View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Juelich Shared Electronic Resources

Master Thesis

Design and Evaluation of an SVM Framework for Scientific Data Applications

Philipp Glock

Master Thesis DKE 15-23

Thesis submitted in partial fulfillment
of the requirements for the degree of Master of Science
of department at the Department of Knowledge
Engineering of the Maastricht University

Thesis Committee:

Dr. Gerasimos Spanakis, Maastricht University
Dr. ir. Kurt Driessens, Maastricht University
Prof. Dr.-Ing. Morris Riedel, Juelich Supercomputing Centre

Maastricht University
Faculty of Humanities and Sciences
Department of Knowledge Engineering
Master Artificial Intelligence

July 6, 2015

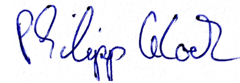
Declaration of Authorship

I hereby declare that the thesis submitted is my own unaided work. All direct or indirect sources used are acknowledged as references.

I am aware that the thesis in digital form can be examined for the use of unauthorized aid and in order to determine whether the thesis as a whole or parts incorporated in it may be deemed as plagiarism. For the comparison of my work with existing sources I agree that it shall be entered in a database where it shall also remain after examination, to enable comparison with future theses submitted. Further rights of reproduction and usage, however, are not granted here.

This paper was not previously presented to another examination board and has not been published.

Jülich, July 6, 2015



Philipp Glock

Abstract

Support vector machines (SVMs) are a popular classification method due to their good accuracy and broad usage domains in scientific applications. The computational complexity is between $O(n^2)$ and $O(n^3)$ for the number of n training samples. The scalability for larger data sets is therefore a problem of SVMs. With the increasing number of large data problems, this disadvantage becomes more and more significant. In order to overcome these scalability issues, this thesis designs and implements a parallel and scalable framework that realizes the cascade SVM approach including specific improvements. A fundamental speed up and increased scalability is gained by splitting up the data set into several sub sets that can be worked on in parallel. The framework is designed to run in modern High Performance Computing (HPC) environments, that provide the necessary massively parallel resources (e.g. large clusters with good node interconnects) to solve large data problems. The framework however also works on a simple computer for smaller problems if needed. To keep the interface usable for non-technical savvy domain scientists, Python is used.

The standard cascade SVM approach is improved with a standardized file format and parallel I/O is introduced that both improve the I/O performance, which besides computing is also often observed to be a bottleneck for large problems. In order to enable enhanced training speed up as well as a better accuracy further improvements such as distance filters and cross-feedback options are realized and evaluated. The resulting improved cascade SVM approach and parallel and scalable framework design is then evaluated on a real world remote sensing data set and compared to another parallel implementation called πSVM. The parallelization strategies of these two implementations are different whereby the cascade SVM is a data processing approach, πSVM follows primarily an algorithmic-driven approach.

Contents

1 Introduction 1

1.1 Motivation . . . 1
1.2 Infrastructure . . . 2
1.3 Objectives . . . 3
1.4 Structure . . . 3

2 Background 5

2.1 Cluster System . . . 5
2.2 Message Passing Interface . . . 6
2.3 Python . . . 7
2.4 Support Vector Machines . . . 11
2.5 Problem Statement . . . 15

3 Related Work 17

3.1 Support Vector Machine Speedup Techniques . . . 17
3.2 Support Vector Machine Tools . . . 18
3.3 Remote Sensing Applications . . . 19
3.4 Summary . . . 20

4 Cascade SVM 21

4.1 Conceptual Cascade Design . . . 21
4.2 Analysis of Scalable and Parallel Approaches . . . 22
4.3 Convergence Condition . . . 24
4.4 Summary . . . 24

5 Improving Cascade SVM 27

5.1 Architectural Design and Basic Implementation . . . 27
5.2 Parallel I/O . . . 32
5.3 Cross Feedback . . . 33
5.4 Distance Filter . . . 34
5.5 Summary . . . 36

6 Evaluation & Use Case **37**
6.1 Remote Sensing Data for Evaluations 37
6.2 Speedup by Improvements using Parallel I/O 38
6.3 Accuracy & Speedup 39
6.3.1 Benchmark on Data Set A 39
6.3.2 Benchmark on Data Set B^* 40
6.4 Cross Feedback Evaluation 42
6.5 Distance Filter 43
6.6 Comparison with πSVM 44
6.7 Summary 45

7 Conclusion **47**
7.1 Future Work 48

A Evaluation Tables **54**
A.1 Tables for Data Set A 54
A.2 Tables For Data Set B 54

Abbreviations **58**

List of Figures

2.1 Judge cluster system of the JSC 5
2.2 MPI Broadcast with four processes 6
2.3 MPI Scatter with four processes 7
2.4 MPI Gather with four processes 7
2.5 Machine Learning workflow supported by scikit-learn. 9
2.6 IPython parallel architecture. 11
2.7 Different possible decision boundaries. 12
2.8 Maximum margin decision boundary. The circle around data points indicated found support vectors. 12
2.9 Nonlinear classification problem. 13
2.10 Nonlinear problem transformed into the third dimension. 14
2.11 Nonlinear classification problem using a rbf kernel. 14

4.1 Binary CascadeSVM 22
4.2 MPI based cascade SVM with idle CPUs (gray) 23
4.3 Visualization of the Convergence Check 25

5.1 Framework architectural design 27
5.2 Basic HDF5 Structure 33
5.3 Early cross feedback in a binary cascade SVM 34
5.4 Filter used before Cross Feedback. 35

6.1 Input time for 1.000.000 samples with serial libSVM and parallel I/O using HDF5. 38
6.2 Training speedup and accuracy on data set A (small workload) . . . 40
6.3 Training speedup and accuracy (table A.5) on data set B^* 41
6.4 Cross feedback on layer 0 42
6.5 Cross feedback design implementation evaluated on different layers. 43
6.6 Training Speedup and Accuracy with Distance Filter. 44
6.7 πSVM speedup 44

List of Tables

3.1 Available SVM Tools 18

6.1 Training Time and Accuracy with multiple iterations on set B^*. . . . 41

A.1 Training Time on data set A 54
A.2 Accuracy on data set A 55
A.3 Training Time on Set B 55
A.4 Accuracy on Set B 56
A.5 Training and Testing Time on Set B* 57
A.6 Training and Testing Time with Feedback on Set B^*. 57

Listings

2.1 libsvm format 9
2.2 mpi4py send and receive 10

5.1 Scattering labels to all processes 28
5.2 Computation of the objective function. 30
5.3 Implemented RBF kernel using matrix operations 31

Chapter 1 Introduction

1.1 Motivation

Today the amount of data that is gathered and stored every day is much larger than in the past contributing to the more recent term 'big data'[34]. More and more people tweet or post status updates. They own smart phones and fitness trackers that gather data all day and cameras create images and videos that have a much higher resolution than in the past. The same is true with scientific devices ranging from small sensors in the field to large instruments such as those at CERN [13].

Researchers and companies want to harness this wealth of data and information. While there are different techniques to learn from data like regression or clustering techniques, this thesis focuses on data classification techniques using selected improvements of support vector machines (SVM)[14]. Some popular approaches for classification of large quantities of data are Mahout [4] for Hadoop [44], MLlib [22] for Spark [55] or Twister [18] and all of them are based on the map/reduce paradigm [17]. Unfortunately, the current version (0.9) of Mahout does not support SVMs and MLlib 1.1 provides only linear SVMs. While these approaches have a high scalability, they often have limitations in terms of stability, robustness and usability compared to serial implementations like scikit-learn [35], Weka [27] or R [38]. These tools are great for building a final model on the complete data, but preparing the data and evaluating different models is often complex for scientists and the important model selection process (e.g. cross-validation [23]) is often very time-consuming.

In this thesis a SVM framework is designed, that parallelizes the training and predicting of a SVM classifier to improve the performance and lower the time to solution. Because the implementation is deployed on a cluster system, frameworks like the message passing interface (MPI) [21] and a portable batch scheduler (PBS) [28] need to be incorporated. Since the target group for the framework is not restricted to programmers, the user interface should be kept as simple as possible. Therefore python was chosen as the primary programming language as it gains momentum in scientific communities, because of its simple usage.

1.2 Infrastructure

The Research Centre Jülich investigates key technologies for the 21st century. Different tools are needed for all kinds of experiments. Because these tools are often very specialized, the Research Centre often develops them itself. Apart from the experiment related software, there are many computational simulation and data problems that have to be solved.

The Jülich Super Computing Centre (JSC) provides the resources for these computations such as the supercomputing capacity and capability needed for a fast calculation. The JSC has several high performance clusters that can be used for compute- and/or data-intensive problems [1]. Besides hardware the institute investigates, explores, and develops different software solutions for parallel computing and therefore offers a comprehensive and powerful infrastructure for the research questions of this thesis.

The division 'Federated Systems and Data' (FSD) of the Jülich Super Computing Centre provides access to parallel and distributed systems that consist on a wide variety of different resources (e.g. HPC systems, parallel file system environments, high throughput computing resources, or clouds). This is done by implementing open standards and simplifying usage and administration of these services. Furthermore the division provides in particular, middle ware services, simple upload / download services, replication services, or data management know-how in general.

The research group on High Productivity Data Processing works on solutions to overcome problems and challenges of applications that specifically require so-called big data analytics solutions. This can be split into three categories, where parts of the thesis contributions fall into.

1. **Investigate Generic Data Methods:** How to overcome limitations of processing and analyzing large amounts of data (e.g. in-memory databases, data privacy methods and query processing) in order to be re-used in different scientific and engineering domains.
2. **Scalable Machine Learning Techniques:** Explore, develop and tune serial or parallel machine learning techniques, like classification and clustering, in order to enable solutions that work with large quantities of scientific data or high dimensional datasets.
3. **Smart Data Analytics Applications:** Find and develop solutions that are specifically applicable in real-world applications for general data analysis including statistical data analysis and feature selection and extraction methods

(e.g. for dimensionality reduction, sampling), data organization, e.g. data access - smart parallel I/O.

1.3 Objectives

In this thesis one support vector machine approach, a widely used and popular classification technique, is parallelized using a known approach. The intention is to develop a framework that is open and freely available. As the framework needs to be operated by domain-specific scientists that are not always technical-savvy, it is also important to have a good and user friendly interface. The framework is designed in such a way that it can therefore be easily used by domain scientists.

In addition improvements such as taking advantage of parallelization techniques are made to accomplish a better performance. The training time is decreased in comparison to a normal, typically referred to as serial SVM and the predicting is parallelized as well. This includes approaches where possible bottlenecks are investigated. For large data sets the I/O could be one of these bottlenecks, because it takes a long time and memory to read in the whole set. To sum up, the objective of the thesis is to design and implement a SVM-based framework that is able to overcome data analysis limitations when dealing with large data sets in scientific environments. This can be summarized into the following points:

- Introducing a standardized data format
- Enabling parallel I/O
- Improvements concerning the load imbalance problem
- Improvements of the feedback method

1.4 Structure

The thesis is structured as follows:

The first chapter gives an introduction to the field of data mining as well as a motivation. It also gives a short description of the infrastructure at the JSC and outlines thesis objectives. Chapter 2 explains the different techniques and frameworks that were used for this thesis and introduces our problem statement. The third chapter describes the related work regarding different techniques, tools as well as work performed in the domain-specific application domain. In chapter 4 the parallelization approach used in this thesis is explained. Chapter 5 reports a number of distinct improvements that were applied to the original approach. In the last chapter the frameworks performance is evaluated on a real world data set.

2 Chapter 2 Background

This chapter introduces some techniques and frameworks that are used in this thesis. At the end the problem statement for this thesis is given.

2.1 Cluster System

The framework is targeted to run on a cluster system [6], which is a collection of computes that are connected in some way, so that a distributed computation is possible. One of the systems deployed by the JSC is called Judge[1] and can be seen in Figure 2.1. It provides 206 compute nodes with a total of 2472 cores and 96gb memory per node.

Figure 2.1: Judge cluster system of the JSC

The job scheduling is done with a portable batch scheduler (PBS) [28]. The available parallelization frameworks are MPI [21] and openMP [15]. Map/Reduce frameworks like Hadoop [44] are not supported, because those frameworks do not take advantage of the extremely fast and powerful network interconnect of JUDGE in particular and cluster systems in general. At the same time these interconnects are the most costly parts of High Performance Computing (HPC) machines and as map-reduce frameworks do not need them those frameworks are to be deployed on High Throughput Computing (HTC) resources with a normal interconnect. Hence,

[1] www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUDGE/JUDGE_node.html

the difference of HPC systems to HTC systems is typically that the interconnect between CPUs/cores is specifically designed for messages (e.g. infiniband [43]).

2.2 Message Passing Interface

The Message Passing Interface (MPI) [21] is a well known standard, that describes the message passing for parallel computations on distributed systems. It provides a programming interface, containing several operations and their semantic.

A MPI application consists of multiple processes that are started in parallel at the beginning. All of them together work on a problem and pass data by sending messages from one process to another one. An advantage of this approach is that the processes can be started on different computers and the application is not limited to a single machine ,e.g. SPMD (Single Program Multiple Data) [6].

