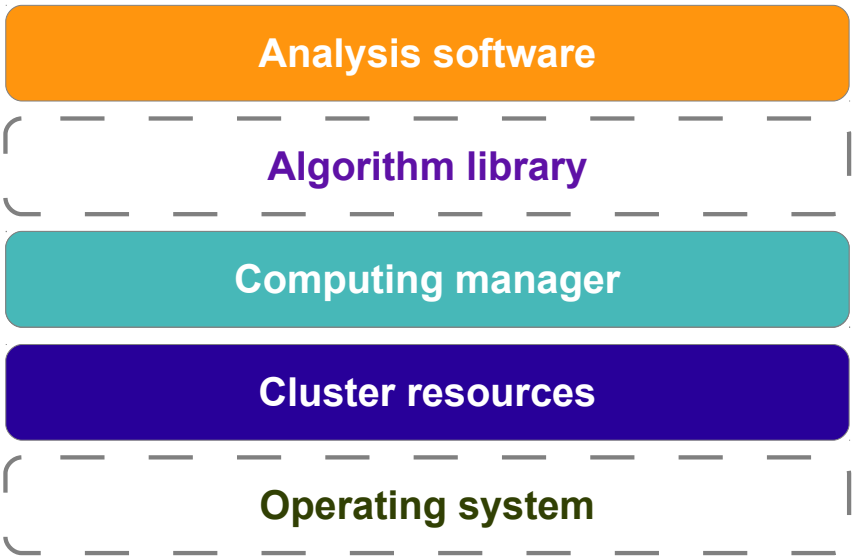

Figure 1: General cluster based computing stack. The abstraction helps to design diverse and flexible data analysis system where each layer has its own responsibilities.

run functions of API offered by the computing manager layer. This helps for developing and relocating different data analysis programs. If available, the analysis program can also benefit from a generally purposed algorithm library developed for distributed computing.

Data analysis considers a large range of different algorithms related to data mining, machine learning, and statistical analysis. When these methods and algorithms have been implemented as a distributed system, such as cloud or cluster of multiple servers, there will be several aspects to take into account. The algorithms might not to be effective when the data is shared with hundreds or even thousands of nodes, for example. All the programming operations will not be reasonable or even possible to implement for a distributed computing system. In this thesis, it is notable that the algorithms themselves will stay centralized, with one output and one controller, only the computing process and the data have been shared between nodes of the distributed system.

This thesis will present some solutions pertaining to distributed Big Data analysis. One of the most popular approaches is the MapReduce paradigm developed by Google and first published in 2004 [16]. From the

academic field, this thesis will focus on Berkeley Data Analysis Stack (BDAS) [5], developed by AMPLab of UC Berkeley after 2010 for expanding the MapReduce paradigm. This thesis will introduce both of these systems and also give a brief overview to the larger field of the distributed data analysis.

Machine learning algorithms are often a very important part of any analysis and data mining tools. Therefore this thesis will present some ideas to implementing machine learning algorithms especially over the BDAS system. As an example, this thesis will present a distributed decision tree for the BDAS system related to a collaborative energy diagnosis project called Carat [6, 31, 32]. Carat has been developed by UC Berkeley and University of Helsinki. Its data consists of energy consumption information of mobile devices from more than half a million devices that submit about half a million data items per week.

Carat data offers information about, for example, battery power, battery health and temperatures, charging durations, network connections, and running applications. The main idea is to find feature chains that might predict particular energy consumption behavior, where a feature is a property of the data with a specific value. For example, if a mobile network is connected and the battery temperature is very high, this combination of features might lead to high energy consumption. The decision tree is one possible solution to find such feature chains.

The thesis has been divided to three main parts: background, introduction to BDAS, and an implementation example. Section 2 will present the background of the Big Data analysis field: basics of cloud computing environments, the MapReduce paradigm and the meaning of distributed machine learning algorithms in the Big Data analysis cases. Section 3 will present the Berkeley Data Analysis Stack as an architecture and two of its layers in detail, cluster resource sharing system Mesos [21] and distribution manager Spark [37]. Section 4 will present the Carat data analysis system and the decision tree algorithm. Section 5 is a description of an implementation of the BDAS decision tree for the Carat data. Section 6 will present results of the decision tree analysis. Finally, Section 7 discusses lessons learned and Section 8 for concludes the thesis.

2 Background

This section presents background of the analysis of large data sets, the distributed computing and the the cloud computing environments. A research trend of the 21st century has been to solve how to combine the knowledge of the virtualized computing environments and Big Data analysis [26]. These areas are well described in the literature, but the purpose of this Section is to expose the main ideas, definitions and subsections behind the distributed Big Data analysis.

Section 2.1 presents definitions for the Big Data. Section 2.2 describes cloud computing environments and services in data analysis point of view. Section 2.3 presents the MapReduce paradigm for distributed data analysis. Section 2.4 focuses on how the Big Data, cloud computing and MapReduce based paradigms has been combined to the distributed data analysis and remarks shortly some ready distributed machine learning libraries and techniques presented in the literature.

2.1 Big Data

"Big Data" is not a well-defined term even though it is widely used. Even its popularity might cause the definition problems: everyone wants to use the fashion terms. How big should the data be to be honored as Big Data?

Ji et al. [26] have gathered some definitions that all have at least one common factor: Big Data is something that is very hard or even impossible to handle with traditional and current management tools such as databases and computing environments. Also the time complexity aspect has been taken into account: Big Data is easily "so big" that its processing in any possible way takes time and performance effectiveness.

When datastores grow and computing environments get faster and more high-powered, it becomes more difficult to give a specific answer to the question "How big is big?" In some cases, even a data set of some gigabytes might be too big to be managed with the current storage or analysis tools. Despite this, there are the data sets of more than terabytes in the world.

Organizations and companies can see the value of Big Data in several ways [29]. Information from customers' behavior can support the creation of new products, services, and business models. Customer segments based on data analysis, clustering for example, can help target the right services to

those who need them. New ways to monetize data are being developed all the time.

The data works also as a training set for learning algorithms and computer-supported decision making. The digitized data could be shared and stored to multiple places, and for use of multiple users. Digitization has also emerged the questions about the data accessibility, privacy, and security – as well as issues of legality – which are still a challenge in the Big Data area [26].

The computing paradigms and environments should be suitable for managing huge amounts of data with diverse of file types and resources [15, 26]. This thesis presents MapReduce based solutions, especially the Spark system [37] of BDAS, as one possibility to solve the paradigm question, and cloud computing as an answer for the problem of environments.

2.2 Clusters and cloud computing environments

Cloud computing is based on hardware clusters and *grids*. A grid is a cluster where a group of distributed computers operate together as a network for mutual computation. The grid is more sophisticated and efficient solution than just one high-performance computer, but it does not have the benefits of virtualization: scalability, resource sharing, and mobility.

A *cloud* is typically a cluster, where resource sharing and runtime computing have been organized in a more or less virtualized way. Foster et al. [19] name four requirements that complete and specialize the cloud as a distributed computing paradigm:

1. The cloud is more scalable than traditional systems such as grids.

2. The cloud could be presented as an abstraction of different services it offers; also, a *service* is an important keyword in the cloud computing area.

3. One advantage of virtualization is its lower cost when compared to grids or supercomputers; anyone with a credit card can buy a part of a cloud without expensive hardware purchases.

4. The cloud is virtually configured, so it is possible to start, remove and reallocate jobs in the cloud without any interest of underlying hardware.

	Grid	Cloud
Architecture	Integrate hardware resources and operation systems via network	Integrate different resources via standard protocols of Internet
Security model	Administration domain, multiple security issues	User accounts are modifiable by web forms, simple to use
Business model	A user has a pre-ordered number of hours or bytes in use	A user pays on consumption basis, e.g., per instance hour consumed, bytes of storage used or data transfered
Programming model	Environment specific	Environment specific or PaaS service applications
Virtualization	Limited, e.g., virtual workspaces	Offers an illusion of a single computing interface
Compute model	Jobs are queued by resource manager	Resources are shared by users at the same time
Applications	High performance computing, different kind of applications	Interactive and transaction-oriented computing, also multiple set of possible applications

Table 1: Some main differences between grid and cloud computing by Foster et al. [19]

Table 1 represents some main differences between grid and cloud computing as Foster et al. have defined [19].

Figure 2 presents example elements of the data analysis cloud. The cloud is based on hardware resources. The relationship between hardware and the cloud depends on the organization model of the hardware layer infrastructure. The cloud works as an environment for the different kind of virtual machines and virtual resources, for example, shared file systems and data storages. In most of the data analysis systems, virtual machines have been organized as a network of a controller node and a set of worker nodes. The controller is responsible for job sharing and communication between the cloud and clients of which there may be several. The worker nodes run the actual computing jobs and return the results to the controller.

Lin et al. [27] present three different organizations for ordering the

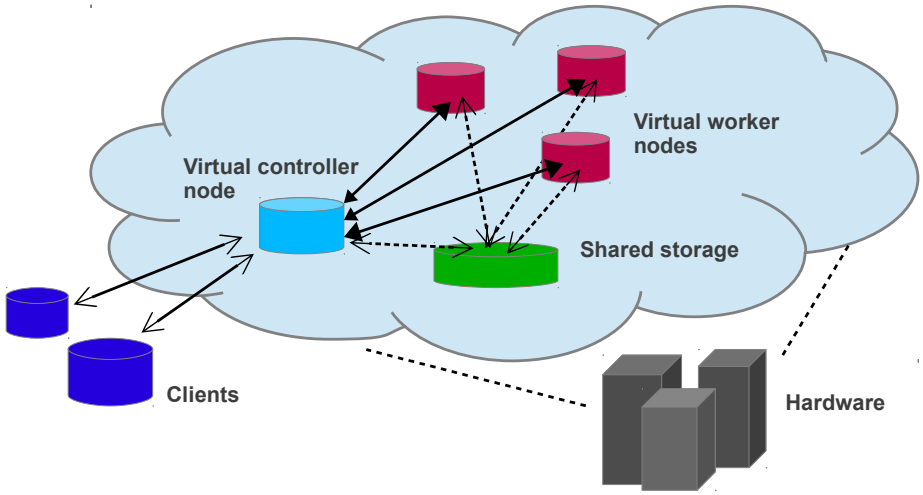

Figure 2: A simple cloud architecture for data analysis scenarios. Placement of, for example, job and tasks schedulers and managers can vary.

cloud over the hardware machines: dedicated, consolidated and hybrid organization. Figure 3 presents an example of these organizations. Their main difference is how independent the applications are of each other. A dedicated organization gives to each application its own infrastructure and responsibility over resources. A consolidated organization involves a management system in cluster resources layer, which globally coordinates and controls all the applications, their computing environments and required resources. A hybrid organization is a collection of orders where some of the applications has their own hardware resources and some of the application are sharing the resources by a cluster management system.

The dedicated organization works well if there are only few and just stable applications running in the cluster, but frequently the consolidated organization is more flexible and adaptable to different and possible variable situations. A significant disadvantage of the consolidated organization is its increased need for scheduling, controlling, decision making and fairness policies. In Section 3, this thesis will present one consolidated cluster system, Apache Mesos [21] that is part of the Berkeley Data Analysis Stack (BDAS). Mesos enables running multiple jobs over it, for example, both of the Spark and Hadoop instances.

In cloud computing, there are frequently used terms and their acronyms

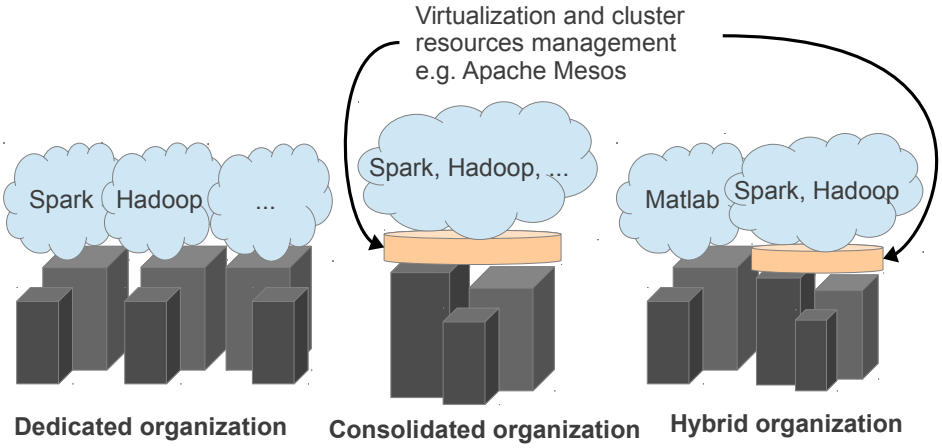

Figure 3: A comparison between the dedicated, consolidated and hybrid cluster organizations [27] with example applications. The main difference is the middleware layer that takes care of, for example, resource managing, data accessing, job scheduling, and load balancing

for different kinds of services the cloud can offer: an Infrastructure as a Service (IaaS), a Platform as a Service (PaaS), and a Software as a Service (SaaS) [30]. These terms are used for describing the cluster organization from the user's point of view.

Figure 4 shows relations of the different services. Infrastructures such as clusters and isolated servers with operating systems, and platforms such as application-hosting environments, offer computing utility to software developers. These softwares, basically web applications, run in the cloud for end users or clients. The service can exploit some public database that is offering Data as a Service (DaaS) [35]. When discussing data analysis, these definitions are not the key elements, but they are useful to know.

