



Masterarbeit am Institut für Informatik der Freien Universität Berlin,
Arbeitsgruppe Intelligente Systeme und Robotik
In Zusammenarbeit mit der Arbeitsgruppe Maschinelles Lernen der
Technischen Universität Berlin



Masterarbeit: Big Data and Machine Learning: A Case Study with Bump Boost

Maximilian Alber
Matrikelnummer: 4452645
Eingereicht bei: Prof. Dr. Raúl Rojas
Betreuer: Dr. Mikio Braun (TU Berlin)

Berlin, 19. Februar 2014

Abstract

With the increase of computing power and computing possibilities, especially the rise of cloud computing, more and more data accumulates, commonly named Big Data. This development leads to the need of scalable algorithms. Machine learning always had an emphasis on scalability, but few well scaling algorithms are known. Often, this property is reached by approximation. In this thesis, through a well structured parallelization we enhance the Bump Boost and Multi Bump Boost algorithms. We show that with increasing data set sizes, the algorithms are able to reach almost perfect scalability. Furthermore, we investigate empirically how suitable Big-Data-frameworks, i.e. Apache Spark and Apache Flink, are for implementing Bump Boost and Multi Bump Boost.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

19. Februar 2015

Alber Maximilian

Acknowledgment

A thesis can hardly be written without any help. In the first place, I would like to thank Mikio Braun for assisting me and for his valuable insights. For her English lessons and for cross-reading my gratitude goes to Grete, for his advice to Marcin. Finally, I thank Nina for her company during the long work hours and my mother for sustaining me.

Contents

1	Introduction	1
1.1	Objectives of this Thesis	2
1.2	Organization of this Thesis	3
2	Big Data	5
2.1	The Term	5
2.2	Provocations for Big Data	6
2.3	Big Data and Machine Learning	7
3	Scaling and Parallelization	9
3.1	Scalability	9
3.2	Parallel Computing	9
3.2.1	Theory	9
3.2.2	Problems	12
3.2.3	Parallelism Characteristics	13
4	Machine Learning and Bump Boost	15
4.1	Background	15
4.1.1	Machine Learning	15
4.1.2	Supervised Learning	16
4.1.3	Regression and Classification	16
4.1.4	Gradient Methods	16
4.1.5	Cross Validation	17
4.1.6	Boosting	18
4.1.7	Kernel methods	18
4.2	Support Vector Machine	19
4.2.1	Implementation	21
4.3	Bump Boost	22
4.3.1	The Algorithm	22
4.3.2	Characteristics	26
4.3.3	Parallelization	28
5	Related Work	36
6	Tools and Frameworks	39
6.1	Parallel Computing Device Frameworks	39
6.1.1	Cuda	41
6.1.2	OpenCl	41
6.2	Cluster Frameworks	41
6.2.1	Apache Big Data Stack	42
6.2.2	Hadoop, HDFS, and YARN	44
6.2.3	Spark and MLlib	45

6.2.4	Flink	46
6.3	Python	46
6.3.1	Scipy, Numpy, Matplotlib	46
6.3.2	CudaMat	46
6.3.3	PyOpenCl	47
6.4	SVM Programs	47
6.4.1	LIBSVM	47
6.4.2	LaSVM	47
7	Implementations	48
7.1	General Framework	48
7.2	Java	49
7.3	Python	49
7.3.1	Development Version	50
7.3.2	Parallelized Version	50
7.3.3	Parallel and Remote LCC	53
7.3.4	Numpy LCC	54
7.3.5	CudaMat LCC	54
7.3.6	PyOpenCL LCC	55
7.4	Big Data Frameworks	55
7.4.1	Spark	55
7.4.2	Flink	58
7.5	Selected Code Comparisons	62
7.5.1	Draw Center	62
7.5.2	R-Prop	64
8	Competitive Solutions	66
8.1	SVM Solvers	66
8.2	MLlib	66
9	Data Sets	67
9.1	Splice	67
9.2	MNIST	67
9.3	Forest Cover	67
9.4	Checkers	68
10	Experiments and Results	69
10.1	Experiment Setup	69
10.1.1	Cycle and Parameters	69
10.1.2	Measurements and Evaluation	70
10.2	Results	71
10.2.1	Basic Results	72
10.2.2	Bump Boost versus Competitors	75
10.2.3	Scaling	77

10.2.4 Spark	81
11 Conclusion and Perspective	84
11.1 Conclusion	84
11.2 Perspective	85
A Computing Systems	88
A.1 GPU-Server	88
A.2 Cluster	88
B Digital Content	89
C Copy of Bump Boost Paper	90

1 Introduction

The development of computer science was always driven by the increasing computing possibilities. Even though the empirical law of Moore [M⁺75] seemed always valid, in the last two decades computers have pervaded the daily life as never before. Beginning with the rise of the personal computer over the laptop evolution to the revolution of smart phones, computers became an essential part of industrial nations. In the developing countries the increased usage of mobile phones leads to new chances and opportunities, mobile banking and micro credits change the economic reality of millions of people [Por06, page 8-18]. For computing backends the development of so called cloud computing infrastructure, f.e. Amazon Web Services, Google App Engine, Microsoft Azure, eased the deployment and handling of computer clusters and distributed applications. This trend increased efficiency, thus decreased the costs of server infrastructure for companies, and, even more, it lowered the barrier for launching services. Several services such as Dropbox do not have physical servers, but rely solely on cloud services, in this case Amazon S3 [dro15]. In the last decades, also the emerging of software companies such as Google, Amazon, and Facebook, to name the most prominent ones, tied millions of users to their services.

From the rise of personal computers to the one of smart phones, from sensor networks to the internet, from deployable cloud computing solutions to the mobile phones in the developing world, from Google to Facebook, more and more computing entities and users are present, and more and more data is created and generated. The broad term “Big Data” names the challenge of handling such data, i.e. data that is too complex or too large for traditional approaches.

Driven by this development, the Apache foundation plays a key role. With the Apache Big Data Stack it provides the most popular solution for Big Data applications. Hadoop, to name the working horse, offers a file system to distribute large data files and resource managers to distribute applications on clusters. The recent development of large data processing lead from the simple Map-Reduce paradigm [DG08] to more complex frameworks such as Apache Spark and Apache Flink, which was and is actively developed at TU Berlin.

In machine learning scaling with increasing data sizes was always an important property, especially because the increase of data often goes hand in hand with increased accuracy. But still few well scaling algorithms are known and most attempts to adapt existing algorithms to larger data sizes were made by approximating partial solutions. Even though this is per se no problem, because the final prediction rate is what matters in machine learning, it indicates that most scaling efforts are made on existing algorithms by “accelerating” them and few are designed to scale. To name an example, even with approximations Support Vector Machines are only able to handle

1.1 Objectives of this Thesis

a moderate quantity of input data. Other solutions, such as deep neural networks can handle large amounts of complex data and benefit from parallel computing[KSH12], but still suffer from their computational cost and model complexity. Apache Spark provides a machine learning library called MLlib with basic and easily scalable solution such as linear Support Vector Machines and logistic regression based on stochastic gradient descent. Even though these solutions scale well, they have restricted success on complicated data due to their simple machine learning models. More successful is the MLlib recommender model [mll15], which is used by Spotify [spo15] for recommending users music they may like, i.e. collaborative filtering. The model approximates a matrix factorization via alternating least squares (ALS, [KBV09]).

In the past, data sets, especially labeled ones, were often of modest size due to missing data sources and small labeling capacities. With the increase of computing entities, the data emergence is increasing, too. Also the problem of missing labels is less prominent due to crowd-sourcing approaches such as Amazon Mechanical Turk [Ipe10]. Both lead to larger data sets.

In other words, with the appearance of Big Data machine learning needs to adapt to the increasing data set sizes, hence has need for scalable algorithms that can cope with complex data.

This thesis introduces Bump Boost and Multi Bump Boost, two boosting based algorithms created by Mikio Braun and Nicole Krämer. We try to make their algorithms scalable. If successful, we would like to implement and empirically test the result, furthermore, show the revised algorithm scaling properties and compare them with state-of-the-art alternatives.

With the popularity and successful application of Big Data frameworks such as Spark on various machine learning tasks, f.e. the Apache Spark MLlib recommender model, the question arises, if Bump Boost and Multi Bump Boost can benefit from them. In this thesis, we try to present a solution for this question by implementing Bump Boost and Multi Bump Boost on the popular/new Big Data frameworks Apache Spark and Apache Flink. Besides a quantitative analysis concerning the training times, we would like to show a qualitative analysis of the implementation effort.

This thesis is written for a reader with a good computer science background and elementary knowledge of machine learning. Even if the research field is quite advanced, we try to ease the understanding by providing the most important background.

In the rest of this chapter, we state in more detail the aim of this thesis and conclude by outlining the structure of it.

1.1 Objectives of this Thesis

The following list represents the objectives of this thesis:

1.2 Organization of this Thesis

Scalability: The first and main purpose is to develop a parallelized and scalable variants of the Bump Boost and Multi Bump Boost algorithms. The theoretical results should be confirmed by testing them on sufficiently large data sets.

To be more precise, we expect from our solution, if possible, the complete same behavior as of the sequential solution. If not, the approximations should be as accurate as possible. The solution should exhibit a good scaling behavior, i.e. when parallelizing the same work on n instances, the run time should be approximately a n -th of the sequential one and the overall run time should scale linearly with increasing data set sizes.

Big Data frameworks: The second objective is to examine the suitability of Apache Spark and Apache Flink for implementing Bump Boost and Multi Bump Boost. Again, next to the empirical evaluation, the resulting programs should be tested on their scaling properties.

To be more accurate, the final solution should be easily understandable and comprehensible. The adaption from the traditional programming model to the semantics imposed by the frameworks should be as small as possible. Finally, we expect the solutions to scale well, i.e. with the same criteria as in the first objective.

1.2 Organization of this Thesis

The thesis is structured as follows:

Chapter 2: Big Data

This chapter clarifies our understanding of Big Data and sketches how Big Data influences machine learning and this thesis.

Chapter 3: Scaling and Parallelization

The broad terms scaling and parallelization are introduced and their meaning in our context is specified. In addition, we present the basic theory and principles of parallelization.

Chapter 4: Machine Learning and Bump Boost

After giving an introduction to machine learning and the theoretical background of Bump Boost and Multi Bump Boost, the algorithms themselves are described. The chapter concludes with the description of the parallelized Bump Boost and Multi Bump Boost algorithms.

Chapter 5: Related Work

In this chapter we show related work in the field of machine learning.

Chapter 6: Tools and Frameworks

The used programming tools and frameworks, for example the Apache Big Data stack, are described in this chapter.

1.2 *Organization of this Thesis*

Chapter 7: Implementations

The first part of this chapter depicts our implementation of Bump Boost and Multi Bump Boost. The second part is on how we try to realize a solution with Apache Spark and Apache Flink.

Chapter 8: Competitive Solutions

This chapter treats the algorithms we used for comparison.

Chapter 9: Data Sets

The data sets used for our experiments are given in this chapter.

Chapter 10: Experiments and Results

After introducing our experiment setups, we show and describe the quantitative results of this thesis.

Chapter 11: Conclusion and Perspective

In this final chapter we conclude and give a perspective on future questions.

Appendix:

In the appendix we provide the original, but never published, paper on Bump Boost and describe the content of the enclosed DVD.

2 Big Data

In the introduction we already mentioned how the field of computer sciences changed in the last decades. One of the latest developments is “Big Data”. This is a rather broad term and in this chapter we try to narrow its meaning and give some examples. After that, we subsume a paper entitled “Six Provocations for Big Data” and conclude by sketching the intersection between Big Data and machine learning.

2.1 The Term

The catch phrase, namely “Big Data”, emerged for problems involving mass of data. While one of the challenges in computer science always has been the adaption to “larger” problems, the novelty in this term is, that it is picked up and made popular by media and marketing, similar to “cloud computing”. The claims and the flexibility of this term can be shown with two definitions.

For example in [MSC13, page 6] the phrase is introduced with the sentence “big data refers to things one can do at a large scale that cannot be done at a smaller one, to extract new insights or create new forms of value, in ways that change markets, organizations, the relationship between citizens and governments, and more.” The most valuable thought in this is, that Big Data requires large scale solutions. For the rest, this definition relates hopes and claims to the term and it shows what some people have in mind concerning Big Data: creating insights out of data.

Regarding this first definition, i.e. analyzing data, some applications of Big Data were widely spread in media. The police of Los Angeles and Santa Cruz, for example, tries to predict where crime is likely to occur by exploiting data accumulated over the years [pol15a] [pol15b]. Another popular example for Big Data is that Google provides an estimator for flu activity around the world by exploiting search data [gool15].

A more precise definition is given in [MCB⁺11, page 1]: ““Big data” refers to datasets whose size is beyond the ability of typical database software tools to capture, store, manage, and analyze. This definition is intentionally subjective and incorporates a moving definition of how big a dataset needs to be in order to be considered big data, i.e., we don’t define big data in terms of being larger than a certain number of terabytes (thousands of gigabytes). We assume that, as technology advances over time, the size of datasets that qualify as big data will also increase. Also note that the definition can vary by sector, depending on what kinds of software tools are commonly available and what sizes of datasets are common in a particular industry. With those caveats, big data in many sectors today will range from a few dozen terabytes to multiple petabytes (thousands of terabytes).”

This definition incorporates following characteristics of Big Data:

2.2 Provocations for Big Data

Datasets: the common denominator is that Big Data is about data, large amounts of data.

Subjective: the term has no actual definition, thus its interpretation is subjective.

Variable Size: the relation of size in Big Data to the current state-of-the-art, thus the actual size will change over time.

Vary by area: depending on the application area, the considered size may vary.

This report relates Big Data to the data base field and gives an estimate of the data sizes falling into the schema, namely from few terabytes to petabytes. Where does this data come from? To name some sources: log files, emails, transactions, social media, photos, videos etc. Just popular service providers such as Google or Facebook have each day billions of users, who generate, i.e. by triggering log messages, posting content, sending messages, uploading pictures or videos etc., a mass of data. The evaluation of these large amounts of data was and is a challenge, even for big companies, and gives a more technical notion to the term Big Data.

We conclude by giving an own definition of Big Data: Big Data names data collections, that impose problems to the state-of-the-art algorithms and applications due to their size and/or complexity. Regarding the field of machine learning, Big Data are data sets that cannot be handled by state-of-the-art algorithms in acceptable time or with an acceptable modeling result.

To clarify, two problems with Big Data mainly arise in this field. Often the known algorithms with sophisticated generalization and modeling capacity have bad scaling properties, f.e. non-linear support vector machines, or a high computational cost, f.e. deep neural networks. On the other hand, well scaling algorithms, f.e. linear support vector machines and other linear models, can handle large amounts of data, but not their complexity. For us, data sets that are concerned by both problems belong to Big Data.

2.2 Provocations for Big Data

In [C⁺11] the authors sketch six problems around Big Data. Even though they are not of technical nature, we consider them as important and with the hype on Big Data they are often ignored. We cite and summarize four of them:

“Automating research changes the definition of knowledge”:

Data is king and numbers speak truth. With the emerge of Big Data, people relate it with phenomenal capabilities. The authors state that such claims reveal an “arrogant undercurrent in many Big Data debates” [C⁺11, page 4]. In such scenarios creativity gets replaced by

2.3 *Big Data and Machine Learning*

the worship of data, data that lacks the “regulating force of philosophy.” [C⁺11, page 4] In other words, we should not forget restrictions imposed to Big Data and the related tools.

“Claims to objectivity and accuracy are misleading”:

“Interpretation is at the center of data analysis. Regardless of the size of a data set, it is subject to limitation and bias. Without those biases and limitations being understood and outlined, misinterpretation is the result. Big Data is at its most effective when researchers take into account the complex methodological processes that underlie the analysis of ... data.” [C⁺11, page 6]

“Just because it is accessible doesn’t make it ethical”:

By tracking public Facebook user profiles, scientists of Harvard tried to analyze changing characteristics over time. Even though, the released data was anonymized, quickly it was shown that deanonymizing is possible, compromising people’s privacy. Other studies give further examples of how individuals can deanonymized with enough data, f.e. [SH12]. Big Data abstracts reality, thus one should not forget to consider that “there is a considerable difference between being in public and being public” [C⁺11, page 11-12] and where the data comes from.

“Limiting access to Big Data creates new digital divides”:

Collecting, cleansing and analyzing data is a tedious task. Besides, this data is a valuable source. Resulting, most companies restrict access to their resources. F.e. Twitter offers access to nearly all Tweets only to a selected number of firms and researchers. In such cases a scientist with access to such data is privileged over others. Next to this access question, handling Big Data imposes requirement to specific knowledge and infrastructure, which again is a potential divide.

2.3 **Big Data and Machine Learning**

Given this introduction to Big Data, the question of how this relates to machine learning is left. First of all, data sets used in machine learning tasks are often inspired by real world scenarios and their accumulated data. In future, more tasks with large data sets will arise as more data will be collected. An example is the Netflix competition of 2006, where researches were challenged to develop a recommender system using a data set with 100 million samples [BL07].

Besides the sole size, data sets with increased complexity were released. The popular ImageNet data set [DDS⁺09] with millions of images is an example. The task is to localize and/or classify objects of 1000 classes inside these pictures. Each year a competition called “Large Scale Visual Recognition Challenge” is held to determine the state-of-the-art.

2.3 *Big Data and Machine Learning*

Both examples fall into our definition, because size and complexity forced and force researcher to create new solutions. But while in these two cases the amount of data may seem justified, due to the complex task, in other cases more data might not lead to new insights. Depending on the complexity, the gain due to this additional data might be negligible. In contrast, additional data should not harm the prediction success, contrariwise, more data usually leads to better generalization.

The scaling properties of popular machine learning algorithms, such as f.e. Support Vector Machines, are a problem when data set sizes are too large. As a solution, data sets can be sampled to one with a smaller size, but if the underlying problem is too complex the machine learning algorithm might be not able to generalize well.

In this thesis, we do not want to examine such questions, i.e. how much data is really needed by an algorithm to generalize well or what the benefit of more data might be. Instead we want to emphasize on the scaling of the algorithms, i.e. of Bump Boost and Multi Bump Boost.

3 Scaling and Parallelization

This section gives an overview over the broad terms scaling and parallelization. We aim to clarify what they mean in our context and to introduce some theoretical constraints and criteria.

3.1 Scalability

As “Big Data” “scalability” is not well defined, i.e. there is no generally-accepted definition [Hil90]. The term itself is related to the word scale and intuitively we connect it to a desirable result after a change of scale in the problem solving capacity or the problem size itself.

For this work we define two notions of scalability. For both we understand as problem the training time of an algorithm and as a worker an independent instance, i.e. process.

The first one is related to the problem solving capacity i.e. in our case we would like the runtime to decrease as the number of workers increases. To be more precise, we would like to have with n workers a speedup of n (see next section), i.e. with n workers the problem should be solved in n times as fast as one worker is able to.

The second one is related to the problem size itself, i.e. generally the number of data samples in the training set. Again to be more precise we would like to have the runtime doubling, if the problem size doubles and the same problem solving setup tackles them.

3.2 Parallel Computing

Under parallel computing we understand the execution of two or more different calculations at the same time. In the further we want to restrict us and assume that these parallel calculations belong to the same algorithm.

“Parallelization” in this context means the execution of an algorithm in parallel instead of sequential manner. The goal of this modification can be a runtime improvement or to enable an algorithm handling larger input dimensions by using more computational power.

The rest of this section describes the most concerning theory, followed by difficulties imposed by parallel computing. Finally, the major ways to parallelize a computer program are shown.

3.2.1 Theory

Theoretical Constraints

At first we would like to introduce the notion of speedup $S(N)$. It is

3.2 Parallel Computing

defined by [Geb11, page 15/16]:

$$S(N) = \frac{T(1)}{T(N)} \quad (1)$$

for N parallel computing instances, where $T(1)$ is the algorithm time using a single one and $T(N)$ is the algorithm time using N .

In a desirable situation, we have no overhead and the algorithm is fully parallelizable, thus $T(N) = T(1)/N$ holds. Which results in $S(N) = N$.

A theoretical limit for the speedup is set by Amdahl's law [Amd67]. Assuming the parallelizable fraction of an algorithm is f , thus the serial one is $1 - f$, then the time on N instances can be written as:

$$T_{Amd}(N) = T(1)((1 - f) + \frac{1}{N} * f) \quad (2)$$

This implies the maximum speedup of:

$$S_{Amd}(N) = \frac{1}{(1 - f) + \frac{1}{N} * f} \quad (3)$$

So to say, the goal must be a big parallelizable fraction of the algorithm, i.e. $1 - f \ll f/N$, to gain a real speedup. Another insight is that the speedup saturates when N gets large:

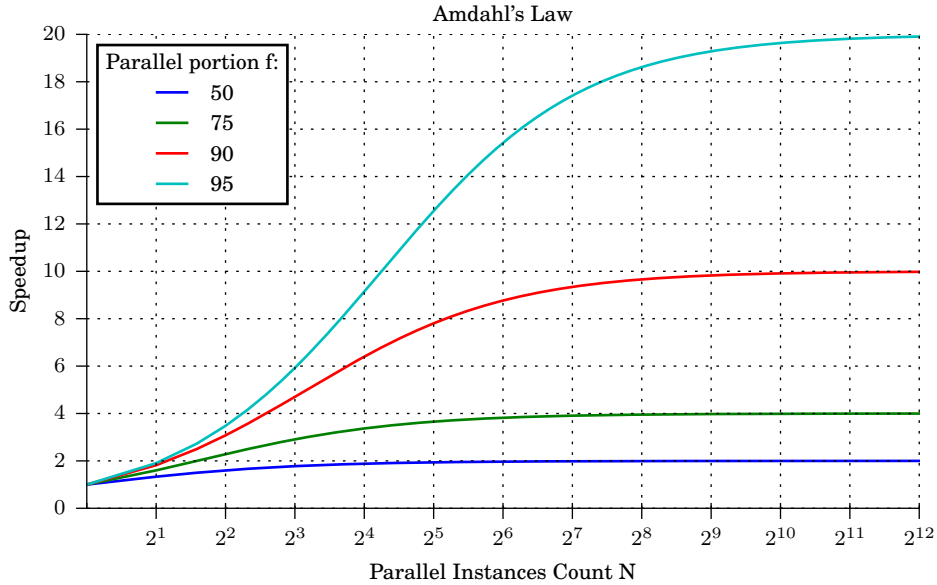


Figure 1: Examples how Amdahl's law evolves with increasing number of parallel instances.

The drawback in Amdahl's law is the fixed input size. Generally, in real world settings the computation can be solved more efficiently when

3.2 Parallel Computing

the problem, i.e. data, size increases. This is addressed by Gustafson-Barsis's law [Gus88]. Given the execution time on N computing instances the computing time on N instances can be written as:

$$T_{GB}(N) = T(N) * ((1 - f) + f) \quad (4)$$

We can describe the computing time on one instance as:

$$T_{GB}(1) = T(N) * ((1 - f) + N * f) \quad (5)$$

This results in the maximum speedup of:

$$S_{GB}(N) = (1 - f) + N * f = 1 + (N - 1) * f \quad (6)$$

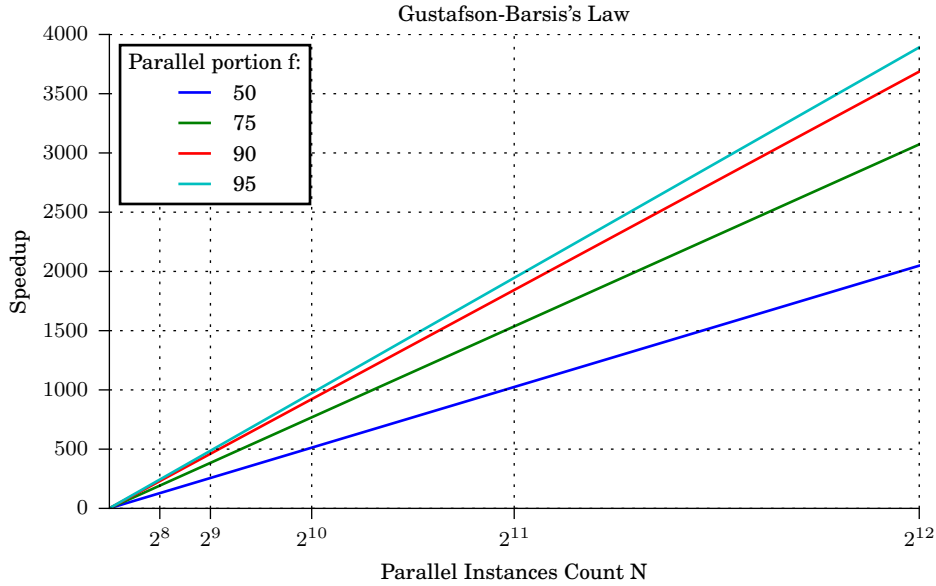


Figure 2: Examples how Gustafson-Barsis's law evolves with increasing number of parallel instances.

“... with a distributed memory-computer, larger size problems can be solved. This model proves to be adapted to distributed-memory architectures and explains the high performance achieved with these problems.” [Roo00, page 228]

Both theories give quite different results due to their viewpoint. Amdahl's law treats the problem size as constant and only the parallel fraction can be reduced, while in Gustafson-Barsis's law the time of the parallel fraction is fixed and the sequential solution scales with N .

Computation Taxonomy

3.2 Parallel Computing

There are several ways to describe computation devices. One of the most popular is Flynn's Taxonomy [Fly66]. Its main focus is on how program instructions and program data relate to each other:

Single Instruction Stream, Single Data Stream (SISD):

This class of computers is characterized by a strict scheme in which a single instruction operates on a single data element at the time. This model fits best with the traditional Von Neumann computing model [VN93]. Early single core processors belong to this class.

Single Instruction Stream, Multiple Data Streams (SIMD):

When several processing units operate on several data elements and all is supervised by a single control unit, a computing device belongs to this class. An example are vector processors, which are used f.e. in GPUs.

Multiple Instruction Streams, Single Data Stream (MISD):

The characteristic of this class is given by several instructions, that are performed simultaneously on the same data. There can be two interpretations: "A class of machines that would require distinct processing units that would receive distinct instructions to be performed on the same data. This was a big challenge for many designers and there are currently no machines of this in the world." [Roo00, page 3] In a more broad definition pipeline processors, i.e. processors, which apply different instructions to one single data stream in a pipeline in one time instance, can be seen as SIMD, if we classify the data stream as one piece of data. [Roo00, page 3]

Multiple Instruction Streams, Multiple Data Streams (MIMD):

Typical multiprocessors or multi computer systems belong to this last class, which is described by several instructions that perform on different data elements in the same time.

3.2.2 Problems

Not all algorithms are parallelizable. Most intuitively two program fragments need to be scheduled in sequential manner, if one's input depends on the output of the other. According to Amdahl's law, no program can execute faster than the longest sequential part. This part is given by the so called critical path, i.e. the longest path of sequential program fragments.

Next to some other possible and less important dependencies, f.e. control dependencies, where execution of an instruction depends on some (variable) data [Roo00, page 115], data dependencies restrict the parallelization success. They are formally described by the Bernstein's conditions [Ber66].

3.2 Parallel Computing

According to them, two program P_i, P_j fragments, with the input and output variables I_i, I_j and O_i, O_j , are independent, i.e. they can be executed in parallel, if the following conditions hold:

$$\begin{aligned} I_i \cap O_j &= \emptyset \\ I_j \cap O_i &= \emptyset \\ O_i \cap O_j &= \emptyset \end{aligned} \tag{7}$$

The last condition represents the case, in which one fragment would overwrite the output of another.

Following to this input/output relations, it is possible to build a directed, acyclic graph representing the data flow and possible parallelization opportunities.

Beside this theoretical barrier, the implementation of in-parallel executed programs can be challenging. Even though the single parallel instances execute their program independently, their results need to get combined together. The major problems are caused by communication, data access and data consistency between the parallel instances. F.e. deadlocks, lifelocks, race conditions can arise, to name some well known problems.

In some theory, f.e. in Amdahl's law, there is no notion of overhead, i.e. effort induced by parallel computing. This overhead generally increases with more parallel instances. Thus, more computing instances do not result necessarily in a faster execution. Usually this overhead is caused by the bigger communication effort. We speak of parallel slowdown, when more parallel instances solve a problem more slowly than less instances.

3.2.3 Parallelism Characteristics

Programs can be parallelized on different levels. Some examples are Bit-parallelism, where single Bit-operations are carried out in parallel by the CPU, instruction-parallelism, where instruction are performed in parallel, and program-parallelism, where different programs are scheduled in the same time instant. [RR13, page 110]

The effort can further be classified by the characteristics of the parallel computations. F.e. in data parallelism several computing units perform the same operations on the different data parts, whereas control parallelism is given when simultaneously performing several different instructions on different data. The former usually is arising on SIMD or MIMD computer systems, the later on MIMD environments. [Roo00, page 117-199]

In practice, data parallelism can be achieved by programming GPU devices or using data parallel programming languages s.a. Fortran 90 [RR13, page 112]. While control parallelism is given in multi process-, thread-, and/or host-programs.

Another characteristic of parallel programs is the way in which they communicate. There are two basic possibilities. The first is to communicate via

3.2 *Parallel Computing*

messages, i.e. communication links, the second is a shared memory space. While message passing models highly depend on the implementation and can be synchronous as asynchronous, shared memory is tightly related to the underlying communication and consistency models. In this case synchronization is a needed characteristic and a likely performance impact. [Roo00, page 120-123]

The practical realization generally is coupled to the operating system capabilities. Network stacks or local inter-process message passing interfaces for message communication or shared memory between local processes are usual features of modern operating systems.

4 Machine Learning and Bump Boost

In this chapter we give an introduction to the machine learning and the backgrounds that concern us most. Besides that, we present an algorithm called Support Vector Machine and conclude with an in-depth description of the Bump Boost algorithms and their parallel variants.

4.1 Background

The background knowledge is organized as follows. After introducing a definition for machine learning and the relationship to other science fields, we confine and specify the in this work treated sub-field of machine learning. At last we describe several basic algorithm techniques.

4.1.1 Machine Learning

To generally describe machine learning we use two citations. “Machine Learning is the field of scientific study that concentrates on induction algorithms and on other algorithms that can be said to “learn.”” [KP] The term “learning” can be further clarified: “In the broadest sense, any method that incorporates information from training samples ... employs learning.” [DHS99, page 16] Summarizing: machine learning is the field of study concentrated on algorithms of whose future behavior is influenced, i.e. learned, from training samples, i.e. data.