The communication can be differentiated into two types, the first one being point-to-point communication. A process sends a message to a specific destination by calling the *send* function and the process that gets the message calls the *recv (= receive)* function. This can be done either with blocking or non blocking calls. A blocking receive call waits until the message has completely arrived. A non blocking receive returns a *request object* that can be explicitly checked if the message has been received or not.

The second type is collective communication representing one of the most powerful approaches in MPI. All processes can be part of the application, but the number of processes can be specifically defined by using an MPI communicator that may be tuned to the communication structure of the application (e.g. cartesian communicator [48]). Because some patterns are often needed for parallel computation, MPI provides some predefined collective functions. As mentioned above all processes of the communicator are involved and there is one process that is initiating the call named as *root*.

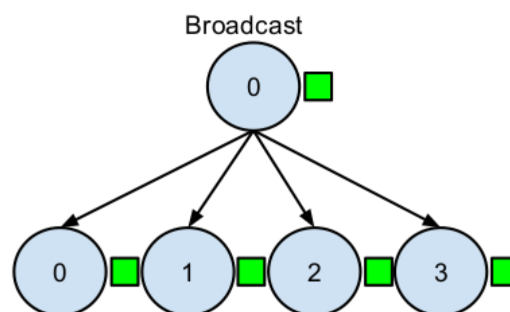


Figure 2.2: MPI Broadcast with four processes

A simple collective function is the *broadcast*, which can be seen in Figure 2.2. It sends the data from the *root* process to all other processes and saves it in the same variable. Like all other MPI functions *broadcast* expects that the memory for the data is already allocated.

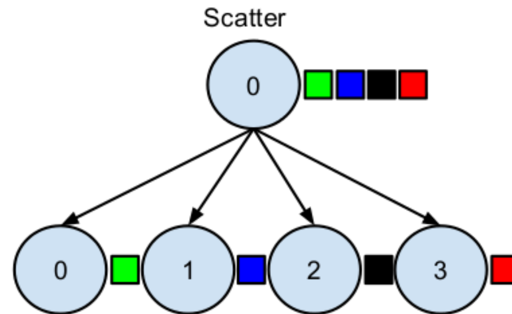


Figure 2.3: MPI Scatter with four processes

Figure 2.3 shows another collective function called *scatter*. While this function sends data to all processes of the communicator like a *broadcast*, the data is not the same for all processes. Instead the data of the *root* process is chunked into n (= number of processes) equally sized parts and distributed to the processes. The *scatterv* function is a variant of *scatter* that can send chunks of variable sizes.

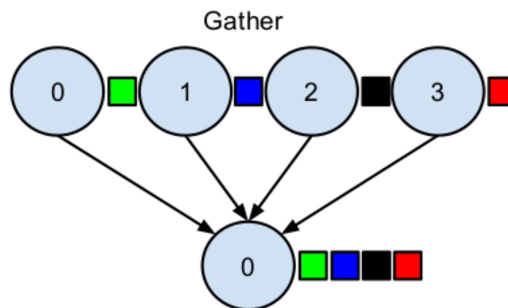


Figure 2.4: MPI Gather with four processes

The complement to *scatter* is the *gather* function shown in figure 2.4. All processes send their *data* to the *root* process which saves the data in an array. The *gatherv* function is again a variant that can send different sized chunks.

2.3 Python

The programming language Python [41], which is primarily used in this thesis, is increasingly used in scientific environments.

Python is a high-level, general-purpose language that emphasizes code readability and a clean and easy syntax. The most used implementation *CPython*[2] is interpreted at run time. Other implementations like *PyPy* [40] use a just-in-time compiler to improve the performance. It also features a large number of available modules, some of them being highlighted in the next part, because they are relevant in this thesis.

Although Python is a slow language (e.g. compared to the traditional C language [30]) it is very popular among scientists, because of its nice syntax. To enable Python for larger computations the performance had to be improved. The numpy [49] module focuses on array computation. It provides classes and functions to handle arrays in a fast way. Next to simple functions like the 'plus' or 'minus' operator numpy also contains more sophisticated ones like 'exp' or 'sin' that work on one or multi-dimensional arrays. The code is not written in Python but in C and only wrapped by Python. Most scientific modules are based on numpy to perform array calculations.

Another module is scikit-learn [35]. The module provides machine learning algorithms. It provides several data mining and analysis tools that are easy to use. The module is very popular because many basic machine learning tasks can be purely solved with it. This can be done because scikit-learn provides functions for the complete workflow of a machine learning task. Figure 2.5 shows the different steps that are needed during a machine learning problem and are provided by the scikit-learn module. It starts with simple data management by supporting popular file formats and sampling of data. In the preprocessing step normalization, like scaling of features, feature extraction and dimensionality reduction techniques are available. The module can handle classification tasks as well as clustering and regression. For the model selection popular methods like grid search and cross validation are provided. Because scikit-learn is compatible with the popular plotting library matplotlib [29] the results can even be visualized.

The main part of scikit-learn does not feature parallelism. This is done to be as independent and easy to use as possible. Only some parts like grid search can be used in parallel on a shared memory system. The performance of the different models may vary depending on the implementation.

For this thesis the SVM classifier is needed, which is implemented in the *SVC* class. It is not written in Python, but wraps the libSVM [11] library, which is designed in C++, and thus provides a good performance. It is thus the de-facto standard and wrapped by most SVM libraries. The *SVC* class can handle numpy arrays as well

[2] *https://github.com/python/cpython*

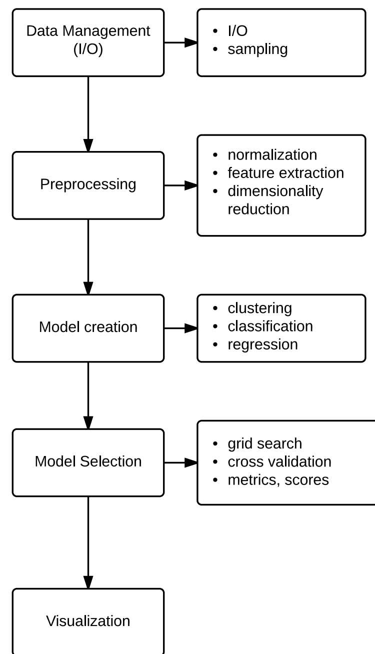


Figure 2.5: Machine Learning workflow supported by scikit-learn.

as sparse matrices which can be saved in the libSVM format as shown in listing 2.1.

```
<class_label> <feature_id>:value ... <feature_id>:value
<class_label> <feature_id>:value ... <feature_id>:value ...
```

Listing 2.1: libsvm format

Serial executions of Python and scikit-learn limits the available memory to one serial system and thus the amount of data that can be handled in an application is limited. While Python has many advantages, it does not have a good support for parallel execution from the start. This problem can be solved by using modules like mpi4py or IPython.

Mpi4py [16] is a module that wraps MPI, so that it can be used within Python. It provides all functions of MPI. To accomplish the performance of a normal MPI implementation numpy arrays have to be used, because they are based on C like arrays. If other objects than numpy arrays are used, the data is transferred by using pickle[3].

[3]https://docs.python.org/2/library/pickle.html

```python
from mpi4py import MPI
comm = MPI.COMM_WORLD rank = comm.Get_rank()

# using python objects (pickle)
if rank == 0:
     data = {"a": 1, "b": 2.5}
     comm.send(data, dest=1, tag=1)
elif rank == 1:
     data = comm.recv(source=0, tag=1)

# using numpy faster, pass explicit MPI datatypes
if rank == 0:
    data = numpy.array([1,2.5], dtype=float)
    comm.Send([data, MPI.DOUBLE], dest=1, tag=42)
elif rank == 1:
    data = numpy.empty(2, dtype=float)
    comm.Recv([data, MPI.DOUBLE], source=0, tag=42)
```

Listing 2.2: mpi4py send and receive

Listing 2.2 shows two small examples for a simple send call. Process 0 sends data to process 1, which saves it by calling the *recv* function. The first one uses Python objects and has a cleaner syntax as the data type can be omitted and no memory has to be allocated by the receiving process. Like mentioned above this includes a performance loss because Pythons pickle is used. The second part uses numpy arrays. The syntax is not as clean, because memory has to be allocated and the data is saved in an output parameter, but the performance is better. While the second example includes the data types as parameters, they are not necessary in this simple case. The data parameters are either just the variable, which is send or received, or a list with additional information. If a list is used, the list's order equals that of the standard MPI interface. More information on mpi4py can be found at the homepage[4].

Another often used framework is IPython [36]. It provides an architecture for interactive computing. This includes an interactive shell and a browser based notebook, that can handle code, text , graphics and mathematical expressions. The most important feature is that IPython supports a powerful architecture for parallel computing, which can also handle MPI. Unlike MPI which is in general used from the command shell, IPython's parallelism can be used from it's interactive shell.

Figure 2.6 shows an overview of the IPython architecture. The engine is a Python instance, that gets commands as well as Python objects over a network connection. While executing commands it is blocked. The hub is a process that keeps track of

[4]http://mpi4py.scipy.org/

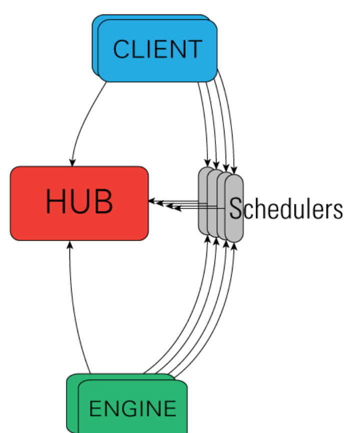


Figure 2.6: IPython parallel architecture. Source: http://ipython.org/ipython-doc/stable/parallel/parallel_intro.html

all engines, schedulers and clients. It also tracks the tasks and their results. Every command for an engines passes through a scheduler. The scheduler hides the fact that an engine block while performing a task and provides a asynchronous interface. The client is the primary object to connect to a cluster and execute tasks.

2.4 Support Vector Machines

Support vector machines (SVMs)[14] are one of the preferred classification methods, because tools are stable and widely available and it is one of the best out-of-the-box methods scientists can use that are not particularly trained in machine learning algorithms. They have a high accuracy, but their training time is quite long especially as the time is related to the number n of samples used in the given problem. A model can easily be described by the found support vectors. Figure 2.7 shows a binary classification problem. It can be seen that many different decision boundaries exist, that separate both classes. The gray band of each boundary is the distance to the closest data point. The larger the distance is the better the decision boundary. The gray band is referred to as the margin and the basic idea of SVMs. They are therefore known as maximum margin classifiers.

An SVM calculates the decision boundary with the maximum distance to the next data point for both classes. Figure 2.8 shows the decision boundary found by an SVM for the previous example.

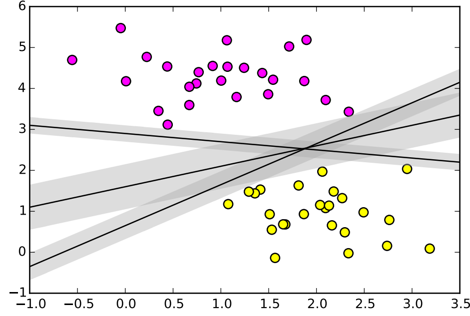


Figure 2.7: Different possible decision boundaries.

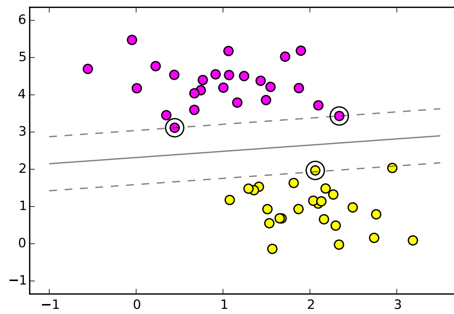


Figure 2.8: Maximum margin decision boundary. The circle around data points indicated found support vectors.

The decision boundary is a hyperplane and can be written as:

$$\vec{w} \cdot \vec{x} - \vec{b} = 0 \tag{2.1}$$

$\vec{w}$ is the normal vector to the hyperplane and $\frac{\vec{b}}{||\vec{w}||}$ is the distance between the origin and the hyperplane. For linear separable training data the two hyperplanes

$$\vec{w} \cdot \vec{x} - \vec{b} = 1 \tag{2.2}$$

and

$$\vec{w} \cdot \vec{x} - b = -1 \tag{2.3}$$

separate the data and have not points between them. The region between these hyperplanes is the margin and the data samples that are on one of the hyperplanes are called support vectors. The distance between both hyperplanes is $\frac{2}{||\vec{w}||}$. Maximizing the margin is therefore the same as minimizing $||\vec{w}||$ under the assumption that no point lies between the hyperplanes. Using some mathematical transformations and

lagrangian multipliers this yields the following function.

$$L = \frac{1}{2} \cdot ||\vec{w}||^2 - \sum \alpha_i (y_i(\vec{w} \cdot \vec{x_i} + \vec{b}) - 1) \quad (2.4)$$

After some more transformations the dual problem can be achieved.

$$L_D = \sum \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \vec{x_i} \vec{x_j} \quad (2.5)$$

By solving this problem a decision boundary like shown in figure 2.8 can be computed. The implementation used in this thesis solves the problem with the sequential minimal optimization (SMO) algorithm [37], which has a complexity between $O(n^2)$ and $O(n^3)$ for n samples.