Armbrust et al. have considered IaaS and PaaS together [12] without a significant difference and they could be handled as a lower-level services. Data analysis software could be understood as a SaaS level service. Then the SaaS user is a client or an application that exploits the analysis results. This thesis presents one such system, Carat, in Section 4. The results provided by analysis software can be also regarded as having their individual worth, for example, for scientific purposes. The definition of the SaaS requires frequently also some application for the end users [12, 30], such as a mobile

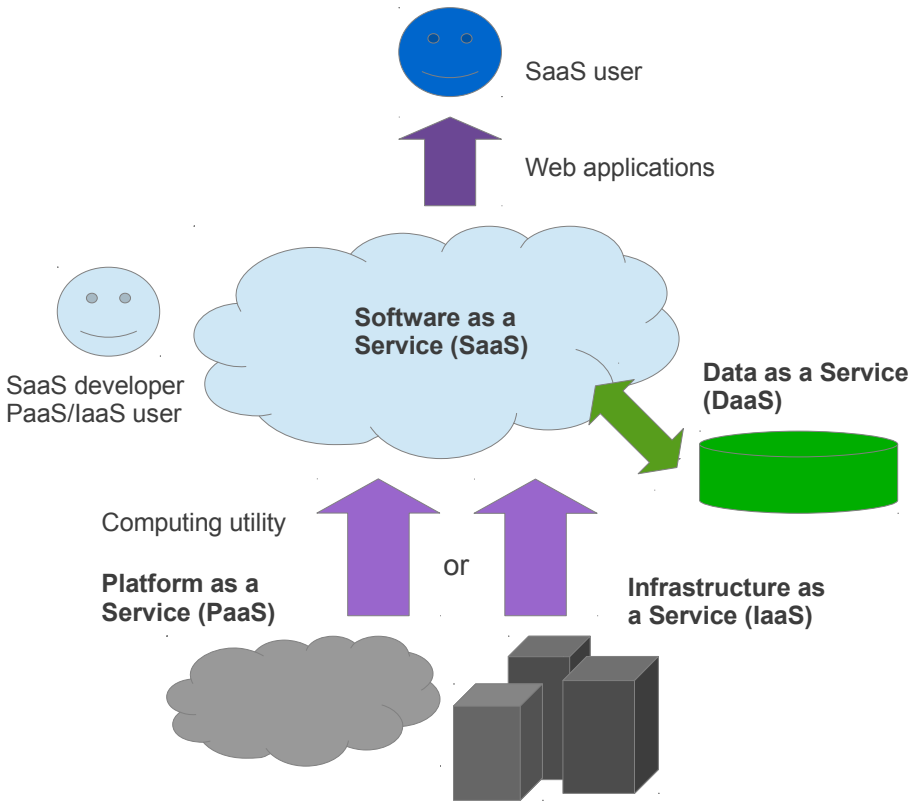

Figure 4: IaaS, PaaS, SaaS, and DaaS parts working together. For a practical example, see also the Carat system in Figure 10 in Section 4.

application that benefits the data analysis results.

Cloud computing has its requirements and challenges. Clouds have to manage large computing facilities and multiple simultaneous requests and operations similarly to grid computing [19]. Because of clouds' layered structure and transparency, resources could seem to be infinite [13], which is not true. Planning the costs of cloud computing can be difficult [13]: how to just use resources that are needed, taking into account data transmission costs, performance and scalability of the cloud environment. Data security and privacy are big issues here, also legality of sharing the data to third-party services [19, 26].

This thesis uses the term *cluster* as an umbrella term for different types of hardware and virtualization solutions. For most of the presented analysis environments, such as MapReduce Hadoop and BDAS, the cloud is the primary environment. But there are no requirements to avoid the grid as a cluster resources layer architecture if the analysis system is still usable that way. In this thesis, the example presented in Section 5 has been implemented using the cloud environments Amazon Elastic Compute Cloud (Amazon EC2) [1] and OpenStack [9] over the private cluster of University of Helsinki.

2.3 The MapReduce paradigm

MapReduce is a popular distributed computing paradigm developed by Google researchers Jeffrey Dean and Sanjay Ghemawat [16, 17, 18]. Its main idea is to concentrate all the computing operations to two functions: *map* and *reduce* which the user has to implement. Computing nodes will specialize so that one of them works as a controller or so called *master* node, and the rest are *workers* participating in the actual computation: map and reduce operations.

Several open source MapReduce implementations have been developed. Hadoop [2] is one of the most popular. Also many other implementations have been presented and Hadoop has got its next generation version, called YARN [27]. Because of versatility of the implementations, this section focuses on MapReduce as a distributed computing paradigm, which is mainly important for understanding systems such as BDAS Spark.

Figure 5 shows how map and reduce functions operate together. The map function is a single operation that is done to each element of the data set, separately and in distributed way. Data items are presented as key and

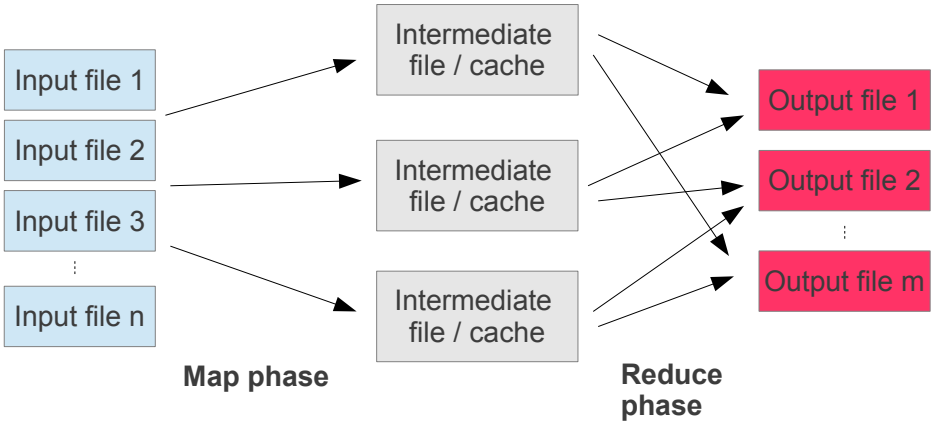

Figure 5: An iteration example from MapReduce. Worker nodes of the map phase read the input files and after the operation save intermediate files to their local disks or caches. Workers of the reduce phase use the intermediate files as their input. Reducer nodes save the final results as output files.

value pairs. Each worker node reads a split of the data items from input files and does the map, or produces another list with the modified data items:

$$map(key1, value1) \rightarrow list(key2, value2).$$

This output of the map function is stored to intermediate files and they will be stored to the local cache or disk. The reduce function reads the intermediate files and merges data items related to the same key. An output of reduce will be a list of values:

$$reduce(key2, list(value2)) \rightarrow list(value3).$$

For a simplified example, there is a list of prices $l = [5.0, 8.5, 11.25]$ related to the same item k as a key. All of them will be increased 5% and after that added together for total costs. The map function is $l_2 = l.map(\_ \times 1.05)$ and the result of the map phase is $l_2 = [5.25, 8.925, 11.8125]$. The reduce function $l_2.reduce(\_ + \_)$ to add the values to 25.9875. If these calculations are used on a list of multiple items, the reduce part would produce, for example, a sum of prices for every item and return them as a list of sums.

When iterating map and reduce phases one after the other and using a

previous output as a next input, it is possible to construct other algorithms. For example, a prominent unsupervised clustering algorithm K-means [28] is easy to implement with map and reduce phases. Algorithm 1 describes the basic K-means based on the book [34, pages 496-498]. The algorithm is initialized with a list of random centroids. A centroid means an average or a central point of each cluster, which the K-means algorithm try to find. In each step of the iteration, each data point will be assigned to the closest centroid (line 4). The algorithm will produce clusters as a set of data points for each of the centroids. After this, new centroids will be recomputed as a mean of the data points in the corresponding cluster (line 5).

Algorithm 1 Basic K-means algorithm

1: Let D be a set of data points
2: Initialize centroids as a set C of size k
3: **repeat**
4: For each data point $d \in D$ assign its nearest centroid $c \in C$
5: For each c, collect assigned data points and recompute a new c_2
6: **until** Centroids do not change

Algorithm 2 describes a K-means algorithm modification for the MapReduce paradigm. There are three parts: first to the master (a *master*() function), second to nodes in the map phase (a *map*() function), and last to nodes in the reduce phase (a *reduce*() function). Note that depending its load and the used system, each node can do both map and reduce work. The master works as a controller node that schedules jobs and collects the results. It starts each iteration when sending map requests to the mappers and takes care that the output will be used as a next input.

The K-means algorithm is divided so that the map phase assigns data points to their corresponding centroids and the reduce phase computes the mean of the cluster to the new centroid. It is also the reducer's task to collect the list of the data points related to the each centroid. One centroid will be reduced by only one reducer node – this guarantees the validity of the results.

Algorithm 2 MapReduce K-means

A master part will be ran by a controller node, functions map and reduce by worker nodes.

function master()

1: Let D be a set of data points (a, d) where a is just some key and d the data point
2: Initialize centroids as a set C of size k
3: **repeat**
4: Broadcast C to the mapper nodes
5: Divide data points to the mapper nodes and let them map
6: Receive new centroids from the reducer nodes and let this list be C
7: **until** Centroids do not change

function map(a, d)

1: for a data point d assign its nearest centroid $c \in C$
2: **return** (c, d) where the centroid c is now a key

function reduce(c, $list[d]$)

1: c_2 = mean of $list[d]$
2: **return** c_2 as a new centroid for the cluster

Depending the implementation, each reducer node will produce a list of elements related to each centroid in C.

2.4 Distributed machine learning and data analysis

Cloud computing environments and the MapReduce paradigm offer a basis for Big Data analysis. One main aim is to ensure sufficient performance and scalability for handling possible large data sets. This challenge sets requirements also for algorithms and techniques reasonable to use for analysis, not only environments and systems. One of the keywords here is distributed computing.

Machine learning techniques are an important part of any data analysis system. When computing is performed on multiple computers, for example, in the cloud between virtual machines, also the algorithm should be implemented in an appropriate way. All the methods may not even be suitable at all because of size or structure of the data [26].

MapReduce based analysis environments, such as Hadoop, and its expansion Spark presented in Section 3, are practical when each data item has been targeted with multiple, separated, and isolated operations [37]. These are easy to implement with a map-like functions. When it is necessary to compute anything through the full data set, there have to be used reduce-like functions. They require more memory and computing performance because of reading through all the data items. For iterative and runtime differences of the Spark and classic MapReduce, see Section 3.3.

There are some ready to use libraries of machine learning algorithms, which are using the MapReduce paradigm. Apache Mahout [3] is a Hadoop-based implementation that offers many algorithms for clustering, classification, and frequent itemset mining. Because Hadoop still requires implementation work, Ghoting et al. have presented SystemML [20] that proposes a higher-level language, algorithm library and performance optimizations for Hadoop jobs. In addition to the machine learning algorithms, SystemML offers statistical methods and linear algebra models for analysis use.

Kraska et al. have presented the MLBase [24] system that is also mentioned with the Berkeley Data Analysis Stack (BDAS) [5] as a part of the projects of AMPLab of UC Berkeley. MLBase offers high-level primitives and operations that help writing machine learning algorithms even without any understanding of the lower level issues such as scalability, load balancing, and data storing.

Apache Mahout, SystemML and MLBase are presented as an example of the trend to produce full libraries or higher-level languages for ease to writing

algorithms. Without taking a position on their optimizations or performance, they are hiding most of the lower level operations, forgetting the situations if developers were interested in observing the distribution system or code an algorithm of their own.

This thesis will present a decision tree algorithm implemented on BDAS Spark [37] in Section 4. Own implementations are necessary if no common libraries exist, which is frequently the situation with Spark today. Also, there can be a need for distribution or memory use management in the code level, even if there would not be any other reasons to avoid existing libraries and frameworks.

3 Berkeley Data Analysis Stack

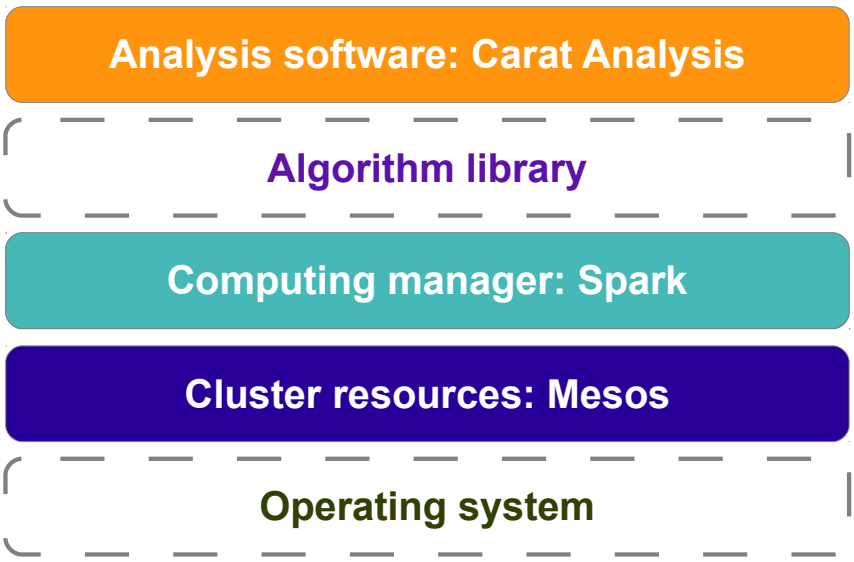

Figure 6: Berkeley Data Analysis Stack (BDAS) layers of cluster resource sharing and a computing manager, and the Carat data analysis as an user application. Compare to Figure 1.

Berkeley Data Analysis Stack (BDAS) [5] is a set of Big Data analysis software components developed by AMPLab of UC Berkeley. In May 2013 BDAS consisted of four different systems: a cluster resource manager called Mesos, a distributed in-memory file system called Tachyon, a cluster computing system called Spark, and an SQL API for data storages called Shark. Probably there will be more coming later. This thesis will focus on two of them: Mesos [21, 4] and Spark [37, 11].

Figure 6 presents how the Berkeley Data Analysis Stack has been used in this thesis. See also the earlier Figure 1 as a comparison. Atop the operating system, Mesos handles cluster resources and offers these resources to frameworks. The frameworks, such as a computing manager Spark, choose the resources they need. Spark works as a distribution interface between the cluster and the actual data analysis application implemented by user. Spark offers a distributed data structure RDD, the Resilient Distributed Dataset, and a lot of functions for data modifications, which are used by the analysis software. In the perfect model, there could also be an algorithmic library for

the flexible working with the analysis software.

Section 3.1 presents the Mesos system and Section 3.2 the Spark system. Spark will be also compared to the MapReduce on Section 3.3. The Carat data analysis software will be presented as an implementation example in Section 5.

3.1 Cluster resource manager: Mesos

BDAS Mesos is a platform for sharing and allocating the cluster resources, for example, CPU and RAM capacities of cloud participating servers. Mesos has been presented in the paper of Hindman et al. [21] in March 2011, but it has been also mentioned earlier, with its original name Nexus, in the workshop report [22] in June 2009. An open source implementation of Mesos [4] has been included in the Apache Incubator project in January 2011.