In order to learn, a notion of “what to learn” is needed, which can be really subjective. Given this “what to learn” we would like to measure how well our algorithm learned it. This usually is done with some objective function, which measures the gap between realized and desired behavior. Thus the goal is to minimize this gap, i.e. the objective function. This can be done in several ways and often it is the actual minimization of mathematical function.

Caused by the use of data and its characteristics as the computational problems machine learning can be seen as subfield of Statistics and Computer Science. Next to that, the field is tightly coupled to fields of Artificial Intelligence and mathematical optimization. The first is a source for algorithms and ideas, the second is a toolbox to optimize the algorithms and functions. In 4.1.4 some basic optimization examples are listed.

For this work we would like to restrict our view on machine learning a bit more. We assume that the algorithm gets provided some train set X_{train} which consists of N_{train} samples. For each sample a correct result or label, see next section, is provided in the set Y_{train} . By using some objective function the algorithm itself can measure the gap between its prediction and the desired result. The result of this training procedure is a prediction model. Finally, there is a test set X_{test} which is never provided to the algorithm or

4.1 Background

model for learning, its solely purpose is to compare the predictions of the model with the actual results Y_{test} and so to rate the algorithm prediction performance.

4.1.2 Supervised Learning

Depending on the feedback, learning itself can be classified [DHS99, page 16-17]:

Supervised Learning: For each training or test result a correct label/result or a cost for it will be provided to the algorithm.

Unsupervised Learning: There are no labels or results for the data samples. In this case the algorithms usually try to cluster similar samples (f.e. the k-means algorithm) or to find pattern in the data (f.e. auto-encoders [EBC⁺10]).

Reinforcement Learning: For each data sample the algorithm only gets binary feedback, thus if the answer is correct or not. In contrast, the feedback in supervised learning usually is enriched by the knowledge of how wrong an answer is or what the desired one would be.

In this work we only use supervised learning.

4.1.3 Regression and Classification

In the case of supervised learning we can further distinct between regression and classification tasks. In regression tasks the result is not constrained and commonly it is a real value. In contrast, classification tasks provide a set of labels and each data sample belongs to one. Classification can be viewed as subproblem of regression. Thus a regression algorithm does theoretically work on a classification problem, vice versa this is not the case.

The most popular and easiest case of classification consists of two cases. All the other so called multi-class classification tasks can be modified into a two class problem, i.e. “Is this sample part of class X?”.

In this work we use only two class problems, because it is the most common denominator for classification algorithms.

4.1.4 Gradient Methods

Let us assume some data X , the desired result vector Y , some prediction function f with a parameter set θ and a cost function $C(Y, f(X; \theta))$. Now we would like to choose the optimal parameter setting $p_{opt} \in \theta$, i.e. the setting with the smallest cost.

One ineffective and usually infeasible way to find p_{opt} would be to try all the possible instances. Another, inspired from Artificial Intelligence, could

4.1 Background

be a genetic algorithm, i.e. keeping a “population” of parameter settings, based on some fitness function drop some and alter some other until a satisfying result is reached. But the most common one is to use the gradient $\frac{\partial \hat{C}(Y, f(X; \theta))}{\partial \theta}$. In some easy cases finding the optimum will be solvable analytically, in most, usually non-linear ones, not.

In this cases gradient descent methods can be used. The general approach is to start with a random parameter setting p , to compute the gradient g , based on the result to modify p , f.e. for a single parameter $p_{new} = p - l * g$ with some “learning rate” l , and then repeat until some stop criterion is satisfied.

There are several different approaches. Some, for example, take into account the second gradient. Most of them have in common that they can not guarantee to find p_{opt} . Here we present R-Prop [RB93] which is used by Multi Bump Boost (see 4.3.1):

The special characteristic is that it modifies the current parameter setting solely on the knowledge of the gradient sign change. At the beginning some random or static value for p will be chosen as some static one for the “update” value u . U_{min} and U_{max} are the minimum and maximum size for the update value u and the values $0 < \eta^- = 0.5 < 1 < \eta^+ = 1.2$ are set empirically. In each step t the parameter setting will be updated according to the gradient g as follows. “ Z^t ” denotes values “ Z ” at iteration step t :

$$u^{t+1} = \begin{cases} \min(u^t * \eta^+, U_{max}) & \text{if } g^t * g^{t-1} > 0 \\ \max(u^t * \eta^-, U_{min}) & \text{if } g^t * g^{t-1} < 0 \\ u^t & \text{otherwise} \end{cases} \quad (8)$$

$$\Delta p^t = -u^t * \text{sgn}(g^t) \quad (9)$$

$$p^{t+1} = p^t + \Delta p^t \quad (10)$$

The informal behavior is following the gradient descent and increasing the speed as long as the gradient sign does not change. If it does, decrease the speed.

4.1.5 Cross Validation

Usually, not all parameters of a model are selected with a gradient descent or another automatic method. Those parameters are set by hand by the developer. Examples would be the number of hidden units in a neural network etc. In order to select them as objectively as possible, m -fold cross validation is often a good choice.

“Here the training set is randomly divided into m disjoint sets of equal size n/m , where n is again the total number of patterns in D . The classifier is trained m times, each time with a different set held out as a validation set.

4.1 Background

The estimated performance is the mean of these m errors.” [DHS99, page 483/484] Important to notice is that train and validation sets are always disjoint and that D would never incorporate the actual test set.

Given n_p parameter settings, to each of them the above procedure would be applied. The setting with the lowest error would be the final choice and used to create the final model by training on the whole training set.

This technique is used to find the best parameters for Support Vector Machines in this work.

4.1.6 Boosting

A special technique to join so called weak learners to an effective predictor is called “Boosting”. Weak learners’ characteristic is that they are only slightly better than chance. In principle, also a better learner could be used, but than the effect of Boosting is not as important.

The general setup is to choose a weak learner, train it on the training set and then train the “successive ... classifiers with a subset of the training data that is “most informative” given the current set of ... classifiers” [DHS99, page 476] (Note: Boosting is not restricted to classification tasks.). In general, this means that the successive training will be done on the training set parts that are predicted worse by the already selected learners.

The final prediction is done together by all learned models. How those votes are joint is part of the actual boosting algorithm, but usually the weighted votes are joint to a final one.

A popular example for Boosting is algorithm of Viola and Jones [VJ01] using AdaBoost [FS95], which uses Haar-like features to rapidly detect complex objects like faces.

In the next sub chapter we will present the Bump Boost algorithm, which is based on Boosting.

4.1.7 Kernel methods

Often, the given features are in raw form and could be separated in more useful ones, i.e. transformed into a higher feature space. High, in this context, means higher dimensional. This could be done manually or preferably by choosing some function $\phi(X_{low}) \rightarrow X_{high}$.

A popular usage example are Support Vector Machines (see 4.2). They try to separate the samples of class 1 from the samples of class 2 with a (hyper-)plane. By mapping the input space into a higher dimensional one, this task can be eased, because certain features can get separable there, while in the original space they are not.

4.2 Support Vector Machine

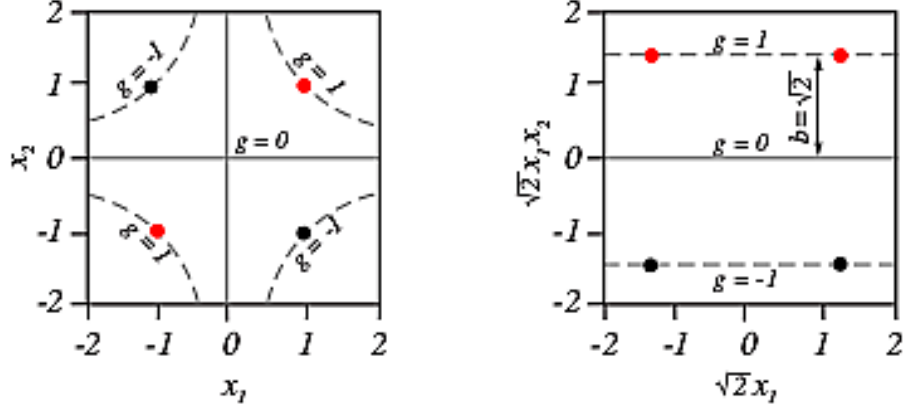


Figure 3: The popular XOR-Problem. On the left side the two-dimensional space, in which no linear function could separate the red and black points. On the right side the feature space using the mapping function $\phi(x_1, x_2) = (1, 2x_1, 2x_2, 2x_1x_2, x_1^2, x_2^2)$, which transforms the two-dimensional input space into a six-dimensional one. In this new space the two classes are easily separable by a linear function. This example and the image are from [DHS99, page 264].

The problem of this mapping is that it can be computationally expensive. Here the so called kernel trick comes into the game. If the algorithm only needs the inner product of the feature space, following function can be imagined:

$$k(x, x') = \langle \phi(x), \phi(x') \rangle_{X_{high}} \quad (11)$$

In this case, the actual representation in the higher feature space is not needed and the result of the kernel is the distance between x and x' in X_{high} . Besides avoiding computational complexity, this procedure replaces, as stated above, the potential handcrafting of additional features with choosing a kernel function.

To create a proper kernel it is sufficient to prove that it is a symmetric positive semidefinite one i.e. Mercer's theorem holds (see page 184 [MMR⁺01]). This holds if the kernel is symmetric i.e. $k(x, x') = k(x', x)$ and positive semidefinite:

$$\sum_{i=1}^n \sum_{j=1}^n k(x_i, x_j) c_i c_j \geq 0 \quad (12)$$

for all finite sequences (x_1, x_2, \dots, x_n) in X_{low} and all choices of n real-valued coefficients (c_1, c_2, \dots, c_n) .

4.2 Support Vector Machine

The general idea of Support Vector Machines (SVMs) is to divide the feature space between two classes, denoted with -1 and $+1$, with a plane. The goal

4.2 Support Vector Machine

thereby is to maximize the distance between the plane and nearest points of each class. This distance is called margin and the plane, that maximizes it, maximum-margin (hyper-)plane.

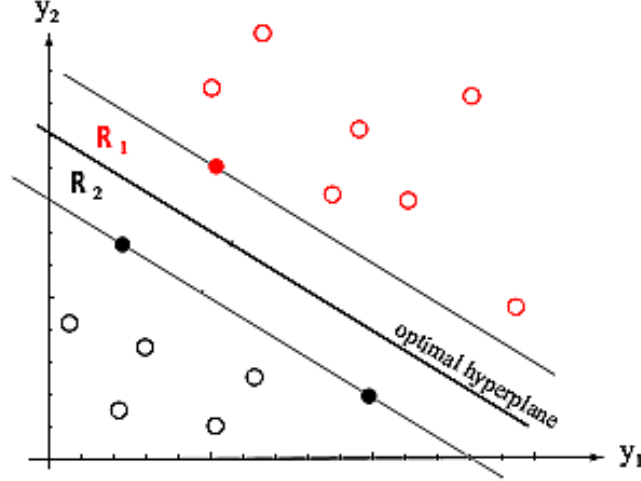


Figure 4: This image shows a two-class separation problem. The optimal hyperplane lies exactly in the middle between the two nearest points of the two classes. In this case, the solid dots would represent the Support Vectors (see below). This example and the image are from [DHS99, page 262].

This algorithm and its soft margin extension were introduced by Vapnik and Cortes in [CV95].

Given this plane (w, b) , a point can be easily classified to class one, if $w \cdot x - b > 0$ holds, else it is of class two.

The optimization problem maximizing the margin can be written as:

$$\begin{aligned} \underset{(w,b)}{\operatorname{argmin}} \quad & \frac{1}{2} \|w\|^2 \\ \text{subject to} \quad & \forall i = 1, \dots, n : y_i(w \cdot x_i - b) \geq 1 \end{aligned} \quad (13)$$

The ≥ 1 guarantees that all points are outside of the margin.

The original form, called primal form, can be rewritten to the dual form by exploiting the facts that $\|w\|^2 = w \cdot w$ and $w = \sum_{i=1}^n \alpha_i y_i x_i$ [CV95, Equation 14]:

$$\begin{aligned} \underset{\alpha}{\operatorname{argmax}} \quad & \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j x_i^\top x_j \\ \text{subject to} \quad & \forall i = 1, \dots, n : \alpha_i \geq 0 \\ \text{constrained by} \quad & \sum_{i=1}^n \alpha_i y_i = 0 \end{aligned} \quad (14)$$

4.2 Support Vector Machine

The plane can be explicitly expressed by $w = \sum_{i=1}^n \alpha_i y_i x_i$. All the points x_i with $\alpha_i \neq 0$ are called “Support Vectors”.

This form also shows that the kernel trick (see 4.1.7) can be applied by replacing the inner product $x_i^\top x_j$ with a valid kernel $k(x_i, x_j)$ leading to a non-linear SVM:

$$\begin{aligned} & \underset{\alpha}{\operatorname{argmax}} && \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j k(x_i, x_j) \\ & \text{subject to} && \forall i = 1, \dots, n : \alpha_i \geq 0 \\ & \text{constrained by} && \sum_{i=1}^n \alpha_i y_i = 0 \end{aligned} \tag{15}$$

The constraint that no point may lie inside the margin can be too restrictive, f.e. if the data is noisy. Due to this reason, a slack variable ξ , in this case as linear penalty, was introduced:

$$\begin{aligned} & \underset{(w, b, \xi)}{\operatorname{argmin}} && \frac{1}{2} \|w\|^2 + C \sum_{i=1}^n \xi_i \\ & \text{subject to} && \forall i = 1, \dots, n : y_i(w \cdot x_i - b) \geq 1 - \xi_i, \xi_i \geq 0 \end{aligned} \tag{16}$$

Depending on the regularization parameter C the exception on the margin constraints are more or less punished.

In the dual form the linear penalty vanishes except one key point:

$$\begin{aligned} & \underset{\alpha}{\operatorname{argmax}} && \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j x_i^\top x_j \\ & \text{subject to} && \forall i = 1, \dots, n : 0 \leq \alpha_i \leq C \\ & \text{constrained by} && \sum_{i=1}^n \alpha_i y_i = 0 \end{aligned} \tag{17}$$

4.2.1 Implementation

Left with these mathematical optimization problems out of the box solvers for these can be used, i.e. quadratic program solvers. These methods can be quite complex and the programs expensive.

Another neat and for SVMs-specialized approach is the “Sequential Minimal Optimization”-algorithm (SMO) [P⁺98] invented by John Platt. By breaking down the problem to two Lagrange multipliers and solving it analytically, the algorithm reduces the optimization complexity. This is done for all Lagrange multiplier as long as they violate the Karush-Kuhn-Tucker conditions.

Linear SVMs can also be solved efficiently with gradient descent methods (see 4.1.4).

4.3 Bump Boost

4.3 Bump Boost

Now we would like to introduce the central algorithms of this thesis. They were invented by Mikio Braun and Nicole Krämer in [BK]. The paper was never published, therefore we provide a copy in the appendix C.

Bump Boost and Multi Bump Boost can be used for classification and regression. The algorithm remains the same. Due to the used data sets, only classification is mentioned.

This section is organized as follows. First we describe the algorithms of Bump Boost and Multi Bump Boost. Then we state some of their characteristics. We conclude with proposals for a parallelized Bump Boost and Multi Bump Boost version.

4.3.1 The Algorithm

Lets begin with the final prediction model. Based on m learned, and so called, bumps, the final prediction function is defined for $x \in \mathbb{R}^d$ as follows:

$$f(x) = \sum_{i=1}^m h_i * k_{w_i}(c_i, x)$$

$$\text{with } k_{w_i}(x, x') = \exp \left(- \sum_{j=1}^d \frac{(x - x')^2}{w_j} \right) \quad (18)$$

One bump is described by the triple center, width, and height $\forall i \in 1, \dots, n : (c_i, w_i, h_i); c_i, w_i \in \mathbb{R}^d; h_i \in \mathbb{R}$.

The kernel k could also be replaced: the algorithm “does not fit all kinds of kernels, but is specialized to “bump-like” kernels like the Gaussian kernel or the rational quadratic kernel which have a maximum when the two points coincide.” [BK, page 2] In this paper, we use only this Gaussian one.

Both algorithms, Bump Boost and Multi Bump Boost, as training input get a feature matrix $X \in \mathbb{R}^{n \times d}$ and a result vector $Y \in \mathbb{R}^n$ with n -samples and base on Boosting (see 4.1.6), namely l_2 -Boosting [BY03]. The general algorithm for l_2 -Boosting is as follows:

```

Initialize residuals  $r \leftarrow Y$ , learned function  $f(x) \leftarrow 0$ ;
for  $i = 1, \dots, m$  do
    | Learn a weak learner  $h_i$  which fits  $(X_1, r_1), \dots, (X_n, r_n)$ ;
    | Add  $h_i$  to learned function:  $f \leftarrow f + h_i$ ;
    | Update the residuals:  $r_j \leftarrow r_j - h_i(X_j) \forall j \in 1, \dots, n$ ;
end

```

4.3 Bump Boost

In our case, the weak learners are Gaussian bumps fitted to the residuals. The following plots show an example of learning bumps. The learned data is the Heavisine function from the Donoho test set [DJKP95]. A noise level of 0.01 is applied on the 500 points:

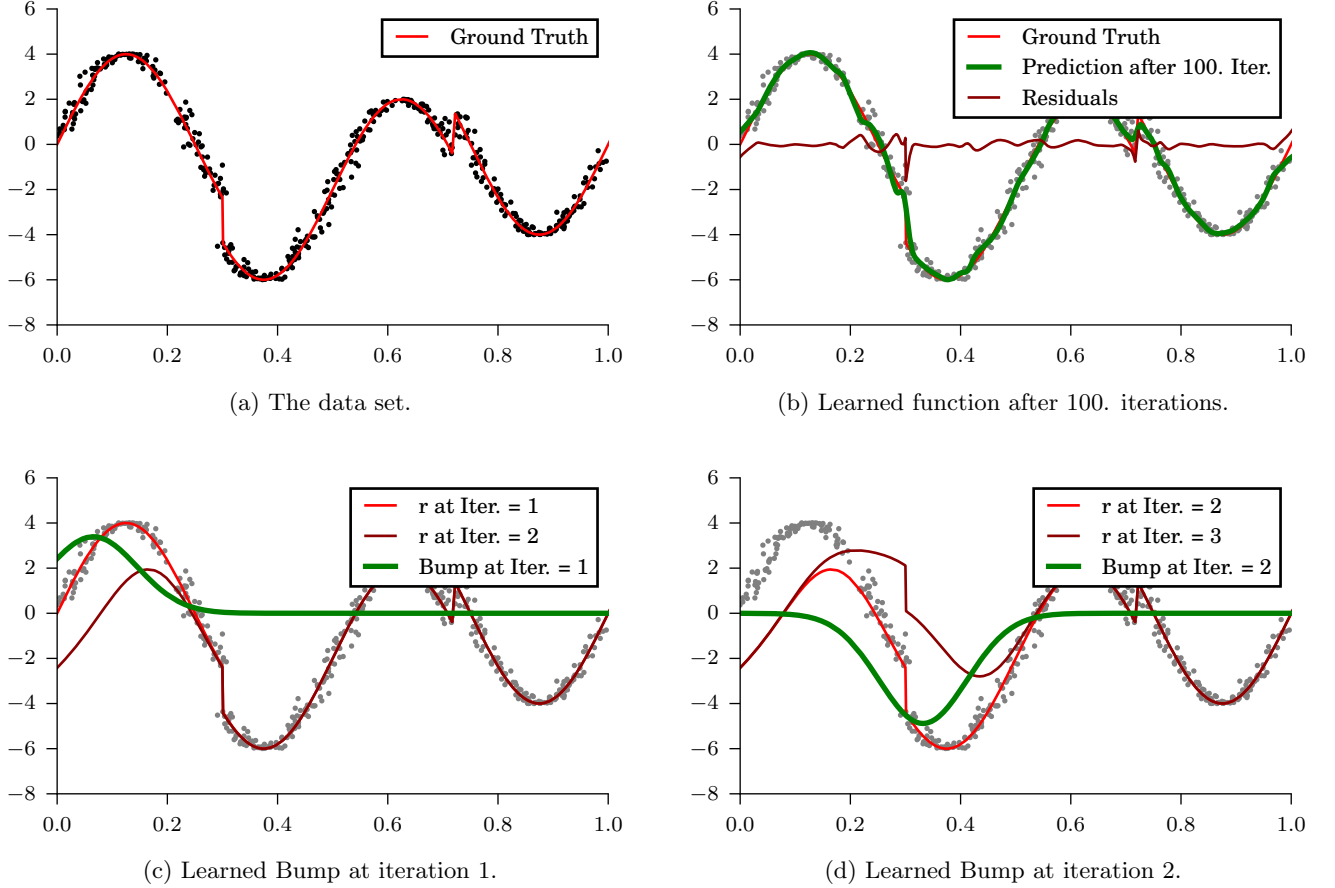


Figure 5: An example of how Bump Boost learns.

As stated above, each weak learner, i.e. Gaussian bump, is described by the parameters center, width, and height (c, w, h) ; $c, w \in \mathbb{R}^d$; $h \in \mathbb{R}$. The parameters are learned in the following order:

Center: The center is drawn using the residual-related probability distribution in equation 19.

Width: In this case, and only in this case, the Bump Boost and the Multi Bump Boost algorithm differ. Bump Boost chooses the best width out of a candidate list, whereas Multi Bump Boost finds the width using R-Prop (see 4.1.4). Both ways base on minimizing the squared error.

4.3 Bump Boost

Height: In the end the height is chosen by minimizing the squared error.

In more detail: the center is chosen from X . The point at index i is drawn with a probability proportional to the squared residual at that point:

$$p(i) = \frac{r_i^2}{\sum_{j=1}^n r_j^2} \quad (19)$$

One way to determine this value is to sum up the squared residuals r_1^2, \dots, r_n^2 and then multiply this value with a random value $\epsilon = \text{random} * \sum_{i=1}^n r_i^2$ with $\text{random} \in [0, 1)$. Given the cumulative sum of the squared residuals $c_j = \sum_{i=1}^j r_i^2$, we draw element X_i with the smallest i for that holds $\epsilon \leq c_i$. This can be programmed by actually creating the cumulative sum or by doing a binary search in the virtual ordering $c_1 \leq c_2 \leq \dots \leq c_n$ by summing up ranges of r_a^2, \dots, r_b^2 . The latter approach will be called “binary search” in the rest of the paper. For code examples see [7.5.1](#).

Given the center c the width w is the next parameter to determine. This is done in either case by minimizing the squared error. We define the so called kernel vector

$$k_{c,w} = (k_w(c, X_1), \dots, k_w(c, X_n)) \quad (20)$$

and the vector of the residuals is named r . Then the squared error is given by:

$$\begin{aligned} \|r - \hat{r}_w\|^2 &= \|r\|^2 - 2 \left\langle r, \frac{k_{c,w} k_{c,w}^\top r}{k_{c,w}^\top k_{c,w}} \right\rangle + \left\| \frac{k_{c,w} k_{c,w}^\top r}{k_{c,w}^\top k_{c,w}} \right\|^2 \\ &= \|r\|^2 - 2 \frac{(k_{c,w}^\top k_{c,w})^2}{k_{c,w}^\top k_{c,w}} + \frac{(k_{c,w}^\top r)^2 k_{c,w}^\top k_{c,w}}{(k_{c,w}^\top k_{c,w})^2} \\ &= \|r\|^2 - \frac{(k_{c,w}^\top r)^2}{k_{c,w}^\top k_{c,w}} \\ &=: \|r\|^2 - C(c, w) \end{aligned} \quad (21)$$

The residuals do not change in this context, thus we are left with maximizing $C(c, w) = \frac{(k_{c,w}^\top r)^2}{k_{c,w}^\top k_{c,w}}$.

The Bump Boost algorithm has a list of candidate widths, thus just needs to select the best one. This is easily done by calculating the reward $C(c, w_i)$ for each width w_i in the candidate list and selecting the one with the highest reward.

In the actual implementation only one dimensional candidates are used. In a higher dimensional case this value is used for all dimensions. It would be possible to choose d -dimensional candidates, even though this would may result in a long list. In this case, Bump Boost would loose his performance

4.3 Bump Boost

advantage against Multi Bump Boost, which performs well with higher dimensional settings without handcrafting candidates.

As mentioned above, in Multi Bump Boost a gradient descent is done. To do so we need the gradient of the reward function $C(c, w)$. Because c is already fixed in this context, we actually need the gradient $\frac{\partial C_c(w)}{\partial w}$. To ease the computation the kernel gets reparameterized by using the logarithm of the actual width:

$$k_w(x, x') = \exp \left(- \sum_{j=1}^d 10^{-w} (x - x')^2 \right) \quad (22)$$

The gradient formula is given in [BK] and looks in our context like:

$$\begin{aligned} \frac{\partial C_c(w)}{\partial w} &= \frac{\partial C_c(w)}{\partial k} \frac{\partial k_c(w)}{\partial w} \\ &= \underbrace{\frac{2k_{c,w}^\top r}{k_{c,w}^\top k_{c,w}} \left(r - \frac{k_{c,w}^\top r}{k_{c,w}^\top k_{c,w}} \right)}_{\frac{\partial C_c(w)}{\partial k}} \underbrace{[k_w(c, x_i)(x_{i,j} - c_j)^2 10^{-w_j} (\ln 10)]_{i=1, j=1}^{n,d}}_{\frac{\partial k_c(w)}{\partial w}} \end{aligned} \quad (23)$$

Given this gradient, R-Prop is used to determine the width w . By using a box constraint, i.e. restricting the minimal and maximal width, the algorithm is slightly modified. According to [BK] doing more than 30 to 100 gradient descent steps in R-Prop does not change the result significantly.

After calculating the center c and the width w , we can easily calculate the remaining parameter height h by again minimizing the squared error:

$$h = \underset{h}{\operatorname{argmin}} \|r - h k_{c,w}\|^2 = \frac{k_{c,w}^\top r}{k_{c,w}^\top k_{c,w}} \quad (24)$$

In the end the residuals are updated for the next iteration:

$$r = (r_1 - h_1 * k_{w_1}(c_1, X_1), \dots, r_n - h_n * k_{w_n}(c_n, X_n)) \quad (25)$$

To summarize, the base algorithm for Bump Boost and Multi Bump

4.3 Bump Boost

Boost is:

Initialize residuals $r \leftarrow Y$;

for $i = 1, \dots, m$ **do**

- Choose a center $c_i = X_i$ according to $p(i) = \frac{r_i^2}{\sum_{j=1}^n r_j^2}$;
- Get the width w_i either by:
 - selecting width w from the candidate list with the maximal $C(c_i, w)$
 - doing R-Prop gradient descent with $\frac{\partial C_{c_i}(w)}{\partial w}$;
- Calculate the height $h_i = \frac{k_{c_i, w_i}^\top r}{k_{c_i, w_i}^\top k_{c_i, w_i}}$;
- Update the residuals: $r_j \leftarrow r_j - h_i * k_{w_i}(c_i, x_j) \quad \forall j \in 1, \dots, n$;

end

Return the final function $f(x) = \sum_{i=1}^m h_i * k_{w_i}(c_i, x)$

Asymptotic Run Time

We want to emphasize that all steps are linear in n , assuming d as constant:

Center: The center can be determined in $O(n)$. Calculating the squared residuals can be done in $O(n)$, after summing them up ($O(n)$), the searched value can be found using a cumulative sum in $O(n)$.

Width: Calculating the kernel vector takes $O(n * d)$. The in Bump Boost needed scalar products can be computed in $O(n)$. In the Multi Bump Boost case we need to multiply a $n \times d$ matrix with a vector of length n , which takes $O(n * d)$. The length of the candidates list as the gradient descent steps are constant, thus finding the center can be done in $O(n * d)$. This can be seen as linear as d is assumed to be constant.

Height: The same yields for the calculation of the height. The kernel vector and the scalar products can be computed in $O(n * d)$ and $O(n)$, resulting in a linear run time.

Residual Update: As calculating the kernel vector this takes $O(n * d)$, thus can be done in linear time.

One iteration can be done in linear time. Because the iteration count does not change and there are no further computations, the whole algorithm can perform in linear time.

4.3.2 Characteristics

In order to efficiently apply the Bump Boost algorithms, it is important to do calculations just once. Especially the term $(x_i - c)^2$ given the center c

4.3 Bump Boost

and for all $x_i \in X$ is time and memory intensive and should be calculated only one time per iteration.

Even though gradient descent can be used, Bump Boost and Multi Bump Boost do not try to minimize a cost function, but instead they try to minimize the squared error via l_2 -Boosting on the residuals. And in contrast to stochastic gradient descent algorithms, which adapt all weights using a single data point, the Bump Boost algorithms adjust one weight using all the data points.

Other algorithms with kernel methods, f.e. Support Vector Machines, that usually have a single global kernel parameter, this algorithm can have several kernel parameters, i.e. in each iteration a different kernel can be and usually is selected. Whereas the global kernel parameter generally is found via cross-validation due to too complex optimization functions, Bump Boost has some sort of cross validation when searching the kernel parameter in an iteration. In Multi Bump Boost this is replaced by the gradient descent.

[BK] claims that for Bump Boost no model selection is needed. We think this claim is inaccurate. Bump Boost seems to be quite robust against parameter selections, but still they need to be set. To be more precise, by boxing the width value in Multi Bump Boost or setting the width candidates we can influence the model behavior. Especially by setting the smallest kernel width, we regularize the model. F.e. assuming that Bump Boost is allowed to use infinitely small widths or really small widths, the algorithm just places a peak bump under each data point, which results in a miserable generalization to not learned data points. The upper bound for the weights is not as important and can be set to a quite high value.

To summarize, Bump Boost and Multi Bump Boost need some parameter space or list as often other kernel methods do, but while those use that list for general cross validation, in Bump Boost and Multi Bump Boost the parameter selection is part of the algorithm.

Usually, also in this thesis, the Bump Boost the width vector has the same value for all dimensions. In general, this works well, but we want to note that in higher dimensional settings different dimension might need a different values. Further research would be needed to examine this problem in more detail.

Another, advantageous, property of Bump Boost is that in principle after learning for m iterations, the learning can be resumed at any time. Or at prediction time only $m' < m$ classifiers can be used until the wished accuracy is reached or the maximal run time is reached.