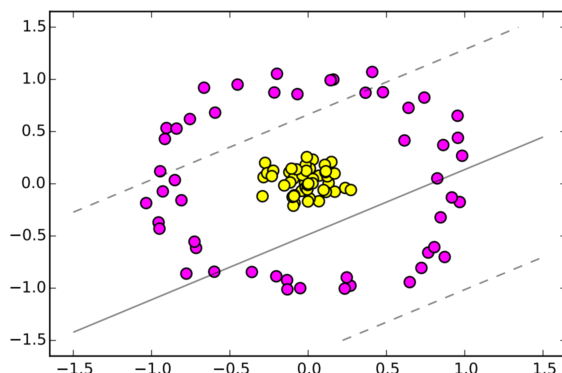


Figure 2.9: Nonlinear classification problem.

If the data is not linear separable as it is shown in Figure 2.9, the classes can not be separated by a linear decision boundary. The problem can be transformed to a problem which is linear separable by increasing the number of dimensions. Figure 2.10 shows the same classification problem with an additional third dimension. By introducing a third variable 'r' next to 'x' and 'y', the problem becomes linear separable. Transforming nonlinear problems into higher dimensions by hand is very expensive and SVMs were not popular because of that for quite a time.

This changed after the kernel trick was introduced. By using the kernel trick the problem does not need to be transformed into a higher dimension anymore. This is possible because only the product of two data points is needed, which can be seen in equation 2.5. Instead of projecting the data into a higher dimension and performing manually non-linear transformations of the data, so that it is linear separable, the dot product is replaced by a non linear kernel method that can be computed.

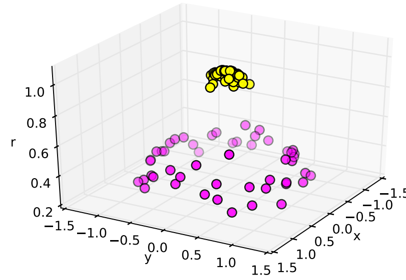


Figure 2.10: Nonlinear problem transformed into the third dimension.

The kernel trick is done by transforming equation 2.5 and using the kernel function $K(x, y)$:

$$L_D = \sum \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j K(\vec{x_i}, \vec{x_j}) \tag{2.6}$$

The most popular kernels are the following:

- linear:

$$K(x, y) = \sum_{i=0}^{n} x_i \cdot y_i \tag{2.7}$$

- polynomial:

$$K(x, y) = (c + \sum_{i=0}^{n} x_i \cdot y_i)^d \tag{2.8}$$

- radial basis function:

$$K(x, y) = \exp(-\frac{||x - y||^2}{2\sigma^2}) \tag{2.9}$$

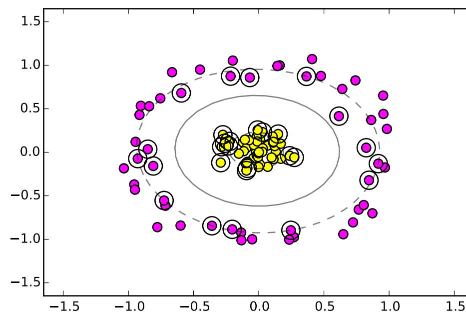


Figure 2.11: Nonlinear classification problem using a rbf kernel.

Using the rbf kernel on the previous nonlinear example, makes it possible to calculate a decision boundary like it is seen in Figure 2.11. After the kernel trick was discovered SVMs gained a huge boost in popularity.

2.5 Problem Statement

Many problems in science can be tackled with data mining and data analysis. While there are many tools for a serial analysis, there is a lack of parallel tools for an increasing number of available larger data sets. A parallel classification algorithm is needed that can handle those data sets. SVMs are one of the best out of the box methods. There are many different fields in science that need to solve a classification task. For example remote sensing in earth science or different task in neuro science or plant science. As a consequence, the framework needs to be generic and reusable. That means it is not optimized for only one task but need to perform well on many varying ones. It should support well known data formats for input and output and not a proprietary one. If a format is not supported there should be a functionality to convert it to a supported type. The framework in this thesis can freely be configured by the user because it is written in Python and all class attributes can be accessed. Most of the times this is not needed because a standard behavior can be used via the Python shell or from the system shell.

When the amount of data increases, training a SVM has a complexity of $O(n^2)$ for n training instances and the training of a SVM becomes very expensive. By parallelizing a SVM the training time can be decreased and larger data sets can be tackled with them, since there is more aggregated memory available on parallel machines than memory on a single system.

To maintain usability by non-technical savvy users the parallelization techniques are hidden from the user. The user does not need to know the technical details of a parallel environment. This includes the low level language C, the batch scheduler of the cluster system, MPI and other cluster tools. Because the amount of data keeps increasing, the implementation needs to scale with big data sets. This includes features like parallel I/O. This leads to the first two research questions.

> **RQ1**: This thesis studies a MPI based Cascade SVM because it is researched if this approach is scalable for larger data sets in order to enable SVMs for big data problems.
>
> **RQ2**: This thesis studies the I/O performance in big data problems to find out how it influences the scalability and how it can be improved.

While the normal cascade SVM approach promises a speedup, a loss in accuracy is to be expected. It therefore needs to be properly investigated if the performance

regarding the training time as well as the accuracy can be improved in some way. This resolves in the second research question.

> **RQ3**: This thesis studies the cascade SVM approach in order to avoid the load balancing problem, so that the framework becomes faster.
>
> **RQ4**: This thesis studies the feedback of the cascade SVM approach and researches possible enhancements to increase the speedup and accuracy.

3 Chapter 3
Related Work

3.1 Support Vector Machine Speedup Techniques

SVMs are popular because they often achieve a high accuracy. While implementations of a linear SVM scale for larger data sets, this is not the case for non linear SVMs. However, most of the real world problems are not linear separable and a non linear SVM has to be used to gain a high accuracy.

There are some approaches to enable non linear SVMs for larger classification problems. One of them is to use a linear SVM on transformed data. Instead of using a non linear kernel like rbf, the data is transformed into a higher dimension by adding additional features. Adding features manually is not feasible. This approach is called kernel approximation. Generated features are added to the data set, so that a linear SVM can be used, which scales better for large data sets.

In some cases the use of dimensionality reduction techniques is a sound method, like applying principle component analysis (PCA) [24] with a subsequent cut of not useful dimensions.

The rbf kernel can be approximated using the Monte Carlo approximation of its fourier transformation. More information on random fourier features and random binning features can be found at [39].

Another approach, that can approximate any kernel is the Nyström method [51]. It uses a subsample of the data set to approximate a kernel. It is shown in that Nyström [53] can achieve a better generalization in some cases.

Because SVMs scale mostly with the number of samples, the performance can also be improved by so called chunking introduced by Vapnik [50]. It reduces the size of the kernel matrix from n^2 (n being the number of training samples) to approximately m^2 (m being the number of training samples with a non-zero lagrange multiplier). The quadratic programming problem is split into smaller problems, that have the goal to identify the non-zero lagrange multipliers and discard the zero ones. While a better algorithm for solving the quadratic programming problem is introduced in [37], the chunking approach may be better suited for parallelization. The approach used for parallelization is similar to chunking.

3.2 Support Vector Machine Tools

This section introduces some popular libraries that implement a SVM. While there are some parallel implementations, most of them are serial. Table 3.1 shows an overview of a wide selection of different libraries and their analyzed properties. The mentioned libraries are open and freely available (commercial libraries and tools have been therefore kept out).

Technology	Platform Approach	Multiclass	Supported Kernels	Parallelization	Stable
libSVM	C/C++, Java	yes	linear, rbf, polynomial, sigmoid	no	yes
Weka	Java	yes	linear, rbf, polynomial, sigmoid	no	yes
R (kernlab)	R	yes	linear, rbf, polynomial, sigmoid, custom	yes	yes
Matlab	Matlab	yes	linear, rbf, polynomial, sigmoid, custom	no	yes
Octave, only libSVM	Octave	yes	linear, rbf, polynomial, sigmoid	no	yes
Apache Mahout	Java, Hadoop	-	-	-	-
MLlib/Apache Spark	Java, Spark	no	linear	yes	yes
scikit-learn	Python	yes	linear, rbf, polynomial, sigmoid, custom	no	yes
Twister/ParallelSVM	Java, Twister, Hadoop	no	linear, rbf, polynomial, sigmoid	yes	no
πSVM	C, MPI	yes	linear, rbf, polynomial, sigmoid	yes	yes
GPU LibSVM	CUDA	yes	linear,rbf, polynomial, sigmoid	yes (rbf)	yes
pSVM	C, MPI	no	linear, rbf, polynomial	yes	no

Table 3.1: Overview of open and freely available parallel SVM tools and their analysis.

The libSVM [11] library is available in C++ and Java. It is very popular and offers bindings for other languages like Matlab/Octave or Python. Apart from that it is wrapped by many other libraries like Weka or scikit-learn. While libSVM is stable and provides all common kernels, it is not parallelized and not suitable for large data sets. It nevertheless represents a stable SVM de-facto standard implementation used in some parallel implementations as well.

A parallel SVM is implemented for Apache Spark [55] in MLlib. Map/reduce [17] frameworks are popular for data intensive applications. However, most of them are not suited for iterative methods, which includes many machine learning algorithms. Apache Spark focuses on this problem while retaining the scalability and fault tolerance of map/reduce. For this a new abstraction is added. Resilient

distributed datasets (RDDs). It can be used with Scala, Java or Python. The SVM implementation of MLlib uses a distributed stochastic gradient descent (SGD) to solve the problem. A drawback of the library is that only linear SVMs are supported by the current version.

Another map/reduce framework that supports iterative algorithms is Twister [18]. There is also a parallel SVM implementation [46] based on the Twister framework and the libSVM Java library. Like the implementation in this thesis it uses a cascade SVM approach which is further explained in chapter 4. Unlike Spark it also provides non linear SVMs, but Twister is not as stable as Spark. It is currently released in version 0.9 and not further developed.

pSVM [12] is a parallel SVM implementation in C and parallelized using MPI. It is based on a parallel incomplete cholesky factorization, that factorizes a matrix A into a smaller matrix H, so that $A \approx H \times H'$. By this the memory usage can be reduced. Furthermore it speeds up the computation with a parallel interior point method. pSVM provides support for non linear SVMs, but the current version is unstable and the development seems to have stopped.

πSVM [3] is also implemented in C and uses MPI. It modified libSVM and uses a distributed SMO algorithm [8] to solve the quadratic programming problem. The library can handle non linear SVMs as well as multiclass problems. Stability as well as performance improvements were made in a bachelor thesis to accomplish a better usability.

3.3 Remote Sensing Applications

Remote sensing [31] is a important source of information for monitoring man-made and natural land covers. The informations are measured analyzed and interpreted to gain new knowledge. Because of the development of sensor resolutions the amount of hyperspectral and high resolution data has increased. An example for a classification task is to classify the land cover types using this data in order to understand the impact of natural disasters or the development of cities.

Remote sensors measure the electromagnetic radiation energy, that is reflected or emitted by earth, at different wavelengths [19]. They are influenced by different sources, e.g. surface material, and are characteristic for the different objects.

The increase in data volume, velocity and variety result in larger data. To enable scaling of established algorithms like Support Vector Machines, a parallelization is needed. Another approach is to reduce the data with different methods (e.g. PCA

[24]) to so called smart data. In [10] a remote sensing data set is analyzed and available methods are evaluated.

Most of the domain scientists are used to frameworks like Matlab or R. Parallel frameworks, however, are often different and hard in usage. A parallel framework with an easy to use interface is needed.

3.4 Summary

This chapter gives an overview over related work in the technical field and the scientific domain. Table 3.1 gives an overview over popular serial SVM implementations and parallel frameworks. The de-facto standard for serial SVMs is libSVM [11]. It is popular because it is stable and provides all common kernels. The performance is good because it is implemented in C++. There is also a Java variant and many bindings for other languages available.

Parallel frameworks based on map/reduce are Apache Spark and Twister. While the Spark [55] framework and MLlib [22] seem promising, only linear SVMs are currently supported. Twister [18] is not further developed and quite unstable.

Apart from that there are also frameworks based on MPI. One of them is pSVM [12], which is based on a parallel interior point method to solve the problem. As for Twister the development is not continued and there are some stability issues. The second approach is πSVM [3] and is based on a parallel sequential minimal optimization algorithm. The implementation has been optimized and can handle non linear as well as multiclass problems.

In addition to an overview of the technical related work a short introduction to the domain specific problem of remote sensing [31] is given. Land covers are measured using remote sensors and different types of land cover have to be classified.

4 Chapter 4 Cascade SVM

This chapter introduces the approach used for parallelization in this thesis. It describes the conceptual design of the cascade SVM , like it was proposed in [26]. After that the basic design for the implementation used in this thesis is proposed. Decisions regarding the design of the framework are discussed and the convergence condition is explained.