The fundamental idea of the Mesos system is to be a fine-grained cluster computing platform that allows running one or more different platforms in the same time in the same cluster. In other words, Mesos is a *multi-tenant* system. The term *framework* means an upper layer software that manages and executes computing jobs, such as Spark or MapReduce Hadoop. In the classification of Lin et al. [27], Mesos represents a consolidated cluster organization, shown in Figure 3 in Section 2.2.

The architecture of Mesos is presented in Figure 7. Mesos has one controller node, called *master*, which communicates with the frameworks. Each framework runs a *job scheduler* that schedules the jobs, which the framework should run. The scheduler sends the job, if there are enough free resources, to the master. The master splits the job to *tasks*, which it gives to the worker nodes, called *slaves*. Slaves run a process called *task executor* that performs the actual computation.

Mesos offers the cluster resources to the frameworks. The frameworks either access the resources or not, depending their current demands. This is also a fairness policy of Mesos: Mesos decides how many resources it can give, and the frameworks choose, which resources they will accept.

3.2 In-memory cluster computing: Spark

Zaharia et al. have presented the ideology of the *Resilient Distributed Datasets* (RDD) in their paper [37] published in April 2012, but also mentioned earlier

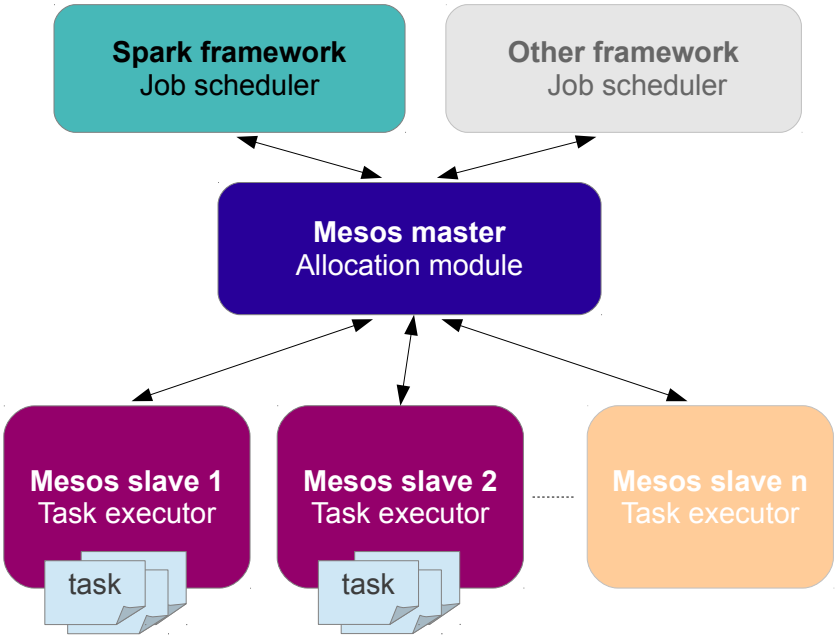

Figure 7: Mesos architecture. An upper layer framework schedules a job and gives it to the master. The master will split the job to the tasks. The master node works as a controller, which allocates the tasks to worker nodes or so called slaves.

as a technical report [36] in July 2011 and a workshop report [38] in June 2010. The Spark system [11] is an open source implementation of the RDDs. Spark can be understood as a computing framework of the distributed system just as MapReduce [16] and its free implementation Hadoop [2]. In fact, the lower layer Mesos can easily run both Spark and Hadoop jobs.

An RDD is a collection of data items. The RDD is *partitioned*, so the same RDD is parallelized to different worker machines. The RDD is *read-only*, which means it is possible to create only from other RDDs or by reading it from a file system. The RDD is only accomplished when necessary: this is called *laziness*, which is also a paradigm of the Spark implementation language Scala [10].

In addition, the RDD has three particular features:

1. *Lineage*. The RDD remembers the operations that are attached to it. This a very powerful feature also in failure cases, for example, if a worker node crashes: the lost parts of an RDD can always be recovered.

2. *Persistence*, or *Caching*. A user can moderate a storage strategy RDD uses, e.g. in-memory only or the memory and the disk. This functionality makes computing faster, when the data is cached in memory. Caching is a fault-tolerant feature, because possibly lost data partitions will be recovered via the RDD's lineage.

3. *Data locality*, or *partitioning*. An user can control also the count of data partitions by the particular functions.

Together these features make RDD/Spark more effective than a basic MapReduce/Hadoop implementation, as Zaharia et al. have shown in their article [37]. However, the reported experiences from Spark are still limited.

Spark API is available in three languages: Scala, Java, and Python. This thesis will consider only the Scala functions for Spark, and no other implementations of Spark will be covered. Spark itself is implemented on Scala and many of its functions seem to be inspired by Scala native functions, such as *map* and *filter*.

Spark offers two different types of operations for RDDs: *transformations* and *actions*. The RDD's lineage saves the operations of the both types, but only the actions are computed instantly. The actions typically also return some value, for example, *count* that returns a number of elements in the

Action	Data operation	Meaning
$reduce(func)$	$RDD[V] \rightarrow V$	MapReduce like reduce, uses a function $func$ to aggregate the data items
$foreach(func)$	$RDD[V] \rightarrow Unit$	Does the same operation $func$ to each data item, does not return anything
$count()$	$RDD[V] \rightarrow Long$	Returns a count of the data items in RDD
$collect()$	$RDD[V] \rightarrow Array[V]$	Returns the data items to the master as an array of elements type V
$first()$	$RDD[V] \rightarrow V$	Returns a first item of the RDD, same as $take(1)$
$take(n)$	$RDD[V] \rightarrow Array[V]$	Returns n first items of RDD as an array
$saveAsText-File(path)$		Saves RDD to the defined file system (local or distributed) as text files
$saveAsObject-File(path)$		As $saveAsTextFile$, but writes object files that are easy to read again to Spark
$broadcast(obj)$	$obj \rightarrow spark.Broadcast[obj]$	Makes the current version of the object available for all the nodes

Table 2: Some of the main Spark RDD actions, which are performed immediately as opposed to the Spark transformations presented in Table 3. The whole API document is available on [11].

RDD and a MapReduce style aggregating function *reduce*. Table 2 presents some other examples of the main actions.

The transformations are operations, which create a new RDD from an existing old one. This means they do not modify the old one, and in the Scala style manner, it is necessary to pick the returning value to the variable. The transformations are executed lazily. Typically they are waiting in the RDD's lineage until some action operation appears. Functions such as *map* and *filter* are transformations; they create a new RDD based on given parameter function. In the case of the *map*, the new RDD consist of the same count of moderated data items, whereas the *filter* gets a boolean function and returns a new RDD whose every element satisfies the boolean function. Table 3

Transformation	**Data operation**	**Meaning**
$map(func)$	$RDD[V] \rightarrow RDD[W]$	MapReduce like map, uses a function $func$ for every item in the data set of type V and returns a new set of type W
$flatMap(func)$	$RDD[V] \rightarrow RDD[W]$	Similar to map, but returns a flatted sequence where every input item can produce zero or more output items
$filter(func)$	$RDD[V] \rightarrow RDD[V]$	Rerturns a selected set of items on which a boolean function $func$ returns true
$groupByKey()$	$RDD[(K,V)]$ $\rightarrow RDD[(K, Seq[V])]$	Collects all the data sets related to each key and returns them as a key and sequence of the corresponding data items
$reduceByKey()$	$RDD[(K,V)]$ $\rightarrow RDD[(K,V)]$	Reduces or aggregates the data items related to each key

Table 3: Some of the main Spark RDD transformations, which are performed lazily. The whole API document is available on [11].

presents some of the main transformations.

Algorithm 3 presents a K-means clustering algorithm introduced in Section 2.3, now in the form of Spark Scala API. Algorithm 3 starts like Algorithms 1 and 2 by initializing the starting set of centroids. In contrast to MapReduce K-means Algorithm 2, the data structure for the data points is an RDD and there is no necessary to implement own map and reduce functions.

All the iteration phases happen in one loop. The centroids have to be *broadcast* to the slave nodes, that means that the current values of variables are shared throughout all the participating nodes. After that, a Spark *map* function can be used for assigning the closest centroid to the each data point in the RDD. Clusters based to the centroids are got by a Spark function *groupByKey*, which returns a set of sequences lead by each key. The new centroids are easy to compute as means of the data points in the clusters. The notation (_._2) inside the map function means that the operation will be run on the second element of the RDD, which is after the *groupByKey* function: $(key, seq[datapoints])$. The function *collect* moves the RDD to an

Algorithm 3 Spark K-means clustering

```
1: Let D be an RDD of data points
2: Initialize centroids as a set C of size k
3: repeat
4:    centroids = broadcast(C)
5:    assigned = D.map(datapoint => {
      closest = centroids.map(centroid => dist(centroid, datapoint)).min
      (closest, datapoint)
      })
6:    clusters = assigned.groupByKey
7:    C = clusters.map(_._ 2.mean).collect
8: until Centroids do not change
```

array. Functions *min* and *mean* are from the native Scala library [10].

3.3 Spark versus MapReduce

Zaharia et al. [37, 36] have evaluated the performance of Spark and two different Hadoop implementations. They measured iteration times of two iterative machine learning algorithms, logistic regression and K-means clustering, in each three systems. In the first iteration, Spark was moderately faster than the Hadoop implementations, and in the later iterations, Spark was clearly faster. Zaharia et al. explain the differences in the overhead of the Hadoop stack, overhead of the HDFS as a data service, and used binary convertion.

In addition to the performance, Zaharia et al. [37] defend Spark's versatility over the other distributed programming interfaces. For example, the MapReduce phases are possible to implement with Spark API: the map phase by the functions *map* or *flatMap*, and the reduce phase by the functions *reduceByKey* or *groupByKey*. Also some other other programming models are easy to implement with the functions of the Spark API, more specifically presented by Zaharia et al. [37]

Spark's RDD model with its transformation and action lineage also offers a possibility to return to any state of the system or separated node in the case of some fault or lost node. So the states of any algorithm are easy to recompute, if necessary. This is one difference between Spark and MapReduce implementations such as Hadoop, that write the outputs separately between the iterations: the next step of computing does not necessarily know what

has happened before it.

Figure 8 presents a data flow of the MapReduce system. MapReduce also actualized each operation one by one as presented in Section 2.3 about the map and reduce phases. The iterations of the algorithm are shown as *tasks*. Each task has one map and one reduce phase, and when the iteration continues, also the map and reduce phases alternate. The controller has to handle the inputs and outputs between the tasks.

Figure 9 presents data flow of the Spark system. The controller node handles the RDD lineage. The operations, both transformations and actions, have been attached to the lineage. The transformations will be actually performed with the actions: for the first action, all the transformations before it will be run in order. This reduces the number of necessary intermediate states. Compared to the MapReduce, only the necessary operations will done to the data point: because of known lineage, earlier operations are not performed to the data point that will be filtered away in some later step, for example.

Section 3 has presented the Berkeley Data Analysis System (BDAS) and two of its main parts, Mesos and Spark, which construct a cloud computing environment operating together. Spark has also been compared to the MapReduce paradigm. As an example of the Spark and Mesos implementations, this thesis will present a decision tree classification algorithm in Section 4.

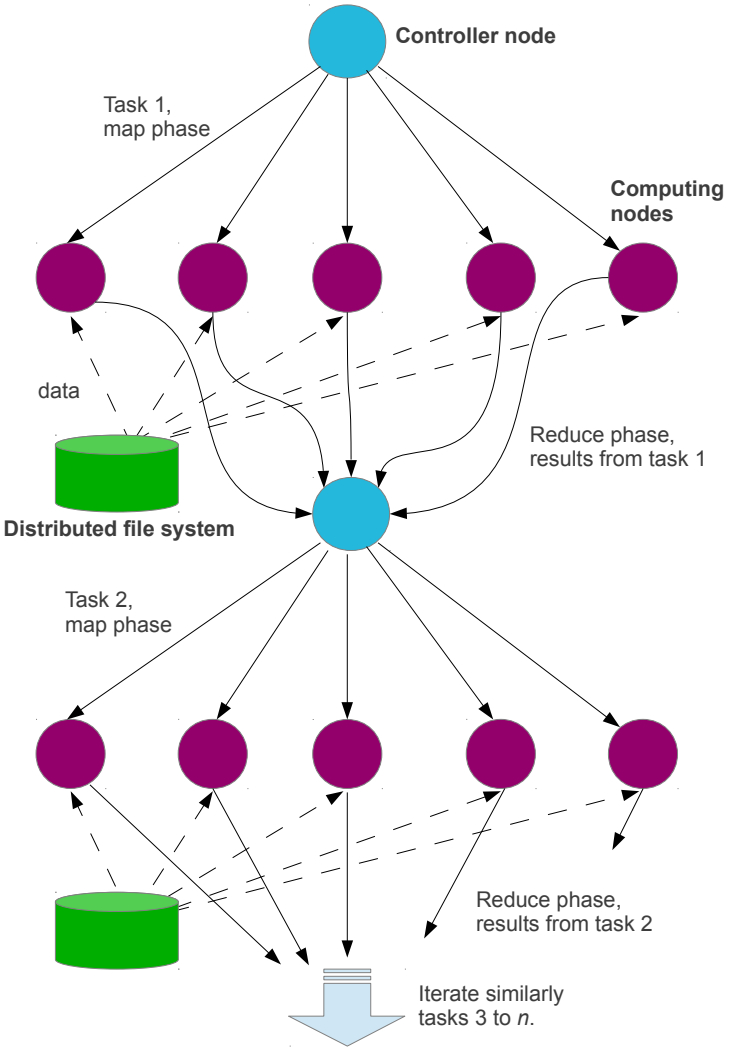

Figure 8: The data flow of MapReduce. Each map and reduce phases are iterated in turns. Computing is managed by controller node, which also organize inputs and outputs of each iteration. More about MapReduce paradigm in Section 2.3.