In this paper we use a Gaussian kernel for Bump Boost and Multi Bump Boost. In principle, it would be possible to use other kernels with a “center” point (see 4.3.1). If Multi Bump Boost would be used with a different kernel only a part of the gradient function needs to be updated. In equation 23, only the second gradient $\frac{\partial k_c(w)}{\partial w}$ is dependent of the actual kernel formula.

4.3 Bump Boost

$\frac{\partial C_c(w)}{\partial k}$ stays the same. We did not investigate which other kernels would suit to Bump Boost or Multi Bump Boost. This is left to further investigation.

4.3.3 Parallelization

Now we would like to describe an, according to us, almost perfect parallelization strategy for Bump Boost and Multi Bump Boost.

First we would like to emphasize that we expect the input parameter n to scale. Even if there are data sets with very high dimensional inputs, they are less frequent and we do not know how well Bump Boost will perform in such settings. Hence, we pay our attention to the sample count.

4.3 Bump Boost

As we have seen in 3.2.1 the parallelization of an algorithm is mainly restricted by the sequential dependencies of the algorithm. The data flows of Bump Boost and Multi Bump Boost simplified look like:

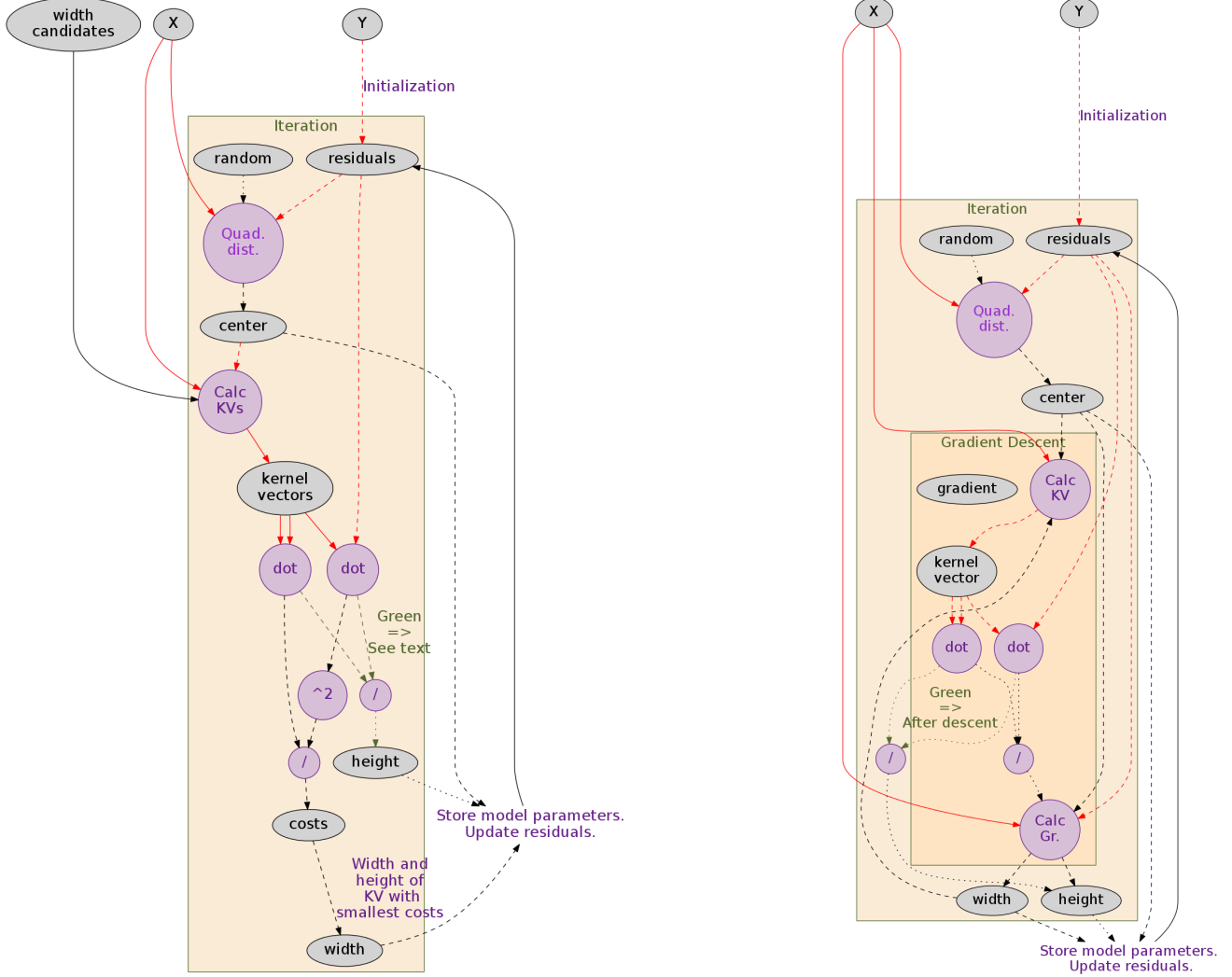


Figure 6: Dependency graph of the major variables in the Bump Boost and the Multi Bump Boost algorithm. Violet marks calculations. The style of the edges marks the delivered value: dotted is a scalar, dashed a vector, and solid a matrix. If an edge is colored red, it means the size of the value grows with $O(n)$ with n sample count.

These dependencies restrict our parallelization effort in several ways:

- Different iterations are not parallelizable, because each relies on the calculated model parameters of the previous one.
- Computations of the width parameter depend on a chosen center, thus both cannot be computed concurrently.

4.3 Bump Boost

- The same holds for the width and height in the Multi Bump Boost case. In Bump Boost the heights could be computed concurrently, and then those parameters of the candidate with the smallest cost are used. On the other hand, the height could also be computed after the width being determined, still taking advantage of the precalculated values. Therefore, the edges are green.

We can summarize that the calculations of each iteration as, inside the iteration, of center, width, and height (with a corner case) have to be done in sequential manner.

Thus, the only way to parallelize (Multi) Bump Boost is to do it during the calculations of the parameters. As we can learn from the graphs, the values belonging to red edges scale with $O(n)$ with n sample count. Expressions involving these values are problematic for scaling.

We assume that each parallel $p = 1, \dots, k$ instance is responsible for some data, i.e. the continuous indexes $I_p = i_p, \dots, i_{p+1} - 1$ with $1 = i_1 < \dots < i_{k+1} = n + 1$. Let us start with Bump Boost:

Center: How a random center can be found is described in 4.3.1. This can be broken into two steps: First calculating the sum of the squared residuals, which can be done efficiently in parallel. The second, finding the center is more challenging. Given $u_p = \sum_{p'=1}^p \sum_{i \in I_{p'}} r_i^2$, the searched value is at worker p with the smallest p for that holds $\epsilon \leq u_p$. At the worker itself, the general search center procedure inside the range I_p can be applied, using a new $\epsilon' = \epsilon - u_{p-1}$ if $p > 1$. Also in the second operation only a fraction of the data needs to be accessed.

Width: By closely examining the sub graph, we identify the computations until the dot products are especially expensive, i.e. they scale with $O(n)$. Ideally, we would like to split those into sub task and indeed we can:

- Each element of a kernel vector can be calculated independently on the other (see equation 20). Thus, the calculation of the whole vector can be parallelized.
- The definition of the dot product for vectors is $v \cdot w = \sum_{i=1}^n (v_i w_i)$. This allows us to do each scalar multiplication in parallel and parts of the summation, too. By denoting $v_{n:m}$ as the sub vector from element n to element m of the vector v , we can easily split the dot product into m smaller dot products, i.e. sub tasks:

$$v \cdot w = \sum_{i=1}^n (v_i w_i) = \sum_{i=1}^{\lfloor n/m \rfloor} \sum_{j=i*m}^{i*(m+1)} (v_j w_j) = \sum_{i=1}^{\lfloor n/m \rfloor} v_{i*m:i*(m+1)} \cdot w_{i*m:i*(m+1)} \quad (26)$$

4.3 Bump Boost

They can be done in parallel and the final value is given by the sum of them.

Height: If the final height is computed in parallel to the widths or afterwards, in both cases the computed dot products in the width calculations can be recycled and the height calculated using a closed-form expression (see 24, mind that the dot products are already computed). Therefore, we do not need to parallelize here.

Residuals Update: As computing the kernel vector given the computed parameters, this can easily be done in parallel, see equation 25.

As Bump Boost is parallelizable to some extent, also Multi Bump Boost it is:

Center: It is the same algorithm as for Bump Boost.

Width: With the knowledge of the Bump Boost parallelization, all the values of the calculate gradient operation can be computed efficiently (in the figure the node is called “Calc Gr.”). Given the “height” value, i.e. $\frac{k_{c,w}^\top}{k_{c,w}^\top k_{c,w}}$, all the operations we need for the gradient (see 23) are element-wise subtraction and multiplication completed by a dot product. This can be done efficiently in two steps by computing the height as described in the Bump Boost parallelization and after distributing that value by doing subtractions and multiplications in parallel. The final value then is given by a final parallelized dot product.

Height: For Multi Bump Boost we cannot recycle the intermediate results of the width calculation, because the actual width is determined by the last gradient update. But we can calculate those results again in efficient manner as for the height computation in the previous step. Hence, the height parameter is parallelizable, too.

Residuals Update: It is the same algorithm as for Bump Boost.

After describing the parallelization, we are left with a last problem: the communication between the sub tasks assigned to some worker. This is especially expensive when they are distributed over several hosts. Fortunately, this is easily solved by distributing the data beforehand. The only “large”, iteration-persistent variables are X and *residuals*. By, as assumed, assigning each worker p a slice of the samples and residuals, i.e. X_{I_p} and $residuals_{I_p}$, he can do all the expensive computation locally and just deliver the result to the master. The master joins the results together to calculate the actual model parameters. Clearly, the work and data load should as balanced as possible to reach a good parallelization.

This is categorized as data parallelism (see 3.2.3), because the same operations and procedures are carried out on different sub sets of data.

4.3 Bump Boost

This parallelization scheme is shown for Bump Boost in the next illustration:

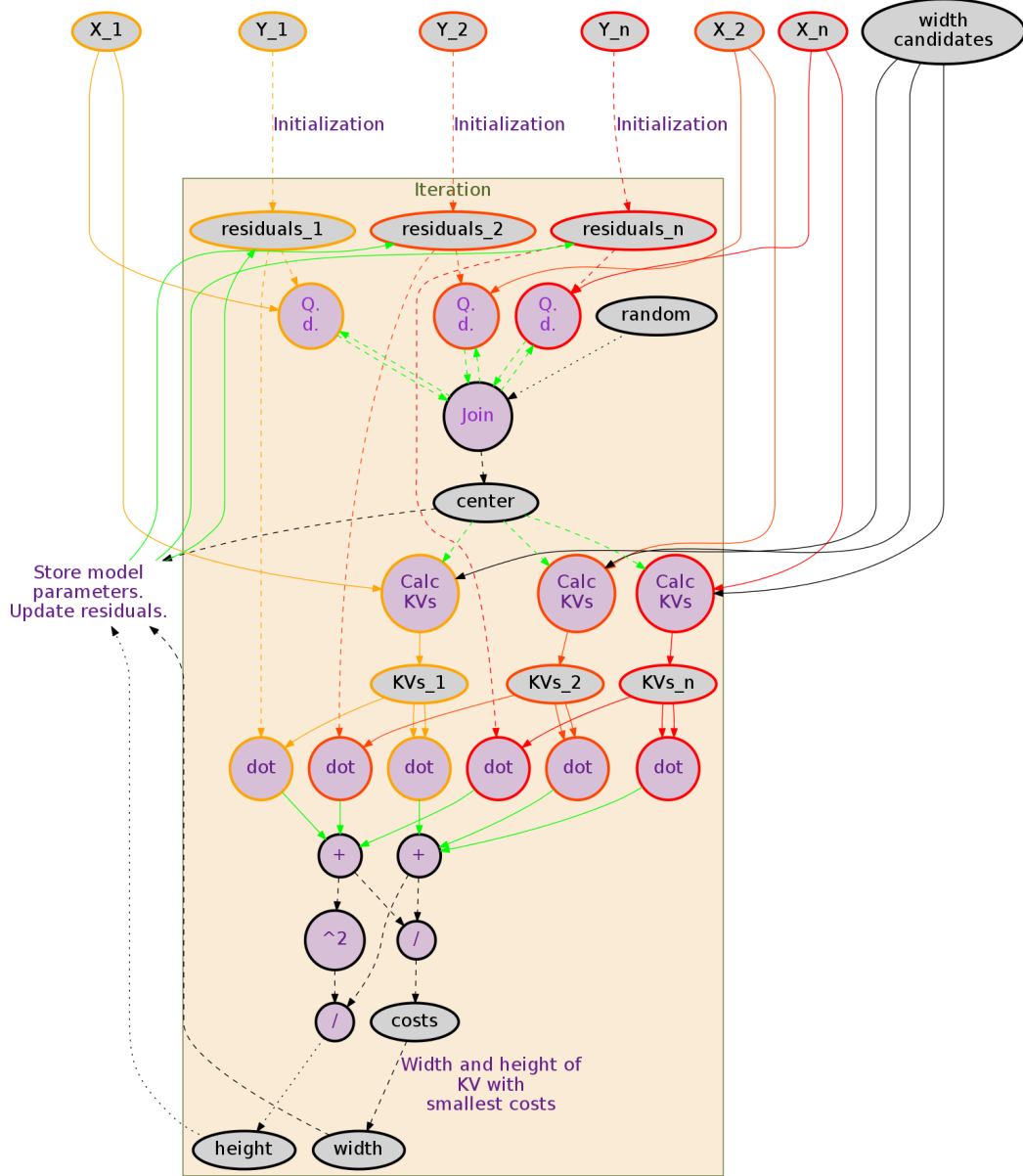


Figure 7: This graph illustrates the calculations subdivision onto different workers for the Bump Boost algorithm. The node border colors orange to red denote different work entities, thus those values and computations were stored/executed on the according workers. Black denotes the master. The edge color green denotes a transfer between master entity and a worker entity. The other graph properties are described in the previous illustration 6.

4.3 Bump Boost

Please note, that only the parallelized operations scale with data set size n , i.e. all the other operations do not depend on n , but on the number of workers. This implicates that with increasing data set size the parallelizable parts of the algorithms increase. Therefore, according to the theoretical laws on scaling (see 3.2.1) Bump Boost and Multi Bump Boost should have better scaling properties with larger data sizes. Furthermore, the amount of data sent between master and workers as the work load at the master stays constant with constant number of workers.

Theoretically, these joins may cause a bottleneck. Imagine having n data points and n workers. In this case, the master needs to sum up n values and the approach would not scale. In more detail, in the graph above the meant operations would be the “join” and additions pointed by the green arrows. Fortunately, all the join and addition operations can be implemented in a tree-like structure. In this case, assuming all nodes have the same child-degree C and each worker has C data points to take care of, the asymptotic growth would be $O(C + \log_C(n/C)) = O(\max(\log_C(n/C), C))$. This is because the operations at the worker nodes grow with C , whereas the join operations grow with the height of the tree $\log_C(n/C)$. More on that in the following paragraph.

How do these versions scale?

As we have seen in 4.3.1, Bump Boost and Multi Bump Boost scale asymptotically in linear manner. How do the parallelized versions scale?

To investigate that, let us assume that the sample set X of size n can be split into m partitions of equal size n_m . The iteration count is named I . We start with Bump Boost and treat the number of width candidates as a constant. In the following description, there is one master and m worker nodes to compute the algorithm. For simplicity’s sake we do not mention constant parts of Bump Boost:

Data fetch: If the workers are distributed and all the data lies at the master, it takes him $O(n)$ to deliver it to the workers, assuming that no parallelization through different network interface etc. is possible.

If the workers are not on the same machine, i.e. perform on different hosts, or the data is distributed, f.e. with HDFS (see 6.2.2), it either takes linear time to load the data into memory or it is somehow parallelized, but the behavior is not predictable. Thus, in worst case each loads the whole data in parallel.

We can summarize that the fetching the data into memory and to distribute it takes $O(n)$.

Center calculation: The parallel effort, as described above, takes $O(n_m)$ for summing up the squared residuals, the work at the master needs

4.3 Bump Boost

$O(m)$ for summing up the partial results and is concluded in $O(n_m)$ for searching the actual searched value at a single worker: $O(n_m) + O(m) + O(n_m) = O(\max(n_m, m))$

Width calculation: To calculate the kernel vectors it takes $O(n_m)$, as it does for the sub dot products on the single workers. The finalized dot product, i.e. summing up the sub results at the master takes $O(m)$: $O(n_m) + O(m) = O(\max(n_m, m))$

Height calculation: The height parameter can easily be calculated out of already computed dot products, thus takes constant time.

Residuals update: After fetching the computed parameters, this can be done at the workers in $O(n_m)$.

We are left with an overall asymptotic run time with I iterations:

$$\begin{aligned} O(n) + I * (O(\max(n_m, m)) + O(\max(n_m, m)) + O(1) + O(n_m)) \\ = O(n) + O(I * \max(n_m, m)) \end{aligned} \quad (27)$$

For Multi Bump Boost the data fetch and the center calculation as the residual update are the same as for Bump Boost. The width needs to be computed in two steps resulting in $O(\max(n_m, m) + \max(n_m, m)) = O(\max(n_m, m))$. The height calculation takes $O(\max(n_m, m))$, it is basically the same effort as a reward calculation in Bump Boost. In this case, the overall computational cost with I iterations and G gradient descent steps is:

$$\begin{aligned} O(n) + I * (O(\max(n_m, m)) + G * O(\max(n_m, m)) + O(\max(n_m, m))) \\ = O(n) + O(I * G * \max(n_m, m)) \end{aligned} \quad (28)$$

We reach the best performance when $n_m = m$ i.e. $m = \sqrt{n}$ holds. Without the data loading, we improved the asymptotic run time of Bump Boost and Multi Bump Boost from $O(n)$ to $O(\sqrt{n})$. As the data loading needs to be performed only once, the final amortized computational cost is $O(\max(n_m, m))$ given $I \rightarrow \infty$.

Given the case that lots of workers are available, i.e. $n_m \ll m$, the join operations scale worse than the main computations or just to reduce the actual run time, it is possible to do the join operations in a (virtual) tree network. This may lead to a smaller run time, if there is lots of data and the overhead does not overwhelm the run time.

Getting back to the proposal of the tree with C children and each leaf is responsible for C data points, in which case the join and addition operations take $O(\log_C(m))$ instead of $O(m)$. In the case the asymptotic run time of Bump Boost boils down to:

$$O(n) + O(I * \max(\log_C(m), C)) \quad (29)$$

4.3 Bump Boost

For Multi Bump Boost this results in:

$$O(n) + O(I * G * \max(\log_c(m), C)) \quad (30)$$

Given $I \rightarrow \infty$ and taking C as constant, we can say Bump Boost and Multi Bump Boost have an amortized computational cost logarithmic in n :

$$O(\log_C(m)) = O(\log_C(n/C)) = O(\log(n)) \quad (31)$$

5 Related Work

This chapter treats related work in the field of machine learning, to be more precise, work that addresses scaling issues.

After the introduction to the popular map-reduce approach in [DG05], [CKL⁺07] shows how to speedup a variety of machine learning algorithms using this simple paradigm. The paper describes how algorithms that fit the Statistical Query model [Kea98] can be rewritten in a certain summation form. Thus by mapping, i.e. calculating the summands, and then reducing, i.e. summing up, the map-reduce approach can be applied. They apply their principle, among others, to logistic regression, naive Bayes, SVM, ICA, PCA, and neural networks. A similar set of algorithm is implemented in the Apache Spark MLlib.

Using a sum to join the in parallel calculated results, Bump Boost and Multi Bump Boost have a similar approach. But the determination of the bump center, for example, is not covered by it. Thus Bump Boost is in some sense too complex for this schema.

While the solution [CKL⁺07] does not rely on approximation, in machine learning it is often used for large scale problems. For example, stochastic gradient descent tries to reduce the actual processed data by sampling on the whole data set. [Bot10] describes how stochastic gradient descent can be used efficiently, f.e. with averaged stochastic gradient descent or computing second order derivatives, for large scale problems. In [LK12] machine learning at twitter is described, where f.e. stochastic gradient descent with logistic regression is used for large amounts of data.

Another example where averaged stochastic gradient descent works well are neural networks, in this case also named mini batch learning. This technique was successfully applied in [DCM⁺12] by massively parallelizing deep neural network learning. Due to its sequential nature, stochastic gradient descent is hard to parallelize, [DCM⁺12] shows how this can be done by asynchronous updates. More precise, in “Downpour SGD” two clusters parallelize the workload. The data is partitioned onto the entities of one cluster, where each performs the gradient calculations. The gradients then get pushed to a second cluster, where each host is responsible for a set of parameters. This cluster is responsible for updating and distributing the actual parameters. A similar approach is used in [LAS14] with a even larger setup.

This idea of a parallel stochastic gradient descent has also been used in several works, such as [FS09], to speedup linear SVMs. In contrast to deep learning, which works well in non-linear settings, non-linear kernels enable SVMs to solve more complex problems [Gär03]. In principle and practice it is possible to use stochastic gradient descent also with kernels, but the question which data points, i.e. support vectors, to prioritize gets prominent [BEWB05, The implementation LaSVM is used in this work.][KSW04]. This

5 Related Work

increases the parallelization complexity.

SVMs suffer from the complex optimization problem, the more data, the slower the state-of-the-art SVM solvers. In [SSS08] they claim that more data should decrease the actual run time when the same prediction error should be reached. The idea is, that even though the optimization problem increases with more data, the actual generalization problem does remain the same, in contrary should be easier solvable with more data. The authors give a theoretical and empirical justification for linear SVMs.

In contrast to stochastic gradient descent, Bump Boost and, in this case more concerning, Multi Bump Boost optimize the parameters always by using the whole data set and synchronously.

Another way to parallelize machine learning algorithms is ensemble learning. There are several ways, but the principle is to learn different models on a data set and then combine their single predictions to a global one [MO99]. Next to stochastic gradient methods, this is a used technique at twitter for large scale problems [LK12]. But it comes with high computational costs, as generally each model is trained on the whole data set. In [CBB02] a mixture of SVMs is proposed to make SVM learning practical, where the output of a set of SVM, each learning on a subset of the data, is combined by a learned gatekeeper. Even though giving good results, the model is still restricted by the size of the subsets, i.e. the size of the subset cannot be larger than practically manageable by a single SVM. Hence, this approach will not work for highly complex and large data sets.

Boosting, which is used in Bump Boost, is also a form of ensemble learning as in each iteration a simple prediction function is learned. But the learning of the bumps is not independent, thus not parallelizable as all the other boosting based approaches.

An interesting solution for large scale learning is given in [RR07]. The authors propose to map the actual feature space into a lower dimensional one and learn on that with fast linear methods. This lower space should be designed so that the resulting inner products are approximately the same in both spaces.

By top-performing in the ImageNet Large Scale Visual Recognition Challenge 2012 (ILSVRC2012) the authors of [KSH12] gave an example on how neural networks can learn features by themselves. The enormous learning task and parameter space, the net has 69 million parameters and 690.000 neurons, is controlled by a highly efficient GPU-implementation and the, back then, new regularization method dropout [HSK⁺12].

Similar our work also includes a fast GPU-implementation. On the other hand, Bump Boost and Multi Bump Boost are as SVMs dependent on a meaningful feature space. The kernel methods merely help to predict more complex problems. This neural network learns these “meaningful” features, too.

In the introduction, we already mentioned recommender systems. They

5 Related Work

relate to our work mainly by the fact that they are a popular Big Data problem and application of Apache Spark. The approach of [KBV09] bases on an approximated matrix factorization and is implemented in Apache Spark MLlib [ml15]. Besides, a proof of concept for Apache Flink is given in [fli15a] and should be released in future. The actual algorithm is of modest complexity, the resulting code [fli15b] in Flink is complicated and, according to us, not implementable without an in-depth knowledge of Flink. Indeed [fli15a] mentions that several features were added to Flink to enable this algorithm.

6 Tools and Frameworks

Before we describe our programs, we would like to introduce the technology they are based on.

Already well-known and established for GPUs is Cuda and for GPUs and other computing devices OpenCl. Both were used to parallelize programs on single devices. Still new are Big Data systems to parallelize computations on computers clusters. In this work we use Apache Spark and Apache Flink. All of them will be described below.

We use the Scipy toolkit to develop a parallelized version of Bump Boost from scratch. To accelerate Bump Boost we try to use the libraries CudaMat and PyOpenCl, both have the aim to provide an easy access to Cuda and OpenCl using Python.

Finally, we describe two very popular SVM-solvers: LIBSVM and LaSVM.

6.1 Parallel Computing Device Frameworks

As stated above Cuda and OpenCl were used to parallelize applications on single devices. While Cuda's job is to enable it for Nvidia GPUs, OpenCl is more general and helps to port applications on different platforms, from CPUs over GPUs to FPGAs. Both are data parallel programming languages (see [3.2.3](#)).

Both of them provide a language to program and an application interface to (cross-)compile the code and access the devices. The programming model of these languages differs from the common one, because according to Flynn's taxonomy (See [3.2.1](#)), it is build on "Single Instruction Stream, Multiple Data Streams" machines. Common sequential programming models assume "Single Instruction Stream, Single Data Stream" devices. This different programming scheme can be a barrier.

To give the reader a broad understanding, let us sketch the following code example from an AMD Developer Blog [[AMD15](#)]. It is a sample dot-product implementation in OpenCl:

6.1 Parallel Computing Device Frameworks

```
1 #define LOCAL_GROUP_XDIM 256
2
3 __kernel __attribute__((reqd_work_group_size(
4     LOCAL_GROUP_XDIM, 1, 1)))
5 void dot_local_reduce_kernel(
6     __global const double * x, // input vector
7     __global const double * y, // input vector
8     __global double * r, // result vector
9     uint n // input vector size
10 ){
11     uint id = get_global_id(0);
12     uint lcl_id = get_local_id(0);
13     uint grp_id = get_group_id(0);
14     double priv_acc = 0; // accumulator in private
15     // memory
16     __local double lcl_acc[LOCAL_GROUP_XDIM]; //
17     // accumulators in local memory
18
19     if ( id < n ){
20         priv_acc = lcl_acc[lcl_id] = x[id] * y[id]; //
21         // multiply elements, store product
22     }
23     barrier(CLK_LOCAL_MEM_FENCE); // Find the sum of the
24     // accumulators.
25
26     uint dist = LOCAL_GROUP_XDIM; // i.e.,
27     // get_local_size(0);
28     while ( dist > 1 ){
29         dist >>= 1;
30         if ( lcl_id < dist ){
31             // Private memory accumulator avoids extra local
32             // memory read.
33             priv_acc += lcl_acc[lcl_id + dist];
34             lcl_acc[lcl_id] = priv_acc;
35         }
36         barrier(CLK_LOCAL_MEM_FENCE);
37     }
38
39     // Store the result (the sum for the local work
40     // group).
41     if ( lcl_id == 0 ){
42         r[grp_id] = priv_acc;
43     }
44 }
```

As hinted above this code gets executed in parallel on different data streams, and is separated implicitly by the results of the identification functions. Again, each thread of this code gets different global and local ids and

6.2 Cluster Frameworks

based on them it should access a different data space and thus use a different “data stream”.

In the code above, first all threads multiply in parallel the according elements (line 17). Then, by halving the working threads in each round, it sums up the results (line 20 to line 30). This code does it until a small vector of summands remains, it could be done also until just the final dot-product result is left over.

The key point we wanted to show, even though the multiplications are easily done in parallel, for the sum we need to use barriers to synchronize the threads and we need to take care which thread does which operation. Next to device-dependent characteristics, this programming model makes these APIs difficult to handle.

The rest of this section will provide some basic information and references on both frameworks.

6.1.1 Cuda

The Nvidia company was the first to offer a general purpose computing interface for a GPU (see [nvi15]) in 2006. This enabled a new form of GPU usage: using high level languages for sequential parts of the application and accelerating the parallelizable parts on graphical interfaces. Today, Nvidia provides interfaces for several languages such as C, C++ and Fortran.

This new development also had a major impact on scientific research. Besides that toolkits as MATLAB support accelerations by GPU usage now ([mat15a]), high end performance gets reached for example in the neural network research. To name one example: the successful convolutional network ([ale15]) of Alex Krizhevsky at the ImageNet([KSH12]).

6.1.2 OpenCl

OpenCl is a project of the Khronos Group (see [ope15a]). Whereas Cuda is only available on Nvidia GPUs, Khronos advertises OpenCl as “first open, royalty-free standard for cross-platform, parallel programming” [ope15a]. All major vendors implement the standard in some way. To name some: Intel, AMD, Xilinx, Altera, ARM, IBM, and Nvidia.

Although this variety of vendors and thus devices providing an OpenCl interface, CUDA still seems to be more popular. F.e. MATLAB does not provide accelerations support by OpenCl devices [mat15b]. A cause maybe is the late following up in the year 2009 [ope15b].

6.2 Cluster Frameworks

A completely different idea compared to Cuda and OpenCl are Spark and Flink. Instead of accelerating the computation as much as possible on a single device, their first goal is to make the computation possible, because of the

6.2 Cluster Frameworks

large data that has to be processed. And second to reduce the computation time by parallelizing the effort on a computer cluster.

As introduction, we describe the Apache Big Data stack they are part of. This includes a short development history, including its first successful project Hadoop. In the end we describe Spark and Flink.

6.2.1 Apache Big Data Stack

¹ Computer clusters, large amounts of data, and distributed programs are for a long time part of computer science. But before the rise of the Apache Big Data stack, these challenges were mostly solved with high end hardware and software systems. The software that is merged into that described stack, especially the most important one, Apache Hadoop, changed the access to Big Data solutions.

We named in this case the Apache Big Data stack, because it is the predominant software bundle for Big Data applications. Notably all this software is open source. To the best of our knowledge, no major software vendor has yet presented a more evolved solution yet.

The following diagram depicts the Apache Big Data stack:

¹Great part of the information of this section are summarized from [\[KFLQ\]](#).