4.1 Conceptual Cascade Design

The idea of a cascade SVM is to filter non support vectors as early as possible. The algorithms that solve the quadratic programming problem of a SVM have a worst case complexity between $O(n^2)$ and $O(n^3)$ for n training samples. By decreasing the number of training samples the training time gets shorter. Different filter techniques like clustering or SVMs were evaluated and SVM filters seem to be the best choice, because other approaches may optimize criteria that are not useful for the global optimum [26]. The filtering can be done in parallel to accomplish further speed up.

Figure 4.1 shows a binary cascade SVM. The training data is split up into several subsets. Each of the eight SVMs of the first layer is then trained on one of these subsets in parallel. After the SVMs are finished only support vectors are passed on to the next layer. The results of two models are combined and used as the training set for the SVM of the next layer. This is repeated until only one SVM model remains. The consequences are a lower accuracy since not all data is used by an SVM. To accomplish a better accuracy another iteration can be initiated. Therefore the result of the last SVM is broadcasted to all SVMs of the first layer. The original input data is merged with the incoming results and a new iteration can be started. This way all SVMs of this iteration take the support vectors of the last iteration into consideration. The approach thus suggests that no SVM is trained on the whole data set but only on parts of it.

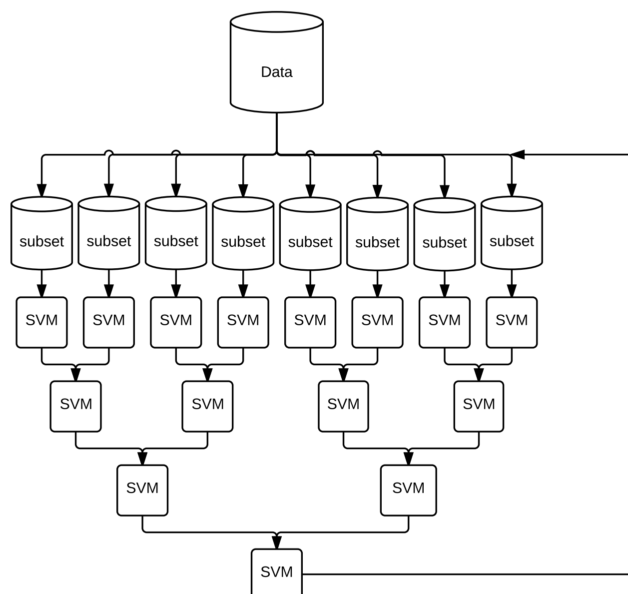


Figure 4.1: Binary CascadeSVM

4.2 Analysis of Scalable and Parallel Approaches

Existing implementations of a cascade SVM like Twister [46] are based on Map/Reduce and Hadoop [44]. As it can be seen in Figure 4.1 the data has to be split and distributed on the different machines. Therefore a data distribution file has to be created. All this has to be done, before a cascade SVM can be trained. During the map job of the Twister implementation a layer of SVMs is trained. The reduce job combines two sets of support vectors into one input set. These sets are used as training data for the map job of the next iteration. The Cascade is finished when there is only one SVM left. Because of this there is no feedback and only one full cascade iteration (deviation from the basic conceptual design seen above).

In this thesis MPI is used instead of Map/Reduce after evaluating Twister with several data sets. This has several advantages. There is no need for any manual preprocessing of the input files by users, since all processes have access to the file, which is located on the network storage. Figure 4.2 shows how the data is distributed in the MPI version that represents an often used standard in message passing within parallel computing. The data is read by the root process and a subset for every process is created. The subsets are distributed among the other processes. This is done automatically and thus hidden from the user. While the message passing itself appears to be more complicated, an iterative cascade SVM can be easily implemented. MPI is also better established in parallel and distributed computing domains in general and better suited for the available clusters at the JSC in particular.

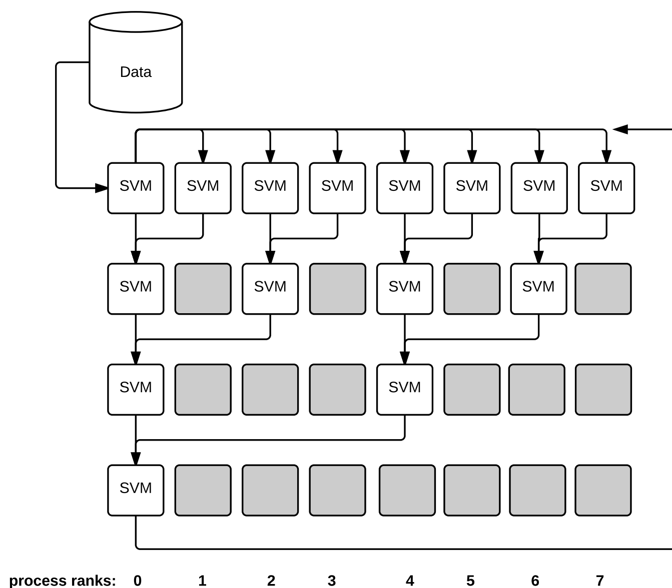


Figure 4.2: MPI based cascade SVM with idle CPUs (gray)

The design partly also shows the disadvantages of a MPI based approach. The data input may be slow for large data sets, because the root process reads in the whole set. This may also lead to memory errors, if there is not enough memory available. The additional message passing needed to send the subsets to the other processes also takes some time. Hence, the following problem can be stated:

Problem 1. The limitation , that the root process is loading all the data, has a bad impact on the scalability for large data sets.

Another disadvantage, which can be seen in Figure 4.2 is the load imbalance. In the first layer all processes are busy calculating a SVM model. Because the results of two SVMs are combined, only one half of the processes are still used in the second layer. The idle processes are colored gray. In the last layer all but the root process are idle and wait for the one computing process. This is problematic because all processes are reserved until the whole MPI application is finished. A Hadoop version does not have this problem because the computing nodes are only reserved while they are calculating (i.e. independent map or reduce tasks). Therefore the second limitation is as follows:

Problem 2. The load imbalance in the later layers causes a loss of computation resources, because reserved processors are idle.

4.3 Convergence Condition

One iteration of a cascade SVM does not always accomplish the same accuracy as a single serial SVM trained on the same data set. In [26] it is proofed that a cascade SVM converges if the support vectors of the last trained SVM are merged with the different input set of the first layer and a new iteration is started. The condition for a convergence check is only vaguely introduced and not described in detail.

The first condition tested in this thesis is based on the comparison of the support vectors. For iteration n the support vectors of the first layer are compared to the support vector of the last layer of iteration $n-1$. An interesting property is that if the two sets of support vectors are equal, the models are the same too and the cascade SVM can finish. While this is working for simple classification tasks, this is not guaranteed for more complicated ones. The problem is that an equivalent decision boundary may be defined by more than one set of different support vectors.

Therefore a new convergence condition is introduced. Instead of comparing the support vectors the objective function of the SVMs is calculated.

$$L_D = \sum \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j K(\vec{x_i} \vec{x_j}) \tag{4.1}$$

This condition solves the problem of multiple sets of support vectors that define the same decision function. Figure 4.3 shows a program flow that illustrates how the convergence check is done. The first layer of SVMs is trained on their input sets. If it is the first iteration, the other layers are trained as well. The SVM of the last layer computes the objective function 4.1 and broadcasts the value to all processes of the first layer in the first iteration. If it is not the first iteration, the cascade does not immediately continue after the first layer. Instead each process of the first layer computes the objective function 4.1 for its trained SVM. These are compared to the value they got from the last layer in the previous iteration. If the values are close enough (i.e. below a certain threshold) , the algorithm terminates with the model of the previous iteration.

4.4 Summary

In this thesis a basic design approach is used to parallelize the popular SVM classification method. This is based on dividing the original data set into sub sets which can be processed on distributed machines. The results of each model are

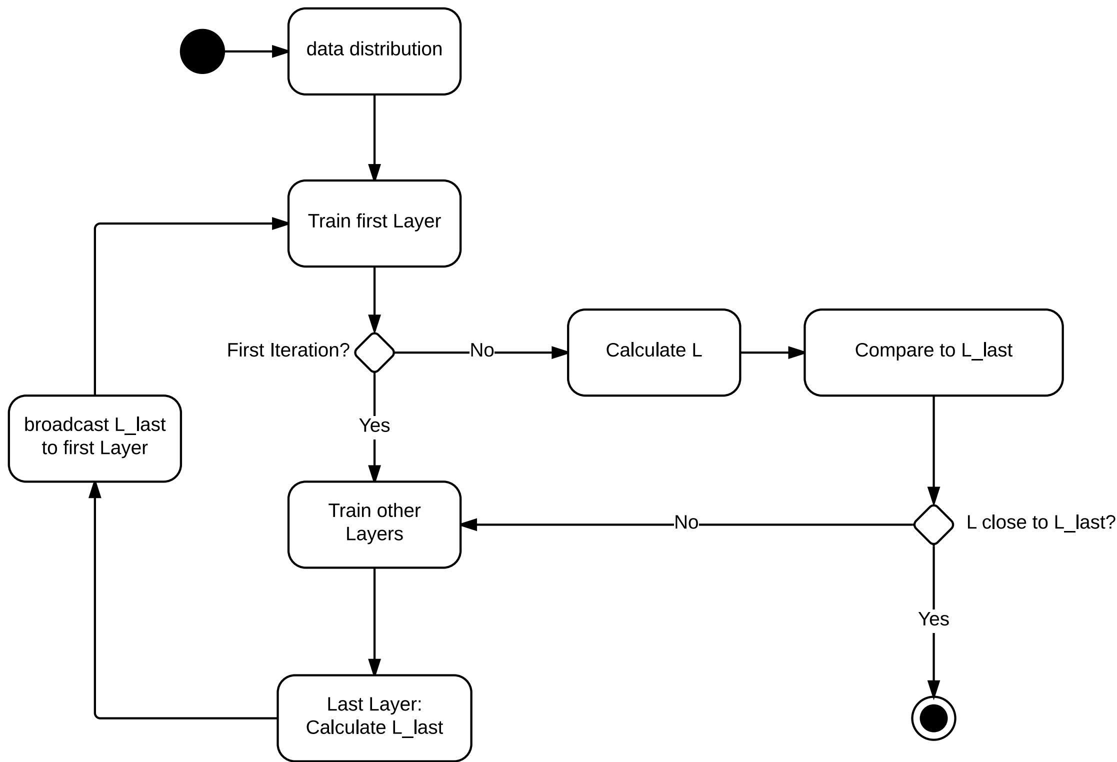


Figure 4.3: Visualization of the convergence check with the program flow of the convergence check. 'L' and 'L__last' are the results of computing the objective functions 4.1 of the SVMs.

then used as an input set for the next iteration of training an SVM. With this no SVM is trained on the complete data set.

The basic approach of a cascade SVM is the same for MPI based implementations as well as the ones based on Hadoop. However, the different parallelization frameworks and their unique properties result in varying designs with different identified challenges. While the MPI version provides an easier preprocessing step and is more flexible than Hadoop versions (e.g. multiple cascade iterations) , the higher levels of a Cascade have a load imbalance problem. This means that many processes, that are allocated, are not actually used and thus idle. An approach is required that lowers the idle time while not loosing sight of the efficient training of the cascade SVM.

Because a cascade SVM is not directly trained on the whole data set, the global optimum may not be reached after one iteration. In contrast to other analyzed implementations the implementation of this thesis enables multiple iterations and the support vectors of the last SVM are used as feedback for the first level of the new iteration. It is proofed in [26] that it converges after some iterations. The number of iterations is designed as a parameter or a convergence check can be configured, that compares the values of the objective functions.

Chapter 5

5 Improving Cascade SVM

This chapter discusses details of the architectural design and implementation and improvements of the basic cascade SVM approach introduced in Chapter 4. This includes low level optimizations concerning the I/O bottleneck as well as direct improvements of the algorithm implementation itself.