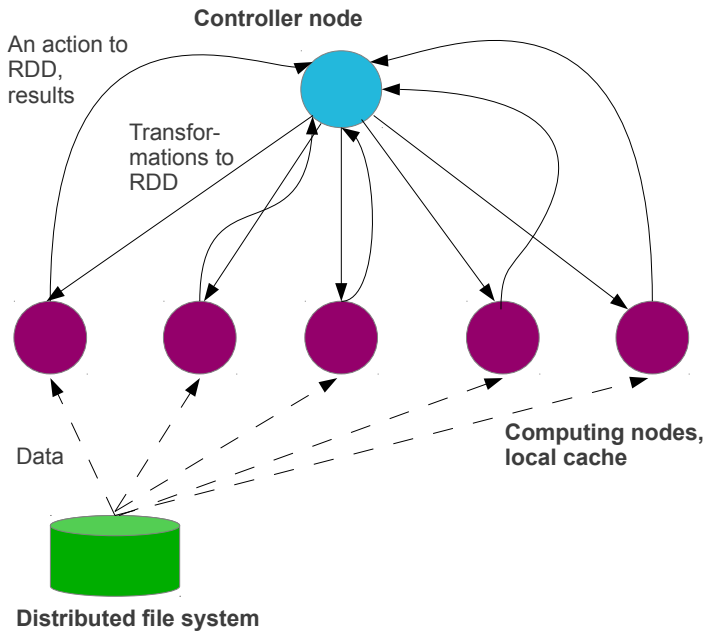

Figure 9: The data flow of Spark. Each transformation has been collected together to RDD's lineage and performed when the next action appears. Compared to MapReduce data flow in Figure 8, the Spark data flow saves unnecessary iterations.

4 An example: Carat data analysis

This section introduces the Carat energy consumption data and gives specification for a decision tree, which is a widely used classification technique. Section 5 will present the implementation for the decision tree algorithm over Berkeley Data Analysis Stack, especially the Spark and Mesos systems.

Section 4.1 introduces shortly the Carat project. Section 4.2 exposes motivation for using data analysis methods for the Carat data and gives an abstract level specification for the analysis process. A decision tree algorithm is presented in Section 4.3 and entropy as impurity measurement in Section 4.4.

4.1 Carat: collaborative energy analysis

Carat [31, 6, 33] is a research project of UC Berkeley and University of Helsinki. Its aim is to discover energy anomalies from mobile devices by collecting and analyzing the energy measurements by users or clients. In addition to the research, Carat offers an application with tips for reducing the energy consumption of the user's device.

Figure 10 presents the structure of the Carat systems. Circa 600.000 clients (in July 2013) have installed the Carat mobile application that measures and sends the data to the Carat project's Amazon cloud. The data is stored and analyzed in the cloud. After the analysis, the cloud returns results to the clients as statistical reports from their energy consumption compared to the other known devices, and actions or tips how to improve own device's energy behavior. A classic example about the actions is to avoid some very energy greedy application, such as a free game with many advertisements.

The Carat analysis software has been implemented on Spark presented in Section 3.2 in Scala language [10]. The analysis software is run over Mesos presented in Section 3.1. Mesos is run in the cloud of Amazon EC2 [1]. Figure 10 shows also a researcher as a Carat developer or data analyst. Her or his aspiration here is to improve the analysis quality and coverage with multiple methods, for example, machine learning algorithms.

After it was published worldwide in June 2012, Carat has collected more than 150 GB of data from iOS and Android devices, both mobile phones and tablets. This crowd of different devices provide about half a million

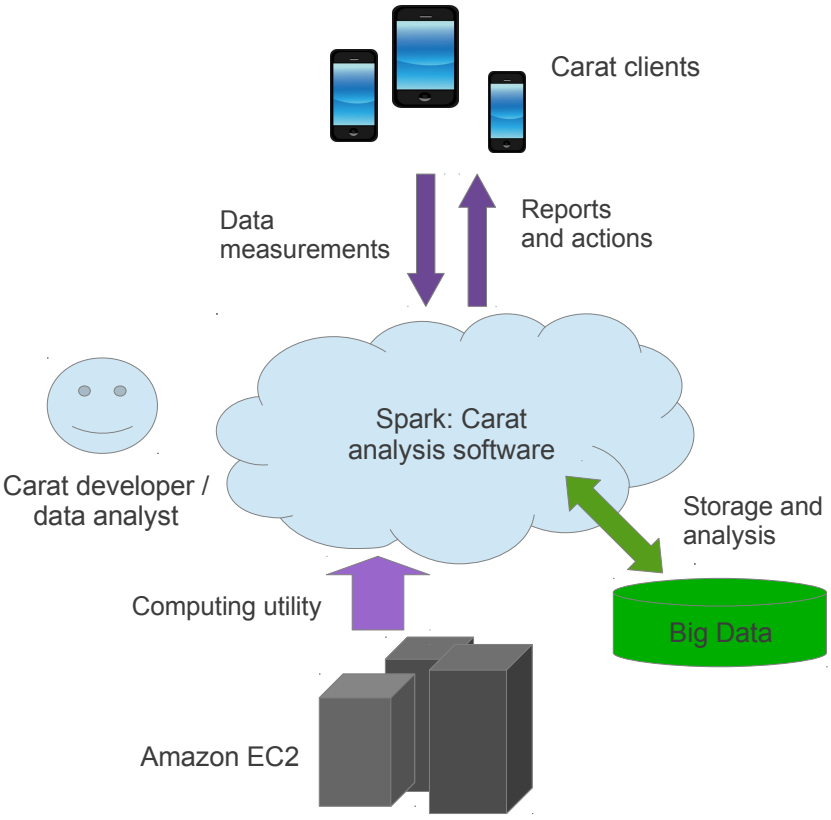

Figure 10: A structure of the Carat analysis system. The services can also be compared to Figure 4 in Section 2.

new samples per week. Each sample includes information from the device's native API, such as a device model, an operating system version, battery state, inside temperature, applications in action, and a set of extra features, such as screen brightness and network connections.

There are multiple research objectives related to the Carat data and the Carat analysis system. The main interest has been in applications that could be associated with increased energy consumption. The current analysis system can find applications that are using more energy altogether – these anomalies are called hogs – or just in some particular device – called bugs. The next step is to take account also the features and other information given by the mobile APIs. Especially the Android devices offer a lot of information from their use.

4.2 Analysis specification

The aim of the Carat analysis is to find combinations of *attributes*, such as running applications or enabled network connections, that could lead to energy anomalies. These attribute combinations might be presented as *attribute chains*, which are easy to 'follow': the chain presents sequentially the combination leading to the anomaly. New actions will be composed to the clients based on the attribute chains. One possible way to construct the attribute chains is a decision tree algorithm presented more detail in Section 4.3.

Each data sample offers following information about the device defined by its API [31, 33]:

- Battery level, in iOS every five percent granularity.
- Event that caused a sample, for example, battery level charged by one percent.
- Battery state, for example, if the device is currently plugged in to the power supply.
- A list of the currently running applications and processes.
- Operating system and its version.
- Model of the device.

- Time stamp.
- Anonymous hash-based identification of the user.
- In Android devices, a list of features related to usage of battery, CPU, memory, network (see Table 4).

This information can be used as attributes for the analysis algorithms, but the presented work with the decision tree is based on the *Android features* given in Table 4. The iOS system is more closed than Android API and most of these features are not possible to obtain from iOS devices. Also, Android battery level is possible to measure in one percent granularity versus iOS just in five percent. Teaching and validation data sets have been constructed from the Android samples that also decreases the data samples to one-third.

For the analysis, samples are organized to *sample pairs*, that means an interval between two temporally sequential samples from the client. The sample pairs can represent the energy consumption and changes in the applications and the features. The data has been cleaned so that only interesting sample pairs are left: for example, eliminated sample pairs include those where the energy use seems to have decreased because of battery charging or where another sample of the pair has been lost. Every sample pair includes all the attributes of both of its samples, energy consumption as a change between two samples, given as percent in a second, and other attributes as a pair or a list of values.

This kind of change in the battery drain or energy consumption will be hence called a *rate value*. If the rate value is high, near 0.04% per second, it means that it is possible to search reasons from the attribute set and vice versa. The decision tree uses the attributes for classifying the rate values and the rate values compose the classes: low, medium, and high energy consumption. The decision tree tries to find the attribute chains, which frequently direct to a certain class of energy consumption.

Applications have been explored in earlier work [31, 33], but the Android features presented in Table 4 have not been approached thus far. In this thesis, the decision tree uses specifically the Android features for attribute chain making. After that, the next step could be to try to combine these aspects of the Carat data even if the number of possible combinations increases quickly with each new attribute.

Feature	Values
Battery charger	String: AC, USB, unplugged
Battery health	String: dead, cold, overheat, good etc.
Battery temperature	Numerical value
CPU usage	Numerical value between 0-100
Distance traveled	Numerical, positive value
Memory active	Numerical
Memory inactive	Numerical
Mobile data activity	String: in, out, none etc.
Mobile data status	String: connected, disconnected etc.
Mobile network type	String: GPRS, EDGE, UMTS etc.
Network type	String: wi-fi, mobile, wimax, etc.
Screen brightness	Numerical value between 0-255, -1 if set to automatic
Uptime	Numerical value
Wi-fi link speed	Numerical, positive value
Wi-fi signal strength	Numerical, positive value
Wi-fi status	String: disabled, enabled, unknown etc.

Table 4: Examples from the Android features. All the values are given by Android API, and they can vary based on the Android version and a phone model.

Section 4.3 introduces the decision tree classification algorithm more detail. Section 4.4 presents the entropy heuristic as impurity measurement and splitting condition. Section 5 presents the Spark decision tree implementation, including Section 5.1, that focuses on the Android features as decision making attributes and gives some examples how to handle both discrete and numerical values.

4.3 The decision tree algorithm

The decision tree is a well-known algorithm for classification and regression. It has been introduced early at least in the book by Breiman et al. [14] but presented many times in the literature.

Algorithm 4 presents a decision tree structure based on the book of Tan et al. [34, pages 164-165]. The algorithm has an input as a set of training data points and a set of attributes. The algorithm builds a tree recursively. In each node, the algorithm makes a split based on the attribute that derive results in minimal impurity. The impurity has been measured by some

heuristic, for example, entropy or gini index. This work uses the entropy heuristic presented in Section 4.4.

Algorithm 4 Basic decision tree

Let D be a set of training data points
Let A be a set of attributes

function growthTree(D, A)

```
1: if stopping condition == true then
2:    leaf = new node
3:    leaf.class = classify()
4:    return leaf
5: else
6:    root = new Node
7:    bestAttribute = findBestSplit(D, A)
8:    let V be a set of values of the best attribute
9:    for all v ∈ V do
10:      D_v = {d|d.bestAttribute = v and d ∈ D}
11:      child = growthTree(D_v, A)
12:      add child as a descendant of the root
13:   end for
14: end if
15: return root
```

The decision tree algorithm 4 starts by checking the stopping condition that could be, for example, the size of the remaining data points or attributes or some other measurement, such as the count of performed iterations. If the stopping condition returns true, a leaf node will be created. A function *classify* gives a class or label to the leaf node. Majority of data points determines to which class the leaf node assign.

If the stopping condition returns false, the iteration continues and a new root node will be created for a subtree and a child of the earlier node. A function *findBestSplit* gets a set of training data points and a set of attributes and returns the attribute that direct to the best split in future. Next, the split will be made based on the best attribute. For each value of the best attribute, a new children node will be created. So a size of the training data point set will increase and, if wanted so, also the used attribute could be removed in order to avoid reusing the attributes.

Figure 11 presents an example decision tree. The root node estimate the attribute *network type* to be the best, or it leads to decreasing impurity in

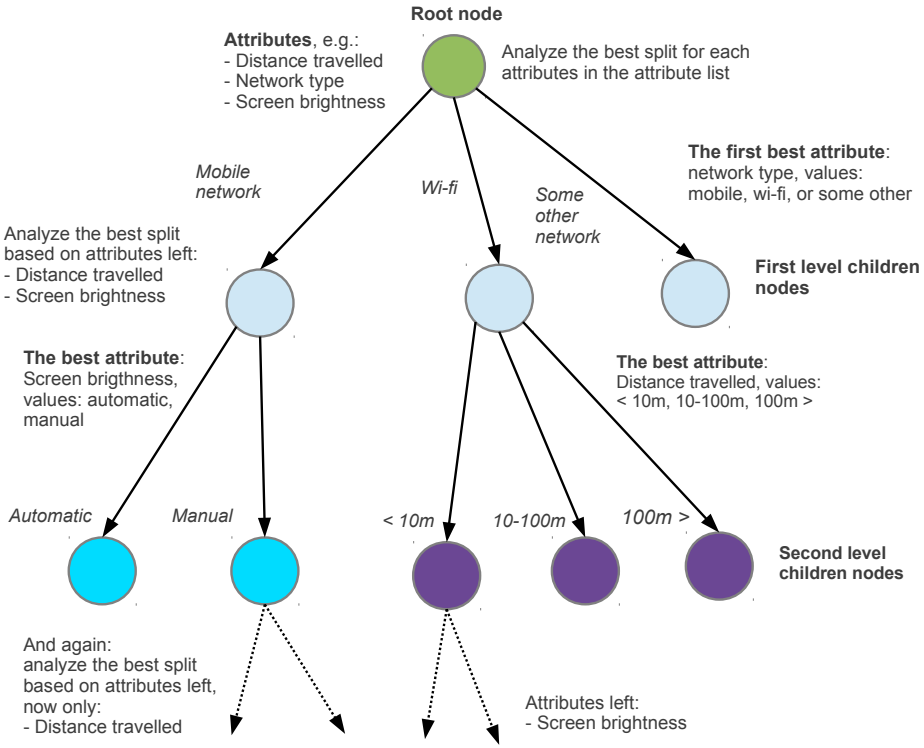

Figure 11: A decision tree example. Each node makes a decision for the next best split. For example, the subtree left has been first split by network type and then by screen brightness.

the tree. Network type has values *mobile*, *wi-fi*, and *some other network*. Each first level child gets a list of remaining attributes and estimates the next best split. The node split by mobile as the network type gets the attribute *screen brigthness* for its next best split. The node split by wi-fi gets the attribute *distrance travelled*. This iteration will be continued until fulfilled the stopping condition.

The decision tree algorithm is possible to implement also without recursion. In this case, nodes should be saved to some helping data structure, such as stack or list. Section 5.2 presents the Spark decision tree that were first implemented with recursion but after that without it because of performance issues.

4.4 Impurity measurement

The decision tree can estimate goodness of the next split by several heuristics. In this work, *entropy* has been used for measuring impurity of the splits. Entropy is presented, for example, in the book of Tan et al. [34, pages 158-160].