6.2 Cluster Frameworks

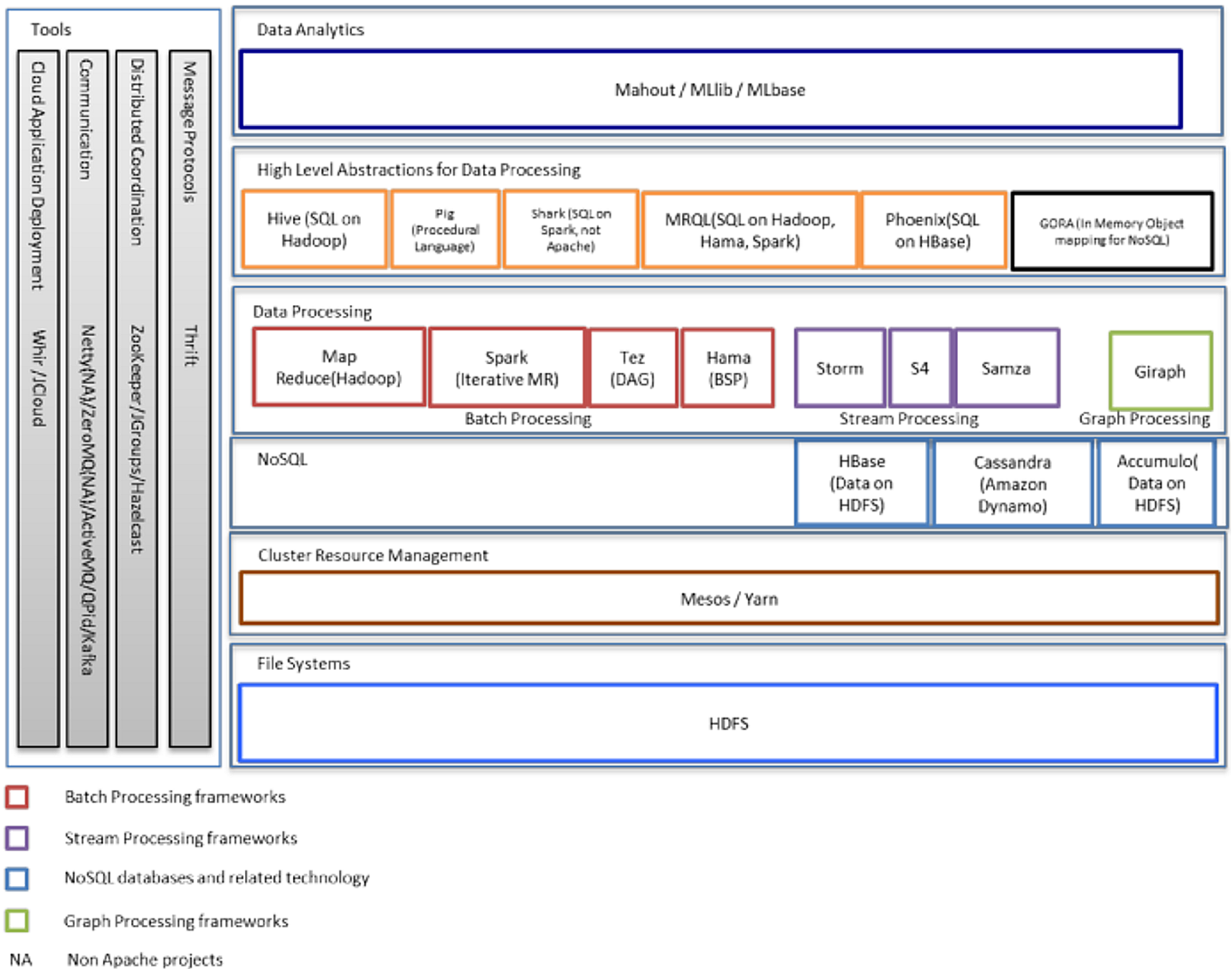


Figure 8: The Apache Big Data stack. Apache Flink is missing and would be in the same place as Apache Spark. (Year 2013. From: [KFLQ])

“Having specialized hardware like Super Computing infrastructures for doing such processing is not economically feasible most of the time. Large clusters of commodity hardware are a good economical alternative ...” [KFLQ]. This leads to new challenges such as hardware heterogeneity, node management, and common and expected hardware failures. These were tackled by Apache Yarn and Mesos. Both’s application purpose is to manage the resources in a cluster.

The next problem is providing the data inside the cluster. Hadoop File System (HDFS) solves this problem in the Apache Big Data stack.

6.2 Cluster Frameworks

The last, general problem is how to distribute the actual work. Here Spark and Flink come into the game, next to several other applications.

Still notably for us are the data analytic tools Mahout and MLlib. Both of them provide popular machine learning algorithms on top of Spark and/or Hadoop. They have a very similar set of algorithms. To us, MLlib seemed a bit more advanced, therefore we have chosen it as example in our experiments.

6.2.2 Hadoop, HDFS, and YARN

Now we introduce the core pieces of the Apache Big Data stack. Namely Hadoop (version 1.0 released in 2011, version 2.0 in 2013 [apa15c]) with the idea of MapReduce and its components HDFS [apa15b] and YARN [apa15e].

New to Hadoop was the simplicity and the short development cycles for developers: “Performing computation on large volumes of data has been done before, usually in a distributed setting. What makes Hadoop unique is its simplified programming model which allows the user to quickly write and test distributed systems, and its efficient, automatic distribution of data and work across machines and in turn utilizing the underlying parallelism of the CPU cores.” [had15] The simplified programming model is the Map-Reduce-approach. The automatic distributions of data and work is done by HDFS and YARN.

Yahoo initially developed Hadoop based on the Google File System [GGL03] and the famous Map-Reduce-approach [DG08] [KFLQ]. After making it open source it became a great success and the corner stone for the Apache Big Data stack.

Map-Reduce-algorithms base on a tuple of functions: a function mapping from A to B $f(A) : B$ and a reducing function $f(B, B) : B$. Thus, they are easily distributable. For example $v \cdot v^\top$ can be expressed with: the mapping $f(x_i) = x_i * x_i$ and the reducing function $f(x_i, x_j) = x_i + x_j$. The fact that the developer just needs to provide these functions is one case for the simplicity, because f.e. there is no need to concern about marshaling data or inter process communication. Another reason is, that the system itself is not the fastest, but it is designed for flat scalability [had15]. This means no need to refactor or rewrite the program if the developer wants to increase the number of cluster nodes.

A very important characteristic of the distribution strategy is the approach of moving computations to the data instead of vice versa.

This simple yet quite powerful programming model suffices for variety of tasks, but for some algorithms it is often not enough, especially machine learning ones.

The very base of Hadoop is the “Hadoop File System” (HDFS). It is designed for large files, which usually are once written and then often read. These large files are split into chunks and those are distributed with some

6.2 Cluster Frameworks

replication factor onto the cluster. The file system interface is similar to the Linux one.

In the first version, YARN did not exist and the functionality was integrated into MapReduce. From Version 2 on they were split and YARN got a fundamental base for other projects. YARN basically offers to reserve cluster resources for an application and uses those to execute the application processes. There is no failure handling included. Thus, this is left to the application developer.

6.2.3 Spark and MLlib

To overcome the Map-Reduce restrictions the Spark [ZCF⁺10][apa15d] program was created. Version 1.0 was released in May 2014 [spa15a]. The programming model of Spark is based on a “resilient distributed dataset (RDD), which is a fault-tolerant collection of elements that can be operated on in parallel.” [spa15d] Thus, it distributes large datasets in the memory of the cluster and computes operations in parallel, if possible.

The usage of Spark is as follows: Spark provides a client library that communicates with a cluster or a standalone version. A program using this library is called master. Inside this program, RDDs can be used and all the operations on those are dispatched to the cluster. To enable optimizations and reduce traffic, those operations are just scheduled when the master wants to retrieve a result of an RDD operation.

As for Hadoop, the user does not need to concern about the distribution of the data or computation and therefore Spark scales flat too.

The most important operations provided by RDD's are:

map: Mapping the set to another.

filter: Filter elements according to the binary value of some function.

union: Union two sets.

intersection: Interset two sets.

join: Join elements with the same key.

reduce: Reduce all elements to one result element.

The RDD sets usually are not sorted and there is no random access, thus the operations are very similar to mathematical operations of sets.

On top of Spark, MLlib [spa15c] was developed. Next to some general math features it provides several stochastic gradient descent algorithms. To name some: linear SVMs, logistic regression, decision trees, k-means clustering, principal component analysis.

6.3 Python

6.2.4 Flink

This project [apa15a] was created at the TU Berlin under the name Stratosphere [ABE⁺14]. After the admission as project in the Apache Software Foundation it was renamed to Flink due to naming conflicts. During the work on this paper, it was still in the incubating phase.

From the first point of view it provides a very similar functionality as Spark, i.e. the distributed manipulation of sets. In fact, nearly the same code semantics belonging to set operations work for both frameworks, but under the hood they are quite different. While Spark programs are executed as master and communicate with a cluster, Flink programs are submitted to a server, which executes them on a cluster. And whereas the Spark program manipulates sets, a Flink program creates a dataflow execution plan. This means, the developer specifies the set sources, the manipulations on them and the sinks, where to store them. Then the server creates out of that plan a real program and executes it on the cluster.

This restricts the programmer to use only operations supported by Flink, whereas in Spark he can fall back to general purpose computations on its master. On the other hand, it allows the Flink compiler to optimize the plan and further to create a pipeline structure, through which the data passes. Thus, those programs can also be used for stream processing [ABE⁺14].

We will discuss the further implications of the Flink programming model later in this work.

6.3 Python

In this two final sections, we describe more common tools. We begin with a short introduction into our chosen python projects.

6.3.1 Scipy, Numpy, Matplotlib

The python project for scientific tasks called Scipy (see [sci15]) got popular in the last years, due to the flexibility of Python and the great work of the community. We mostly use the Numpy (see [num15]) module which provides a n-dimensional array object backed by fast C-implementations and other useful mathematical features, such as linear algebra or random number functions.

Related to the popular MATLAB plotting functions, Matplotlib (see [mat15c]) provides a very powerful plotting library in Python. We use it to analyze and visualize our results.

6.3.2 CudaMat

The module CudaMat (see [Mni09]) created by Volodymyr Mnih is a simple yet powerful library to do matrix computations on CUDA-enabled devices.

6.4 SVM Programs

It is aligned to Numpy arrays and provides conversion functions between these types. There is no open interface to compile CUDA code.

6.3.3 PyOpenCl

PyOpenCl does a somewhat different job. Similar to CudaMat it enables acceleration on GPU-devices and provides some basic support for one-dimensional arrays. But the main scope is to provide an easy way to create and compile OpenCl code. For this purpose, it can abstract several OpenCl management tasks, especially memory management.

This project was created by Andreas Klöckner (see [pyo15]), who also maintains a similar module named PyCuda (see [pyc15]).

6.4 SVM Programs

Now we would like to conclude by presenting two popular SVM-solvers. For the general SVM-Algorithm please consider 4.2.

Both of them use the popular LIBSVM/SVM-light file format [las15]. Because of its popularity and to have the same starting point for all algorithms, we use it for the Bump Boost implementations, too.

6.4.1 LIBSVM

The first solver is called LIBSVM (see [CL11]), now in its third version (see [lib15a]). This library supports, next to support vector classification, also regression and distribution estimation. The implementation is a SMO-like algorithm described in [FCL05]. According to [CL11, page 26] the computational cost of this algorithm is in the worst case (all cache misses) $O(I * n * d)$ with I iterations, n data samples of d dimensions. Unfortunately, “empirically, it is known that the number of iterations may be higher than linear to the number of training data.” [CL11, page 26]

6.4.2 LaSVM

The second one is called LaSVM (see [BEWB05]) and uses an approximate online learning approach, which delivers already a good result after a single pass on the data. Even though some parts have “asymptotic cost ... like n^2 at most.” [BEWB05, page 10], the final algorithm has runtime of “ n^3 behavior of standard SVM solvers.” [BEWB05, page 10] The developers claim to use considerably less memory than LIBSVM [las15].

7 Implementations

This chapter describes the various implementations of the Bump Boost algorithm. The aim is to give the reader a good understanding of how the code base works and further how different computing framework influence the development. Of special interest is the usage of the Spark and Flink framework, which we describe and compare in more detail. The final sub chapter gives an informal impression on some coding fragments .

7.1 General Framework

Caused by the usage of different frameworks and technologies as by the lengthy and numerous runs it was necessary to create a general framework. The main purposes of this framework are:

Installation: An easy installation of all components, programs and datasets on the given computer.

Single interface: Providing a single interface for all algorithms and implementations to enable comparable tests and runtime measurements.

Testing: The functionality of the algorithms should be ensured by automatic tests.

Automatizing: The experiments with all their different implementations, configurations and datasets as their repetitions need to be scheduled and supervised automatically.

Analysis: To get a quick overview, the automatic creation of plots and tables is useful. For further understanding, an interface to customize the plots is, too.

In order to achieve these goals the following technologies were chosen. GNU Make (see [\[gnu15\]](#)) for the installation process, for all the rest we rely on Python and the marvelous Scipy environment with Numpy, Nosetests, and the Matplotlib (see [\[sci15\]](#)).

To achieve our goals the following design was used:

Installation: Using GNU Make all the needed libraries and datasets were downloaded, compiled, created, and installed automatically.

Single Interface: A python class hierarchy provides a single interface for the test and experiment procedures. All the implementations use this interface. For those not using Python, the implementation of this interface is more a stub, which converts the input, output data and launches the according programs.

7.2 Java

Testing: With Nosetests all the algorithms were tested. This is especially useful for Bump Boost as we are using various implementations and would like to ensure that all work in the complete same way.

Automatizing: Different self-made Python-programs allow us to declare and launch experiments by specifying the configured implementations, datasets, repetitions etc. For each run the runtime and test performance is measured.

Analysis: For each experiment the framework is able to automatically create the most important plots. For more enhanced analysis tasks, an interface for an enhanced plotting is provided.

7.2 Java

This code was originally written by Mikio Braun and is used for the evaluation in [BK]. The obtained code was not runnable, thus different fixes were needed to get it back to work. This affected mainly input and output mechanisms. At this point, we would like to emphasize that nothing of the algorithm implementation itself was changed, except some minor changes described below. Therefore, the run time should be comparable to the one in [BK].

For the framework alignment we added different configuration parameters. They allow us to change each control argument and to get complete same behavior as for the other version.

Achieving this, a small change has been introduced in the procedure for choosing the bump center. In the original implementation the first value of the sample array could not been drawn and the last one's probability was increased by the one of the first sample. All in all, this does not influence the effectiveness of the program much, we have changed to get the same behavior as in the other versions.

In the end we point out the particular characteristics of this version. First of two it uses the double data type, i.e. 64-Bit floating point numbers, whereas the other versions use the float data type, i.e. 32-Bit floating point numbers. The consequence is a slower program and the use of memory for samples etc. doubles. Second, this version uses JBlas ([jbl15]), whereas other implementations might use other linear algebra backends.

7.3 Python

The language of our choice is Python. Next to the scripting capabilities, useful for the general framework, Python's Scipy offers a fast and advanced toolkit for scientific and mathematical computations. Besides this core, numerous libraries provide interfaces to other technologies, f.e. CudaMat for GPU-enabled computations (See 6.3).

7.3 Python

All the following programs implement the above presented interface. Thus, the algorithm gets executed in the same process as the experiment scheduler. Other external implementations get invoked in an own one. This should have negligible influence on the runtime.

Below, we describe the characteristics of the various implementations.

7.3.1 Development Version

This initial version is just used for development reasons. It provides a fully functional and correct implementation of Bump Boost and is therefore used as reference point in the tests and further development. Besides the correctness, it is up to the order of a magnitude slower than the other python versions. Therefore it is not used in the final experiments.

7.3.2 Parallelized Version

Based on the theoretically best parallelization of Bump Boost (see [4.3.3](#)) two interfaces were created. The basic work flow between them is sketched here:

7.3 Python

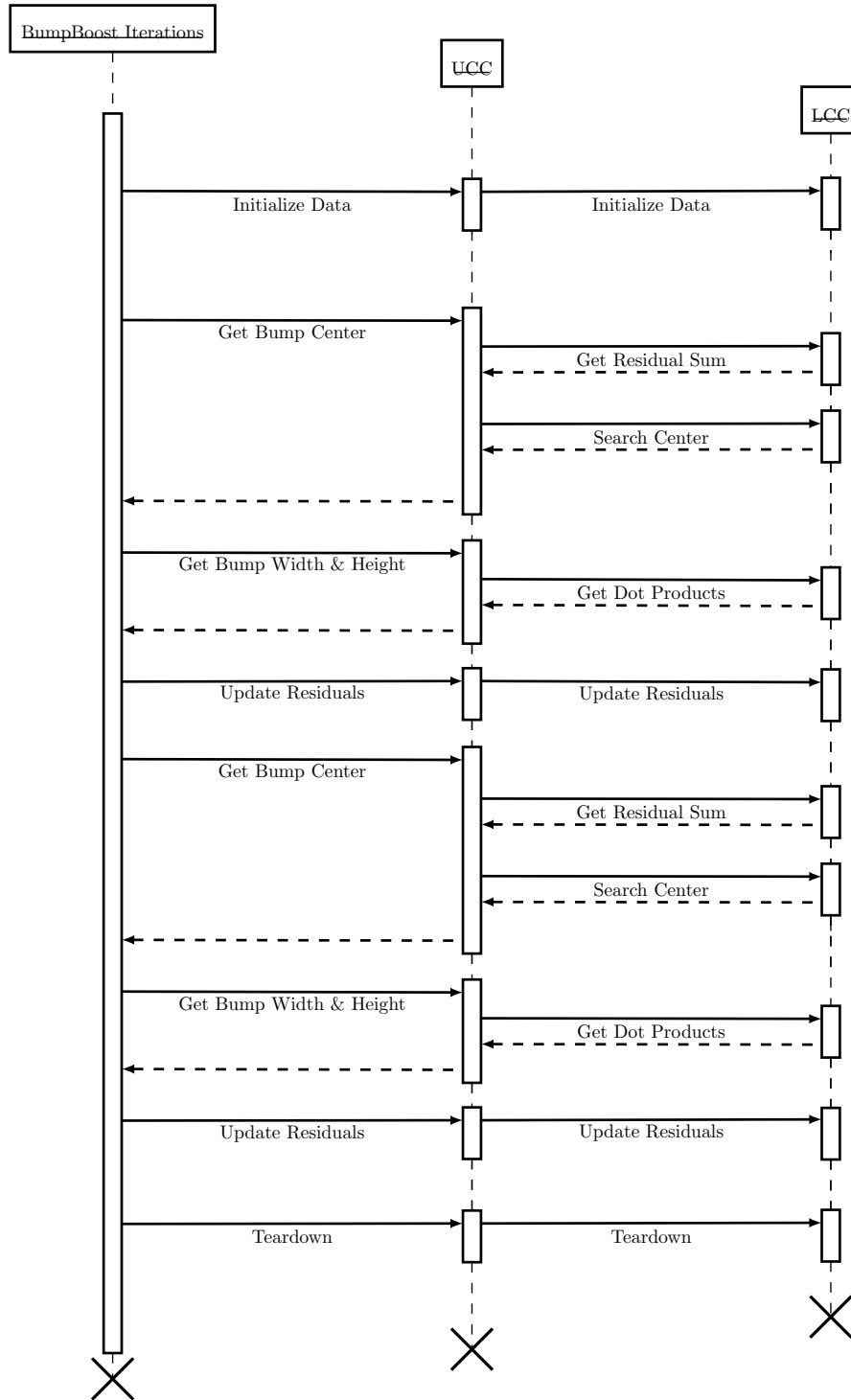


Figure 9: UML Sequence Diagram with basic work flow for two iterations between the algorithm implementation, the UCC, and the LCC in the Bump Boost case. For further descriptions, see below.

7.3 Python

The first one, named “Upper Computing Core” (in the following UCC), mainly abstracts the calculation of the various parameters in a single iteration. Next to that, functions to setup and tear down the component as update the residuals are provided.

The second one is named “Lower Computing Core” (in the following LCC). The LCC is designed to represent the leaves and nodes inside the computing tree i.e. the distributed computation parts. While the work of the UCC is still done at master node, work of the LCC can be done distributed and remotely.

To describe the processes in more detail:

Center Search: the LCC calculates the sum of the squared residuals, the UCC then chooses the random element and finally it is left to the LCC to find it.

Bump Boost Width and Height: to calculate the height and width in the Bump Boost case the LCC calculates the needed dot products and the UCC computes out of them the costs and the final height.

Multi Bump Boost Width and Height: in the Multi Bump Boost case it is a bit more complicated. In the UCC the gradient descent steps were done, the LCC mainly helps to complete them by first calculating the dot products for the needed height and then finishing the gradient calculations (taken from the file “python/implementation-s/numpy_bbcc” inside the code repository, see appendix [B](#)):

7.3 Python

```
1 # Here holds: actual_width = 10**width
2 def get_gradient(width):
3     kv_dot_kv, kv_dot_u = lcc.
4         compute_first_dot_products(center, width)
5
6     height = 0
7     if kv_dot_kv != 0:
8         height = kv_dot_u / kv_dot_kv
9
10    grad_width = 2 * height * lcc.
11        compute_second_dot_products(center, width,
12            height)
13    return grad_width
14
15 # Do gradient descent
16 ...
17
18 for i in range(self._gradient_descent_iterations):
19     grad_width = get_gradient(width)
20     # ...
21     width = ...
22
23 # Calculate final height
24 kv_dot_kv, kv_dot_u = self._lcc.
25     compute_first_dot_products(center, width)
26 height = 0
27 if kv_dot_kv != 0:
28     height = kv_dot_u / kv_dot_kv
29
30 width = 10**width
```

Where the UCC is implemented only once using numpy, the LCC is the computation intensive part and thus realized in different versions. They are described below.

7.3.3 Parallel and Remote LCC

The parallel LCC creates just n Python-threads and distributes the work to them. In addition to forwarding the calls and arguments, this core splits the work i.e. arguments apart and joins the results meaningful together. This LCC would be an inner node in the tree description of the Bump Boost parallelization (see 4.3.3).

The remote LCC is in fact just a stub, which forwards the call and argument to a remote server. A simple but powerful idea. In the tree description of the Bump Boost parallelization, this one is not visible as it just forwards data.

7.3 Python

As Python does not support real threads because of the Global Interpreter Lock (see [pyt15]), we use a more general approach by first splitting the work using the parallel LCC and then forwarding it using the remote LCC. These remote points, each an own process, can be either on the local or on a remote host. In the local case, this is an easy way for IPC using Python. In the remote one, it is an easy and powerful way to parallelize Bump Boost on different hosts.

The following graph shows an example structure. At the top is the UCC controlling the root LCC. A parallel LCC splits the data flow to its children. While a remote LCC just forwards the data. The leafs then do the actual computation, besides the merging in the parallel LCC:

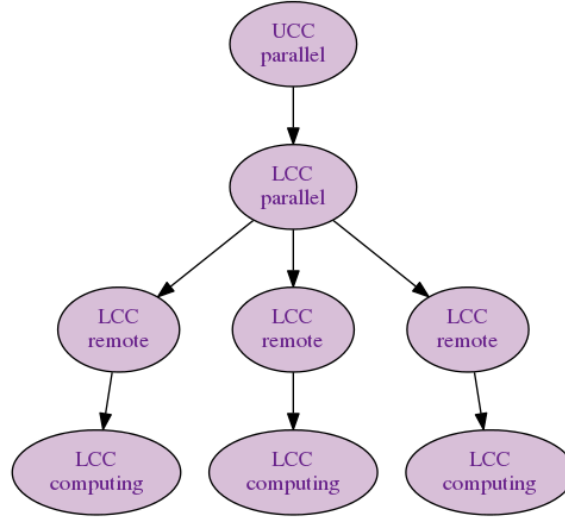


Figure 10: An example tree of LCCs.

7.3.4 Numpy LCC

The easiest implementation is this one. Using the Numpy toolkit, it implements the interface in a few lines of code, but is still highly effective.

7.3.5 CudaMat LCC

It uses the CudaMat project described in 6.3.2.

One of CudaMat's benefits is the easy and Numpy-like interface. Thus, porting code written in Numpy to the CudaMat library is quite easy. Assuming all the functionality is available.

In our implementation this is not the case, caused by a missing cumulative sum procedure used in the center search operation. Therefore, we used the alternative approach doing some sort of binary search (see 4.3.1). Even though much more data needs to be accessed, the performance penalty should be modest thanks to the massive parallel characteristics of GPUs.

7.4 Big Data Frameworks

7.3.6 PyOpenCL LCC

Unfortunately, we were not able to control the complexity of the OpenCL framework and create a performance boost. Therefore even having a working implementation, we refused to use it for the experiments.

Here we would like to describe in a short way the main advantage of PyOpenCL (see 6.3.3) and our problems. The benefit of working with OpenCL is that, the code can be used in a parallelized way on different computing devices i.e. CPU and GPU. On the CPU, the code is parallelized on different cores by the OpenCL framework. Another benefit is that OpenCL code can run on ATI, Intel and Nvidia GPU-devices, whereas Cuda code can only run on Nvidia GPU's. Next to this benefits inherited from OpenCL, PyOpenCL provides a neat library with a lot of abstractions for memory management, compilation processes etc. and a vector class with fast operations.

So far so good, on the other hand the OpenCL computing model can be tricky and for different devices different options behave in different ways. Without specific knowledge it is hard to get working code and especially a performance boost (For an example see 6.1.). A drawback of PyOpenCL itself is the missing support for matrices. F.e. CudaMat supports it. In PyOpenCL, all operations that can not be done on a flatten array need to be implemented in OpenCL code.

7.4 Big Data Frameworks

One of the initial objectives of this thesis was the examination of the Big Data frameworks Spark and Flink and their applicability for implementing machine learning algorithms. Whereas in the previous descriptions we left out most coding details, in this sub chapter we want to describe it in more detail.

In the chapter 6.2 we already described in the general environment around and the idea behind those frameworks. Now we would like to do the next step and describe the impact of these models on our work flow. This is done by first describing the implementation effort for Spark and Flink. In the next sub chapter a specific code section is compared.

Please note that only a simplified subset of the code is shown here, for the whole and working Scala-code please see the code base referred in appendix B.

7.4.1 Spark

The Spark (see 6.2.3) way is a very pragmatic one. The main idea of Spark is to have a “resilient distributed dataset“ called RDD and to make transformations to it. All of them get cached until the result is explicitly requested. That's the only impact on the programming structure. This means, the developer can program as he would like to, especially in cases where the

7.4 Big Data Frameworks

framework does not solve the problem well, he can fall back to, in our case, Scala.

On the other hand, this restricts the possibility of Spark to optimize the code and increases the impact of the developer on the program efficiency.

As we already stated, for Spark exists the machine learning library ML-LIB. We decided not to use it, because it did not seem very promising, in other words evolved. One of the reasons was the missing dot product between matrices and vectors which is needed in Multi Bump Boost. The other was that for Flink no such library exists, but for comparisons sake we wanted to keep the code as similar as possible.

The solution to our own math library was subclassing the Spark RDD class in combination with Scala implicits [OSV08, See sub chapter 6.12]. This allowed us to code more easily. In the rest of the section we describe our design choices.

Each of our RDD's represents a matrix of shape $n \times d$, thus possibly of shape $n \times 1$, and each element of the set represents a row of that matrix. Due to the missing ordering in a set, each element consists of a row index and an array containing the row elements. The same design choice was made by the creators of the MLLIB (see [spa15b]). As a consequence, for each operation including two matrices or vectors they need to be “zipped” together i.e. the elements need to be joined by index. This can be a major impact on the performance during math operations, especially if the according elements are not stored on the same hosts.

In the listing we show the simplified code for element-wise math operations:

```
1 object VectorRDD {
2   def +(X1: RDD[Vector], X2: RDD[Vector])=X1 zip X2 map{x => x._1 + x._2}
3   def -(X1: RDD[Vector], X2: RDD[Vector])=X1 zip X2 map{x => x._1 - x._2}
4   def *(X1: RDD[Vector], X2: RDD[Vector])=X1 zip X2 map{x => x._1 * x._2}
5   def /(X1: RDD[Vector], X2: RDD[Vector])=X1 zip X2 map{x => x._1 / x._2}
6
7   ...
8
9   def zip(X1: RDD[Vector], X2: RDD[Vector]) =
10     ((X1 keyfy) join (X2 keyfy)) map {case (k, (v1, v2)) => (v1, v2)}
11   def keyfy(X1: RDD[Vector]) = X1 map {x => (x.id, x)}
12 }
```

Given two vector RDDs, first the row elements are “zipped” together and then the according operations are applied on the single vectors(line 2-5). As stated, this means that the elements are joined by their row indexes. In Spark joins require a RDDs with elements of the form (key, value) and return a RDD with elements of the form (key, (value1, value2)). Knowing

7.4 Big Data Frameworks

this, we can follow the flow, where in line 10 the vector RDDs were mapped to the required form using `keyfy` (line 11) and then joined together. We don't need the key, so the mapping in line 10 removes it.

The actual operations on the vector are a loop over the array elements performing the desired operation.

Thanks to the flexibility of Scala we can overload operators such as “+”, “-” etc. and thus create a quite readable code. F.e. in the next listing, two vector RDDs can be element-wise multiplied just by writing “`X1 * X2`”, which invokes the operation in line 4 above.

The other math operations worth noting are:

```
1 object VectorRDD {
2   ...
3   def absV(X1: RDD[Vector]) = X1 map {x => x.abs}
4   def dot(X1: RDD[Vector], X2: RDD[Vector]) = X1 * X2 sumV
5   def sumV(X1: RDD[Vector]) = X1 reduce {_ + _}
6   def minV(X1: RDD[Vector], dimensions: Int) =
7     X1.fold(Vector.MaxValue(dimensions))(({acc, element}=>acc min element})
8   def maxV(X1: RDD[Vector], dimensions: Int) =
9     X1.fold(Vector.MinValue(dimensions))(({acc, element}=>acc max element})
10  ...
11 }
```

The code above illustrates how three of the basic operation types, namely “map”, “reduce”, and “fold”, work.

Line 3 shows a mapping of a vector RDD to the a vector RDD with it's absolute values.

In 6.2.2 we already described how reduce functions work. Here, we present an actual example. In the lines 4-5, first the vector RDDs get multiplied element-wise (see previous listing) and then the rows are summed up using the reduce function of the RDD class, resulting in a vector or matrix-with-vector dot-product.