5.1 Architectural Design and Basic Implementation

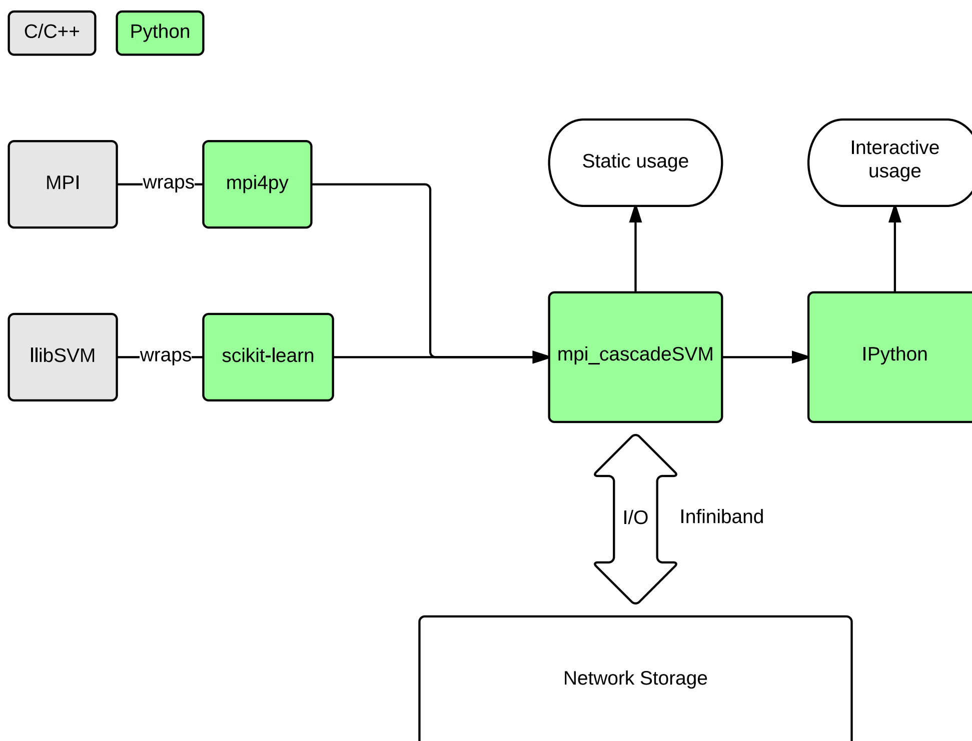


Figure 5.1: Framework architectural design

Figure 5.1 shows the design of the framework in more detail, the cascade SVM (see fig. 4.2) is implemented in 'mpi_cascadeSVM' and uses the 'SVC' class of scikit-learn [35] for the single core (i.e. serial) SVM models. The 'SVC' class itself wraps libSVM, which is implemented in C. This way a clean Python interface can be used while maintaining C-like performance.

```
def _scatter_labels(y, comm=world_comm, root=0):
    if comm.rank == root:
        slices = _calc_slices(y, comm=comm)
    else:
        slices = np.zeros(comm.size, dtype=np.int64)
    # Broadcast information for scatterv
    comm.Bcast(slices, root=root)
    # pos for labels
    pos = np.array([sum(slices[:i]) for i in range(comm.
        size)], dtype=int)
    # number of rows for each process
    row_cnt = slices[comm.rank]
    # allocate memory for split samples and scatter it
    split_y = np.zeros(row_cnt, dtype=np.int64)
    comm.Scatterv([y, tuple(slices), tuple(pos), MPI.
        INT64_T], split_y, root=root)
    return split_y
```

Listing 5.1: Scattering labels to all processes

The message passing is handled with the mpi4py module [16], which provides Python bindings to C MPI implementations. Most of the communication is done with the powerful collective operations (e.g. Broadcast, Scatter, etc.) .

It is out of scope to provide the full listings of the code, but as an example listing 5.1 shows the function that distributes the class labels to all processes. The communicator and root process are parameters, so that the function can be used more flexible and not only with the world communicator. The world communicator is the default one and contains all processes that are started by MPI. The root process calculates the number of labels (*slices*) for each process and broadcasts the information to all processes. These are needed to create a fitting numpy array (*split_y*) on all processes and calculate the starting index (*pos*) for each process. The labels are then scattered. The difference between scatter and scatterv [33] is that varying number of items can be send with the latter. The first parameter of scatterv is the send buffer as a list with additional information. The second parameter is the receive buffer and the third is an integer declaring the root process.

The data points are distributed in a similar way. The size of the subsets is equivalent to the ones of the labels with the exception that the data points are vectors and not single values. The array containing the data points is treated as a one dimensional array for the message passing and the subsets are reshaped after they are received. Therefore the number of features n_f has to be known at every process. Instead of sending a two dimensional array with m data points, a one dimensional array with $m \cdot n_f$ elements is sent.

This explains how the processes are initialized with training data, but this is not all the communication which is needed. After the SVMs of the first layer are trained the support vectors are sent to the next layer as it can be seen in Figure 4.1. This is done using gatherv [33], which is the counterpart to scatterv. Figure 4.2 shows that two processes take part at the communication. For every group a communicator is created and gatherv is used.

The alternative to collective communications on different communicators is to perform point to point communication on the world communicator. The support vectors are manually send to the process that calculates the next SVM. For a binary cascade SVM one send/receive operation is used. For different formed Cascades this is however not the case. A loop would be needed to send the results of the first layer to the next one. This would result in more error prone code because the communication is done manually and more complex. Furthermore the collective communications of MPI are optimized on performance and therefore used in this framework.

To enable multiple iterations of a cascade SVM the results of the last layer have to be sent to the first layer. Hence, once the iteration is finished in the last layer the broadcast function [33] is used with a communicator that contains all processes of the whole Cascade.

An additional step that requires communication is the convergence check described in section 4.3. The first implementation was based on the support vectors and the second one is based on the objective function. The advantage of this is that only one scalar has to be communicated instead of a matrix. The downside is that the computation of the objective functions as mentioned in 4.1 is quite expensive, because two loops over all SVs are needed to calculate the double sum.

$$L_D = \sum \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j K(\vec{x_i}, \vec{x_j}) \tag{5.1}$$

Listing 5.2 describes how the lagrangian shown in equation 5.1 is computed. For a linear SVM $K(\vec{x_i}, \vec{x_j})$ is equal to the dot product, but for more elaborate cases a different kernel is used. The first three parameters are the support vectors (x), their labels (y) and their coefficients (α). The last parameter is the kernel function, which has only the support vectors as a parameter. The function can currently only handle a binary problem and expects the labels to be either 0 and 1 or -1 and 1. Numpy is used to handle all the values. This provides good performance, if numpy functions are used to operate on the data. The disadvantage is that looping over numpy arrays is very inefficient. Equation 5.1 shows that a for loop (i.e. the sum of the coefficients) and a nested for loop (sums from i and j) is needed to

```python
def lagrangian_fast(X, y, coef, kernel):
    set_y = set(y)
    assert len(set_y) == 2, "Only binary problem can be handled"
    new_y = y.copy()
    new_y[y == 0] = -1
    C1, C2 = np.meshgrid(coef, coef)
    Y1, Y2 = np.meshgrid(new_y, new_y)
    double_sum = C1 * C2 * L1 * L2 * kernel(X)
    double_sum = double_sum.sum()
    W = -0.5 * double_sum + coef.sum()
    return W
```

Listing 5.2: Computation of the objective function. X: Support Vectors, y: class labels, coef: coefficients, kernel: kernel function

calculate the objective function. By replacing these loops with matrix operations a huge speedup can be achieved, because numpy is based on C and Fortran code. Accessing each element in a for loop costs much more than these matrix operations.

The sum of the coefficients can easily be computed with a numpy function. To compute the nested loop meshgrids of the labels and coefficients are created to compute $y_i y_j$ and $\alpha_i \alpha_j$. For a given list of labels $a = [1, 2, 3]$ the meshgrid function returns two matrices A_1 and A_2.

$$A_1 = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}, A_2 = \begin{pmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{pmatrix}$$

Multiplying A_1 and A_2 element wise results in a matrix containing all pairwise products.

To compute $K(\vec{x_i}\vec{x_j})$ the kernel matrix for the given support vectors has to be computed. For a linear kernel the dot function of numpy can be used. Aside from the linear kernel, the rbf kernel as defined in equation 2.9 is implemented because it is one of the most used kernels. Available Python libraries like numpy and scikit-learn don't provide an implementation. Like above also matrix operations are used.

Equation 2.9 includes the term $||x - y||^2$ that is not suited for matrix operations. The pairwise differences have to be calculated for every vector pair yielding another vector as a result. In order to store values according to this structure, a three dimensional matrix had to be used. Therefore it is transformed and scalar products are calculated instead of vectors.

$$||x-y||^2 = \sqrt{(x-y)\cdot(x-y)}^2 \tag{5.2}$$
$$= (x-y)\cdot(x-y) \tag{5.3}$$
$$= x^2 - 2xy + y^2 \tag{5.4}$$

Based on Equation 5.4 and using Equation 5.5, the rbf kernel function can be further transformed.

$$\gamma = -\frac{1}{2\sigma^2} \tag{5.5}$$
$$K(x,y) = exp(-\frac{x^2 - 2xy + y^2}{2\sigma^2}) \tag{5.6}$$
$$= exp(\frac{2xy - x^2 - y^2}{2\sigma^2}) \tag{5.7}$$
$$= exp(\frac{xy}{\sigma^2} - \frac{x^2}{2\sigma^2} - \frac{y^2}{2\sigma^2}) \tag{5.8}$$

The function in listing 5.3 uses equation 5.8 to compute the rbf kernel. It has two input parameters. The first one is a list of vectors. The second one is γ, which is a parameter of the rbf kernel. The three fractions seen in equation 5.8 are calculated. The first fraction xy is the dot product between every support vector and results in a matrix K. The matrix is divided by σ^2. K_{ij} is equal to the dot product of the support vectors $x_i \cdot x_j$. Therefore the diagonal elements of K are exactly equal to x^2 and y^2 of equation 5.8. This is taken as the input for numpy's exponential function in the last step and the kernel matrix is returned.

```
1 def rbf_kernel(x, gamma):
2     sigmaq = -1 / (2 * gamma)
3     n = x.shape[0]
4     K = np.dot(x, x.T) / sigmaq
5     d = np.diag(K).reshape((n, 1))
6     K = K - np.ones((n, 1)) * d.T / 2
7     K = K - d * np.ones((1, n)) / 2
8     K = np.exp(K)
9     return K
```

Listing 5.3: Implemented RBF kernel using matrix operations

The steps above enable the convergence check that can be used for linear kernels but more notably for rbf kernels without losing too much performance.

5.2 Parallel I/O

Reading the data is currently done serial. This means that the root process reads all samples, although it only needs a part of them in order to compute a fraction of the input data. For small data sets this is not a problem, but for larger ones the following issues occur. The loading time as well as the time needed to distribute the subsets to their belonging processes increases with an increasing input set. If the input set is too large, it may even cause a memory error. Thus the I/O becomes a bottle neck for large data sets and thus we work on improvements of the basic cascade model putting the focus also on parallel I/O besides solely parallel computing.

In order to avoid these problems an alternative format is introduced. The libSVM format stores the data points in text form. While this enables the user to read the data file, it is slower than a binary format when it is read by a program.

One binary file format is HDF5 [47], that is also a broadly used standard format for science and engineering. It is popular in science, because it enables storing multidimensional tables or arrays in an efficient way. Next to the data meta data can be stored in the same file. The hierarchical structure of a HDF5 file, shown in Figure 5.2, can be basically compared to a file system. It consists of data sets containing the 'actual data' but also 'groups', which are container structures and can hold groups or data sets. Meta data is stored as user defined attributes of data sets or groups.

The format is used by many scientific simulations (e.g. [45]) to store data. One machine learning algorithm using HDF5 is HPDBSCAN [25], which is a parallel implementation of the clustering algorithm DBSCAN [20]. A list with users of HDF5 can be found at their website[2].

The design in the thesis takes advantage of HDF5 as follows. Improved data files in this framework consist of a 'Data' data set and a 'Label' data set.

In addition HDF5 also support parallel I/O, which is often used on clusters and increases the I/O performance as shown in [54]. If it is compiled against the correct MPI driver, a HDF5 file can be read by multiple processes. Each process calculates the starting position and size of its data set according to its rank. Each MPI process is identified by its rank, which is an integer between zero and the number of processes minus one. Instead of one process reading all data and scattering it, each process only reads the sub set it needs. Because of this less MPI communication is needed and less memory is used while the data can be read in parallel from different processors.

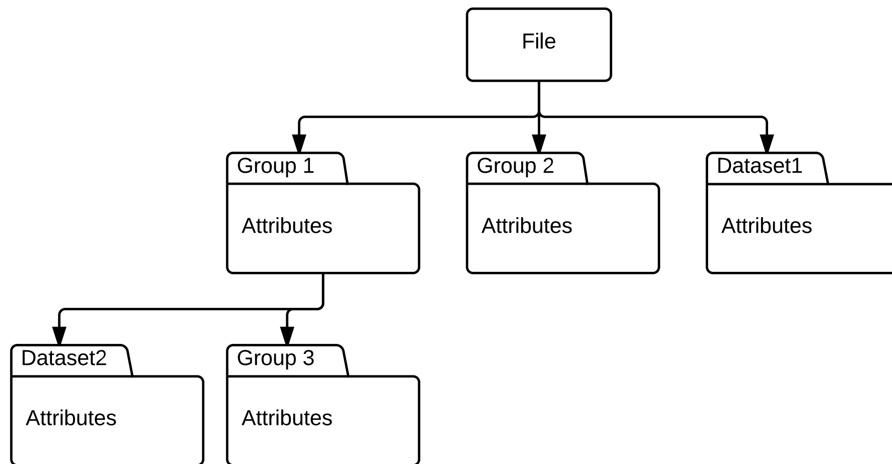


Figure 5.2: Basic HDF5 Structure

The framework is designed in such a way, that the I/O is hidden from the user. This means, that parallel I/O is automatically used if it is available. This improvement enables the framework to handle larger data sets than with a serial I/O design.

5.3 Cross Feedback

The design of the cascade SVM based on MPI as seen in Figure 4.2 has the problem of load imbalance. This means that only a few processors are working at the later layers and the rest of them is idle that we already identified as a limit in chapter 4. One improvement in the design is thus to reduce the time spent at the later layers and decrease the total idle time of all processes. One possible improved training algorithm is proposed in [56] and [52]. Instead of using the result of the last SVM as feedback for all SVMs of the first layer, the feedback is done at an earlier stage.