Entropy is defined so that

$$Entropy = -\sum_{i=0}^{c-1} p(i|t) log_2 p(i|t)$$

where c is a count of possible classes, i is an iteration over data points and t presents a count of data point entries in the given node. This denote that the notation $p(i|t)$ means a fraction of data points in class i appearing in the node t. The value of entropy is in the range $[0, 1]$ so that 0 means all the data points of the node belongs to the same class and 1 means the data points are divided equally between the classes. For simplicity, it is defined that $0 log_2 0 = 0$.

When growing the tree, variances in the entropy are aggregated to the *information gain* that presents a difference of impurity between the parent and the children nodes. The information gain IG is defined so that

$$IG(A, a) = Entropy(A) - \sum_{v=0}^{n} (A_v/A) Entropy(A_v)$$

where A is a set of attributes and $a \in A$, and values of each attribute are

presented as $v \in values(a)$ and n is a number of values of attribute a.

For each attribute $a \in A$ it will produce an information gain. The gain is a difference between the node's current entropy before the split, which could be given also as a parent note's entropy, and the sum of entropies of every children node made by attribute values. The entropies of the children nodes are weighted with the count of data points belonging on this current children node.

After producing the information gains for each attribute, the attribute with the highest information gain will be picked up for the splitting condition. This defines the best attribute for the next split:

$$bestAttribute = max\{IG(A, a)|\forall a \in A\}.$$

Because trying to minimize the entropy in the decision tree, the best information gain is such that where the difference in impurity of the parent and the child node is maximized. The child consists of a fraction of the data points of the original parent, so the child will inevitably have a entropy measurement leading to more pure results.

5 The Spark decision tree for Carat

This section presents the Spark implementation of the decision tree algorithm which has been used for analyzing the Carat data. The analysis specification has been presented in Section 4. Section 5.1 introduces to preprocessing of the Carat data. Section 5.2 presents the Spark decision tree implementation. Section 5.3 focuses on validation process of the decision tree algorithm. Analysis results will be presented in Section 6.

5.1 Attributes and data preprocessing

Table 4 shows examples from the Android features of the Carat data. Some of the features has discrete values, given as strings, for example, mobile network type has values "GPRS", "EDGE", or "UMTS". Some of the features has numerical values, for example, screen brightness is always an integer from the range 0 to 255, or -1 if the screen brightness has set to automatic in the device. Some numerical attributes has floating values, for example, distance traveled, or wi-fi link speed.

Multiple diversity of different attribute values has to be handled in suitable way. In this work, it seemed to be a sufficient solution to discretize all the attributes. This means that all the attributes with numerical values are presented as value ranges, which describe classes such as the discrete values describe a class. For example, possible classes of the attribute distance traveled can be zero to one meter, one meter to hundred meters, and all the values more than hundred meters. Sometimes a single value may be enough to present a class, for example, the screen brightness has value -1 that describes devices where the screen brightness has been set to be automatic instead of manually by user. The attribute classes used in this work are presented in Table 5.

The value classes for each attribute should be results of some automated method, such as clustering or statistical analysis, so the classes will base on the Carat data. They are also possible to type by expectations based on natural groups, such as low, high, and automatically set screen brightness values. In terms of the implementation, all the attributes are given to the algorithm as a parameter, so they are possible to modify without modifications to the decision tree algorithm's implementation.

The decision tree algorithm uses the entropy heuristic as an impurity

Attribute	**Value classes**
Network status	connected, other
Battery voltage	0-2.5, 2.5-5, 5->
Mobile network type	GPRS, EDGE, UMTS, 3G, other
Battery temperature	0-20, 20-40, 40-100
Wi-fi status	enabled, other
Network type	wifi, mobile, wimax, other
CPU usage	0-20, 20-40, 40-60, 60-80, 80-101
Battery health	dead, cold, overheat, over voltage, good, other
Mobile data activity	none, in, out, inout, dormant, other
Screen brightness	-1, 0-101, 101-255, 255
Distance traveled	0-101, 101->
Mobile data status	connected, disconnected, suspended, other

Table 5: Attributes used in the example of this work. Numerical ranges used so that the lowed bound includes to the range, but the upper bound does not.

measurement function for splitting decisions. The entropy measurement is presented better in Section 4.4. For working correctly, entropy measurement needs to know the ending classes beforehand. This means the rate classes mentioned in Section 4.2. They represent if energy consumption has been low, medium, or high – possible in more detail groups.

Different rate values and their amounts of the Android data are presented in Figure 12. Figure shows that there are lot of samples with just a small rate value, which means a little energy consumption, maybe the devices have been idle. There are fewer rate values with very high energy consumption, but the distribution is not smooth and some increased values are possible to observe. Rate values can be interpreted as hours by the formula

$$h = \frac{\frac{100}{rate}}{3600}$$

so that the energy consumption 0.015 per second means circa 1,85 hours of total battery life, for example. Vice versa, the rate value can be interpreted to hours by the formula

$$rate = \frac{100}{h \cdot 3600}$$

Figure 12 also shows that there are no clear clusters naturally in the data. In this thesis, natural split are used for present energy consumption groups: low as more than 24 hours of total battery life, high as fewer than eight

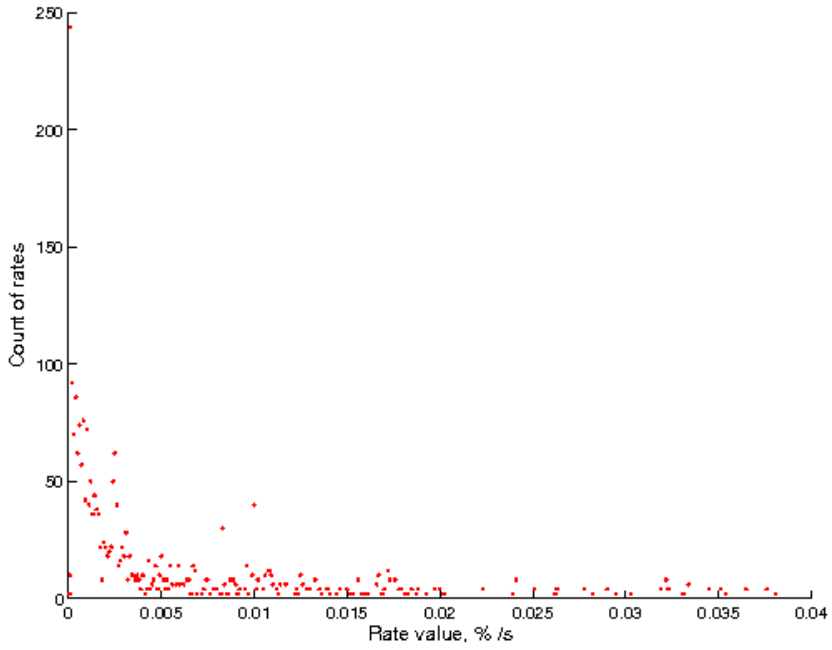

Figure 12: Counts of different rate values in the Android data set.

hours of total battery life, and medium as their intermediate. These number of hours also represent how often the device should be charged, roughly. For more information, also rates that predict less than an hour battery usage form a class. So the rate classes the approach of this thesis will be:

1. Low consumption, more than 24 hours of total battery life: rate values < 0.001157
2. Medium consumption, eight to 24 hours of total battery life: rate values $\in [0.001157, 0.003472[$
3. High consumption, less than eight hours of total battery life: rate values $\in [0.003472, 0.027777[$
4. Only an hour of total battery life: rate values ≥ 0.027777

5.2 The Spark decision tree implementation

The basic structure of the decision tree algorithm has been presented in Section 4.3. It was recursion-based, and also a recursion version of the Spark

decision tree has been developed. Because of multiple performance issues, also the non-recursive version has been tested. Algorithm 5 presents this non-recursive version which based on a feature chain list as the helping data structure.

Algorithm 5 Non-recursive Spark decision tree

Let D be an RDD of training data items (sample pairs)
Let A be a set of attributes (Android features)
Let C be a set of rate classes

Let *chainSet* to to be a set of ready feature chains
Let n be maximum depth of the tree
There can be used also other stopping conditions.

function makeDecisionTree()

```
1:  Broadcasting the attributes can make this faster.
2:  (feature, values) = bestSplit(D, A, C)
3:  chainSet ++= values.map(value => new Chain((feature, value)))
4:  iteration = 1
5:  iterationRDD = null
6:  while iteration <= n do
7:    chainSet.filter(_.length == iteration).foreach(chain => {
8:    attributes = broadcast(chain.attributes)
9:    iterationRDD = D.filter(chain)
10:   (feature, values) = bestSplit(D, A, C)
11:   if feature != null then
12:     attributes = chain.attributes - feature
13:     chainSet ++= values.map(value => {
14:     newFeatures = chain.features ++ (feature, value)
15:     new Chain(newFeatures)
16:     })
17:   end if
18:   iteration += 1
19:   })
20: end while
21: return chainSet
```

The bestSplit function is given in Algorithm 6.

The training set D of the data items consists of sample pairs presented in the specification in Section 4.2. A sample pair contains a set of Android features and a rate value from one measurement interval. The attribute set A

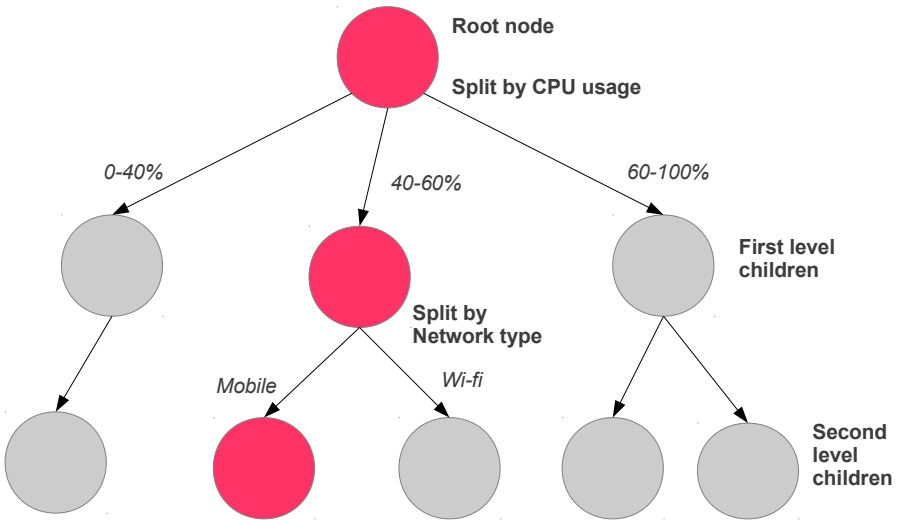

Figure 13: One feature chain of decision tree is colored with red. After two splits, this chain consists of feature and value pairs *(CPU usage, 40-60%)* and *(Network_type, mobile)*.

contains Android features given in from $a = (attribute\_name, Set[values])$. The class set C represents the classes of rate values introduced in Section 5.1. The rate classes are used for measuring entropy presented in Section 4.4.

The set called *chainSet* is the helping data structure that saves the chains discovered. A chain consists of a path from the root node to some children node as presented in Figure 13. The chain is presented as an ordered list of features and their values which the splits are based. The tree contains chains from the root to the leafs or from the root to some children in the middle of the tree. For example, the tree in Figure 13 consists of eight chains: five from the root to the leafs and three from the root to the first level children.

The main function *makeDecisionTree* makes first a split for the root node. The *bestSplit* function, presented in Algorithm 6, returns the best attribute as $(feature, values)$ where the second part is a set of values. Every $(feature, value)$ pairs form the first chains.

After the first split, the algorithm continues iterating the children node levels in order of breadth-first search presented in Figure 14. Iteration continues until the stopping condition has been fulfilled. Now, there is only the iteration depth used as a stopping condition, but also others are possible, such as the size of the training items left. At the beginning of each iteration

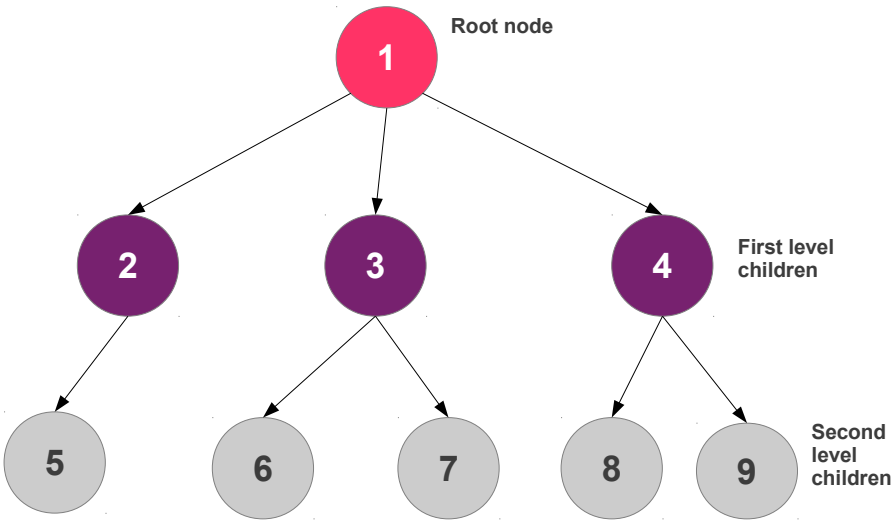

Figure 14: Breadth-first search. Nodes are handled in the order of numbering. The levels are colored for clarity.

level, the algorithm separate chains that are as long as the given index. This aims only chains of the previous iteration level can be used as a seeds for the next level chains.

The attributes of the seed chain are broadcast. The training set D is filtered by the attributes of the chain, so that the appropriate sample pairs are separated to the variable $iterationRDD$. For example, if the chain consists of the feature and value pairs $(network\_type, mobile)$ and $(battery\_healt, good)$, the filtering operation returns the sample pairs where the network type is mobile and the battery health is good. In the recursive version of the decision tree algorithm, this filtering would be managed when splitting the training set to the children nodes, for example.

The best split is measured by the function *bestSplit* called by the main function *makeDecisionTree*. The function *bestSplit* also uses the function *measureGoodness* for perform the entropy heuristic. They both are represented in Algorithm 6.