While the reduce operations can be performed in parallel, the fold operations can not. Given an initial accumulator element a , for all elements e_i in a set following update operation is performed: $a = f(a, e_i)$. In our case, this semantic is used for a minimum and maximum function. After initializing the fold operation in line 7 with the maximum possible value the minimum is found with a function that always returns the smaller element. In line 8 it is done vice versa for the maximum.

If the result is a single element, f.e. using fold or reduce procedures, Spark returns the value instead of dataset with one element. Contrary to Flink as we will see later.

Backed by this mathematical functionality implementing Bump Boost and Multi Bump Boost was not a big deal as we could rely on Scala's power.

7.4 Big Data Frameworks

For searching the bump center we make the same design choice as in GPU case (see 7.3.5) and do some sort of binary search (see 4.3.1), but the reasons differ. Whereas in the GPU case we could not make a cumulative sum and thus we took advantage of the GPU parallel architecture, in Spark we could do it by fetching each element/row of a vector/matrix. The result would be really slow as it includes n transfers from the nodes to the master of a single element. We are faster to make only $\log n$ transfers, even though we might sum up to the half of the elements in the vector at the nodes.

7.4.2 Flink

The Flink way is the idealistic one. Flink(see 6.2.4) programs base on a data flow model built around data sets. This restricts the power of the developer on one hand, on the other the compiler has much more control, thus is potentially able to optimize the program better than f.e. in the Spark case.

Spark can be seen as a feature to a general purpose language. Flink programs are not general purpose. A general purpose language s.a. Scala is used to describe a Flink-program, but the Flink program itself is tied to the data flow paradigm.

For Flink, a math or machine learning library does not yet exist (a Mahout implementation is planned [mah15]). Fortunately, due to the similar semantics, we can use most of the code from Spark, with all its benefits and drawbacks. The only bigger difference is that we can join the rows/elements of matrix/vector more easily by addressing their elements:

```
1 object VectorDataSet {  
2   ...  
3   def zip(X1: DataSet[Vector], X2: DataSet[Vector]) =  
4     X1 join X2 where "id" equalTo "id"  
5     ...  
6 }
```


7.4 Big Data Frameworks

One of the major drawbacks of Flink is that everything is a data set. In Spark, the result of a dot product is a vector, in Flink, it is a data set with a single vector. If we would like to subtract a single vector from all vectors in a data set, this vector is a single one in a data set. In this case, either a cross join needs to be performed or a special annotation operation called “withBroadcastSet” needs to be applied. This broadcasts the passed data set (line 10) to all executing nodes, where it is converted into a Scala collection and extracted into a member object(line 7). Finally, in the map function the actual subtraction is applied:

```
1 object VectorDataSet {
2   ...
3   def subVector(X1: DataSet[Vector], X2: DataSet[Vector]) = X1.map(
4     new RichMapFunction[Vector, Vector]{
5       var v: Vector = null
6       override def open(config: Configuration) = {
7         v = getRuntimeContext.getBroadcastVariable("v").toList.head
8       }
9       def map(x: Vector) = {x - v}
10    }).withBroadcastSet(X2, "v")
11   ...
12 }
```

A more realistic example is given in the next listing. It shows the final update of the R-Prop algorithm (see 4.1.4) i.e. $width = width + update * sign(gradient)$. As everything is a data set, also the values for the update and gradient variable are data sets, thus need to be broadcasted into the mapping function of the data set:

```
1 ...
2 width = width.map(new RichMapFunction[Vector, Vector]{
3   var update: Vector = null
4   var gradient: Vector = null
5   override def open(config: Configuration) = {
6     update = getRuntimeContext.getBroadcastVariable("update").toList.head
7     gradient = getRuntimeContext.getBroadcastVariable("gradient").toList.
8       head
9   }
10   def map(x: Vector) = {(x + update * (gradient sign))}
11 }).withBroadcastSet(update, "update").withBroadcastSet(gradient, "gradient")
12 ...
```

This is already quite cumbersome for such simple operations. Furthermore, Flink does not offer control statements for operations on data sets.

7.4 Big Data Frameworks

In more detail, the content of data sets can be added, modified, and removed, but if a certain operation is applied to a data set or not is fixed [fli15d]. Imagine a pipeline structure which controls the passed content, but the structure itself cannot be modified. An example of the code explosion is given in the next section 7.5.2.

This workaround of using data sets with single values, broadcasting data sets and putting the program logic into “open” function of these rich map and filter functions make Flink quite flexible. But the programming is hard and the compiler has problems with the increased amount of nodes, i.e. data sets, the program contains. More on that below.

Other restrictions are imposed when using loops. Flink offers two types of loops. The first one is called “Bulk Iteration”, which allows to modify a data set with an iteration function for n times. The second one is called “Delta Iteration”, which allows to provide a work data set and cumulates the results in a solution set. The loop performs until the work set is empty or the developer set maximum iteration count is reached. For both, also a custom “aggregator” can be used to control the termination (see [fli15c]).

The following example shows how bulk iterations work by calculating the factorial of 100. First of all, an iteration needs a working set, i.e. a set used for the iteration, created in line 1. Line 3 states that we want to make 100 iterations. What each iteration does, is determined by the so called step function, line 4 to line 8, which gets as input in the first round the input data set, in our case “initial”, then in each round the output of the last iteration. Please note that Scala functions implicitly return the last object in a function, in this case “result” in line 8. In lines 4-7 the input data “iterationInput” gets mapped to the “result” data set. In the map function the factorial gets calculated by multiplying the only number in the data set with “getIterationRuntimeContext.getSuperstepNumber”, which returns the current iteration index beginning at 1. As mentioned above, everything is a data set, thus to update the factorial value we need to use a mapping.

```
1  val initial = env.fromElements(1)
2
3  val factorial = initial.iterate(100) {
4    iterationInput: DataSet[Int] =>
5      val result = iterationInput.map { i =>
6        i * getIterationRuntimeContext.getSuperstepNumber
7      }
8      result
9  }
```

7.4 Big Data Frameworks

The first problem using these iterations is that only one variable, i.e. changed during one iteration, data set can be passed into the loop, above called working set. This results in strange workarounds. An example is the following loop in the R-Prop algorithm. First, all needed variables, i.e. actual width, update value, and last gradient, need to be merged into one set, and then in each iteration they need to be separated at the beginning (line 12-14) and merged in the end (line 18-22):

```
1  ...
2  val startWidth = env.fromCollection[Vector](Seq(startWidthVector))
3    map {x => new Vector(0, x.values)}
4  val startUpdate = env.fromCollection[Vector](Seq(startUpdateVector))
5    map {x => new Vector(1, x.values)}
6  val startLastGradient = env.fromCollection[Vector](Seq(zerosVector))
7    map {x => new Vector(2, x.values)}
8
9  var stepSet = startWidth union startUpdate union startLastGradient
10 stepSet = stepSet.iterate(config.gradientDescentIterations){
11   stepSet =>
12   var width = stepSet filter {_.id == 0} neutralize;
13   var update = stepSet filter {_.id == 1} neutralize;
14   var lastGradient = stepSet filter {_.id == 2} neutralize;
15
16   ...
17
18   width = width map {x => new Vector(0, x.values)}
19   update = update map {x => new Vector(1, x.values)}
20   lastGradient = lastGradient map {x => new Vector(2, x.values)}
21
22   width union update union lastGradient
23 }
24 val width = stepSet filter {_.id == 0}
25 ...
```

In addition, it is not supported to nest loops, nor will it be in the near future [\[fi15j\]](#). Thus it's hardly possible to implement Bump Boost and Multi Bump Boost. Unfortunately, we discovered this during the development, because the fact has not been stated in the official documentation(Not in [\[fi15c\]](#) nor in [\[fi15d\]](#) as of Flink version 0.8, January 26. 2015).

During the implementation of Bump Boost and Multi Bump Boost we encountered several bugs ([\[fi15e\]](#), [\[fi15f\]](#), [\[fi15g\]](#), [\[fi15h\]](#), [\[fi15i\]](#)) which needed to be fixed by the Flink developers. Even though Flink is still under development, some of the bugs were of general nature. Because of this and after some discussions with the Flink developers, we have the impression that nobody tried to implement something similar in Flink before.

7.5 Selected Code Comparisons

In the end, we were not able to produce a working Bump Boost or Multi Bump Boost program using Flink. After completing the implementation for a single iteration, which worked fine for both algorithms, we would have needed a nested loop to repeat that iteration. Unfortunately, this is not supported by Flink.

We tried to replicate the loop code using a template script. It was possible to do two iterations. Doing the step from two to three iterations the Flink server did not stop computing. We suspect that the compiler could not cope with the complex computation graph, because each variable is a data set and Flink tries to optimize the flow of them all, not knowing that only one element per time will be inside each of these data sets and no optimization is needed.

To summarize, the biggest problems using Flink are a missing linear algebra library and the restrictions of unordered sets when creating an efficient matrix implementation as it is for Spark. Furthermore, the loop semantics and the mantra “everything is a data set” make the coding cumbersome and hard. Even more, not all algorithms can be expressed in Flink as Bump Boost and Multi Bump Boost show.

Due to all these problems, it was not possible to implement and test Bump Boost and Multi Bump Boost on the Flink framework. An example of how complex Flink code can be compared to Spark code is given in the next section.

7.5 Selected Code Comparisons

In this final section we compare some code fragments of selected implementations we discussed above. The scope of this is to introduce the reader to the code complexity depending on the language and platform choice, not to introduce the reader to language, platform, or code details.

We begin with the code to draw a center according to the given distribution comparing the implementations in Numpy and CudaMat. We conclude by illustrating an impression of the code for R-Prop in Multi Bump Boost using Spark and Flink.

For the whole and working code please see the code base referred in appendix [B](#).

7.5.1 Draw Center

The probability distribution and the algorithm, how to determine a center in Bump Boost and Multi Bump Boost are outlined here [4.3.1](#). Two different versions are named. One is using a cumulative sum, the other a binary search approach. The former is implemented in the left listing using Numpy. The later is used in the right one using CudaMat, because a cumulative sum is not supported:

7.5 Selected Code Comparisons

```

1  ...
2  # common code
3  def get_center(self):
4      s = self._lcc.sum_u_2()
5      x = s * numpy.random.rand(1).
        astype(self._data_type)[0]
6      c = self._lcc.search_center(x)
7      return c
8  ...
9  def sum_u_2(self):
10     self._u_cumsum = (self._u**2).
        cumsum()
11     return self._u_cumsum[-1]
12
13 # returns center which belongs to this
    random state
14 def search_center(self, x):
15     ret = self._X[numpy.sum(x > self.
        _u_cumsum)]
16     self._u_cumsum = None
17     return ret
18 ...

```

```

1  ...
2  # common code
3  def get_center(self):
4      s = self._lcc.sum_u_2()
5      x = s * numpy.random.rand(1).
        astype(self._data_type)[0]
6      c = self._lcc.search_center(x)
7      return c
8  ...
9  def sum_u_2(self):
10     self._u_2 = cudamat.empty(self._u.
        shape)
11     self._u.mult(self._u, target=self.
        _u_2)
12     return self._u_2.sum(0).sum(1).
        asarray()[0, 0]
13
14 # returns center which belongs to this
    random state
15 def search_center(self, x):
16     r = [0, self._X.shape[0]]
17     while r[1]-r[0] > 1:
18         middle = int(math.ceil((r[0]+r
            [1])/2.0))
19
20         rs = self._u_2.get_row_slice(r
            [0], middle)
21         s = rs.sum(0).sum(1).asarray()
            [0, 0]
22         self._free(rs)
23         if x > s:
24             x -= s
25             r[0] = middle
26         else:
27             r[1] = middle
28
29         ret = self._X.numpy_array[r[0]]
30         self._u_2 = self._free(self._u_2)
31         return ret
32 ...

```

For a general understanding, the “get_center” function draws the actual center, “sum_u_2” computes the sum of the squared residuals and “search_center” retrieves the actual center value.

7.5 Selected Code Comparisons

7.5.2 R-Prop

This gradient descent algorithm is described here [4.1.4](#). It is used in Multi Bump Boost to calculate the width parameter. The width is boxed into the range $minWidth < width < maxWidth$. We would like to leave the reader with a general impression, thus do not further explain the coding details.

In Spark, the implementation is quite clean:

```
1  ...
2  def calcMBBWidth(residuals: RDD[BumpBoost.scalarType], center: Vector) = {
3    // spark closure problem workaround
4    val centerX_tmp = centerX
5
6    var width = Vector.ones(config.dimensions) * config.startWidth
7    var update = Vector.ones(config.dimensions) * 0.01F
8    var lastGradient = Vector.zeros(config.dimensions)
9
10   for(i <- 0 until config.gradientDescentIterations){
11     val gradient = BumpBoost.getGradient(centerX_tmp, residuals, center,
12       width)
13     val term = gradient * lastGradient
14     lastGradient = gradient
15
16     update = update.condMul(term.isLess(0), 0.5F)
17     update = update.condMul(term.isGreater(0), 1.2F)
18     update = update.clip(config.minWidthUpdate, config.maxWidthUpdate)
19
20     width = width + update * (gradient sign)
21     width = width.clip(config.minWidth, config.maxWidth)
22   }
23   width = width invPow 10
24   width
25 }
26 ...
```

7.5 Selected Code Comparisons

While in Flink the code is much longer due to the above stated problems:

```

1  ...
2  def calcMBBWidth(env: ExecutionEnvironment, centerX: DataSet[Vector], residual: DataSet[Vector
3    ], center: DataSet[Vector]): (DataSet[Vector], DataSet[Vector]) = {
4    val startWidth = env.fromCollection[Vector](Seq(Vector.ones(config.dimensions) * config.
5      startWidth)) map {x => new Vector(0, x.values)}
6    val startUpdate = env.fromCollection[Vector](Seq(Vector.ones(config.dimensions) * 0.01F))
7      map {x => new Vector(1, x.values)}
8    val startLastGradient = env.fromCollection[Vector](Seq(Vector.zeros(config.dimensions))) map
9      {x => new Vector(2, x.values)}
10
11    var stepSet = startWidth union startUpdate union startLastGradient
12    stepSet = stepSet.iterate(config.gradientDescentIterations){
13      stepSet =>
14        var width = stepSet filter {_.id == 0} neutralize;
15        var update = stepSet filter {_.id == 1} neutralize;
16        var lastGradient = stepSet filter {_.id == 2} neutralize;
17
18        val gradient = getGradient(centerX, residual, center, width) //neutralize
19        val term = gradient * lastGradient
20
21        lastGradient = gradient
22
23        val minWidthUpdate = config.minWidthUpdate;
24        val maxWidthUpdate = config.maxWidthUpdate;
25        update = update.map(new RichMapFunction[Vector, Vector]{
26          var term: Vector = null
27          override def open(config: Configuration) = {
28            term = getRuntimeContext.getBroadcastVariable("term").toList.head
29          }
30
31          def map(x: Vector) = {x.condMul(term.isLess(0), 0.5F).condMul(term.isGreater(0), 1.2F).
32            clip(minWidthUpdate, maxWidthUpdate)}
33        }).withBroadcastSet(term, "term")
34
35        val minWidth = config.minWidth;
36        val maxWidth = config.maxWidth;
37        width = width.map(new RichMapFunction[Vector, Vector]{
38          var update: Vector = null
39          var gradient: Vector = null
40          override def open(config: Configuration) = {
41            update = getRuntimeContext.getBroadcastVariable("update").toList.head
42            gradient = getRuntimeContext.getBroadcastVariable("gradient").toList.head
43          }
44
45          def map(x: Vector) = {(x + update * (gradient sign)).clip(minWidth, maxWidth)}
46        }).withBroadcastSet(update, "update").withBroadcastSet(gradient, "gradient")
47
48        width = width map {x => new Vector(0, x.values)}
49        update = update map {x => new Vector(1, x.values)}
50        lastGradient = lastGradient map {x => new Vector(2, x.values)}
51        width union update union lastGradient
52      }
53    val width = stepSet filter {_.id == 0} map {_ invPow 10}
54    width
55  }
56  ...

```

8 Competitive Solutions

In order to compare Bump Boost’s effectiveness, we have chosen two different sorts of competitors. The first ones are SVM solvers and the second ones are part of the MLlib toolbox. This chapters aims to describe how they were used.

8.1 SVM Solvers

The SVM solvers LaSVM and LIBSVM, described here [6.4](#), are used in the same setup as in [\[BK\]](#). This means we use K-fold cross validation to find the best parameter configuration for a training set and finally train with this configuration on the whole set before we predict the test sets results.

In contrast to LaSVM, LIBSVM provides an interface for K-fold cross validation. For better comparability, we do not use it and implement it in Python. I.e. our python program splits the training set and uses the SVM solvers to cross validate on the parameter space. The best configuration is then used to get the final and tested model.

8.2 MLlib

MLlib, see [6.2.3](#), as competitor stands for the Big Data machine learning. The common approach is to use simple algorithms and hope that they work well with lots of data. To stay on the track, we use two stochastic gradient descent algorithms with following objective function:

$$f(w) := C R(w) + \frac{1}{n} \sum_{i=1}^n L(w; x_i, y_i) \quad (32)$$

Once with a hinge loss i.e. linear SVM:

$$L(w; x, y) := \max\{0, 1 - yw^\top x\}, \quad y \in \{-1, +1\} \quad (33)$$

and once logistic loss i.e. logistic regression:

$$L(w; x, y) := \log(1 + \exp(-yw^\top x)), \quad y \in \{-1, +1\} \quad (34)$$

The regularization $R(w)$ can be no, the “L1” ($\|w\|_1$), or the “L2” ($\frac{1}{2}\|w\|_2^2$) function.

As with the SVM Solvers, we use K-fold cross validation to achieve the best performing parameter set. Contrary to them, the cross validation is not done in Python, but in the Spark master application. This decision is caused by the long setup time of Spark applications in contrast to the SVM solvers and the need to write a Spark master program anyway.

9 Data Sets

The aim of this chapter is to summarize and describe the data sets used in our experiments. Overall, the selection is oriented at the one of [BK]. The major setup modification, besides different data set sizes, is that the forest cover sets are not scaled and have a fixed test set.

9.1 Splice

The splice data set [spl15] consists of 3.190 instances, each a DNA-Sequence of length 60 and a label. The labels describe if the sequence is a splice side or not. The task can be subdivided by classifying the splice sides into “exon/intron” and “intron/exon” boundaries. Which is not done in our experiments.

For our tests we divided the data set into a training set of size 1.000 and a test set of size 2.175. The DNA-labels e.g. “A”, “C”, “G”, and “T” are encoded to numbers between 1 and 4.

9.2 MNIST

The MNIST database of handwritten digits is a famous data set created and first used in [LBBH98]. Composed of 60.000 training and 10.000 test images chosen from NIST, it is a very popular classification problem. Each image consists of an 28x28 pixel-array. The pixels itself are described by a floating point value indicating their gray level.

In our setup we use the official test set and the task is to classify “1” vs the rest. In addition, we use smaller training sets of size n , where each repetition is a randomly chosen subset of the official training set. Following values for n were used: 1.000, 5.000, 10.000, 20.000, 50.000, and 60.000.

9.3 Forest Cover

The problem of the “Covertypes Data Set” [for15] is to classify the cover type of a 30x30 meter forest cell by evaluating cartographic values. The 54 features are not scaled and some are real values, some not. Moreover some features are qualitatively independent from the other. All in all, there are 7 labels and 581.012 examples.

In our setup, we have randomly chosen a fixed test set of size 181012. The rest is used as training set. As for the MNIST data sets, the task is classify class 1 vs the rest and we randomly sub sample again the training set for sizes of n equal to 1.000, 5.000, 10.000, 50.000, 100.000, 200.000, and 400.000.

9.4 Checkers

9.4 Checkers

This data set was invented in [BK] to benchmark Bump Boost. Given a checkers board, divided by a 20x20 grid, randomly one of two classes gets assigned to each of the equally sized squares. Random points were then classified by the square's class, in which they lay. Even though the Bayes error is zero, a lot of information is needed to classify a field correctly and to find out that each field is square.

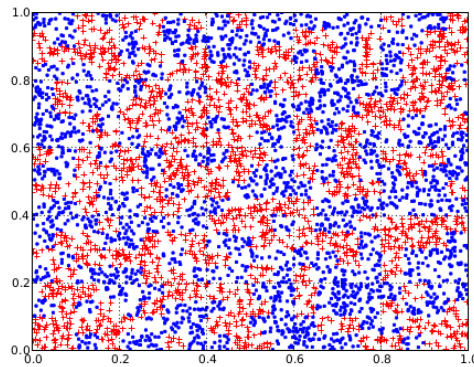


Figure 11: An example of a Checkers data set instance with 5000 points (From: [BK]).

As we can choose the size of our data set, this one is well suited to test the scaling properties of Bump Boost. To do so, we have randomly generated ten different data set collections, each made out of an own ground truth i.e. label assignment. The individual collections consist of a test set with 100.000 samples and training sets of size n equal to: 100, 200, 300, 400, 500, 600, 700, 800, 900, 1.000, 2.000, 3.000, 4.000, 5.000, 6.000, 7.000, 8.000, 9.000, 10.000, 20.000, 30.000, 40.000, 50.000, 60.000, 70.000, 80.000, 90.000, 100.000, 200.000, 300.000, 400.000, 500.000, 600.000, 700.000, 800.000, 900.000, 1.000.000, 2.000.000, 3.000.000, 4.000.000, 5.000.000, 6.000.000, 7.000.000, 8.000.000, 9.000.000, 10.000.000.

10 Experiments and Results

This chapter first describes the experiment setup, followed by an in-depth evaluation of the experiment results.

10.1 Experiment Setup

The description of the general experiment cycle is followed by the their technical and model parameters. We conclude with a depiction of the measurements made and of the final evaluation.

10.1.1 Cycle and Parameters

All the experiments are scheduled automatically by our framework (see 7.3). Depending on the current experiment configuration, for each selected data set and for each selected algorithm implementation a model training is performed. For each data set, 10 different repetitions are made. If the data set is a sub set of a bigger one, 10 different sub sets of equal size are used. After the training, all the learned models, i.e. 10 for each data-set-implementation tuple, are tested on the test sets. Plotting and other evaluation is then done on purpose by using the serialized experiments.

Except for experiments involving a GPU-enabled implementation, all the experiments are executed on a cluster of 4 machines. On the cluster HDFS and YARN are installed. While HDFS is used by all implementations to fetch the data sets, YARN is only used by those using Spark. Experiments with GPU usage are launched on a dedicated computer, which also has HDFS installed to provide the data sets. The details to all machines are listed in appendix A.

For each algorithm we have a different parameter setting. For the sake of comparison we use the same as in [BK] for Bump Boost, Multi Bump Boost and non-linear SVMs. For Spark MLlib we choose our own parameters.

In more detail, the iteration count for Bump Boost and Multi Bump Boost varies from setup to setup and is given in the description of the results. For all Multi Bump Boost experiments the following R-Prop settings are in common: start value of 1.0, minimum update value of 10^{-8} and a maximum update value of 10. If not stated otherwise, we use 30 gradient descent steps.

The SVM solvers always use a Gaussian kernel, i.e. $\exp(-\gamma\|x-y\|^2)$, and a cache size of 512 MB. We use 5-fold cross validation to find an appropriate value for γ and the regularization constant C . For both of them, 5 values are provided, resulting in 125 training runs. This setup is justified by the fact that Bump Boost and Multi Bump Boost have built-in parameter selection (see 4.3.2), while this generally is not the case for SVM with non-linear kernels, i.e. it is common practice to use cross validation. Furthermore, the kernel parameter, i.e. width size, is some sort of cross validation performed

10.1 Experiment Setup

for each iteration. In any case, the asymptotic run time will not be influenced by this decision.

A similar setup is used for MLlib, where we use a linear SVM and logistic regression. Also in this setup, 5-fold cross validation is used to determine the best parameters. Both linear SVM and logistic Regression base on the same stochastic gradient descent algorithm. In the SVM case, a hinge loss is applied, while for logistic regression a logistic loss function is applied. Each setting is used once with L1- and once with L2-loss, while for all of them 50 gradient descent steps are made.

Now the table reflects the data set specific parameters. All the values should be interpreted as exponents to the basis 10, i.e. $-3..1$ means $10^{-3}..10^1$. For Bump Boost and Multi Bump Boost, the ranges stand for the width parameter selection. In Bump Boost 20 values with logarithmic spacing are chosen out of the range, while for Multi Bump Boost the range is the box constraint for the modified R-Prop algorithm. Similar to Bump Boost, the SVM parameters are chosen with logarithmic spacing, but in this case 5 apiece for γ and C . The same applies to MLlib, where the regularization parameter C is listed and 6 values are chosen:

Algorithm	Parameters	Splice	MNIST	Forest	Checkers
Bump Boost	w	$-1..3$	$5..10$	$-2..2$	$-4..0$
Multi Bump Boost	w	$-1..3$	$5..10$	$-2..2$	$-4..0$
SVM	γ, C	$-3..1, 0..2$	$-10..5, -2..2$	$-3..1, 0..2$	$0..4, -2..2$
MLlib	C	$-2..2$	$-2..2$	$-2..2$	$-2..2$

Table 1: Data set depending parameters of the algorithms. Please see the text above for further explanations.

10.1.2 Measurements and Evaluation

It is not the main purpose of this thesis to show the effectiveness of Bump Boost in terms of absolute classification or regression error. This has already been done in [BK]. Classification errors are still recorded and compared, but we mainly want to see at which time instant they are reached and how the training time does evolve with increasing data and/or changing computing power.

Given the importance of the run time, a detailed examination is justified.

First to mention is that we also measure the data loading. This means fetching the data out of an HDFS storage, where all the data set files are stored in the LIBSVM-format (see [lib15b]). The reason for this choice is the heterogeneity of our platforms i.e. Python programs usually load the data from disk, whereas frameworks as Spark and Flink do it from a distributed storage.

Besides, in Spark and Flink it is not easily possible to measure the run time of specific code regions, due to their programming models and frame-

10.2 Results

work implementations, while in the Java SVM-implementations LIBSVM and LaSvm it is not possible without modifying them.

Next to that, different applications have different start-up times. For Python and Java it is nearly negligible, for Spark and Flink it is not. Thus, measuring the whole run time gives a more complete picture regarding the different loading techniques and computing frameworks.

The classification error is measured by the amount of wrongly classified predictions, i.e. given the correct labels $Y \in \{-1, +1\}^n$ and the predictions $\hat{Y} \in \{-1, +1\}^n$ the error is $\frac{1}{2n} \sum_{i=1}^n \text{sgn}(Y_i - \hat{Y}_i) + 1$.

Each training or test run has one hour to complete, otherwise it will be stopped.

All the specified values in the rest of this chapter are the mean out of 10 repetitions, if not stated otherwise.

10.2 Results

In this sub chapter we present the results of our experiments. At the beginning, the Bump Boost algorithms are evaluated and then they were compared to the SVM solutions and Spark MLlib. Subsequent, the emphasis on scaling is valued, until a more detailed evaluation of the Spark programs concludes this section.

In the following, these abbreviations are used:

“BB” and “MBB”: Bump Boost and Multi Bump Boost.

“It.” and “it.”: iterations.

“Numpy”, “default”, “Numpy NxM”, “NxM”, “GPU”, and “Spark”:

“Numpy” describes the default single-threaded implementation of Bump Boost, also called “default”. “Numpy NxM” or just “NxM” stands for the distributed version, running on N hosts, on each with M instances. “2x3” would mean 6 instances in total. The final two abbreviations are dedicated for the CudaMat and the Spark implementation.

“Lin. SVM” and “Log. Reg”: stand for the Spar MLlib linear SVM and logistic regression algorithms.

“Splice”, “MNIST”, “Forest”, and “Checkers”:

The Splice, MNIST, forest cover and Checkers data sets.

The used data set, leading to the plotted data, is mentioned in the plot title. Please mind that for the forest cover and the MNIST data sets we the classification task is “class 1 vs the rest”.

10.2 Results

10.2.1 Basic Results

At the beginning, we show that Bump Boost and Multi Bump Boost do indeed scale linearly. Then the results of the algorithms are compared, preceding to the comparison of different Bump Boost implementations.

Linear Scaling

The figure below empirically shows that Bump Boost and Multi Bump Boost scale linearly with increasing data size. Furthermore, it shows that Multi Bump boost performs much slower than Bump Boost. Even if it grows linearly, the factor is much worse. This is mainly due to increased computing effort of the gradient descent. Especially in the parallelized version, each gradient descent step imposes 2 communication attempts to the nodes, while for Bump Boost it is only one during the whole width determination. We therefore still expect Multi Bump Boost to scale as well as Bump Boost, but with more overhead and an implicated later pay off.

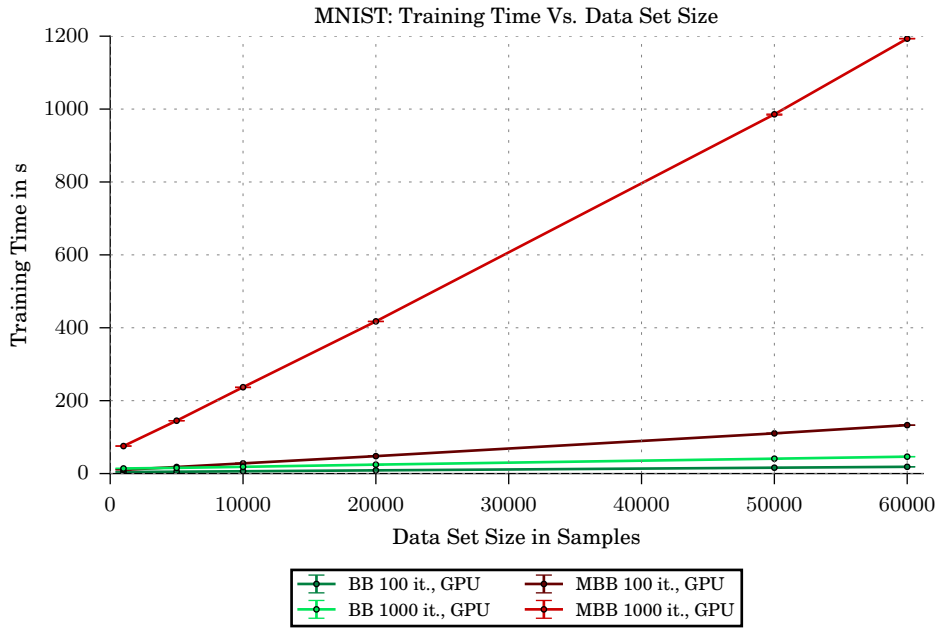


Figure 12: How the training times of Bump Boost and Multi Bump Boost evolve with increasing data set size.