Figure 5.3 visualizes the cross feedback at the second layer. The support vectors of 'SVM5' merged with the input sets of 'SVM3' and 'SVM4'. There is no need to merge them with the ones of 'SVM1' and 'SVM2' because 'SVM5' is trained on their results. Likewise the support vectors of 'SVM6' are merged with the input sets of 'SVM1' and 'SVM2'.

Taking advantage of cross feedback in order to improve the parallel framework design yields two advantages. The first one is, that presumably less time is used to train a Cascade, if cross feedback is used instead of the initially proposed last layer feedback. This is due to the fact that less layers have to be trained. The second advantage is, that the load imbalance is reduced because more time is spent at the early layers and less processes are idle.

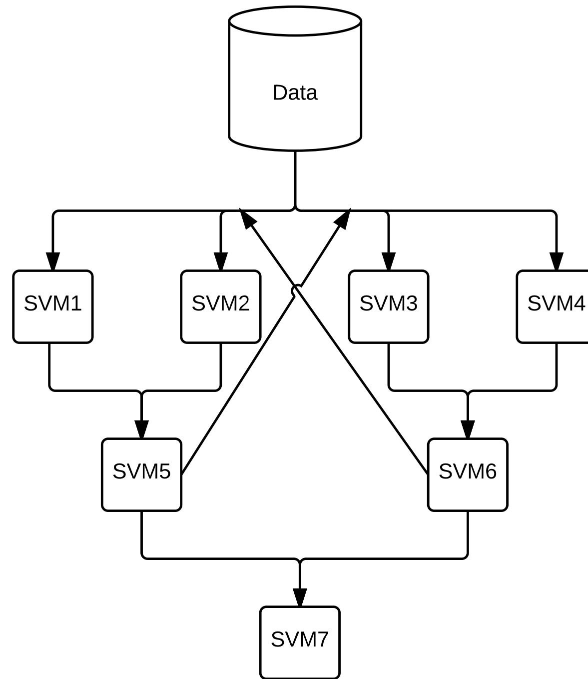


Figure 5.3: Early cross feedback in a binary cascade SVM

The framework is generally designed in such a way, that cross feedback can be done at any layer and thus can be configured as applications require it. The number of cross feedbacks that is done is not limited to one but can also be set by the user to an arbitrary number. Therefore the framework enables the user to optimize the configuration of the cascade SVM for a given data application.

5.4 Distance Filter

The feedback and cross feedback are methods used to improve the accuracy of a cascade SVM without loosing too much speedup. Another further interesting enhancement to feedback is a so-called filter function. This function takes the support vectors as an input and computes a score for everyone of them. Instead of using all support vectors only a sub set of them is used. The sub set is determined by the filter function. The function itself is not fixed, except that it returns a sub set of the support vectors. This approach is easier to understand when one considers the basic cascade SVM approach, that over the different layers is also filtering non SVs out of all data sets. Using a specific filter criteria early in the process speeds up this process.

One approach is to use spatial information of the vectors. A cluster of support vectors has presumably redundant information, because many vectors are close

together and there is a high potential that only one will remain in order to support the final decision boundary. In contrast a vector that has a higher distance to others, may hold more important information in order to contribute to the decision boundary of the classifier. A distance filter, that calculates the mean distance for every support vector to the support vectors of the same class, can be used. After that the vectors are ordered with an decreasing mean distance and the first k vectors are returned and used for feedback. In this case the euclidean distance is used. Parameter k also depends on the application and is thus configurable.

Figure 5.4 shows how the filter can be used before the actual cross feedback is done. A large number of SVs goes into the filter and a smaller number k is used as feedback. This is visualized by the thick and thin lines connecting to the filter. What kind of filter is used and on which layer can be decided by the user.

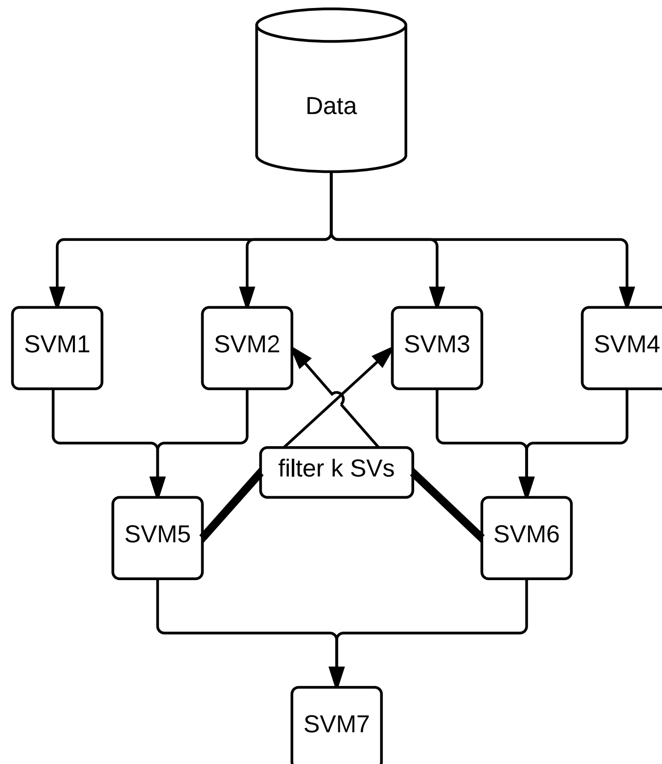


Figure 5.4: Filter used before Cross Feedback.

A more sophisticated method would be to cluster the support vectors using a clustering algorithm, e.g. kmeans [5]. However, more complex approaches also take more time and reduce the speedup. In this thesis a simple distance filter is used to keep a good speedup while the framework can be easily extended with other filters depending on application needs.

5.5 Summary

In this chapter the design of the framework is described. The communication is mainly done with the collective functions of MPI. Therefore multiple communicators have to be created for the different communication groups. Because of the arbitrary number of data points the vector versions of scatter and gather are used.

The convergence check is done by comparing the objective functions of the different SVMs. To calculate the functions in reasonable time, matrix operations are used. Matrix operation replace 'for loops' and provide a speedup if numpy arrays are used because they are based on compiled C and Fortran code. However, the algorithms have to be transformed to be able to use matrix operations. This is done for the rbf kernel 2.9 and the objective function 5.1, so that the convergence check can be done for binary classification problems.

Apart from these design decisions, specific improvements to the original cascade SVM algorithm are designed and implemented. One of them is the improvement of the I/O. Problem 1 states, that the I/O becomes a bottleneck for large data sets. This is solved by using the standard HDF5 file format. It is a binary format, that supports a file system like structure. In addition it features parallel I/O on cluster systems, which can be used to improve the I/O performance and decrease the memory usage.

Another problem of the basic cascade SVM is the load imbalance at later layers as stated in problem 2. A solution to this problem is cross feedback. Instead of performing the feedback at the last layer, it is done at an earlier point in time, so that the total time the idle processes is less. It also decreases the training time in comparison to the initial feedback approach.

The distance filter can be used to filter the support vectors before feedback is done. This reduces the training time because less data points are used, while a increase in accuracy is still likely.

Chapter 6
6 Evaluation & Use Case

In this chapter the framework is tested and evaluated on a remote sensing data set (cf. Section 3.3). The speedup and accuracy are measured and compared to a serial SVM. In order to ensure a proper evaluation, a specific experimental setup is used with the different enhancements of the implemented framework design. In addition it is compared to πSVM that enables a comparison with another parallel approach in a similar execution environment.

For the evaluation many runs had to be performed on the cluster. Therefore the tool Jube2 [32] was used. A benchmark run can be defined using the markup language XML [7]. One benchmark can contain many single runs of the program with different parameters. The results can be parsed using regular expressions, which can also be defined in the XML file, and the output is saved in form of CSV [42] files. This enables that runs and parsing their output can be automated. The evaluation is done on the cluster system introduced in Section 2.1.

6.1 Remote Sensing Data for Evaluations

The data set used for evaluation is the Rome data set, which is available at B2SHARE[1]. Images of Rome were taken by a satellite and 16 different types of land cover (e.g. building, road, tram) are labeled. The problem is reduced to a binary classification task by only using the two classes, that have the most samples. The first type is the 'building' class and the second one is the 'road' class. There are three versions of the data set with different features. The raw data (set A) has five features. The two additional sets take the neighboring pixels into account and have 15 (set B) and 55 (set C) features. After reducing the data set to a binary problem the training set is about 34.000 pixels large and the test set about 340.000. Each pixel is equivalent to one sample and the task is to classify a pixel as belonging to a building or to a road. In addition to the three data sets A, B and C slightly differing sets (A^*, $B^*$and C^*) are created by switching the training and test sets in order to enable a relative straightforward understanding of different

[1]b2share.eudat.eu/record/111

workloads (i.e. small and big) without having the problem of explaining and using many different data sets from different domains. This is done because the original training set is small and a larger training set is needed to more properly measure the speedup.

6.2 Speedup by Improvements using Parallel I/O

Figure 6.1 shows the time needed to read dense and sparse input data for a set with 1.000.000 samples. The data is created artificially in order to show significant speed up and design towards big data. While the serial input of the libSVM file needs 29.811s, the HDF5 version of the same data set is read in a fraction with 0.347s for the dense set. The effect of parallel I/O also is visible when evaluating the sparse set whereby libSVM needs 4.563s and HDF5 0.308s.

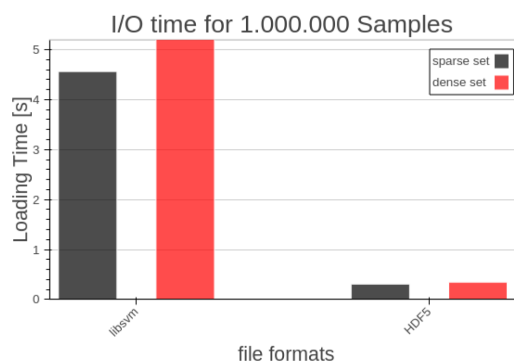


Figure 6.1: Input time for 1.000.000 samples with serial libSVM and parallel I/O using HDF5.

This shows that the I/O bottleneck problem can be solved by introducing the HDF5 standard as an input format, because each process needs less memory and the I/O can be done in parallel, so that it takes less time. For dense data HDF5 as used in this framework is much faster than libSVM, which is optimized for sparse data. This is done by storing the features with their 'id' and only those features that are non zero. While libSVM is faster for the sparse set than for the dense set, it is still slower than HDF5. Therefore it can be concluded that HDF5 is a better alternative if the file itself does not need to be human readable. While today many may inspect libSVM file format data manually, this is probably not possible anymore once big data sets are used for which this improvement is important.

6.3 Accuracy & Speedup

6.3.1 Benchmark on Data Set A

A first benchmark was performed on data set A. A standard binary cascade SVM is used. The data is saved in the libsvm format and no cross feedback or filtering was performed yet and thus it represents an evaluation of the basic cascade SVM framework implementation. The speedup is measured in regard to the number of processes used. If T_p is the time needed for training with p processes, the speedup S_p is calculated as follows:

$$S_p = \frac{T_1}{T_p} \tag{6.1}$$

Figure 6.2 shows the speedup with the number of processes ranging from one to 32 performed on JUDGE (See 2.1). The blue line represents a linear speedup. That means $S_p = p$. It is shown because a linear speedup is often the ideal case. The red line is the speedup of a cascade SVM with one iteration. For two to eight processes the speedup is above linear speedup. This can be explained by the fact that SVMs have a complexity of $O(n^2)$ for n samples. By splitting the samples into multiple subsets a super linear speedup can be achieved. If the number of processes is further increased the speedup falls below a linear speedup. The reason for this is that the size of the training set stays the same, while the number of processes is increased. At some point the communication between the processes and the additional layers take more time than is saved by spitting the data set. For a larger training set more processes could be efficiently used. For small data sets the cascade SVM should only be used on a small number of processes. In general the bigger the data set, the more benefits the parallelization effect in using the cascade SVM approach.

If multiple iterations over the cascade SVM are performed, the speedup drops even more, but is often required in order to increase the overall accuracy. For two iterations a marginal speedup is reached with four and eight processes. The other runs are even slower than the serial SVM. This is due to the small number of samples of set A. After the first iteration the support vectors of the last SVM are merged with the subsets. The subsets grow larger, because an additional cascade run is needed. For a small data set this costs more time than training a single SVM on the original set, thus demonstrating again that the parallel design implies that it makes only sense to use for bigger workloads.

For three iterations this gets even worse and the speedup is lowest of all three runs.