The function *bestSplit* has a sets of sample pairs in the RDD, attributes and rate classes. At first, the sample pairs are ordered by each feature given in the attribute set so that each feature has a set of the related sample pairs. Second, the sets of features and sample pairs are organized to the sets of the

Algorithm 6 Best split functions

These functions are used for measuring the best split. In addition to them, there are also helping functions for computing entropy heuristic as presented in Section 4.4.

function bestSplit(samples, attributes, classes)

```
1: byFeature = map samples by each feature ∈ attributes
2: byFeatureAndValue = map samples by values of each feature
3: if byFeatureAndValue != empty then
4:    entropiesAfterSplit = byFeatureAndValue.map(samples => {
5:    measureGoodness(samples, classes)
6:    })
7:    entropyBefore = measure entropy after split
8:    bestIG = entropiesAfterSplit.map(entropyAfter => entropyBefore -
      entropyAfter).max
9:    return  bestIG as a pair (bestFeature, values)
10: end if
```

function measureGoodness(samples, classes)

```
1: split = get a real class for each item ∈ samples
2: entropies = measure entropy for each part ∈ split
3: entropyAfterSplit = sum of entropies weighted by part size
```

feature, the feature values and the related sample pairs, so that:

$$byFeatureAndValue = RDD(feature, Map(value, Set(samplepair)))$$

where for each feature there is a map of the feature values and the sample pair set. This form of the RDD helps in the next steps of the algorithm.

The algorithm prunes the feature and value combinations which have no sample pairs. If the sample pairs exist, the algorithm measures a entropy for each possible splits. The splits are based on the features and their values, so the function *measureGoodness* is attached to each sample pair set in the *byFeatureAndValue* RDD. For the information gain presented in Section 4.4, the algorithm also measures entropy after the split. It can also be given as a parameter from the earlier iteration, except to the root node that has no parents. The best information gain is chosen by maximizing the difference of entropy after and before the splits. The function *bestSplit* returns the best information gain as a pair of the best feature and its values.

The function *measureGoodness* implements the actual entropy heuristic with necessary helping functions. At first, each sample pair is assigned to one of the given rate value classes. Second, entropy is measured for each set of the assigned sample pairs, that are comprehended to the parts of the possible split or the next child nodes. Finally, the entropy after split is measured as a sum of entropies of the parts weighted by size of each part.

5.3 Validation with the synthetic data set

The Spark decision tree has tested with the synthetic data set generated by Matlab [8]. This data set consists of thousand data items that have been separated to six rate classes. Each data item includes a limited set of Carat like attributes that are divided so that the decision tree results are possible to assume.

The Matlab classification and regression tree function *classregtree* [7] has been ran also with the same synthetic data set. Because the *classregtree* function produces only binomial decisions, also the synthetic data set has been designed so that the Spark decision tree should also return a binary tree. For the Spark decision tree, the synthetic data set has been written as an RDD.

Figure 15 presents the output of the function *classregtree* based on the

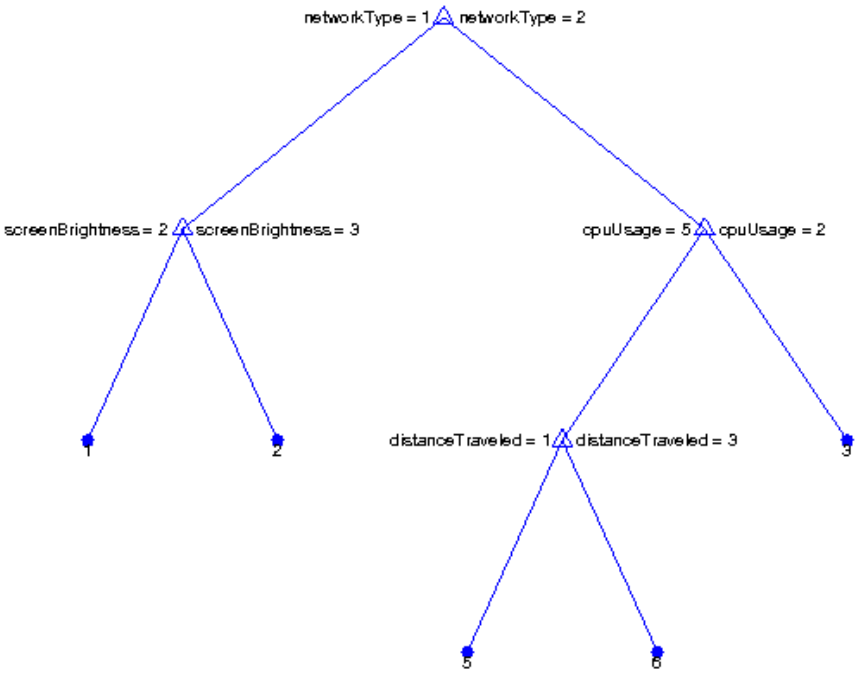

Figure 15: The output of the Matlab classification and regression tree function *classregtree* based on the synthetic data set.

synthetic data set. For simplicity, the attribute classes are presented as numbers. The equivalent value names are:

- Network type: 1 = wi-fi, 2 = mobile
- Screen brightness: 2 = 101-254, 3 = 255
- CPU usage: 2 = 20-40%, 5 = 80-100%
- Distance travelled: 1 = 0 to 100 meters, 3 = more than 100 meters

With the synthetic data set, the Spark decision tree produces the following attribute chains:

- Network type = mobile
- Network type = wi-fi
- Network type = mobile, CPU usage = 20-40
- Network type = mobile, CPU usage = 80-100

- Network type = wi-fi, Screen brightness = 101-254
- Network type = wi-fi, Screen brightness = 255
- Network type = mobile, CPU usage = 80-100, Distance traveled = 100-10000000000
- Network type = mobile, CPU usage = 80-100, Distance traveled = 0-100

which are equivalent to the results of the Matlab function *classregtree*.

5.4 Cross validation with the real Carat data

K-fold cross validation has been run with the real Carat data. K-fold cross validation, presented by Kohavi [23] et alia, is a validation technique where the data set has been divided to two parts: a training set and a test set. The decision tree is growth by the training set and after that validated by the test set. This operations are iterated K times, so that each time the test set is individual $1/K$ part of the full data and the training set consists of $K-1$ data folds. Figure 16 shows how test and training data sets alternate during iterations.

The basic cross validation technique measures the error rate for each test set. The full error rate can be given as an average of errors of the iterations, for example. Basically, the test set error means that how many items in the test set are classified wrong by the decision tree. This requires the attribute chains always lead just in one possible class. In the Carat data set, it is even

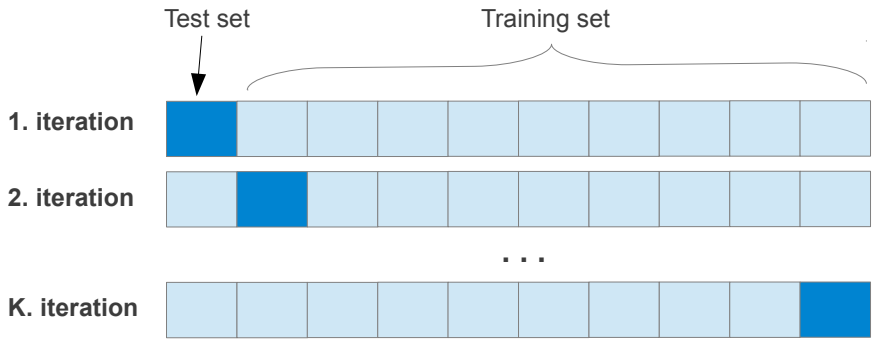

Figure 16: An example of the K-fold cross validation.

suitable that one chain can lead to multiple classes – the class distributions are more interesting than choosing one specific class for each of the chains.

In this work, K = 10 that leads to ten folds and iterations. The cross validation aims to handle the full data set divided into K parts, but because of performance issues, the folds of this work are initialized randomly selecting a data fraction of the right size to be a fold. So the validation is possible to execute in hours instead of days. All the measurements have been executed in Amazon EC2 cluster of ten nodes of eight vCPU and 30GB RAM.

The fold size was first measured circa 12 000 sample pairs, which means size of the training set is circa $9 \cdot 12\,000 = 108\,000$ sample pairs. The training set of this size seems to make the decision tree really slow. At 108 000 sample pairs, the run did not completed before 48 hours when the maximum depth of the tree is four only. In comparison, a tree of circa 12 000 sample pairs and the same depth has generated in less than an hour, and a tree of circa 1 200 sample pairs in circa five minutes. So the scalability is superlinear.

Because of this quick-growing time complexity, the fold size has been reduced by random sampling to circa 1 200 sample pairs. This fold size leads to the training set of circa 10 800 sample pairs. Hence, the cross validation is not a comprehensive to all the Carat data samples, but it can give a small approach, at least.

Results of the cross validation analysis are represented in Figures 17 and 18, as differences in class distributions between the training set and the test set. After the observation that each attribute chain of the decision tree can lead multiple classes, there is no possibility to *classify* the sample pairs to any kind of *truth* classes. That is because it is more relevant to compare, in which distributions the sample pairs have been separated when growing the decision tree by the training set, and then, when validating the decision tree by the test set.

Figure 17 presents average Kullback–Leibler divergences for each of the cross validation folds. The Kullback–Leibler divergence [25] is a widely known, statistical measurement that offers a tool for comparing two distributions P and Q, where P is the 'truth' and Q is the distribution whose impurity should be measured. The Kullback–Leibler divergence is defined so that

$$KLd(P||Q) = \sum_{i=0}^{n} \log\left(\frac{P(i)}{Q(i)}\right) \cdot P(i)$$

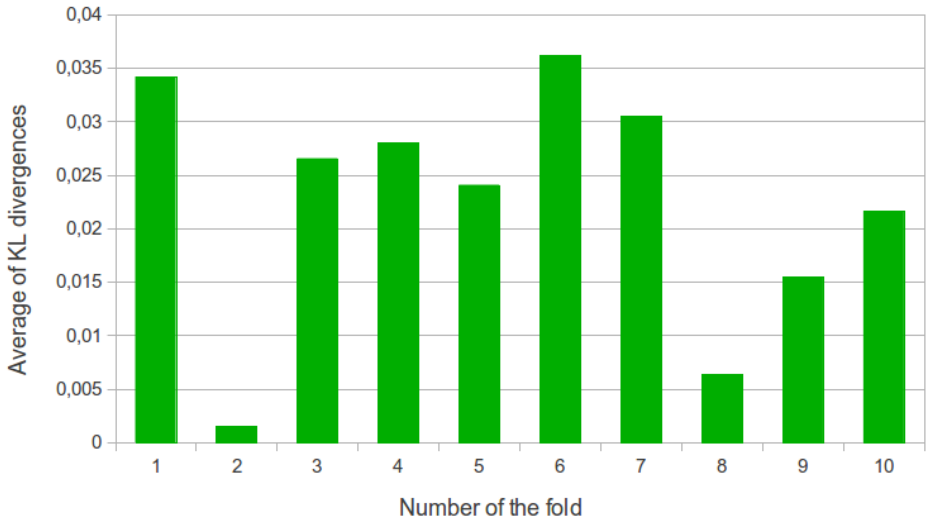

Figure 17: Averages Kullback–Leibler divergence in the 10-fold cross validation for the Carat data.

In this work, P is defined to be the distribution of the training set and Q to be the distribution of the test set, where distributions mean how the sample pairs of each chain have been separated to the different rate classes. The Kullback–Leibler divergence works so that if the distribution Q reminds the distribution P much, the sum in the formula approach towards zero. If the distributions Q and P are the same in each measured point, the Kullback-Leibler divergence is zero.

The Kullback-Leibler divergence has been measured for the rate class distributions per chain. As mentioned, the chain can lead to the multiple rate classes, and parts of each rate class are given as a distribution. For each chain in the decision tree, there is

$$KLd(chain) = KLd(training(chain)||test(chain))$$

and for each decision tree or for each fold, there is

$$AverageKLd = \frac{\sum_{chain}^{n} KLd(chain)}{n}$$

where n is a count of chains. Absolute value averages of each fold are presented in Figure 17.

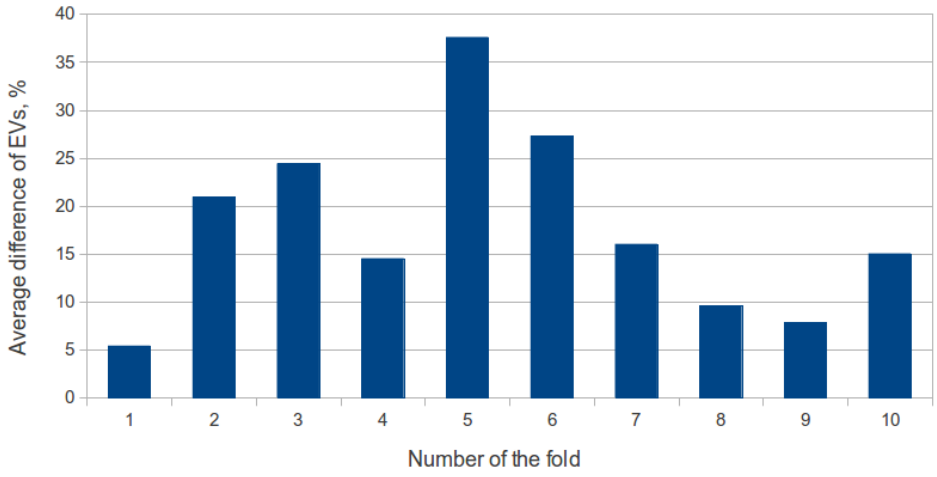

Figure 18: Average differences of expected values in the 10-fold cross validation for the Carat data.

Figure 17 represents that in the cross validation experiment, the folds two and eight have been lead to the minimum impurity, if measured by the Kullback-Leibler divergence. The average Kullback-Leibler divergence over all the folds is 0.0224485835 and the standard deviation is 0.0114797276 that both seems to be near zero. The decision tree of the fold eight will be represented in Section 6 in more detail.

Figure 18 presents the average differences in expected values between the training set and the test set, measured by each fold separately. The differences are given as percentages. The expected values in the Carat analysis present the total battery life of the device under under existing conditions. That means, for example, that in the fold number eight, there are 10% difference in the total battery life between the average case in the training set and the average case in the test set. Hence, the smaller difference aims to the larger similarity.