Bump Boost versus Multi Bump Boost

The next table shows the best results of Bump Boost and Multi Bump Boost on the used data sets.

For MNIST and the forest cover data set, both have similar classification results. While for the Splice data set, Multi Bump Boost performs much

10.2 Results

better than Bump Boost and shows it's abilities.

The opposite is the case for the Checkers data set, where Multi Bump Boost does not learn well. The examined cost function seemed to be of convex nature and unfortunately, we lacked the time for further investigations. Thus, we cannot give a reliable cause for this behavior.

Splice:	BB 100		MBB 100	
	23.72 ± 0.20		4.54 ± 0.88	
MNIST:	BB 100	BB 1000	MBB 100	MBB 1000
	0.44 ± 0.03	0.28 ± 0.02	0.23 ± 0.03	0.20 ± 0.03
Forest:	BB 1000	BB 5000	MBB 1000	MBB 5000
	18.48 ± 0.23	11.76 ± 0.09	13.32 ± 0.11	10.43 ± 0.05
Checkers:	BB 500	BB 2000	MBB 500	MBB 2000
	12.45 ± 0.94	3.80 ± 0.33	30.32 ± 1.85	22.83 ± 1.49

Table 2: The classification error and standard deviation in percentage for the Bump Boost and Multi Bump Boost algorithms trained with the GPU implementation.

On the MNIST data set, both perform similarly, but the better classification rate of Multi Bump Boost is paid with an increased training time:

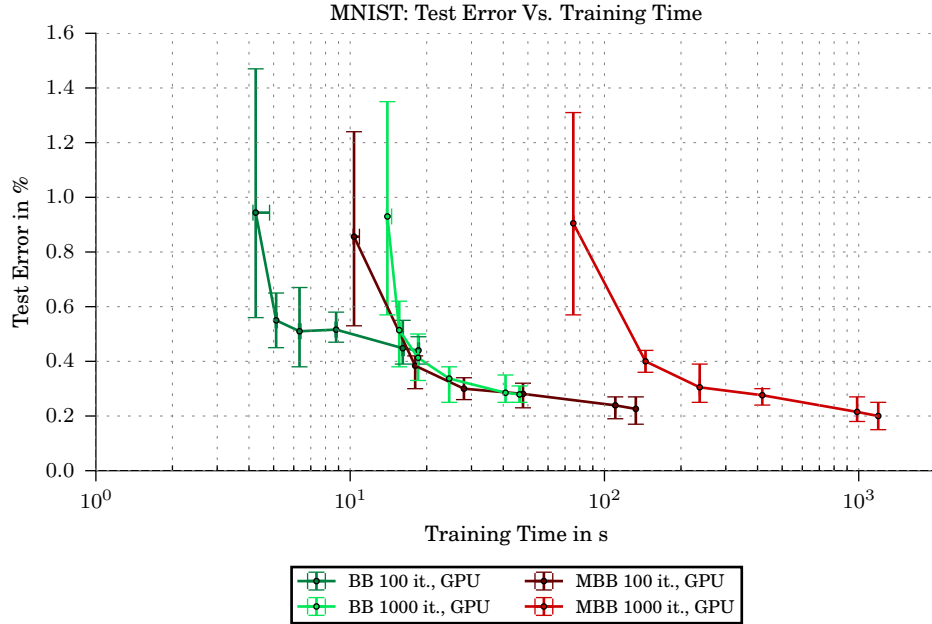


Figure 13: Plot on how the run times of Bump Boost and Multi Bump Boost are related to the test error.

As the table and the plot illustrate, Multi Bump Boost is able to learn better with less iterations, except for Checkers. As result, the final prediction function takes less time to complete, which can be an advantage.

10.2 Results

Equally Functional Implementations

The sole aim of table 3 is to show that all Bump Boost implementations work equally well. Due to its increased training time, Multi Bump Boost was only tested on the Splice data with all implementations. All performed equally well. Where this results can be found is noted in appendix B.

BB Impl.	Splice	MNIST	Forest	Checkers
Default	23.66 ± 1.19	0.27 ± 0.02	11.81 ± 0.12	12.45 ± 0.91
Java	22.86 ± 1.49	0.28 ± 0.02		12.46 ± 0.91
GPU	23.72 ± 0.20	0.28 ± 0.02	11.76 ± 0.09	12.45 ± 0.94
1x1	23.15 ± 0.60	0.28 ± 0.03	11.85 ± 0.09	12.49 ± 1.17
1x2	22.40 ± 0.86	0.27 ± 0.03	11.85 ± 0.06	12.39 ± 1.21
1x3	23.07 ± 0.96	0.29 ± 0.02	11.91 ± 0.10	12.55 ± 1.13
1x4	22.65 ± 0.50	0.26 ± 0.02	11.88 ± 0.11	12.60 ± 1.07
1x5	22.71 ± 0.90	0.29 ± 0.02	11.86 ± 0.07	12.18 ± 0.96
1x6	22.62 ± 1.10	0.26 ± 0.02	11.85 ± 0.13	12.62 ± 1.12
2x1	22.99 ± 0.96	0.27 ± 0.02	11.74 ± 0.09	12.57 ± 1.20
3x1	23.55 ± 1.35	0.28 ± 0.04	11.78 ± 0.07	12.32 ± 1.10
4x1	23.23 ± 0.93	0.28 ± 0.02	11.71 ± 0.12	12.33 ± 1.32
4x2	23.82 ± 0.98	0.27 ± 0.03	11.70 ± 0.09	12.68 ± 1.07
4x3	23.60 ± 1.43	0.27 ± 0.02	11.67 ± 0.07	12.61 ± 1.41
4x4	23.55 ± 1.37	0.27 ± 0.02	11.72 ± 0.08	12.57 ± 0.99
4x5	23.25 ± 1.44	0.27 ± 0.02	11.69 ± 0.07	12.41 ± 1.28
4x6	23.43 ± 0.66	0.28 ± 0.02	11.78 ± 0.09	12.47 ± 1.08

Table 3: Classification error in percentage for the different Bump Boost implementations. For the Splice data set each implementation made 100, for MNIST 1000, for Forest 5000, and for Checkers 500 iterations. The java implementation could not be tested successfully with the forest cover data set, due to memory errors.

Java versus Numpy

We re-implemented the algorithm using Numpy. The original code of [BK] was written in Java. The main difference is, that we use 32-Bit floating point numbers, while with Java 64-Bit were used. As shown in the table above, the lack of precision does not influence the training or the prediction accuracy. More, the next plot shows how Numpy outperforms Java with increasing iterations. To the best of our knowledge, this is caused by the differing data types.

10.2 Results

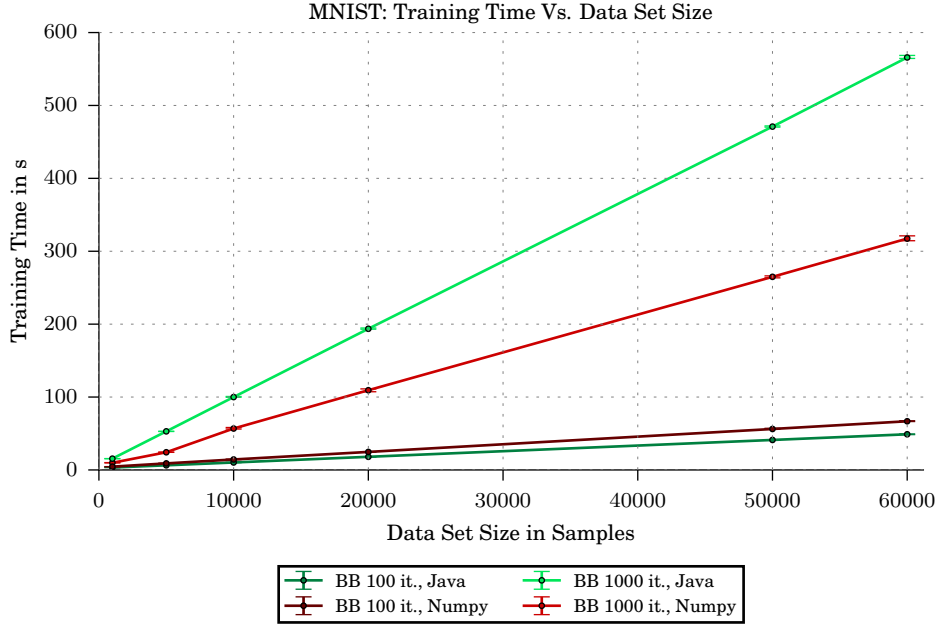


Figure 14: How the training times of the Java and Numpy implementation differ.

10.2.2 Bump Boost versus Competitors

Now we compare Bump Boost and Multi Bump Boost against the chosen competitors. In the first paragraph the classification results are examined, while in the second we show the efficiency of Bump Boost compared to the SVM solvers. The scaling of Spark MLlib is discussed in this chapter later on.

Classification Error

The next table reveals that Bump Boost and Multi Bump Boost are able to achieve similar classification rates as the state-of-the-art SVMs. As expected, the models on Spark, i.e. linear SVM and logistic regression, give bad results. The mantra that with a lot of data also simple solvers can do well, does not seem to be valid. For example on the Checkers data set, the algorithms were trained with up to 900.000 samples without an improvement on the classification error.

As stated, Bump Boost is able to reach the performance of current state-of-the-art SVM solvers. In the next paragraph, we will see that Bump Boost is able to reach them in a fraction of time without parallelization and using more data samples.

10.2 Results

Splice:	BB 100	MBB 100	LaSvm	LIBSVM	Lin. SVM	Log. Reg.
	23.72 ± 0.20	4.54 ± 0.88	9.79 ± 0.00	10.58 ± 0.00	15.72 ± 0.00	15.86 ± 1.32
MNIST:	BB 1000	MBB 1000	LaSvm	LIBSVM	Lin. SVM	Log. Reg.
	0.28 ± 0.02	0.20 ± 0.03	0.23 ± 0.02	0.28 ± 0.03	1.01 ± 0.12	1.31 ± 0.16
Forest:	BB 5000	MBB 5000	LaSvm	LIBSVM	Lin. SVM	Log. Reg.
	11.76 ± 0.09	10.43 ± 0.05	15.33 ± 0.29	14.91 ± 0.17	36.38 ± 0.00	36.38 ± 0.00
Checkers:	BB 2000	MBB 2000	LaSvm	LIBSVM	Lin. SVM	Log. Reg.
	3.80 ± 0.33	22.83 ± 1.49	4.79 ± 0.28	4.15 ± 0.23	47.34 ± 1.74	47.58 ± 1.76

Table 4: The classification error and standard deviation in percentage for the Bump Boost algorithms and competitors.

Efficiency

Especially for a data set like forest cover, where much data is needed for a good classification rate, Bump Boost performs well. This is because Bump Boost is able to train on all the data within the timeout limit of 1 hour. The SVM solvers could only handle 10.0000 samples on the MNIST, forest cover, and Checkers data set. For example on the Checkers data set, Bump Boost with 2000 iterations needed 33 seconds for 10.000 data points, achieving the same prediction error. LaSVM needed 1533 seconds and LIBSVM 1121 seconds. For the next larger set with 30.000 elements, the first run of LaSVM took nearly 5 hours. Bump Boost with 2000 iterations trained in 93 seconds.

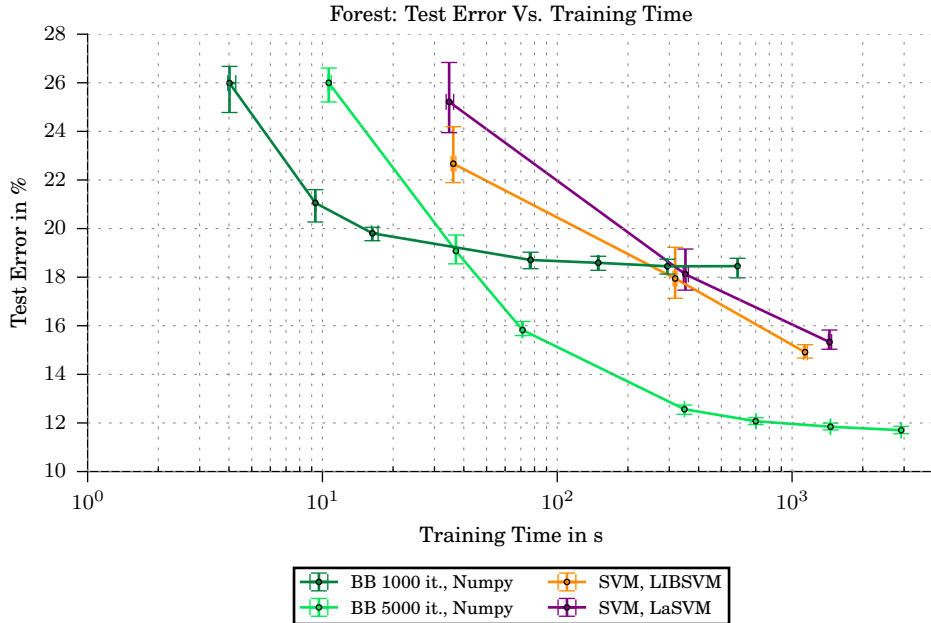


Figure 15: The training time/test error relation of the default Bump Boost implementation compared to the SVM solvers on the forest cover data set.

10.2 Results

In the figure 15 above we can see this well. Bump Boost with 5000 iterations reaches the same prediction error more than ten times faster as the SVM solvers. Moreover, Bump Boost handles the 200.000 data samples sets at the same time as the SVM solvers handles the 10.000 samples (The largest forest cover data sets are of size 200.000 and 400.000).

10.2.3 Scaling

We begin with a description of the scaling behavior on the smaller forest cover data set and passing on the synthetic data set Checkers with up to 10 million data points.

10.2 Results

Forest

In section 3.2.2, we already mentioned the slowdown characteristic, i.e. due to the overhead, more parallel instances perform worse than their sequential counterpart. Figure 16 shows this phenomena well. The parallelized versions of Bump Boost need around 20.000 data points to catch up with the default implementation. This overhead relation is even better visible, when considering the version with one “parallel” instance. The only difference to the default one is, that invocations for usually parallelized operations are issued over the network. In this case, up to 50.000 data samples are needed until the overhead is nearly negligible.

Another noteworthy insight of this plot is the better speedup, if the parallel instances execute on different hosts rather than on one. This is not as distinct in case of the Checkers data set, presented in the next paragraph. The main difference between these data sets is the higher dimensionality of the forest cover set, thus we suspect cache congestion as cause.

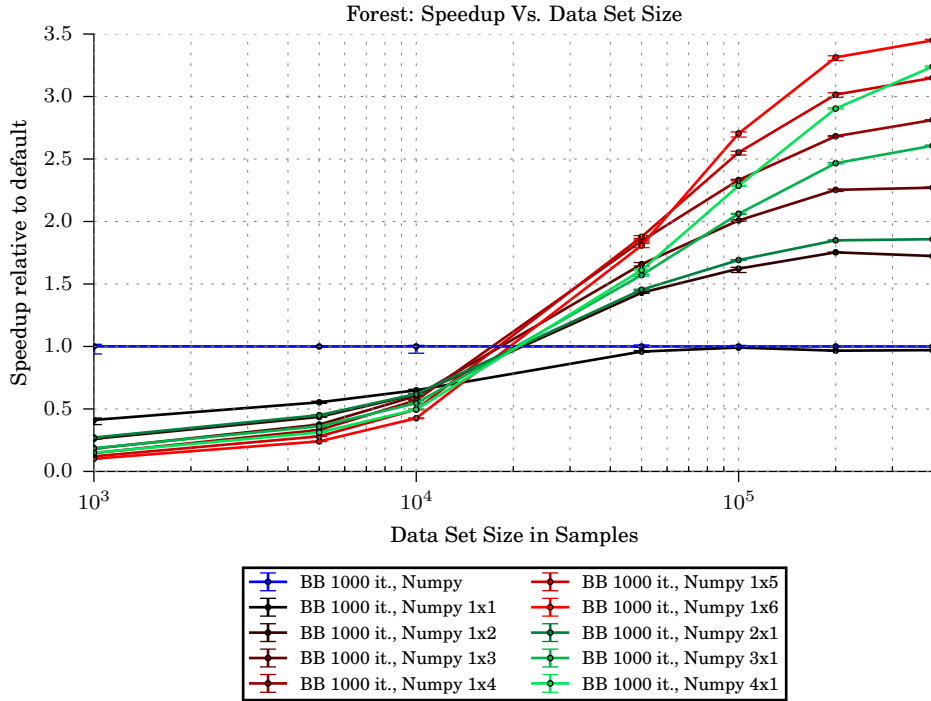


Figure 16: The speedup with increasing data set sizes of various Bump Boost implementations on the forest data set.

Checkers

Bump Boost already showed a good parallelization degree with the forest cover data set. But as we noticed, the more data the better Bump Boost scales. This can be seen in the following plots.

10.2 Results

Given the same Bump Boost implementations, the next plot shows a similar speedup development with similar data set sizes as for the forest cover data set. With more data, f.e. 10 million samples, 4 instances nearly reach a perfect speedup. Some implementations have a speedup above the theoretical limit given this data set size. The difference is quite small and we do not suspect Bump Boost to scale even better than theoretically justified. On the contrary, we assume that some basic system service such as HDFS has biased the measurement of the default Bump Boost version increasing the speedup of the other versions.

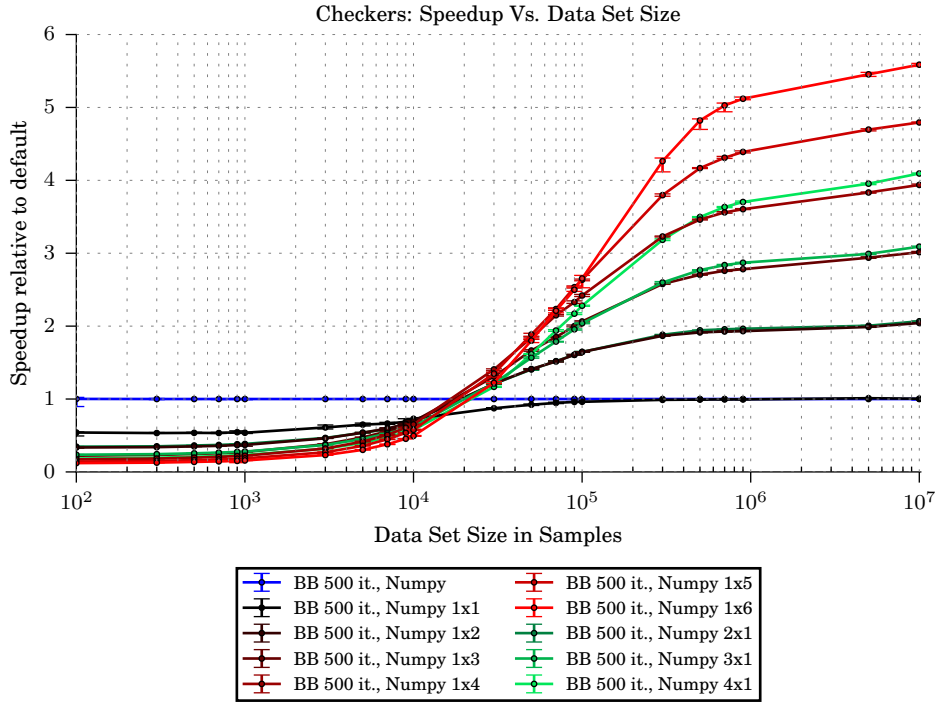


Figure 17: The speedup with increasing data set sizes of various Bump Boost implementations on the Checkers data set.

On the forest cover data set, we already mentioned the worse speedup on a single machine. Here again, we suspect cache congestion and the system overload throttling our parallelized versions. The reason for this suspect is, that on a single machine 5 and 6 instances are not able to scale nearly perfectly, in distributed manner up to 12 instances are able, as shown in the next plot.

The speedup of implementations with even more instances seems to not saturate yet. Therefore, we assume Bump Boost to scale even better with even more data.

A special case is the GPU implementation. It does not impose network overhead, thus catches up faster with the default implementation, and it

10.2 Results

scales fast. Nor does the speedup seem to saturate, even though it is already faster than the theoretically justified speedup 24 of our cluster with 24 cores. The main restriction of GPUs is the limited main memory compared to computers, but nowadays computers with several GPUs are already available and there is no theoretical or practical barrier to parallelize Bump Boost with several GPUs. In this case we expect a much higher speedup.

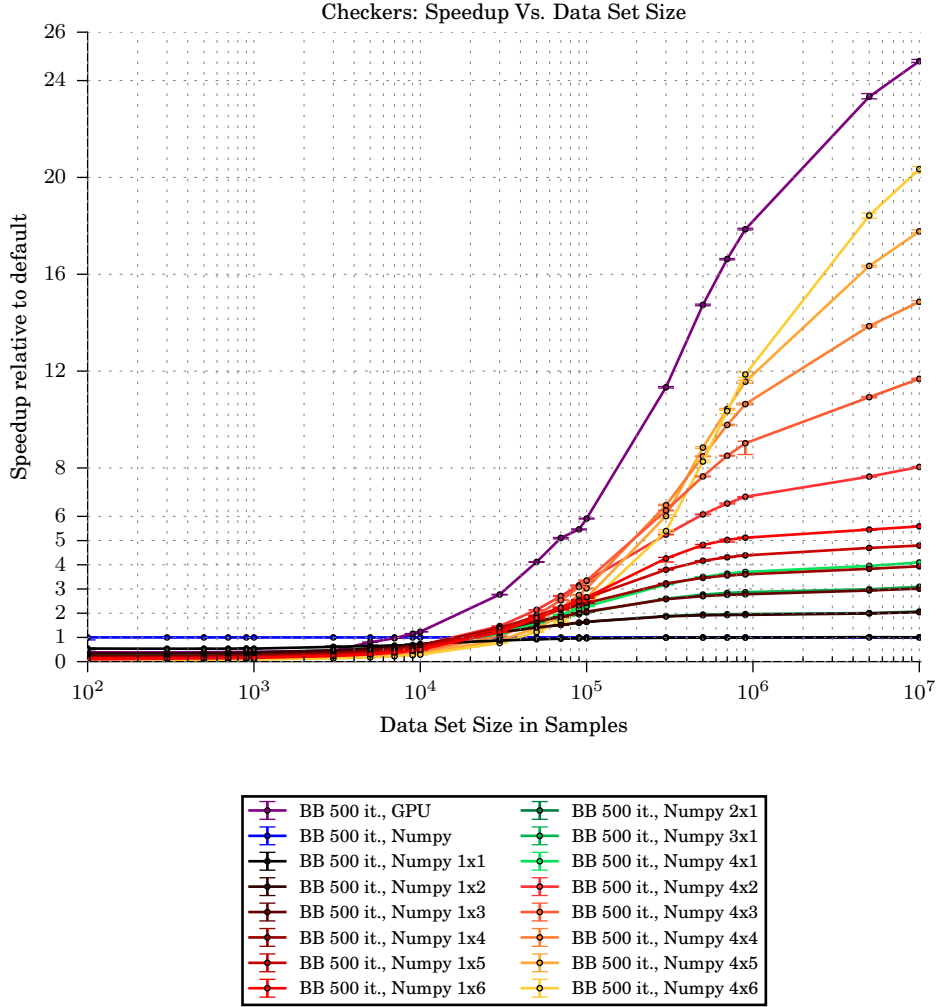


Figure 18: The speedup with increasing data set sizes of all Bump Boost implementations, except Spark, on the Checkers data set.

Amdahl's law and Gustafson-Barsis's law

Now we would like to revise the scaling theory (see 3.2.1). As we stated in 4.3.3, Bump Boost scales better the more data is trained on, as the parallel fraction of the algorithm increases. In figure 19, we can see this phenomenon.

10.2 Results

More, the scaling of Bump Boost seems much better represented by the law of Amdahl's, as by the law of Gustafson-Barsis's, because we can notice a decay of the speedup the more instances are used. Comparing the curves of Bump Boost trained with 10 million samples and Amdahl's with a parallel fraction of 99%, let us assume that Bump Boost has a very high parallelization fraction with this data size, i.e. higher than 99%.

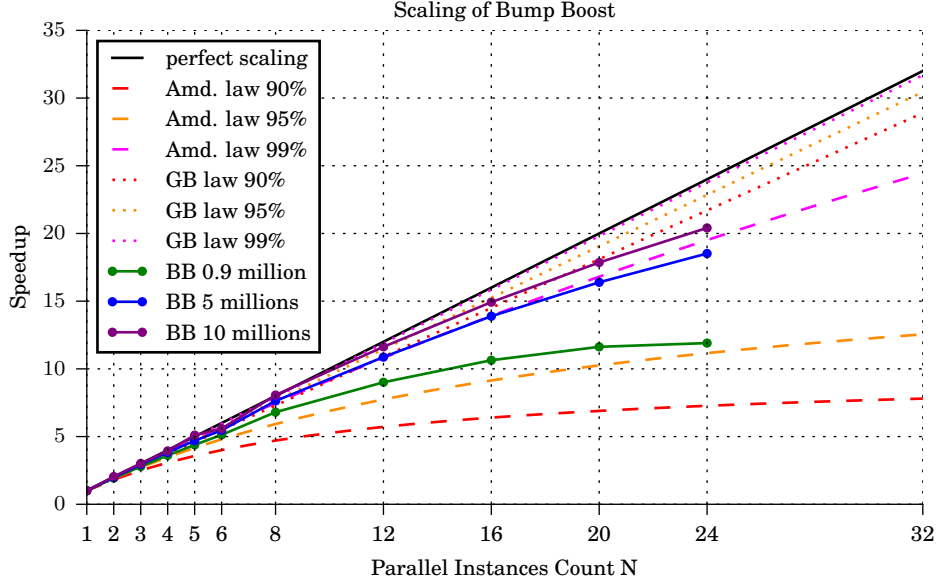


Figure 19: The speedup of Bump Boost with increasing parallel instances on the Checkers data set. “Amd. law” and “GB law” stand for Amdahl’s law and Gustafson-Barsis’s law. The number after “BB” states on how much data the Bump Boost instances have trained.

10.2.4 Spark

Until now, we excluded the Spark programs from our results. This was due to their either bad classification or training time performance. Hence, the emphasis of this paragraph is on Spark. First we give a look on the training time of the Bump Boost implementation and then we show the performance of MLlib.

Bump Boost Scaling

Due to its bad training times the Bump Boost Spark implementation was not mentioned up until. The following plot shows, that also on Spark Bump Boost scales linearly, but its factor is much worse than of our default implementation. We suspect the missing support for ordered sets and the need of joins for most linear algebra operations as main cause for this behavior. The initial setup overhead is shown by the training time with small sample

10.2 Results

sets. Again it is larger than the one of the default implementation. This is caused by the Spark framework.

Apache Spark was created to handle very large amounts of data. In our settings, data sizes do not exceed the size of some Gigabyte in LIBSVM-format. But even if Bump Boost scales on Spark linearly, the efficiency is so bad that we where not able to test them in appropriate time. A larger cluster might increase the performance of this Spark application, but according to us, given the small data sizes, this would not be appropriate.

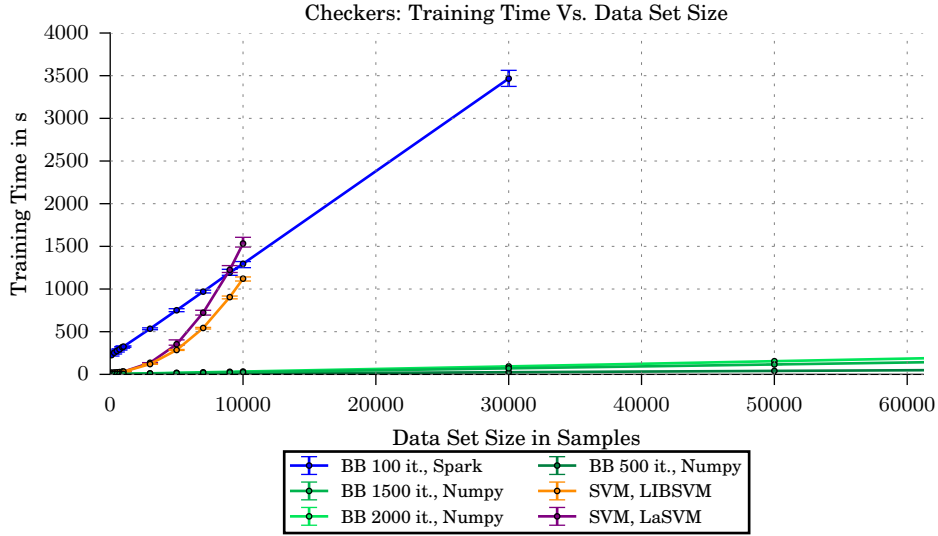


Figure 20: The training time with increasing data set sizes of Bump Boost on Spark compared to SVM Solvers and Bump Boost on Numpy.

MLlib Scaling

In table 4, we already saw that the MLlib algorithms have a high classification error, i.e. on complex data sets they do not learn anything. While SVMs do not scale and thus can not take advantage of all the data, this is not the case for MLlib, which does scale well, as shown in the next plot. But the programming model of Spark makes it hard to implement machine learning algorithm. The result is that only simple models are available and, f.e., the SVM implementation does use stochastic gradient descent instead of an established algorithms as SMO.