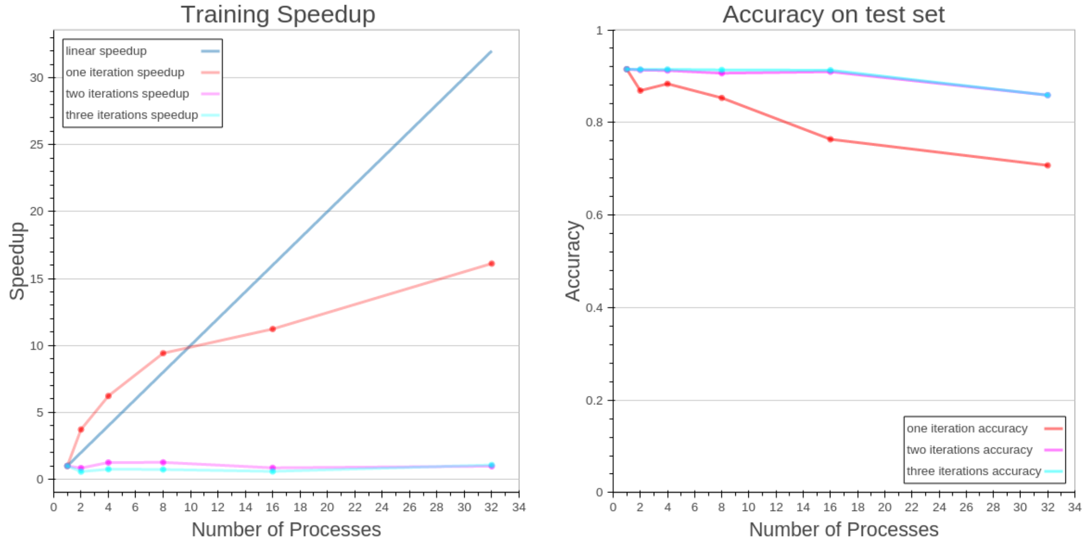


Figure 6.2: Training speedup and accuracy on data set A (small workload)

The right side of Figure 6.2 shows the accuracy on the test set of data set A. With one iteration the accuracy drops with an increasing number of processes. This is due to the fact that the different SVMs are trained on subsets and therefore the optimal decision function can not be found in one run (e.g. data points that contribute to the optimal decision boundary might be in the other disjunct subset). The accuracy can be improved by using multiple iterations. For two as well as three iterations the accuracy is more stable with an increasing number of processes. Only with 32 processes a larger drop can be observed, again due to the extremely small data in the subsets in this particular evaluation run.

To conclude, the evaluations on data set A underline the fact that the cascade SVM is no tool for smaller data sets. While a speedup can be achieved, the accuracy also drops. A compromise can be made by only using a small number of processes to accomplish a speedup while maintaining a reasonable accuracy. In production runs however, there is the expectation that the cascade SVM will be rather used for big data sets and just for the purpose on proper evaluation we here have shown details on a small data set for the sake of completeness.

6.3.2 Benchmark on Data Set B^*

While a benchmark was made on the original set B as seen in tables A.3 and A.4 in the appendix, this section focuses on B^* to investigate bigger workloads as before, which uses the larger test set as a training set and thus adds more insights for the evaluation of the thesis contributions. The results of this benchmark can be found

in table A.5. Because the training time has increased to approximately six hours for a serial SVM, each run was only performed one time.

In Figure 6.3 the speedup and the accuracy for set B^* are shown for one iteration. Like in the previous figure the blue line is the linear speedup and the red one is the training speedup. The benchmark on this larger training set shows that a large speedup can be achieved by using the cascade SVM framework design implementation on big data sets. The training time drops significantly from 21257.6 seconds for a serial SVM to 90.4 seconds for a cascade SVM with 32 processes.

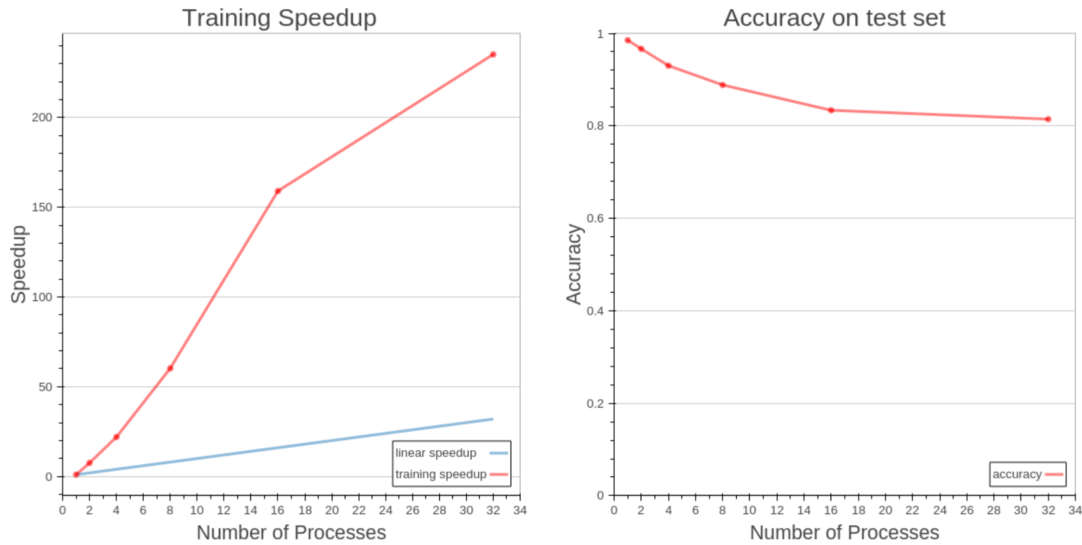


Figure 6.3: Training speedup and accuracy (table A.5) on data set B^*

Figure 6.3 also shows the accuracy in regard to the number of processes. With an increasing number of processes the accuracy drops. Like in the previous benchmark a trade off can be made between the speedup and the accuracy, that should be reached.

# Cascade Iterations	# processes	training time [s]	test time [s]	accuracy
2	16	2047.51	1.10	0.946
2	32	1925.95	0.57	0.925
3	32	5422.09	0.65	0.953

Table 6.1: Training Time and Accuracy with multiple iterations on set B^*.

Table 6.1 is a small part of the table A.5 and shows runs with multiple iterations. The missing runs needed more than 12 hours and were terminated as they thus not provide very much valuable insight here. With two iterations the runs with 16 and 32 processes finished. While the training time is higher than with one iteration, the

accuracy is closer to the one of a serial SVM. The run with 16 processes and two iterations ranges between the runs with two and four processes and one iteration. This goes for the accuracy as well as for the training time.

6.4 Cross Feedback Evaluation

The cross feedback (cf. Section 5.3) is evaluated on data set B^*(larger data set) , so that it can be compared to the single iteration and multiple iteration runs on B^*. As shown above, as it does not make much sense to evaluate on small data sets, the evaluations are performed on the larger data set as the small data set does not add to insight. The goal of cross feedback is to achieve a better speedup than a whole additional iteration would and still increase the accuracy. The cross feedback also addresses the load imbalance problem.

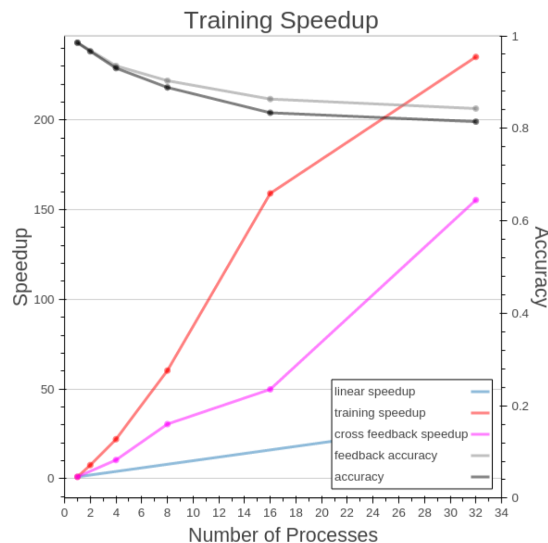


Figure 6.4: Cross feedback on layer 0

Figure 6.4 shows the results of the normal run on B^* and the runs using cross feedback at layer 0, which is the first layer. The speedup is between the linear speedup and the speedup of a plain cascade SVM. Interestingly, the accuracy is only slightly above the one of the plain cascade SVM. This shows that a better accuracy can be achieved with the same amount of processes while loosing some of the performance.

Figure 6.5 shows the speedup and accuracy for cross feedback at different layers. It shows that the speedup gets worse if the cross feedback is done at a later layer that can be expected because of the additional overhead in distributing messages

for implementing cross feedback. On the other side there is a tendency for a higher accuracy. But even if the cross feedback is done at the fourth layer the speedup is still super linear. This shows that the cross feedback method also scales for bigger cascade SVMs and a feedback at high numbered layers.

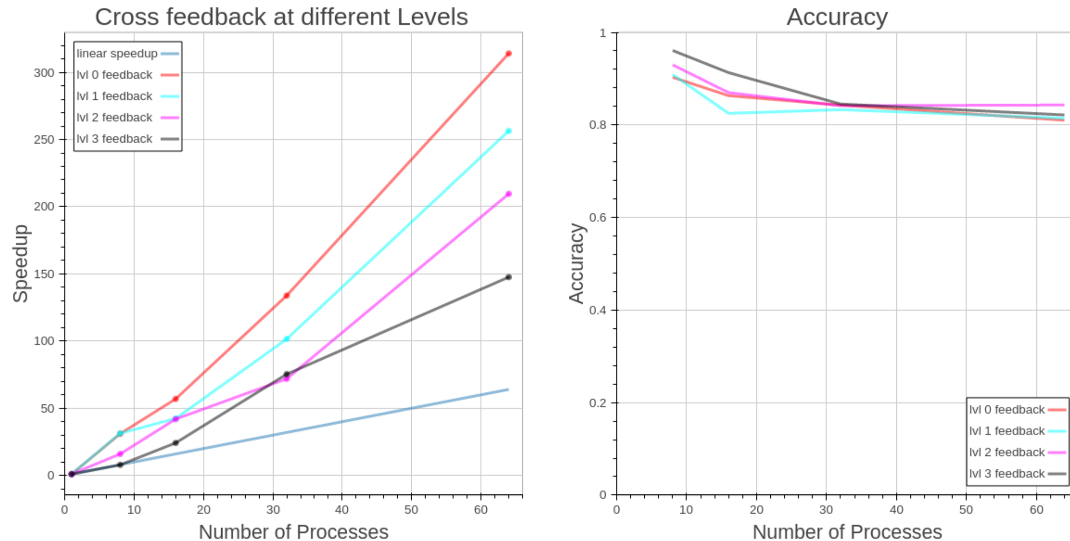


Figure 6.5: Cross feedback design implementation evaluated on different layers.

6.5 Distance Filter

For the evaluation of the distance filter improvements the larger data set $B^*$was used. Figure 6.6 shows the training speedup and the accuracy of a cascade SVM, that uses cross feedback and an additional distance filter (cf. Section 5.4). The distance filter is used to filter each set of SVs and uses the top K SVs as feedback. The feedback is done after the first layer for this benchmark.

The left image shows that the speedup decreases if K is increased. On the right it can be seen that the accuracy increases if K is increased. This method yields another approach to regulate the trade off between speedup and accuracy. Hence, it is adding another parameter option K to the parameter set of the SVM that can be configured depending on data set and intended application goal.

Also other filters can be easily integrated into the framework, e.g. such as using those SV with a large α, while smaller values are neglected according to some configurable threshold. Note that all α computed need to be above zero to represent a SV, but those points of them considered most important are the SVs with the highest values of α, since they exert the highest forces on the decision boundary [9].

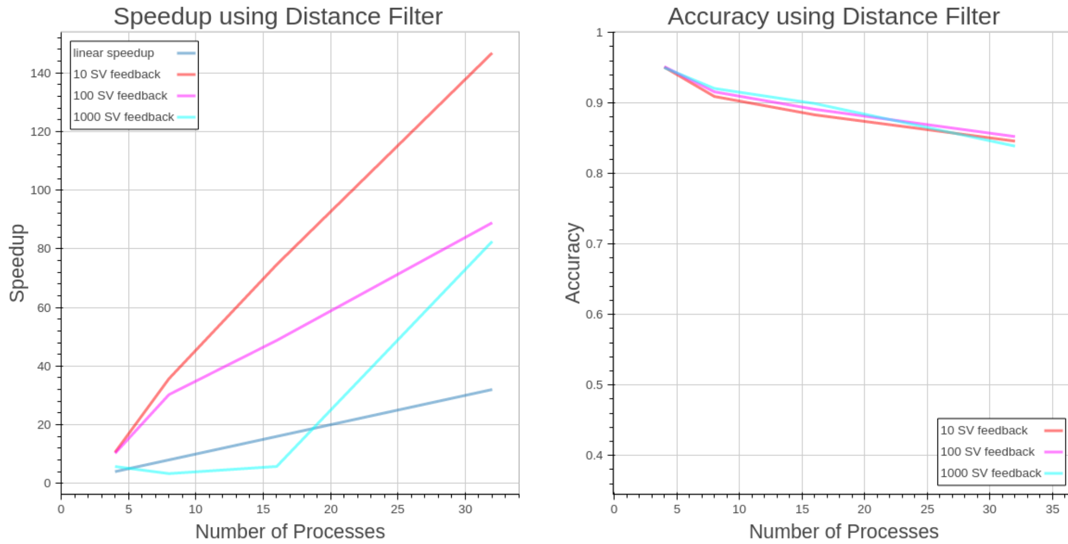


Figure 6.6: Training Speedup and Accuracy (zoomed in) with Distance Filter. The top $K = [10, 100, 1000]$ SVs are used for feedback.