Because of limited size of the data sets used to the cross validation experiment, it is hard to analyze how comprehensive these measurements are in a broader perspective. The performance issues and superlinear time complexity make any wider measurements arduous to perform. One of the key elements of the distributed Big Data analysis in the cloud computing environments should be low execution time – and in this form, the distributed decision tree will not be as fast as desirable. Because the Spark system has

been found to have good performance [37], it is probable that the Spark decision tree implementation needs more optimizations.

Regardless the performance issues, Section 6 will introduce some results of the decision tree analysis in the form of decision trees found. Section 7 will summarize lessons learned and discuss some future work prospects.

6 Results

Figure 19 represents an example of a decision tree for Carat data. This decision tree has a depth of three and it is based on the data fold number 8 from the cross validation experiment in Section 5.4. The tree has been grown by circa 10 800 sample pairs as a training set, and also information presented next is based on this training set. When making any conclusions about the results, it is important to notice the results are based on a small part of the whole data set.

In Figure 19, the attribute that has lead to the split is presented inside the node. Values of the attribute are presented in connection with edges. For example, the attribute of the first split is battery temperature with values 0-20, 20-40, and 40-100, given in degrees Celsius. When the battery temperature has the value 0-20, the child node 2.1. determines that its next attribute for a split will be screen brightness. The attribute sets are individual per chain. For example, after the split by the attribute distance traveled in node 2.2. both of its children nodes 3.3. and 3.4. are split by the attribute screen brightness.

Rate class distributions of root to leaf chains are presented in boxes after each leaf node. The percentage shows a proportion of sample pairs belonging to each rate class defined in Section 5.1 and given in hours, for clarity. According to the percentage proportions, the chains with the most sample pairs in the lowest rate class (only an hour of total battery life), are:

- Battery temperature = 20-40, Battery voltage = 0-2.5, Screen brightness = -1 – 25%
- Battery temperature = 20-40, Battery voltage = 0-2.5, Screen brightness = 101-255 – 12%
- Battery temperature = 20-40, Battery voltage = 2.5-5, Network type = wimax – 66%

All these chains contain only a few sample pairs in the aggregate. That can be seen in Figure 20, that presents the decision tree of Figure 19 when the nodes have been scaled by the data set size in each split by common logarithm $log(x)$. A chain in the middle of the tree includes significantly more sample pairs than the other chains. In the depth of four, this large

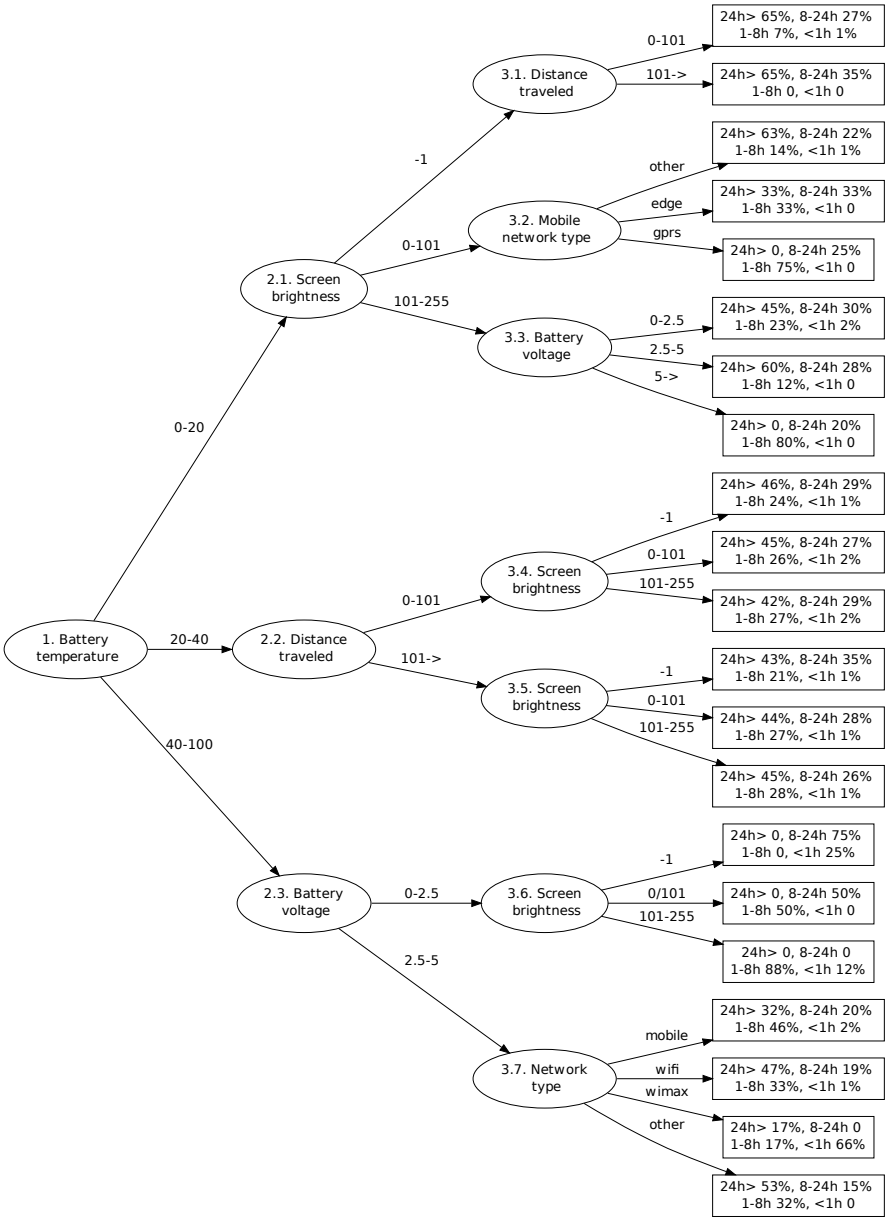

Figure 19: A result graph of depth three based on the data fold 8 from the cross validation experiment in Section 5.4. Rate class distributions of each root to leaf chain are presented in boxes.

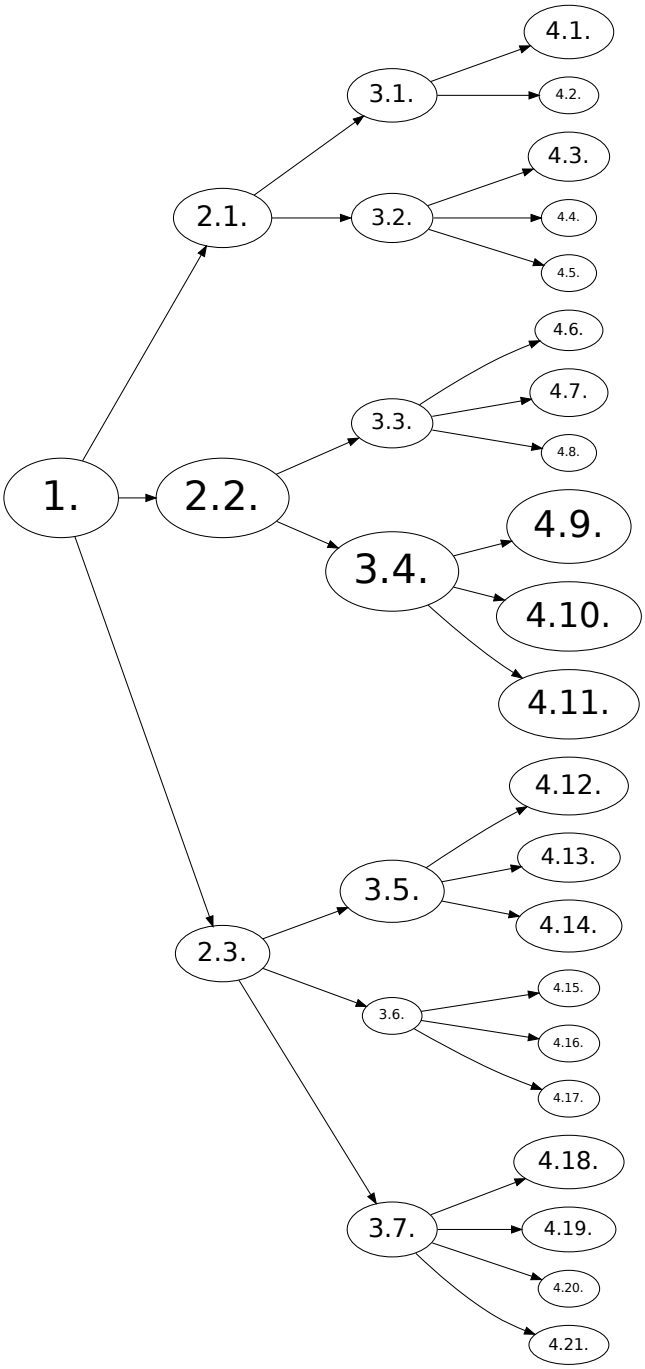

Figure 20: The decision tree of Figure 19 where the node size in the nodes is scaled by size of the data set in the current split and by common logarithm $log(x)$.

Attribute chain	**Average battery life**	**Sample pairs (class / total)**
Battery temperature = 20-40	16.17h	134 / 8137 = 1.6 %
Battery temperature = 20-40, Distance traveled = 0-101	16.11h	134 / 8070 = 1.7 %
Battery temperature = 20-40, Distance traveled = 0-101, Screen brightness = -1	19.05h	37 / 3781 = 0.98 %
Battery temperature = 20-40, Distance traveled = 0-101, Screen brightness = -1, Mobile data activity = none	19.06h	33 / 3414 = 0.96 %
Battery temperature = 20-40, Distance traveled = 0-101, Screen brightness = 0-101	16.96h	31 / 2089 = 1.4 %
Battery temperature = 20-40, Distance traveled = 0-101, Screen brightness = 101-255	14.82h	37 / 1721 = 2.1 %

Table 6: Attribute chains that have lead to a rate class of 0.027777 or higher, that means at most an hour of total battery life.

set of sample pairs starts also to scatter. Next, there are analyzed chains that do not have a high proportion of the lowest rate classes but do have a number of sample pairs belonging to these classes: an hour or less of total battery life and an hour to eight hours of total battery life.

According to the rate classes presented in Section 5.1, a rate value more than 0.027777 represents the highest energy consumption profile and the lowest total battery life, an hour only. In the training set of the decision tree in Figure 19, there are attribute chains that have led to this class with more than 30 sample pairs. They are presented in Table 6 together with the average total battery life of the chain in the entirety and the number of the sample pairs of this rate class compared to the full number of sample pairs of this chain.

Table 6 shows the following: The two first attribute chains have a same number of sample pairs, so in every time the battery temperature with the value 20-40 leads to the highest energy consumption class, there exists also the attribute distance traveled with the value 0-101. It seems that the device has been in place and in room temperature. After that, the class has been

divided by the values of the attribute screen brightness. Compared to the values of the average total battery life, the sample pairs of this rate class seems to be incorrectly classified. Regarding the number of the sample pairs in the class and in total, they are certainly in the minority.

The rate value 0.003472 to 0.027777 means that there are less than eight hours but more than an hour of total battery life in the device. The attribute chains that lead to this class are presented in Table 7. Only attribute chains of more than a hundred sample pairs are taken into account.

Table 7 shows the following results: There are much more sample pairs in the rate class 0.003472 to 0.027777 than in the rate class 0.027777 or higher. In both of the rate classes, there are mostly sample pairs where the attribute battery temperature has a value of 20-40 and the attribute distance traveled has a value 0-101, which possibly means these devices are in place and in room temperature.

After the attributes battery temperature and distance traveled, the attribute screen brightness appears again. There are 926 sample pairs in this class where the screen brightness has been set to automatic, that means the value -1. In contrast, there are 548 sample pairs in which the screen brightness has been set between the values 0 and 101, but also 407 sample pairs in which it has set between the values 101 to 255. For simplicity, the bound value 101 belongs to higher class only. This observation might be interesting, because it is a common hypothesis that the automatic screen brightness might save the battery life.

Compared to the average battery life of the chains, the averages are more positive than the observed class. When the part of the sample pairs in the class increases, also the average battery life decreases. Against previous assumptions, there are no attributes related to network connections on the top of the decision tree.

Figure 21 represents the expected values of each node in the decision tree of Figure 19. These values are hours of total battery life in the device: the more red, the lower expected value of total battery life in current node. Figure 21 is comparable to the mentioned chains that have the highest proportion of sample pairs related to the rate class of just an hour of total battery life.

The decision tree is possible to use for giving advices to the devices of the Carat community. Attributes of the new data measurements are compared

Attribute chain	**Average battery life**	**Sample pairs (class / total)**
Battery temperature = 20-40	16.17h	2177 / 8137 = 26.7 %
Battery temperature = 20-40, Distance traveled = 0-101	16.11h	2166 / 8070 = 26.8%
Battery temperature = 20-40, Distance traveled = 0-101, Screen brightness = -1	19.05h	926 / 3781 = 24.5%
Battery temperature = 20-40, Distance traveled = 0-101, Screen brightness = -1, Mobile data activity = inout	16.98h	253 / 894 = 28.3%
Battery temperature = 20-40, Distance traveled = 0-101, Screen brightness = -1, Mobile data activity = none	19.06h	801 / 3414 = 23.5%
Battery temperature = 20-40, Distance traveled = 0-101, Screen brightness = 0-101	16.96h	548 / 2089 = 26.2%
Battery temperature = 20-40, Distance traveled = 0-101, Screen brightness = 0-101, Network type = mobile	16.90h	309 / 1155 = 26.8%
Battery temperature = 20-40, Distance traveled = 0-101, Screen brightness = 0-101, Network type = wifi	19.71h	214 / 828 = 25.8%
Battery temperature = 20-40, Distance traveled = 0-101, Screen brightness = 101-255, Mobile network type = other	15.84h	407 / 1559 = 26.1%
Battery temperature = 20-40, Distance traveled = 101->	19.72h	201 / 802 =25.1%
Battery temperature = 40-100	11.09h	154 / 332 = 46.4%
Battery temperature = 40-100, Battery voltage = 2.5-5	11.96h	142 / 314 = 45.2%
Battery temperature = 40-100, Battery voltage = 2.5-5, Network type = mobile, Mobile data status = connected	12.07h	108 / 235 = 46.0%

Table 7: Attribute chains that have led a rate class 0.003472 to 0.027777, that means one to eight hours of total battery life.