10.2 Results

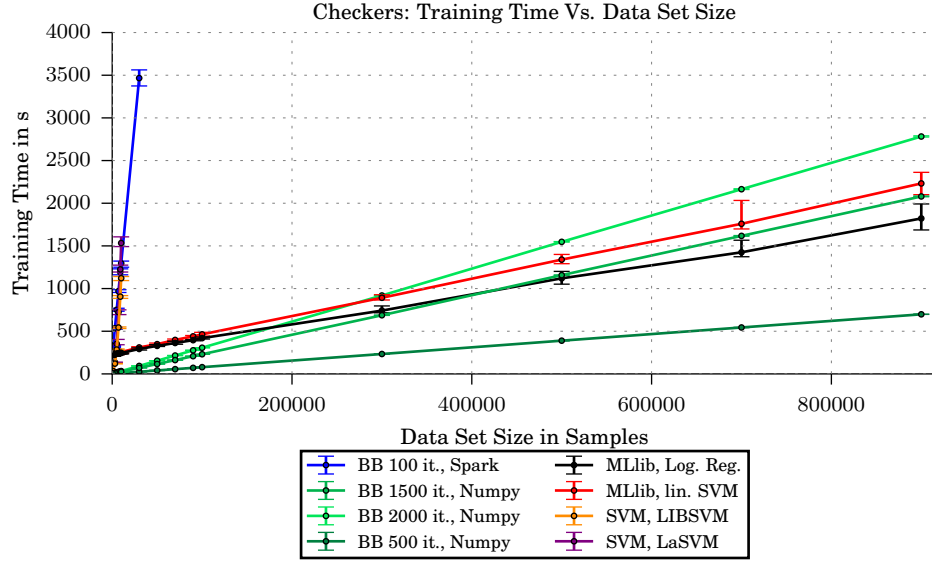


Figure 21: The training time on increasing data set sizes of Spark MLib compared to SVM Solvers and Bump Boost on Numpy and Spark.

The next plot shows clearly that the idea, more data, better performance, does not hold for complicated data sets, if the model is too simple.

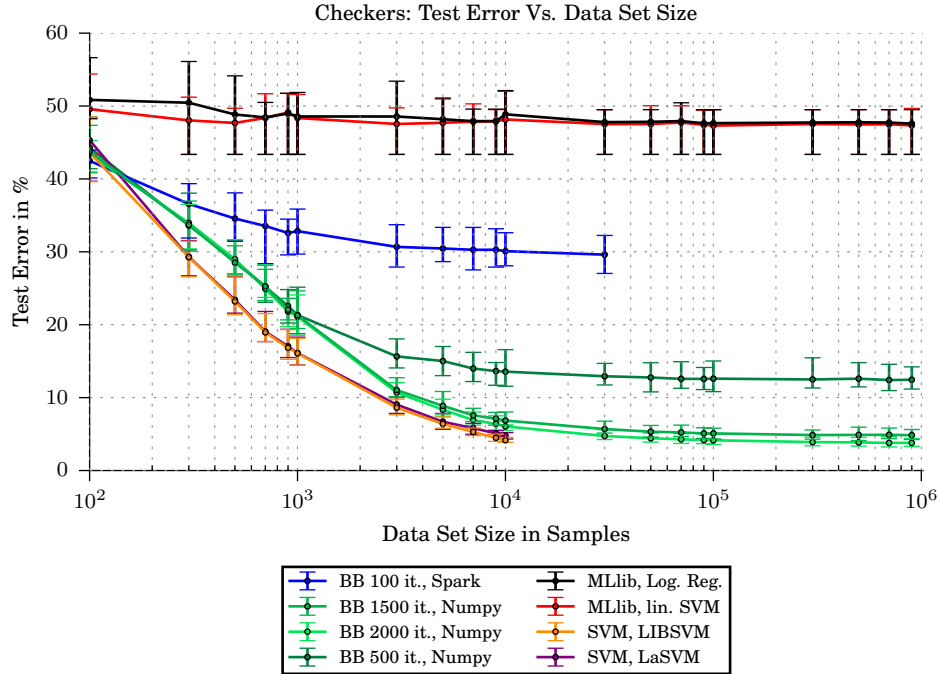


Figure 22: The classification error on increasing data set sizes of Spark MLib compared to SVM Solvers and Bump Boost on Numpy and on Spark.

11 Conclusion and Perspective

In this thesis, we have shown theoretically and empirically that Bump Boost and Multi Bump Boost are able to scale nearly perfectly with no loss of accuracy. Furthermore, we examined the suitability of Big Data frameworks for our tasks. With a disappointing result. While for Apache Spark the implementation was rather easy, but the result lacked the efficiency, we were not able to implement Bump Boost nor Multi Bump Boost using Apache Flink.

In the following, we summarize the results of this thesis and in the end we give an outlook to future questions and estimates.

11.1 Conclusion

Reminding the initial objectives of this thesis (see 1.1), we can state that the first objective was reached to our full satisfaction. Unfortunately, the second one revealed useful insights, but not the desired results. In more detail:

Scalability: We were able to extend the Bump Boost and Multi Bump Boost algorithms with parallel versions. Whereas the sequential algorithms need linear asymptotic run time, the parallel versions theoretically scale logarithmic with the sample count. The parallelized algorithms calculate the exact same results as the sequential one, proven theoretically and empirically.

We have shown empirically, that Bump Boost is able to handle up to several millions of data points, and there is no obvious barrier that Bump Boost should not be able to process even more data in reasonable time. With increasing data set sizes, Bump Boost scales better and better. We have shown that with 10 million samples Bump Boost can reach with up to 12 parallel computing instances perfect speedup.

In addition, we described how Bump Boost is able to scale by using different instances on a multi core machine as it is able to scale with instances spread on CPUs across a cluster of computers. Our GPU implementations showed an especially good speedup with increasing data, combined with less overhead imposed due to the missing network communication.

Summarized, Bump Boost is able to scale linearly with increasing data sizes and Bump Boost is able to scale, given enough data, nearly perfectly with increasing computing resources.

Even if the scaling behavior is less favorable for Multi Bump Boost, due to a larger computing and overhead factor, we cannot name any

11.2 Perspective

obvious reason why Multi Bump Boost should not scale asymptotically as well as Bump Boost.

Big Data frameworks: With the best of our knowledge and effort, we were not able to meet the goals of the second objective. The first challenge was the missing support for linear algebra operations. We solved the problem by implementing, inspired by Apache Spark MLlib, the needed operations.

On Apache Spark we implemented Bump Boost and Multi Bump Boost. Unfortunately, even if we showed empirically that the training time scales linearly with increased data sizes, the linear factor is so big that a practical use of the solution is not justified. According to us the biggest problem lies in the nature of the framework, i.e. the not ordered data sets impose lots of unnecessary join operations, resulting in too much overhead.

In contrast, we were not able to implement Bump Boost or Multi Bump Boost on Apache Flink. After long and tedious workarounds, we could not finish the program due to missing nested loop support, which is necessary to process Bump Boost and Multi Bump Boost. A detailed justification is given in 7.4.2. In the next sub chapter, solution proposals are given.

11.2 Perspective

For Bump Boost and Multi Bump Boost we name three interesting questions, worth further investigation:

New Kernels: In this thesis we only used Bump Boost and Multi Bump Boost with a Gaussian kernel. A lot of large data sets are encoded with text. An interesting question is, if it is possible to also achieve good classification performance with structured kernels like they exist for SVMs [Gär03] and if Bump Boost and Multi Bump Boost are able to be competitive to the state-of-the-art solutions.

Other data sets: Related to the first question, to further examine how the local kernel approach of Bump Boost and Multi Bump Boost perform on other data sets. Especially, is there a type of data which is well suited for this approach?

Multi Bump Boost and the Checkers data set: Due to time reasons, we were not able to find the actual cause for the bad performance of Multi Bump Boost on the Checkers data sets. All the instances of cost function we have seen were of convex nature and therefore well suited for gradient descent. This behavior is still an open question.

11.2 Perspective

Regarding our second objective, we have several solution proposals. While Apache Spark is only concerned with the first, all of them are valid for Apache Flink (see 7.4.2 for examples and a better understanding.):

Linear Algebra:

- A matrix or vector class, i.e. a data set of vectors, which abstracts efficiently the most important mathematical functions, would be a great enhancement. We created our own one, which seemed rather inefficient. According to us, this is mainly the cause of the unordered data sets. In linear algebra, strict ordering is common and numbers are a less flexible data type than strings. Two characteristics that the underlying system can take advantage of. We expect a great impact on the efficiency of the frameworks for machine learning tasks given such a feature.

Loops:

- Iterations on several “working” sets, i.e. no need to join all the variables in one set in order to do a loop.
- In the end, we were not able to implement Bump Boost or Multi Bump Boost on Flink due to the missing nested (rolled out) iterations support. In order to successfully implement a wide range of algorithms, such a feature is essential.
- The realization of the above proposal, i.e. nested iterations, can be challenging in models such as the data flow model of Flink, thus we propose a more realistic feature. An outer looping mechanism could be established, i.e. being able to restart an application without resubmitting it to the cluster or reloading the needed data sets.

Flexibility:

- In 7.4.2 we have shown the tediousness of Flink application. Especially the mantra that everything is a data set can be a problem when only variables are treated. Therefore a notion of distributed variables is needed. Besides the easier coding syntax and semantic, Flink would have less problems with the increased number of nodes as it could save the needless optimization attempts for data sets with one sample.
- For now, the structure of Flink data flows is static. It would be a neat feature, if it would be possible to control the data flow structure itself and not just the content of the data flows.

11.2 *Perspective*

For the future, Bump Boost and Multi Bump Boost can be a serious alternatives for tasks where, f.e., state-of-the art SVM solvers are not able to cope with the amount of data. Especially in such Big Data cases, Bump Boost and Multi Bump Boost benefit from their characteristic to scale the better the more data is processed. In addition, Bump Boost and Multi Bump Boost are out-of-the-box able to also do regression. As mentioned above, a text-enabled kernel could expand the application range, as lots of Big Data is text-based.

Big Data frameworks such as Apache Spark and Apache Flink come out of the data base field and for machine learning application they still impose serious challenges. According to us, the biggest impact is the set semantic, i.e. all data sets are unordered, which results in reduced application range and for linear algebra tasks, where lots of information is available on the data structures and operations, all this knowledge is ignored. We claim those frameworks need specialized implementations to cope with this problem and to be more attractive for mathematical and machine learning tasks.

A Computing Systems

In this section, the two used computing systems are described.

A.1 GPU-Server

The computer for GPU-enabled implementation is composed of:

Component:	Short	Long Description:
Mainboard:		MSI 970A-G43, AMD Socket AM3+, ATX, DDR3
CPU:	6x3.5 GHz AMD64	AMD FX-6300 Processor
GPU:	980 MHz, 2GB RAM	GeForce GTX 660, GK106, 2 GB DDR5
RAM:	8 GB	2 x 4GB DDR3 G.Skill RipJaws PC3-12800U CL9
Harddrive:	1 TB	Toshiba DT01ACA Series 1TB, SATA
Network:		Not used.

A.2 Cluster

The cluster is composed of 4 machines with the following configuration:

Component:	Short	Long Description:
Mainboard:		Dell PowerEdge M605
CPU:	6x2.2 GHz AMD64	AMD Opteron Processor 2427
GPU:		Not used.
RAM:	16 GB	16 GB DDR3
Harddrive:	500 GB	2xSATA RAID 0
Network:	Gigabit Eth.	Broadcom Corporation NetXtreme II BCM5708S

B Digital Content

This thesis includes an enclosed DVD. On it there are two directories.

All the code created for this thesis is stored in the directory named “repo”. After specifying various key directories, the makefile can be used to setup an environment for testing the code or to create the various data sets used. The main code is placed in the directory “python”, while the Apache Spark and Flink code is save in the directories “spark” and “flink”.

Due to the rapidly changing world wide web we saved each cited web page. These saves can be found in “/repo/master/docs/web_archive”.

The second, named “experiments”, contains a series of archive files containing the results of all experiments made. One can simply extract them and by using the “Experiment” class of the python code deserialize them.

For more details, please refer to the DVD and the source code.

C Copy of Bump Boost Paper

C Copy of Bump Boost Paper

The following nine pages are the original Bump Boost [BK] paper written by Mikio Braun and Nicole Krämer. It was never published, therefore we provide a copy with their consent.

BumpBoost – Fast and Large-Scale Learning for Non-Linear Kernels

Anonymous Author(s)

Affiliation

Address

email

Abstract

We introduce BumpBoost, an iterative kernel based learning method that scales multi-linearly in the number of observations, dimensions and iterations. BumpBoost (a) iteratively minimizes a quadratic loss using the gradient descent view of Boosting, (b) fits single kernel bumps in each iteration step and (c) locally adapts the kernel parameters for each bump. This results in a fast and large-scale algorithm where model selection is already included. Together with the local adaptivity of kernel parameters, this feature distinguishes BumpBoost from state-of-the-art large-scale solvers, which efficiently approximate the objective function, but rely on time-consuming cross-validation for model selection. We show on various benchmark data that BumpBoost outperforms other large-scale learning algorithms in terms of prediction accuracy versus training time.

1 Large Scale Learning Revisited

Large scale supervised learning has mostly been the domain of fast solvers to the support vector machine (SVM) problem. Over time, many specialized solvers for the SVM optimization problem have been invented, among the most popular are $\text{SVM}^{\text{light}}$ [1], LIBSVM [2], and LASVM [3]. Many algorithms have also been proposed to treat the case of linear SVMs (that is, with a linear kernel), including SVM^{perf} [4] (which also provides many other interesting features like optimizing complex performance criteria), or LIBOCAS [5]. In particular for the case of linear SVMs, stochastic gradient methods have proven to be very efficient, including Vowpal Wabbit [6] and the SGD implementations by L. Bottou [7].

Clearly, all state-of-the-art solvers depend on some model parameters (e.g. the regularization constant C for SVMs) which need to be selected, say via cross-validation. The problem of model selection and the computational overhead associated with it becomes more severe for non-linear kernels than for their linear counterparts, as these introduce at least one more additional parameter. For example, for the Gaussian kernel, we need to select the regularization constant C as well as the kernel width γ appropriately. Even if we are very frugal and only choose five different candidates for C and γ and restrict ourselves to five-fold cross-validation this means that we need to train 125 SVMs for model selection. In other words: Even when learning on a data set takes 1 second, we will have to wait for more than two minutes for the model selection.

While research initially focussed on finding faster algorithm for the original optimization problem, there has lately been some argument whether finding the exact global solution of the learning problem is really necessary in order to learn well. This discussion has also been active in the neural network field [8]. In the Pascal Large Scale Challenge [9], many submissions relied on only one iteration step of a general optimization method—and still performed competitively. Still, to our knowledge, it is not yet resolved how to extend these ideas to non-linear learning (that is, learning with a non-linear kernel) and how to make model selection efficient.

To overcome these challenges, we propose *BumpBoost*, an iterative learning method that (a) approximately minimizes a quadratic loss using gradient descent, (b) fits single kernel bumps in each iteration step, and (c) can also deal with multi-scale information by locally adapting the kernel parameters.

How BumpBoost Works and How it Differs from Existing Large Scale Approaches

We consider a supervised learning problem with n observations $(X_i, Y_i) \in \mathbb{R}^d \times \mathcal{Y}$, where $\mathcal{Y} = \{\pm 1\}$ (classification) or $\mathcal{Y} = \mathbb{R}$ (regression).

BumpBoost differs from existing large scale approaches in a number of ways.

Local kernel parameters. BumpBoost iteratively learns a kernel function of the form

$$f(x) = \sum_{i=1}^n \alpha_i k_{w_i}(X_i, x)$$

which differs from the function e.g. learned by an SVM in that each point can have its own kernel parameter. As we discuss below, this step allows to locally adapt kernels and to obtain faster learning rates. This is demonstrated on the *bumps* data set by Donoho and Johnstone [10] in Section 3.3. Furthermore, it is also possible to use multi-scale information by optimizing multivariate kernel parameters (see Section 3.4 for an evaluation on the *splice* data set [11]). Also note that in general, the expansion is sparse: Since one kernel function is added in each iteration step, the number of non-zero kernel coefficients α_i is as large as the number m of iterations. As a result, the computational demands for prediction are of similar magnitude than those for SVMs, and in particular much smaller than those of memory based methods like k -nearest-neighbor classification.

Loss function. Similar to other approaches, BumpBoost does not attempt to find the exact minimum of a regularized cost function. Instead it approximately minimizes the squared error via ℓ_2 -Boosting [12], that is by iteratively fitting residuals. The generic ℓ_2 -Boosting algorithm is displayed in Algorithm 1. For BumpBoost, the weak learner is a single kernel bump. Its center is chosen by random,

Algorithm 1 ℓ_2 -Boosting [12]

- 1: Initialize residuals $r \leftarrow Y$, iteration counter $m \leftarrow 1$, learned function $f(x) \leftarrow 0$.
 - 2: **for** $i = 1, \dots, m$ **do**
 - 3: Learn a weak learner h_m which fits $(X_1, r_1), \dots, (X_n, r_n)$.
 - 4: Add h_m to learned function: $f \leftarrow f + h_m$.
 - 5: Update the residuals: $r_i \leftarrow r_i - h_m(X_i)$ for $1 \leq i \leq n$.
 - 6: **end for**
-

with the probability proportional to the size of the residual. This heuristic does not fit all kinds of kernels, but is specialized to “bump-like” kernels like the Gaussian kernel or the rational quadratic kernel which have a maximum when the two points coincide. The kernel widths is chosen such that it minimizes the squared error to the residuals. (See Section 2 for more details.) Now, unlike existing iterative methods like stochastic gradient descent or sequential minimal optimization, a single iteration takes into account the whole data set. In other words, a single BumpBoost iteration adjusts one weight based on all training examples, whereas methods like stochastic gradient descent adjust all weights based on one training example.

Model selection. BumpBoost already includes model selection, as it automatically adapts the kernel parameters in each iteration locally. If the kernel has only one parameter (like the widths of a Gaussian kernel), BumpBoost selects the parameter from a list of candidates. If the kernel has more than one parameter (as, for example, a Gaussian with individual kernel widths), the parameter values are optimized by gradient descent for each point. Unlike existing approaches which boost single kernel bumps (for example, [13]), we put more effort on optimizing the kernel parameters than placing the kernels well. In summary, BumpBoost performs model selection where it is computationally cheap, instead of adding it as an afterthought to the learning process. We emphasize that the number m of iterations is not a regularization parameter. In all our experiments, we find that the test error decreases with the number of iterations (see also Section 3.1). So, the number m of iterations rather controls the time-budget that we have for learning.

Run-time. BumpBoost’s run time and memory requirements are linear in all parameters: The size of the data set n , the number of iterations m , the number of kernel parameters k , and the dimensionality of the space d . Since k and d are fixed for a data set, we get an algorithm which is linear in m and n . Empirically, it seems that m should also increase with n such that BumpBoost can make better use of more data, but this dependency is definitely sub-linear, such that the overall BumpBoost algorithm is sub-quadratic in n . In practice, in particular due to the included model selection, BumpBoost is very fast and outperforms existing SVM solvers in terms of achievable test error given a training time constrain (see Sections 3.2 for an experimental comparison). Finally, one can always add further iterations to an already learned BumpBoost model. This means that one can further refine a model if necessary, or inspect an intermediate solution without penalty.

2 The BumpBoost Algorithms

While the outline of the BumpBoost algorithm is given in the last section, the remaining question is how to fit a single kernel function *efficiently*. This amounts to selecting a base point x_c , selecting appropriate kernel parameters and then computing the weight α .

For selecting the kernel parameters we propose two alternatives. If the kernel has only one parameter, we simply test all candidates and use the one minimizing the squared error (this variant will simply be called “BumpBoost”). If there exist more than one real-valued parameter, we use a modified version of Rprop [14] to optimize again the squared error (“MultiBumpBoost”).

Here, we specify how to perform Step 3 in the above Algorithm 1.

Algorithm 2 Step 3: Learning a single kernel function

- 1: Choose a base point x_c using probability distribution (1).
 - 2: Select kernel parameters using one of the following approaches:
 - (i) If there are finitely many candidates, compute criterion \mathcal{C} (2) for each candidate and select the maximum.
 - (ii) If the kernel has finitely many real valued parameters, optimize the parameters using Rprop. For an example of the gradient, see Proposition 1.
 - 3: Set the weight α as in (2) for the selected kernel parameters.
-

Choosing the base point We draw an index c at random from $\{1, \dots, n\}$ with probability

$$p(i) = \frac{r_i^2}{\sum_{j=1}^n r_j^2}, \quad (1)$$

that is proportional to the squared residual at that point.

Choosing the weight If we know the base point and the kernel parameter w , then we can easily compute the weight α such that the squared error is minimized. The solution is given by the projection of the vector of residuals r to $k_w = (k_w(x_c, X_1), \dots, k_w(x_c, X_n))$:

$$\hat{\alpha} = \underset{\alpha}{\operatorname{argmin}} \|r - \alpha k_w\|^2 = \frac{k_w^\top r}{k_w^\top k_w} \quad (2)$$

Choosing the kernel parameter, finite version (BumpBoost) If we have a finite candidate set for w , we select the best kernel simply by minimizing the squared error. Since one kernel function has very limited complexity, this choice hardly leads to overfitting.

Note that we can further simplify the criterion as follows.

$$\begin{aligned} \|r - \hat{r}_w\|^2 &= \|r\|^2 - 2 \left\langle r, \frac{k_w k_w^\top r}{k_w^\top k_w} \right\rangle + \left\| \frac{k_w k_w^\top r}{k_w^\top k_w} \right\|^2 \\ &= \|r\|^2 - 2 \frac{(k_w^\top k_w)^2}{k_w^\top k_w} + \frac{(k_w^\top r)^2 k_w^\top k_w}{(k_w^\top k_w)^2} = \|r\|^2 - \frac{(k_w^\top r)^2}{k_w^\top k_w} =: \|r\|^2 - \mathcal{C}(w). \end{aligned} \quad (3)$$

Since $\|r\|^2$ does not change, we can simply maximize $\mathcal{C}(w) = (k_w^\top r)^2 / k_w^\top k_w$ to select the kernel.

Choosing the kernel parameters, finite dimensional version (MultiBumpBoost) Now we assume that the kernel has parameters $w \in \mathbb{R}^p$. Since the criterion (2) is in general not convex, we resort to Rprop [14] in order to optimize the criterion. Rprop is a gradient method which adapts its own step sizes per dimension based on sign changes in the gradient. If the sign is the same as in the previous iteration, the step size is increased by a factor, while the step size is halved if the sign has changed. We modify this Rprop algorithm by adding box constraints which are enforced strictly after each iteration.

We now discuss BumpBoost for a Gaussian kernel with individual weights per dimension. Note that the weight has to be positive and is more naturally expressed on an exponential scale: A change from 10^{-2} to 10^{-1} is similar to a change from 10^1 to 10^2 . Therefore, we re-parameterize the kernel as follows:

$$k_\tau(x, y) = \exp \left(- \sum_{j=1}^d 10^{-\tau_j} (x_j - y_j)^2 \right). \quad (4)$$

Next, we have to compute the derivative of the criterion (2) with respect to τ .

Proposition 1 *The gradient of the criterion $\mathcal{C}(\tau) = (k_\tau^\top r)^2 / k_\tau^\top k_\tau$ for the kernel (4) with respect to the kernel parameter vector τ is given by*

$$\begin{aligned} \frac{\partial \mathcal{C}(\tau)}{\partial \tau} &= \frac{\partial \mathcal{C}(\tau)}{\partial k_\tau} \frac{\partial k_\tau}{\partial \tau} \\ &= \frac{2k_\tau^\top r}{k_\tau^\top k_\tau} (r - \pi_{k_\tau} r) \left[k_\tau(x_i, \mu) (x_{ij} - \mu_j)^2 10^{-\tau_j} (\ln 10) \right]_{i=1, j=1}^{n, d} \end{aligned} \quad (5)$$

where $\pi_x y$ is the orthogonal projection of y on x .

Proof The gradient of $\mathcal{C}(\tau)$ with respect to k_τ is [15]

$$\begin{aligned} \frac{\partial}{\partial \tau} \frac{(k_\tau^\top r)^2}{k_\tau^\top k_\tau} &= \frac{2(k_\tau^\top r) r k_\tau^\top k_\tau - 2(k_\tau^\top r)^2 k_\tau}{(k_\tau^\top k_\tau)^2} \\ &= \frac{2(k_\tau^\top r)}{k_\tau^\top k_\tau} \left[r - \frac{k_\tau k_\tau^\top r}{k_\tau^\top k_\tau} \right] = \frac{2(k_\tau^\top r)}{k_\tau^\top k_\tau} (r - \pi_{k_\tau} r). \end{aligned}$$

Furthermore, the derivative $\partial k_\tau / \partial \tau$ is straightforward to compute, and the chain rule yields the desired result. \square

Note that $\mathcal{C}(\tau)$ is the composition of the mapping kernel parameter vector τ to the kernel vector k_τ , and mapping k_τ to the criterion score $(k_\tau^\top r)^2 / k_\tau^\top k_\tau$. Therefore, by the chain rule, the derivative requires one multiplication between a $n \times d$ matrix and a vector of length n . Also note that the second gradient is independent of the kernel, so only the gradient of the kernel vector by the kernel parameters needs to be computed for other kinds of kernels.

A few words on the implementation In order to speed up the algorithm, it is imperative to cache kernel evaluations as much as possible. If using a Gaussian kernel (with the same width for all dimensions), all squared distances between the base point x_c and the data points should be evaluated only once for each choice of x_c , and then reused to compute the Gaussian kernels for different widths. Likewise, for individual kernel widths, the matrix with entries $(x_{ij} - \mu_j)^2$ should be computed only once and then used both for the evaluation of the gradient and the kernel function.

Different stopping options exist for the Rprop algorithm, which can also have a huge impact on the overall run-time of the kernel widths selection. However, we have found that iterating for more than 30 to 100 steps does not significantly improve the results. In this paper we set the number Rprop iterations to 50.

Name	Subset	No. Features	Training	Test	URL
mnist [16]	1 vs. rest	784	60000	10000	http://bit.ly/mnist
forest-cover	1 vs. rest	54	581012	—	http://bit.ly/forestcover
ida [17]	flare-solar	9	666	400	http://bit.ly/ida-benchmark
	image	18	1300	1010	
	splice	60	1000	2175	
donoho [10]	bumps	1	2048	—	http://bit.ly/ftnonpar

Table 1: Overview of Benchmark Data Sets.

Method	ida	checkers	forest-cover	mnist	donoho-bumps
SVM (γ, C)	-3..1, 0..2	0.4, -2..2	-3..1, 0..2	-10..-5, -2..2	—
BumpBoost (w)	-1..3	-4..0	-2..2	5..10	-6..1
MultiBumpBoost (w)	-1..3	—	—	—	—
KRR (w, C)	-1..3, -6..2	—	—	—	—

Table 2: Parameter choices for the different algorithms. Given are the exponents to basis 10 (i.e. $-1..3$ means $10^{-1}..10^3$). For SVM type algorithms (LIBSVM [2], SVM^{light} [1], lasvm [3]), 5 candidate were chosen with logarithmic spacing, for BumpBoost 20. For MultiBumpBoost, the ranges are the box constraint passed to Rprop.

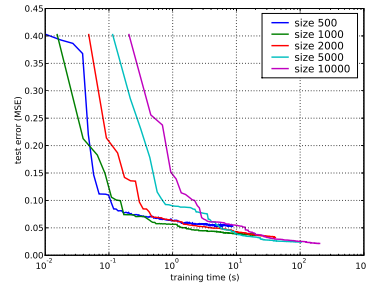
3 Experiments

In the following, we illustrate the convergence behavior of BumpBoost, compare it to state-of-the-art SVM algorithms on large scale data sets, and finally discuss how local kernel parameter adaptation and individual weights for each dimension can lead to improved prediction accuracy. The used data sets and parameters are summarized in Tables 1 and 2. Unless we mention it explicitly, we use BumpBoost with a one-dimensional kernel parameter.

3.1 Convergence of BumpBoost on different data sizes

The figure on the right shows the decrease of the mean squared error on a test set for BumpBoost with 1000 iterations and a Gaussian kernel on subsamples of the mnist data set, task “1 against the rest”. The increase of training time is quite moderate and the test error decreases as more and more data becomes available.

As we discuss in more detail below, it is precisely this combination of being able to deal with large data sizes which enables BumpBoost to deliver better prediction accuracy with less training time.



3.2 Comparison of Test Error vs. Training Time on Large Scale Data Sets

We now compare BumpBoost against the SVM implementations LIBSVM [2], SVM^{light} [1], and lasvm [3] using a Gaussian kernel. The kernel weight (specified by γ in $\exp(-\gamma\|x - y\|^2)$) is chosen together with the regularization constant C from 5 candidate values each using 5-fold cross-validation, resulting in 125 training runs to perform model selection plus an additional training run using the found parameters. We use the implementations provided by the respective authors of the methods. For LIBSVM, we use the Java version with an additional modification which restricts the number of iterations to 10000. The cache size is set to 128MB for all methods.

While this setup increases the run times of SVM by more than two orders of magnitude, we consider the comparison fair nonetheless since BumpBoost also already includes model selection. Choosing only 5 candidate values is also the least one would practically consider. Of course, in practice, one will likely resort to other heuristics in order to improve run time, for example by performing model selection on a subsample only. However, such heuristics are also thinkable for BumpBoost, therefore performing model selection on the whole data set gives a fair comparison between the methods.

	BumpBoost ₁₀₀₀		BumpBoost ₅₀₀₀		lasvm		LIBSVM	
training time in seconds								
1000	3.29±	0.19	16.27±	0.60	83.75±	2.96	80.62±	2.15
5000	16.35±	0.57	84.15±	2.37	1004.49±	133.19	3083.52±	337.84
10000	33.23±	0.21	185.46±	9.69	5845.37±	550.01	8400.09±	104.99
50000	191.34±	6.57	965.86±	28.19	217174.08±	10602.73	167487.60±	21217.73
100000	384.69±	10.33	2292.59±	122.03	—		—	
test error in percent								
1000	29.98±	1.14	30.46±	1.80	27.86±	1.44	25.50±	2.04
5000	24.79±	0.40	22.26±	0.67	20.43±	0.51	20.04±	0.55
10000	24.06±	0.49	20.15±	0.54	18.14±	0.40	16.78±	0.43
50000	22.87±	0.42	17.50±	0.21	(13.51±	0.47)	(17.94±	4.65)
100000	22.79±	0.24	16.93±	0.12	—		—	

Table 3: Results for the *forest-cover* data set. Shown are results over 10 random subsamples from the full forest cover data set. Results for 50,000 points for lasvm and LIBSVM are based on only 2 resamples.

	BumpBoost ₁₀₀		MultiBumpBoost ₁₀₀		KRR		LIBSVM	
training time in seconds								
flare-solar	0.22±	0.02	1.15± 0.05		4.91± 0.04		9.42± 0.30	
image	0.48±	0.02	2.73± 0.08		31.62± 0.20		48.12± 2.44	
splice	0.51±	0.01	7.65± 0.27		38.84± 3.68		105.91± 12.15	
test error in percent								
flare-solar	35.87±	1.84	35.89± 1.84		34.08± 1.71		32.83± 2.18	
image	7.29±	1.15	2.19± 0.58		2.70± 0.52		3.57± 0.72	
splice	23.22±	1.49	4.73± 0.54		11.15± 0.67		11.15± 0.60	

Table 4: Results for the *ida* data sets.

	BumpBoost ₁₀₀		BumpBoost ₁₀₀₀		SVM ^{light}		lasvm	
training time in seconds								
1-1000	1.89±	0.11	17.30±	1.53	381.96±	12.01	273.92±	39.09
1-5000	8.30±	0.14	82.56±	3.10	1447.77±	49.54	1675.80±	58.79
1-20000	34.77±	0.33	354.11±	21.02	—		(>50000)	
1-50000	96.12±	8.52	908.56±	28.17	—		—	
test error in percent								
1-1000	1.24±	0.19	1.04±	0.18	0.62±	0.11	0.51±	0.08
1-5000	0.85±	0.16	0.54±	0.04	0.32±	0.02	0.27±	0.03
1-20000	0.76±	0.12	0.41±	0.04	—		0.20±	NaN
1-50000	0.71±	0.11	0.38±	0.04	—		—	

Table 5: Results for the *mnist* data set, “1” against the rest.

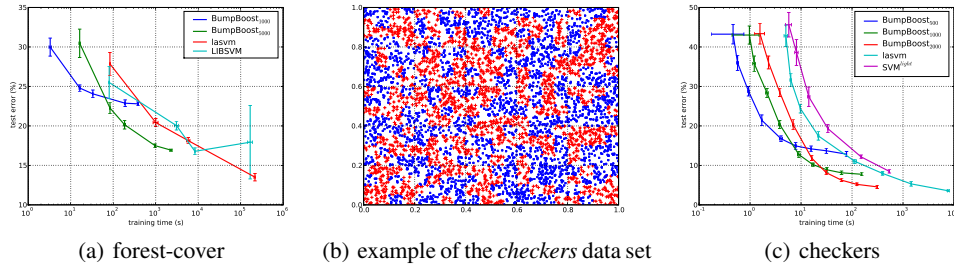


Figure 1: Training time versus test error for the *forest-cover* and the *checkers* data set.

This setup also reflects the fact that, in practice, model selection for SVM can actually be quite demanding computationally. Certain choices of parameters can lead to a very slowly converging solution (for example, large kernel width and only little regularization).

In the *forest-cover* data set, the task is to distinguish different tree types based on a number of parameters. We consider the task of distinguishing class 1 from the rest similar as in [3]. The data set consists of more than 500,000 data sets, but we consider random subsamples of up to 100,000 points here (with a test set of the same size). For preprocessing, we scale each feature such that the values lie between 0 and 1 and remove features which have a constant value on the training set.

Table 3 shows the training times for BumpBoost with 1000 and 5000 iterations and lasvm, and LIBSVM. One clearly sees that the SVM methods scale roughly quadratically in the number of training examples, whereas BumpBoost scales linearly. For the data set with 50,000 points, LIBSVM takes about 46 hours (again, including model selection), while lasvm takes more than 60 hours. Note that the results for 50,000 data sets are unreliable as they are based on only 2 resamples (instead of 10 as the other ones).

The experiments also show that BumpBoost performs slightly inferior if we compare the results for a given number of training points. However, if we consider the prediction accuracy we can obtain after a certain amount of training time, we see that BumpBoost₅₀₀₀ trained on 100,000 data points in about 2300 seconds leads to a test error of 16.93% which is on par with the test error obtained by training LIBSVM on 10,000 data points, which required more than 8000 seconds for training. Figure 1(a) plots the training time against the test error and shows clearly that BumpBoost outperforms the SVMs in terms of prediction accuracy after a given training time.

Next, we compare the methods on the larger data sets from the *ida* benchmarks (see Table 4). We also include BumpBoost with a Gaussian kernel with individual weights per input dimension here (called “MultiBumpBoost”). It is remarkable that MultiBumpBoost leads to much better results on the *image* and *splice* data set. We discuss this finding below in Section 3.4. For these experiments, we also use kernel ridge regression (KRR) with efficient computation of the leave-one-out error for the selection of the regularization constant. Although KRR scales cubically with the size of the training set, for the modest training set sizes it can compete with the SVM if one includes model selection.

Table 5 shows the results for the “1” against the rest task from the *mnist* data set. We use no preprocessing on this data set, and subsample data sets only for training and always using the whole test set. Again, BumpBoost delivers competitive performance in less training time (about 15 minutes compared to 28 minutes for lasvm).

One recurring finding was that if we fix the number of data points, BumpBoost performs inferior to SVMs. It seems that BumpBoost makes less effective use of available data. As we have already seen, BumpBoost is nevertheless able to deliver better prediction accuracy in less training time if there is abundant data. We wish to illustrate this point on a toy example. The data set *checkers* consist of a 30-by-30 grid with randomly chosen labels per field (see Figure 1(b)). This data set has Bayes risk zero, but to predict the class memberships well, one has to be able to cope with quite large data sets in order to see the actual structure. Figure 1(c) plots the test error against the training error and clearly shows that BumpBoost performs much better in terms of prediction accuracy vs. training time. The reason is that BumpBoost is able to deal with more data, giving it a statistical advantage to estimate the class memberships well.

3.3 Advantages of Local Kernel Width Adaption

In BumpBoost, model selection takes place for each individual kernel function placed around a data point. However, being able to locally adapt the widths can also lead to drastically better predictions. We consider the *bumps* data set created by Donoho and Johnstone [10] to discuss local adaptivity of wavelets. Figure 2(a) shows the resulting fit of BumpBoost with 100 iterations and the parameters shown in Table 2. Below in black, the logarithms of the kernel widths around each data point (weighted by the contribution of the kernels to the prediction of at that point) show how BumpBoost is able to adapt to the spikes in the data, leading to a much smoother fit in between. In comparison, the SVR fit shown in Figure 2(b) using a kernel width small enough to fit the spikes leads to much noisier predictions in the areas between the spikes.

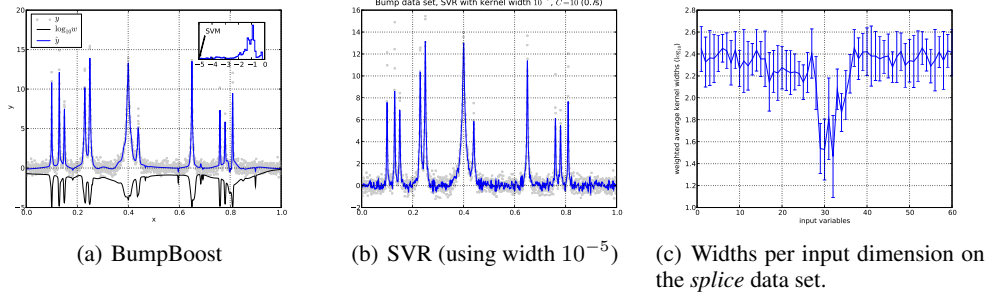


Figure 2: Locally adapting kernel widths leads to better fits. (a) shows the BumpBoost fit and the average kernel widths at each point. Note that the SVR fit (b) can use only one kernel width, and also a very small one. The histogram over the log-widths (inset in Figure (a)) clearly shows that the majority of the data points require a large scale than necessary to fit the spikes well.

3.4 Advantages of Individual Widths per Direction

In Section 3.2, we saw that MultiBumpBoost, the BumpBoost version using a Gaussian kernel with individual weights per input dimension (see Equation 4) leads to much better results on the *splice* data set from the *ida* benchmark.

In the *splice* data set, each input point encodes a piece of DNA of size 60 where the location of interest is centered at position 30. Figure 2(c) plots the logarithms of the widths per input dimensions kernel for 20 iterations of MultiBumpBoost. We see how BumpBoost focusses on locations on the DNA close to the position of interest. Biologically, it makes sense that this area is highly relevant for distinguishing the splice sites (although areas further away from the splice site also important to some degree, see [18]). Thus, the improved prediction accuracy of MultiBumpBoost results from its ability to focus on the input variables of interest, effectively removing the other ones from the input.

4 Summary: BumpBoost Outperforms SVMs for Large, Complex Data Sets

The two main contributions of BumpBoost are (a) its ability to solve large-scale problems with non-linear kernels *including* model selection and (b) the local adaptivity of kernel parameters using multi-scale information.

(a) *Large Scale* BumpBoost works very well on large, complex data sets. On such data sets, it can take advantage of the fact that its training time is *multi-linear* in each parameter: number of training examples, dimensionality of the data set, and number of iterations. BumpBoost can then make better use of larger data sets and deliver more accurate predictions faster than state-of-the-art SVM solvers. This turns it to an attractive alternative for large scale applications such as computer visions, where exponential kernels based on histogram distances have proven to work very well, and where a major challenge is the size of the data. Note furthermore that a parallel computation of the BumpBoost training can be achieved. BumpBoost is based on operations like vector addition and evaluation of rows of the kernel matrix, both of which can be readily computed in a distributed manner.

From a conceptual point of view, BumpBoost addresses the point of learning on a fixed time budget. Given a fixed amount of time, there is a real statistical advantage in using the largest possible data set. In fact, BumpBoost consistently delivers equal or even better prediction accuracy in terms of training time compared to SVMs.

(b) *Local Kernel Parameters* BumpBoost deals with multi-scale information without inducing additional computational overhead, leading to predictions whose smoothness adapts locally. Such behavior has traditionally been the domain of methods like wavelets. BumpBoost however also extends naturally to multivariate input data. Similar considerations hold for data with different scales for individual input variables. In such cases, MultiBumpBoost is able to use finer scales on input variables which are informative, also leading to significantly better prediction accuracy.

References

- [1] T. Joachims. Making Large-Scale SVM Learning Practical. In B. Schölkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods - Support Vector Learning*. MIT Press, 1999.
- [2] C.-C. Chang and C.-J. Lin. *LIBSVM: a library for support vector machines*, 2001. Software available at <http://www.csie.ntu.edu.tw/~cjlin/libsvm>.
- [3] A. Bordes, S. Ertekin, J. Weston, and L. Bottou. Fast kernel classifiers with online and active learning. *Journal of Machine Learning Research*, 6:1579–1619, September 2005.
- [4] T. Joachims. Training linear svms in linear time. In *Proceedings of the ACM Conference on Knowledge Discovery and Data Mining (KDD)*, pages 217–226. ACM, 2007.
- [5] V. Franc and S. Sonnenburg. Optimized cutting plane algorithm for large-scale risk minimization. *Journal of Machine Learning Research*, 2009. (accepted).
- [6] J. Langford, L. Li, and A. Strehl. Vowpal wabbit (fast online learning). <http://hunch.net/~vw>, 2007.
- [7] L. Bottou and Y. LeCun. On-line learning for very large datasets. *Applied Stochastic Models in Business and Industry*, 21(2):137–151, 2005.
- [8] G. Orr and K.-R. Müller. *Neural Networks: Tricks of the Trade*, volume 1524 of *Lecture Notes in Computer Science*. Springer, 1998.
- [9] S. Sonnenburg, V. Franc, E. Yomtov, and M. Sebag. The pascal large scale learning challenge. 2008.
- [10] D. L. Donoho and I. M. Johnstone. Ideal spatial adaptation by wavelet shrinkage. *Biometrika*, 81(3):425–455, 1994.
- [11] G. Rätsch, T. Onoda, and K.-R. Müller. Soft margins for AdaBoost. *Machine Learning*, 42(3):287–320, 2001.
- [12] P. Bühlmann and B. Yu. Boosting with the L2-Loss: Regression and Classification. *Journal of the American Statistical Association*, 98:324–339, 2003.
- [13] G. Rätsch, A. Demiriz, and K.P. Bennett. Sparse Regression Ensembles in Infinite and Finite Hypothesis Spaces. *Machine Learning*, 48(1):189–218, 2002.
- [14] M. Riedmiller and H. Braun. Rprop - a fast adaptive learning algorithm. In *Proceedings of the International Symposium on Computer and Information Science VII*, 1992.
- [15] N. Krämer and M.L. Braun. Kernelizing PLS, degrees of freedom, and efficient model selection. *Proceedings of the 24th international conference on Machine learning*, pages 441–448, 2007.
- [16] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
- [17] G. Rätsch, T. Onoda, and K.-R. Müller. Soft margins for AdaBoost. *Machine Learning*, 42(3):287–320, March 2001. also NeuroCOLT Technical Report NC-TR-1998-021.
- [18] S. Sonnenburg, A. Zien, P. Philips, and G. Rätsch. POIMs: positional oligomer importance matrices — understanding support vector machine based signal detectors. *Bioinformatics*, July 2008.

References

- [ABE⁺14] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, et al. The stratosphere platform for big data analytics. *The VLDB Journal*, pages 1–26, 2014.
- [ale15] Alex krizhevsky cuda-convnet homepage. <https://code.google.com/p/cuda-convnet/>, 06. January 2015.
- [Amd67] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.
- [AMD15] Amd developer - opencl dot webpage. <http://developer.amd.com/community/blog/2012/07/05/efficient-dot-product-implementation-using-persistent-threads/>, 08. January 2015.
- [apa15a] Apache flink incubator homepage. <http://flink.incubator.apache.org/>, 07. January 2015.
- [apa15b] Apache hadoop hdfs homepage. http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html, 07. January 2015.
- [apa15c] Apache hadoop homepage. <http://hadoop.apache.org/>, 07. January 2015.
- [apa15d] Apache spark homepage. <https://spark.apache.org/>, 08. January 2015.
- [apa15e] Apache yarn homepage. <http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html>, 07. January 2015.
- [Ber66] Arthur J Bernstein. Analysis of programs for parallel processing. *Electronic Computers, IEEE Transactions on*, (5):757–763, 1966.
- [BEWB05] Antoine Bordes, Seyda Ertekin, Jason Weston, and Léon Bottou. Fast kernel classifiers with online and active learning. *Journal of Machine Learning Research*, 6:1579–1619, September 2005.
- [BK] Mikio Braun and Nicole Krämer. Bumpboost - fast and large-scale learning for non-linear kernels. Unpublished. In the appendix.
- [BL07] James Bennett and Stan Lanning. The netflix prize. In *Proceedings of KDD cup and workshop*, volume 2007, page 35, 2007.

References

- [Bot10] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.
- [BY03] Peter Bühlmann and Bin Yu. Boosting with the l_2 loss: regression and classification. *Journal of the American Statistical Association*, 98(462):324–339, 2003.
- [C⁺11] Kate Crawford et al. Six provocations for big data. 2011.
- [CBB02] Ronan Collobert, Samy Bengio, and Yoshua Bengio. A parallel mixture of svms for very large scale problems. *Neural computation*, 14(5):1105–1114, 2002.
- [CKL⁺07] Cheng Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. *Advances in neural information processing systems*, 19:281, 2007.
- [CL11] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at <http://www.csie.ntu.edu.tw/~cjlin/libsvm>.
- [CV95] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.
- [DCM⁺12] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pages 1223–1231, 2012.
- [DDS⁺09] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.
- [DG05] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04*, pages 137–150, 2005.
- [DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
- [DHS99] Richard O Duda, Peter E Hart, and David G Stork. *Pattern classification*. John Wiley & Sons, 1999.
- [DJKP95] David L Donoho, Iain M Johnstone, Gérard Kerkycharian, and Dominique Picard. Wavelet shrinkage: asymptopia? *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 301–369, 1995.

References

- [dro15] Dropbox uses amazon s3 webpage. <http://www.makeuseof.com/tag/dropbox-review-invites-and-7-questions-with-the-founder/>, 10. Februar 2015.
- [EBC⁺10] Dumitru Erhan, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, Pascal Vincent, and Samy Bengio. Why does unsupervised pre-training help deep learning? *The Journal of Machine Learning Research*, 11:625–660, 2010.
- [FCL05] Rong-En Fan, Pai-Hsuen Chen, and Chih-Jen Lin. Working set selection using second order information for training support vector machines. *The Journal of Machine Learning Research*, 6:1889–1918, 2005.
- [fli15a] Apache flink als article. <http://data-artisans.com/computing-recommendations-with-flink.html>, 12. February 2015.
- [fli15b] Apache flink als code. <https://github.com/tillrohrmann/flink-perf/blob/ALSJoinBlockingUnified/flink-jobs/src/main/scala/com/github/projectflink/als/ALSJoinBlocking.scala>, 12. February 2015.
- [fli15c] Apache flink doc.: Iterations webpage. <http://flink.apache.org/docs/0.8/iterations.html>, 26. January 2015.
- [fli15d] Apache flink doc.: Programming guide. http://flink.apache.org/docs/0.8/programming_guide.html, 26. January 2015.
- [fli15e] Apache flink mailing list: Bug 1. <http://apache-flink-incubator-user-mailing-list-archive.2336050.n4.nabble.com/The-given-strategy-does-not-work-on-two-inputs-td403.html>, 26. January 2015.
- [fli15f] Apache flink mailing list: Bug 2. <http://apache-flink-incubator-user-mailing-list-archive.2336050.n4.nabble.com/No-Nested-Iterations-And-where-is-the-Nested-Iteration-td213.html>, 26. January 2015.
- [fli15g] Apache flink mailing list: Bug 3. <http://apache-flink-incubator-user-mailing-list-archive.2336050.n4.nabble.com/Class-not-found-exception-in-user-defined-open-function-without-open-function-td558.html>, 26. January 2015.
- [fli15h] Apache flink mailing list: Bug 4. <http://apache-flink-incubator-user-mailing-list-archive.2336050.n4.nabble.com/It-is-currently-not-supported-to-union-between-dynamic-and-static-path-in-an-iteration-td540.html>, 26. January 2015.

References

- [fli15i] Apache flink mailing list: Bug 5. <http://apache-flink-incubator-user-mailing-list-archive.2336050.n4.nabble.com/Illegal-State-in-Bulk-Iteration-td492.html>, 26. January 2015.
- [fli15j] Apache flink mailing list: No nested iterations. <http://apache-flink-incubator-user-mailing-list-archive.2336050.n4.nabble.com/java-lang-IllegalStateException-This-stub-is-not-part-of-an-iteration-step-function-td603.html>, 26. January 2015.
- [Fly66] Michael Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.
- [for15] Forest cover data set webpage. <https://archive.ics.uci.edu/ml/datasets/Coverttype>, 05. Januray 2015.
- [FS95] Yoav Freund and Robert E Schapire. A desicion-theoretic generalization of on-line learning and an application to boosting. In *Computational learning theory*, pages 23–37. Springer, 1995.
- [FS09] Vojtěch Franc and Sören Sonnenburg. Optimized cutting plane algorithm for large-scale risk minimization. *The Journal of Machine Learning Research*, 10:2157–2192, 2009.
- [Gär03] Thomas Gärtner. A survey of kernels for structured data. *ACM SIGKDD Explorations Newsletter*, 5(1):49–58, 2003.
- [Geb11] Fayez Gebali. *Algorithms and parallel computing*, volume 84. John Wiley & Sons, 2011.
- [GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.
- [gnu15] Gnu make homepage. <http://www.gnu.org/software/make/>, 05. January 2015.
- [goo15] Google flu trends hompage. <http://www.google.org/flutrends/about/how.html>, 11. February 2015.
- [Gus88] John L Gustafson. Reevaluating amdahl’s law. *Communications of the ACM*, 31(5):532–533, 1988.
- [had15] Yahoo hadoop tutorial webpage. <https://developer.yahoo.com/hadoop/tutorial/module1.html>, 07. January 2015.
- [Hil90] Mark D Hill. What is scalability? *ACM SIGARCH Computer Architecture News*, 18(4):18–21, 1990.
- [HSK⁺12] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

References

- [Ipe10] Panagiotis G Ipeirotis. Analyzing the amazon mechanical turk marketplace. *XRDS: Crossroads, The ACM Magazine for Students*, 17(2):16–21, 2010.
- [jbl15] Jblas homepage. <http://mikiobraun.github.io/jblas/>, 05. January 2015.
- [KBV09] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, (8):30–37, 2009.
- [Kea98] Michael Kearns. Efficient noise-tolerant learning from statistical queries. *Journal of the ACM (JACM)*, 45(6):983–1006, 1998.
- [KFLQ] Supun Kamburugamuve, Geoffrey Fox, David Leake, and Judy Qiu. Survey of apache big data stack.
- [KP] Ron Kohavi and Foster Provost. Glossary of terms. *Machine Learning*, 30(2-3):271–274.
- [KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.
- [KSW04] Jyrki Kivinen, Alexander J Smola, and Robert C Williamson. Online learning with kernels. *Signal Processing, IEEE Transactions on*, 52(8):2165–2176, 2004.
- [las15] Lasvm homepage. <http://leon.bottou.org/projects/lasvm>, 06. January 2015.
- [LAS14] Mu Li, David G Andersen, Alex J Smola, and Kai Yu. Communication efficient distributed machine learning with the parameter server. In *Advances in Neural Information Processing Systems*, pages 19–27, 2014.
- [LBBH98] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
- [lib15a] Libsvm homepage. <http://www.csie.ntu.edu.tw/~cjlin/libsvm/>, 06. January 2015.
- [lib15b] Libsvm readme file. <https://github.com/cjlin1/libsvm/blob/master/README>, 02. February 2015.
- [LK12] Jimmy Lin and Alek Kolcz. Large-scale machine learning at twitter. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 793–804. ACM, 2012.
- [M⁺75] Gordon E Moore et al. Progress in digital integrated electronics. *IEDM Tech. Digest*, 11, 1975.

References

- [mah15] Mahout features by engine webpage. <https://mahout.apache.org/users/basics/algorithms.html>, 25. January 2015.
- [mat15a] Matlab cuda support webpage. <http://de.mathworks.com/discovery/matlab-gpu.html>, 06. January 2015.
- [mat15b] Matlab opencl support webpage. http://de.mathworks.com/products/matlab/choosing_hardware.html#_Graphics_Processing_Unit_1, 06. January 2015.
- [mat15c] Matplotlib homepage. <http://matplotlib.org/>, 06. January 2015.
- [MCB⁺11] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, and Angela H Byers. Big data: The next frontier for innovation, competition, and productivity. 2011.
- [mll15] Apache mllib recommender system with als. <https://spark.apache.org/docs/latest/mllib-collaborative-filtering.html>, 10. Februar 2015.
- [MMR⁺01] K Muller, Sebastian Mika, Gunnar Ratsch, Koji Tsuda, and Bernhard Scholkopf. An introduction to kernel-based learning algorithms. *Neural Networks, IEEE Transactions on*, 12(2):181–201, 2001.
- [Mni09] Volodymyr Mnih. Cudamat: a cuda-based matrix class for python. *Department of Computer Science, University of Toronto, Tech. Rep. UTM TR*, 4, 2009.
- [MO99] Richard Maclin and David Opitz. Popular ensemble methods: An empirical study. *Journal of Artificial Intelligence Research*, 1999.
- [MSC13] Viktor Mayer-Schönberger and Kenneth Cukier. *Big data: A revolution that will transform how we live, work, and think*. Houghton Mifflin Harcourt, 2013.
- [num15] Numpy homepage. <http://www.numpy.org/>, 06. January 2015.
- [nvi15] Nvidia cuda homepage. http://www.nvidia.com/object/cuda_home_new.html, 06. January 2015.
- [ope15a] Opencl homepage. <https://www.khronos.org/opencl/>, 06. January 2015.
- [ope15b] Opencl registry homepage. <https://www.khronos.org/registry/cl/>, 08. January 2015.
- [OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in scala*. Artima Inc, 2008.
- [P⁺98] John Platt et al. Sequential minimal optimization: A fast algorithm for training support vector machines. 1998.

References

- [pol15a] Bbc the age of big data. <http://www.bbc.co.uk/programmes/b01rt4c7>, 11. February 2015.
- [pol15b] Predpol results homepage. <http://www.predpol.com/results/>, 11. February 2015.
- [Por06] David Porteous. The enabling environment for mobile banking in africa, 2006.
- [pyc15] Pycuda homepage. <http://mathematician.de/software/pycuda/>, 06. January 2015.
- [pyo15] Pyopencl homepage. <http://mathematician.de/software/pyopencl/>, 06. January 2015.
- [pyt15] Python global interpreter lock homepage. <https://wiki.python.org/moin/GlobalInterpreterLock>, 05. January 2015.
- [RB93] Martin Riedmiller and Heinrich Braun. A direct adaptive method for faster backpropagation learning: The rprop algorithm. In *Neural Networks, 1993., IEEE International Conference on*, pages 586–591. IEEE, 1993.
- [Roo00] Seyed H Roosta. *Parallel processing and parallel algorithms: theory and computation*. Springer Science & Business Media, 2000.
- [RR07] Ali Rahimi and Benjamin Recht. Random features for large-scale kernel machines. In *Advances in neural information processing systems*, pages 1177–1184, 2007.
- [RR13] Thomas Rauber and Gudula Rünger. *Parallel programming: For multicore and cluster systems*. Springer Science & Business, 2013.
- [sci15] Scipy homepage. <http://www.scipy.org/>, 05. January 2015.
- [SH12] Mudhakar Srivatsa and Mike Hicks. Deanononymizing mobility traces: Using social network as a side-channel. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 628–637. ACM, 2012.
- [spa15a] Spark history webpage. <https://spark.apache.org/news/index.html>, 08. January 2015.
- [spa15b] Spark mllib data types webpage. <https://spark.apache.org/docs/1.1.0/mllib-data-types.html#distributed-matrix>, 25. January 2015.
- [spa15c] Spark mllib homepage. <http://spark.apache.org/docs/1.1.1/mllib-guide.html>, 07. January 2015.
- [spa15d] Spark programming guide webpage. <http://spark.apache.org/docs/latest/programming-guide.html>, 08. January 2015.

References

- [spl15] Splice data set webpage. [https://archive.ics.uci.edu/ml/datasets/Molecular+Biology+\(Splice-junction+Gene+Sequences\)](https://archive.ics.uci.edu/ml/datasets/Molecular+Biology+(Splice-junction+Gene+Sequences)), 05. January 2015.
- [spo15] Spotifies recommender system, slides and pdf. <http://spark-summit.org/2014/talk/music-recommendations-at-scale-with-spark>, 10. Februar 2015.
- [SSS08] Shai Shalev-Shwartz and Nathan Srebro. Svm optimization: inverse dependence on training set size. In *Proceedings of the 25th international conference on Machine learning*, pages 928–935. ACM, 2008.
- [VJ01] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, volume 1, pages I–511. IEEE, 2001.
- [VN93] John Von Neumann. First draft of a report on the edvac. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.
- [ZCF⁺10] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

List of Tables

1	Data set depending parameters of the algorithms. Please see the text above for further explanations.	70
2	The classification error and standard deviation in percentage for the Bump Boost and Multi Bump Boost algorithms trained with the GPU implementation.	73
3	Classification error in percentage for the different Bump Boost implementations. For the Splice data set each implementation made 100, for MNIST 1000, for Forest 5000, and for Checkers 500 iterations. The java implementation could not be tested successfully with the forest cover data set, due to memory errors.	74
4	The classification error and standard deviation in percentage for the Bump Boost algorithms and competitors.	76

List of Figures

1	Examples how Amdahl's law evolves with increasing number of parallel instances.	10
2	Examples how Gustafson-Barsis's law evolves with increasing number of parallel instances.	11
3	The popular XOR-Problem. On the left side the two-dimensional space, in which no linear function could separate the red and black points. On the right side the feature space using the mapping function $\phi(x_1, x_2) = (1, 2x_1, 2x_2, 2x_1x_2, x_1^2, x_2^2)$, which transforms the two-dimensional input space into a six-dimensional one. In this new space the two classes are easily separable by a linear function. This example and the image are from [DHS99, page 264].	19
4	This image shows a two-class separation problem. The optimal hyper-plane lies exactly in the middle between the two nearest points of the two classes. In this case, the solid dots would represent the Support Vectors (see below). This example and the image are from [DHS99, page 262].	20
5	An example of how Bump Boost learns.	23
6	Dependency graph of the major variables in the Bump Boost and the Multi Bump Boost algorithm. Violet marks calculations. The style of the edges marks the delivered value: dotted is a scalar, dashed a vector, and solid a matrix. If an edge is colored red, it means the size of the value grows with $O(n)$ with n sample count.	29
7	This graph illustrates the calculations subdivision onto different workers for the Bump Boost algorithm. The node border colors orange to red denote different work entities, thus those values and computations were stored/executed on the according workers. Black denotes the master. The edge color green denotes a transfer between master entity and a worker entity. The other graph properties are described in the previous illustration 6.	32
8	The Apache Big Data stack. Apache Flink is missing and would be in the same place as Apache Spark. (Year 2013. From: [KFLQ])	43
9	UML Sequence Diagram with basic work flow for two iterations between the algorithm implementation, the UCC, and the LCC in the Bump Boost case. For further descriptions, see below.	51
10	An example tree of LCCs.	54
11	An example of a Checkers data set instance with 5000 points (From: [BK]).	68

List of Figures

12	How the training times of Bump Boost and Multi Bump Boost evolve with increasing data set size.	72
13	Plot on how the run times of Bump Boost and Multi Bump Boost are related to the test error.	73
14	How the training times of the Java and Numpy implementation differ.	75
15	The training time/test error relation of the default Bump Boost implementation compared to the SVM solvers on the forest cover data set.	76
16	The speedup with increasing data set sizes of various Bump Boost implementations on the forest data set.	78
17	The speedup with increasing data set sizes of various Bump Boost implementations on the Checkers data set.	79
18	The speedup with increasing data set sizes of all Bump Boost implementations, except Spark, on the Checkers data set. . .	80
19	The speedup of Bump Boost with increasing parallel instances on the Checkers data set. “Amd. law” and “GB law” stand for Amdahl’s law and Gustafson-Barsis’s law. The number after “BB” states on how much data the Bump Boost instances have trained.	81
20	The training time with increasing data set sizes of Bump Boost on Spark compared to SVM Solvers and Bump Boost on Numpy.	82
21	The training time on increasing data set sizes of Spark MLlib compared to SVM Solvers and Bump Boost on Numpy and Spark.	83
22	The classification error on increasing data set sizes of Spark MLlib compared to SVM Solvers and Bump Boost on Numpy and on Spark.	83