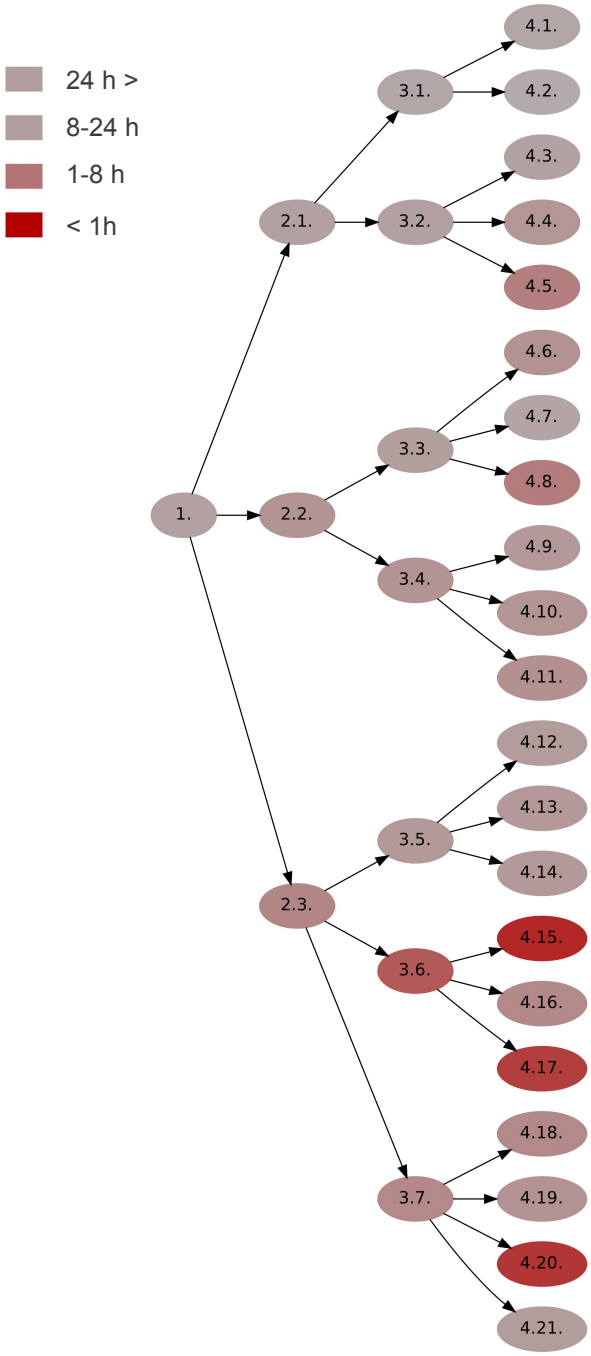

Figure 21: The decision tree of Figure 19 where the color of the node indicates the expected value of total battery life: the more red, the lower expected value of total battery life.

to the decision tree. If the new measurements seems to be fitting to the chains, which are known for leading to increase in total battery life, an advice will be sent to the device. For example, if the new measurement contains an attribute chain *battery temperature = 40-100, battery voltage = 0-2.5, screen brightness = -1*, the advice can be such as *"put screen brightness under 101"*. That is because the decision tree of this attribute combination leads to the low total battery life, but the near chain where screen brightness has the value 0-101 leads higher total battery life. For searching the decision tree, also other improving advices can be given, depending on how much the improvement should be and how the walk through the decision tree has been organized.

7 Discussion

Even with multiple optimizations, the experiments of this thesis have shown that the Spark implementation of the distributed decision tree slows down significantly when the number of items in the training data set increases. This means that it does not make sense to grow a decision tree that covers the full Carat data set or even a large subset of it. This is unfortunate, because the Spark system has been designed specifically for large scale cluster computing. Some more optimizations or even some other, more suitable or efficient algorithms, are needed.

An additional optimization might be to change the data format in which the sample pairs are represented. Currently, there are many string comparisons that are heavy to execute and generate Java objects. One effective solution would be to represent information of the sample pairs in binary format as bit vectors. With a simple API around for the binary vectors, they should be as easy to use in the algorithmic code as the objects where the attributes are presented in String format.

For the Carat data analysis system, it might be possible to try to find a decision tree that describes the data set as well as possible. When new data items are measured, they can work as an input for improving the accuracy of the old decision tree, for example, evaluating the tree towards lower error rates via cross validation technique. This evaluation model can also work when the whole data can not be used for growing the tree because of performance or memory complexity issues.

One of the aims of distributed data analysis is to develop solutions, which do not require more time or memory to be executed, even when the size of the data set grows. Certainly, the time or memory complexity for handling the new data items should not depend on the size of the old data set.

If the distributed decision tree will work efficiently in the future, an interesting approach might be to expand the decision tree for handling also other attributes than Android features. A possible choice is to handle also running applications as attributes of the decision tree. This increases the number of possible chains or attribute combinations rapidly, but it might offer a better overview to energy consumption of the devices in the Carat project.

There are also other data mining algorithms that can be even more

effective when trying to find attribute combinations. For example, some of the frequent itemset mining algorithms or association rule mining algorithms might be interesting to implement for the Spark distributed computing environment. The decision tree is focused on organized combinations, because the splits have been made sequentially. This can lead some information losses, for example, if the split has been made based on the attribute of minimal difference to the second best. For example, the frequent itemset mining algorithms should provide a solution for these kind of cases, because they do not take the attribute order in the combinations into account [34].

A broader perspective, the data mining and machine learning algorithms offer a set of tools for better understanding and summarizing the information of the large data sets. Implementing these algorithms efficiently requires deep understanding of the distribution system and paradigms, such as Spark and MapReduce. This thesis has been a great lesson on how to implement more complex algorithms for Spark, and how to take into account the size of the large data set and the limited memory even in the cloud computing environment.

8 Conclusion

This thesis has introduced the current fields of Big Data, cloud computing and distributed machine learning. It has presented what cloud computing environments can provide for analyzing large data sets efficiently. Especially, this thesis has presented Berkeley Data Analysis Stack and two of its systems: the dynamic cluster resource manager Mesos, and the in-memory cluster computing system Spark.

Spark extends a well-known computing paradigm MapReduce. In contrast to MapReduce that provides two functions only, map and reduce, the Spark system offers a diverse library of functions and memory control operators. MapReduce requires that the developer implements both the map and reduce functions, whereas Spark offers the API of easy to use Scala-like functions.

This thesis has presented the Carat project, that collects energy consumption data from mobile devices, such as smart phones and tablets. The devices send their measurements to the cloud, where the Carat analysis system performs data analysis, and returns advices for energy control as feedback. The Carat analysis system has been implemented on the Spark system.

This thesis has presented the Spark implementation of a distributed decision tree algorithm. As an example, the decision tree has been used on the Carat data. The decision tree uses the features of Android devices, such as screen brightness, distance traveled, and network connections, as attributes of the algorithm. The decision tree algorithm produces attribute chains, or paths of the decision tree, that describe combinations that might lead to particular energy consumption behavior. The aim of the decision tree analysis is to find and predict possible energy anomalies.

The Spark decision tree algorithm has been validated by the synthetic data set, where the results are comparable to the Matlab decision tree implementation, and by the K-fold cross validation technique, where it has been used on the real Carat data samples. Some interesting results of the decision tree analysis have also been represented at the end of this thesis, together with advantages and disadvantages of the decision tree analysis.

In conclusion, some other algorithms have been proposed to compensate or supplement the decision tree analysis, for example frequent itemset mining algorithms and association rule mining algorithms. Some other interesting

algorithms to implement on Spark would be linear regression and nearest neighbor classification algorithms. They can offer a broader perspective for the entire data.

When the suitable algorithms have been recognized and implemented on Spark, the cloud environment can offer data analysis and data mining as services such as any other computing properties. This means that the mobile application can contact to the cloud, send the data measurements and, after the data processing and executing the algorithms, get some feedback from the cloud based on the new measurements and all the data gathered in the past. This process can be named as *Data Mining as a Service* (DMaaS).

The DMaaS systems should have some requirements related to efficiency and performance. Frequently, the devices need feedback in short order without any unnecessary delays. Approximated results with sufficient feedback can be enough if the exact results require more time to be executed – it is possible to improve the results afterwards. Short response times advocate streaming systems and algorithms, for example. The Spark system has been developed to the Streaming Spark system already.

The sensor data provides its own challenges. There are lot of different sorts of sensors. Mobile devices have sensors for location, position, acceleration, battery temperature and voltage, for example. Heterogeneity of the sensing devices can be wide. For example, mobile devices have multiple models, operating systems and hardware components. The sensors can measure incorrectly or imprecisely and the devices are frequently moving, which can lead to possible low data reliability.

For handling large amounts of the sensor data, the DMaaS system should provide appropriate data locations with sufficient distributed file systems, efficient data mining algorithms for bringing some sense to the versatile data, and also machine learning tools for predicting the future behavior of the sensing devices. To reduce the unnecessary data measurements or computing load of the analysis system, it can be relevant to discuss how the sensing devices as clients can also participate to the data processing, for example, by preprocessing the measurements already in the mobile device.

After the work of this thesis, the concept of Data Mining as a Service holds great potential for future work. The possible research questions would be related to the topics of data streaming, collaboration of client-side computing and cloud computing, approximated and immediate results from the analysis

system, machine learning algorithms, and appropriate data locations. At its best, the DMaaS systems would combine the best efforts of cloud computing, data mining algorithms and Big Data, for use of different kind of applications, but especially sensing applications in mobile devices.

9 References

[1] Amazon elastic compute cloud (Amazon EC2). http://aws.amazon.com/ec2.

[2] Apache Hadoop. http://hadoop.apache.org.

[3] Apache Mahout. http://mahout.apache.org.

[4] Apache Mesos: Dynamic resource sharing for clusters. http://incubator.apache.org/mesos.

[5] Berkeley data analysis stack (BDAS). https://amplab.cs.berkeley.edu/bdas.

[6] Carat: Collaborative energy diagnosis. http://carat.cs.berkeley.edu.

[7] Matlab classification and regression tree. http://www.mathworks.se/help/stats/classregtree.html.

[8] Matlab: the language of technical computing. http://www.mathworks.se/products/matlab/.

[9] OpenStack cloud software. http://www.openstack.org.

[10] Scala language. http://www.scala-lang.org.

[11] Spark: Lightining-fast cluster computing. http://www.spark-project.org.

[12] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[13] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A Berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[14] Leo Breiman, Jerome Friedman, Richard Olshen, and Charles Stone. *Classification and regression trees*. Wadsworth, Belmont (CA), 1984.

[15] Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M. Hellerstein, and Caleb Welton. MAD skills: New analysis practices for big data. In *Proceedings of the VLDB Endowment*, pages 1481–1492. VLDB Endowment, 2009.

[16] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation*, OSDI '04, 2004.

[17] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[18] Jeffrey Dean and Sanjay Ghemawat. MapReduce: A flexible data processing tool. *Communications of the ACM*, 53(1):72–77, 2010.

[19] Ian Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. Cloud computing and grid computing 360-degree compared. In *Proceedings of Grid Computing Environments Workshop*, GCE '08, pages 1–10, 2008.

[20] Amol Ghotingm, Rajasekar Krishnamurthy, Edwin Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. SystemML: Declarative machine learning on MapReduce. In *Proceedings of IEEE 27th International Conference on Data Engineering*, ICDE '11. USENIX Association, 2011.

[21] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of NSDI '11: 8th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '11, pages 295–308. USENIX Association, 2011.

[22] Benjamin Hindman, Andy Konwinski, Matei Zaharia, and Ion Stoica. A common substrate for cluster computing. In *USENIX Workshop on Hot Topics in Cloud Computing*, HotCloud '09. USENIX Association, 2009.

[23] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, IJCAI 95, pages 1137–1143. Morgan Kaufmann, 1995.

[24] Tim Kraska, Ameet Talwalkar, John Duchi, Rean Griffith, Michael F. Franklin, and Michael Jordan. MLbase: A distributed machine-learning system. In *Proceedings of 6th Biennial Conference on Innovative Data Systems Research*, CIDR '13, 2013.

[25] Solomon Kullback and Richard A. Leibler. On information and sufficiency. *Annals of Mathematical Statistics*, 22(1):79–86, 1951.

[26] Yu Li, Wenming Qiu, Uchechukwu Awada, and Keqiu Li. Big data processing in cloud computing environments. In *Proceedings of 12th In-*

ternational Symposium on Pervasive Systems, Algorithms and Networks, pages 17–22. IEEE, 2012.

[27] Jian Lin, Li Zha, and Zhiwei Xu. Consolidated cluster systems for data centers in the cloud age: A survey and analysis. *Frontiers of Computer Science*, 7(1):1–9, 2013.

[28] James MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*. University of California Press, 1967.

[29] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, and Angela Hung Byers. Big data: The next frontier for innovation, competition, and productivity. Technical report, McKinsey Global Institute, May 2011.

[30] Peter Mell and Timothy Grance. The NIST definition of cloud computing. Technical Report NIST Special Publication 800-145, National Institute of Standards and Technology, Sep 2011.

[31] Adam Oliner, Anand Padmanabha Iyer, Ion Stoica, Eemil Lagerspetz, and Sasu Tarkoma. Collaborative energy debugging for mobile devices. In *Eighth Workshop on Hot Topics in System Dependability*, HotDep '12. USENIX Association, 2012.

[32] Adam Oliner, Anand Padmanabha Iyer, Ion Stoica, Eemil Lagerspetz, and Sasu Tarkoma. Carat: Collaborative energy diagnosis for mobile devices. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, 2013. To appear.

[33] Adam Oliner, Anand Padmanabha Iyer, Ion Stoica, Eemil Lagerspetz, and Sasu Tarkoma. Carat: Collaborative energy diagnosis for mobile devices. Technical Report UCB/EECS-2013-17, EECS Department, University of California, Berkeley, Mar 2013.

[34] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Pearson Education, 2006.

[35] Hong-Linh Truong and Schahram Dustdar. On analyzing and specifying concerns for data as a service. In *Proceedings of Services Computing Conference*, APSCC '09, 2009.

[36] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Technical Report UCB/EECS-2011-82, EECS Department, University of California, Berkeley, Jul 2011.

[37] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of NSDI '12: 9th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '12, pages 15–28. USENIX Association, 2012.

[38] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing*, HotCloud '10. USENIX Association, 2010.