6.6 Comparison with πSVM

Unlike cascade SVM, which is a data-processing approach for parallelizing SVMs, πSVM is a algorithmic approach. While the cascade SVM distributes the data to multiple processes to parallelize the training, πSVM parallelizes the solving of the quadratic problem itself. Because of this the accuracy of πSVM does not drop with an increasing number of processes, but the parallelization is much more complex and it does not scale as well as the cascade SVM. Like the previous benchmark this one was made on data set B^*.

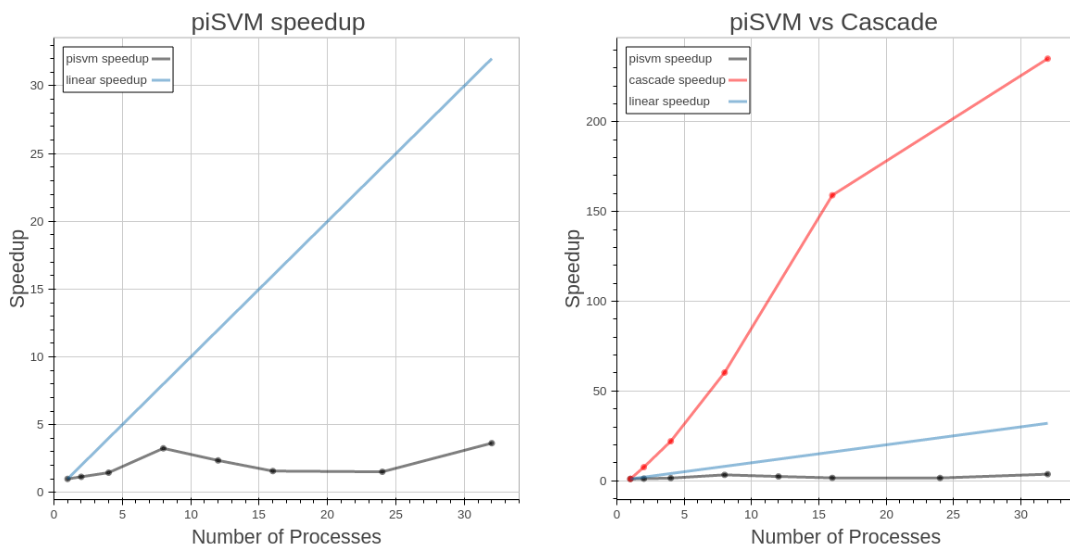


Figure 6.7: πSVM speedup with an increasing number of processes.

Figure 6.7 shows the speedup of the πSVM with up to 32 processes. The speedup is lower than the one of the cascade SVM. This can be clearly seen at the right image of the Figure. One reason is that while the cascade SVM can easily filter out non-SV quickly. It this also decreases the handling of them in further layers, however this has a side effect, that a high number of cores are effected by load imbalance while the πSVM has a rather constant load balancing scheme. In return the accuracy achieved by the cascade SVM drops if more processes are used, while the accuracy of πSVM stays the same.

6.7 Summary

In this chapter the cascade SVM framework of this thesis is evaluated. At first the I/O speed is measured when the libSVM format and the HDF5 format are used. The benchmark shows that HDF5 is faster for dense as well as sparse data sets. If the data does not have to be human readable, HDF5 is the preferred format, thus indirectly taking advantage of parallel I/O.

The benchmark on data set A (small) shows that a single iteration of the cascade SVM has a speedup, which is super linear for a few number of processes and drops if more processes are used. If two or three iterations over the cascade are used the speedup drops even more, so that it is not reasonable to use multiple iteration at all for small data sets. For a larger problem like B^* the benchmark shows that the cascade SVM scales to a higher number of processes and also for more iterations. The cascade SVM has a good performance for larger data sets, that proves our expectations that the cascade SVM framework is well designed for upcoming big data sets (e.g. in neuro sciences with higher number of pixels).

The benchmark on cross feedback shows that it can be used to enable a better accuracy while having a better training time than a whole additional iteration would need. An additional method to decrease the training time is the distance filter, which can be used to regulate the number of SVs that are used as feedback. An evaluation with a simple distance filter shows that the accuracy increases if more SVs are used while the speedup decreases.

At the end the cascade SVM is compared to another parallel approach πSVM, which is also currently developed at the JSC. The benchmark shows that the cascade SVM framework has a much better speedup on data set B^*. In return the accuracy achieved by the cascade SVM drops if more processes are used, while the accuracy of πSVM stays the same.

Chapter 7
Conclusion

More and more large data problems arise in the different scientific and non scientific fields (e.g. neuro sciences with high number of pixels for high resolution post-mortem brain scans) . SVMs are a popular tool to solve classification tasks. In this thesis a framework is proposed that implements the cascade SVM parallelization approach for HPC clusters and simple computers. The parallel communication is based on MPI and the main programming language used is Python. This way the framework is usable for non-technical savvy users.

One domain that generates large data problems is the remote sensing field. Images taken by satellites are to be classified according to the different land cover types and future data sets from satellites expect to have much better resolutions and a higher number of bands going from multi-spectral to hyper-spectral bands. A real world problem from this domain was used to evaluate the proposed research questions regarding the performance and accuracy of a cascade SVM. The evaluation shows that a cascade SVM provides a speedup for smaller as well as larger data sets. While the cascade SVM does not scale for many processes on the smaller data set, tests on the larger set show that the cascade SVM in general does scale for large data sets. However, the huge speedup comes with a loss in accuracy. The first research question 1 (cf. Section 2.5) was about the scalability of an MPI based cascade SVM. The evaluation has shown that the framework designed in this thesis is scalable for larger data sets and multiple processes.

Different improvements were made and evaluated to increase the accuracy and speedup. At first a standardized and parallel I/O was introduced to accomplish a better performance on cluster systems and thereby answering research question 2. Apart from that cross feedback is used to improve the accuracy without losing as much performance as a normal reiteration causes. The evaluation showed that the accuracy is between a single cascade run and a cascade with multiple runs. Furthermore it partly addresses the load imbalance problem stated in research question 3. The same goes for the speedup. The speedup can be further improved by using an additional filter function for the cross feedback which is related to research question 4. The framework is flexible and can easily integrate other filter options.

In general it can be seen, that a speedup in training time is achieved, but a trade off between training time and accuracy has to be made. Future research can be made, so that the drop in accuracy is not as large as currently. Comparing the cascade SVM to other parallel approaches like πSVM shows that cascade SVM outperforms πSVM in terms of training speed but with the expense of loosing accuracy.

7.1 Future Work

There are some possible features that can be implemented and investigated in the future. At the moment cascade SVM only partially supports multiclass problems. While a cascade iteration is possible, the convergence check is limited to a binary problem. Aside from that the multiclass strategy (e.g. One vs One) can be parallelized to achieve a better performance.

While the cascade SVM per cascade itself is currently based on a serial SVM, this may be changed. Instead of using a serial SVM on every node, a parallel SVM like πSVM can be used to accomplish another parallelization layer and a better speedup or by using shared memory approaches within the node. Due to the time limitations this was no option for this thesis. Another speedup possibility is to create a hybrid code leveraging the power of openMP and shared memory together with the already achieved MPI parallelization.

Bibliography

[1] http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/supercomputers_node.html.

[2] Hdf5 users. https://www.hdfgroup.org/HDF5/users5.html.

[3] πsvm homepage. http://pisvm.sourceforge.net/.

[4] Apache Mahout, http://mahout.apache.org.

[5] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics, 2007.

[6] Blaise Barney et al. Introduction to parallel computing. *Lawrence Livermore National Laboratory*, 6(13):10, 2010.

[7] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (xml). *World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210*, page 16, 1998.

[8] Dominik Brugger. Parallel support vector machines. 2006.

[9] Christopher JC Burges. A tutorial on support vector machines for pattern recognition. *Data mining and knowledge discovery*, 2(2):121–167, 1998.

[10] G. Cavallaro, M. Riedel, J.A. Benediktsson, M. Goetz, T. Runarsson, K. Jonasson, and T. Lippert. Smart data analytics methods for remote sensing applications. In *Geoscience and Remote Sensing Symposium (IGARSS), 2014 IEEE International*, pages 1405–1408, July 2014.

[11] 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.

[12] Edward Y. Chang, Kaihua Zhu, Hao Wang, Hongjie Bai, Jian Li, Zhihuan Qiu, and Hang Cui. Psvm: Parallelizing support vector machines on distributed

computers. In *NIPS*, 2007. Software available at http://code.google.com/p/psvm.

[13] Adrian Cho. Breakthrough of the year. the discovery of the higgs boson. *Science (New York, NY)*, 338(6114):1524–1525, 2012.

[14] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[15] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, January 1998.

[16] Lisandro Dalcín, Rodrigo Paz, Mario Storti, and Jorge D'Elía. Mpi for python: Performance improvements and mpi-2 extensions. *J. Parallel Distrib. Comput.*, 68(5):655–662, May 2008.

[17] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[18] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: A runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 810–818, New York, NY, USA, 2010. ACM.

[19] Charles Elachi and Jakob J Van Zyl. *Introduction to the physics and techniques of remote sensing*, volume 28. John Wiley & Sons, 2006.

[20] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[21] Message P Forum. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

[22] Michael Franklin, Joseph Gonzalez, Michael I. Jordan, Xinghao Pan, Virginia Smith, Evan Sparks, Ameet Talwalkar, Shivaram Venkataraman, and Matei Zaharia. Mllib, 2013. http://mloss.org/software/view/516/.

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

[24] Keinosuke Fukunaga. *Introduction to statistical pattern recognition*. Academic press, 2013.

[25] Markus Götz, Matthias Richerzhagen, Christian Bodenstein, Gabriele Cavallaro, Philipp Glock, Morris Riedel, and Jón Atli Benediktsson. On scalable data mining techniques for earth science. *Procedia Computer Science*, 51:2188–2197, 2015.

[26] Hans P Graf, Eric Cosatto, Leon Bottou, Igor Dourdanovic, and Vladimir Vapnik. Parallel support vector machines: The cascade svm. In *Advances in neural information processing systems*, pages 521–528, 2004.

[27] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009.

[28] Robert Henderson. Job scheduling under the Portable Batch System. In Dror Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pages 279–294. Springer Berlin / Heidelberg, 1995.

[29] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[30] Brian W. Kernighan. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[31] Siamak Khorram, Frank H Koch, Cynthia F van der Wiele, and Stacy AC Nelson. *Remote sensing*. Springer Science & Business Media, 2012.

[32] Sebastian Lührs. JUBE - A Flexible, Application- and Platform-Independent Environment for Benchmarking. Cy-Tera/LinkSCEEM HPC Administrator Workshop, Nicosia (Cyprus), 01/19/2015 - 01/21/2015.

[33] Snir Marc, S Otto, Steven Huss-Lederman, D Walker, and Jack Dongarra. Mpi: the complete reference. *http:'/www. netiib. org/utk/papers/mpi-book/mpi-book. html*, 1996.

[34] Viktor Mayer-Schnberger. *Big Data: A Revolution That Will Transform How We Live, Work and Think. Viktor Mayer-Schnberger and Kenneth Cukier*. John Murray Publishers, UK, 2013.

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

[36] Fernando Pérez and Brian E. Granger. IPython: a system for interactive

scientific computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007.

[37] John Platt et al. Fast training of support vector machines using sequential minimal optimization. *Advances in kernel methods-support vector learning*, 3, 1999.

[38] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014.

[39] A. Rahimi and B. Recht. Random features for large-scale kernel machines. http://www.eecs.berkeley.edu/~brecht/papers/07.rah.rec.nips.pdf.

[40] Armin Rigo and Samuele Pedroni. Pypy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 944–953. ACM, 2006.

[41] Guido Rossum. Python reference manual. Technical report, Amsterdam, The Netherlands, The Netherlands, 1995.

[42] Yakov Shafranovich. Common format and mime type for comma-separated values (csv) files. 2005.

[43] Tom Shanley. *Infiniband*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[44] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[45] Volker Springel. The cosmological simulation code gadget-2. *Monthly Notices of the Royal Astronomical Society*, 364(4):1105–1134, 2005.

[46] Zhanquan Sun and Geoffrey Fox. Study on parallel svm based on mapreduce. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 16–19. Citeseer, 2012.

[47] The HDF Group. Hierarchical Data Format, version 5, 1997-NNNN. http://www.hdfgroup.org/HDF5/.

[48] Jesper Larsson Träff. Implementing the mpi process topology mechanism. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 28–28. IEEE, 2002.

[49] Stefan van der Walt, S. Chris Colbert, and Gael Varoquaux. The numpy array: A structure for efficient numerical computation. Computing in Science & Engineering 13, 22-30, 2011.

[50] Vladimir Naumovich Vapnik and Samuel Kotz. *Estimation of dependences based on empirical data*, volume 40. Springer-verlag New York, 1982.

[51] Christopher Williams and Matthias Seeger. Using the nyström method to speed up kernel machines. In *Advances in Neural Information Processing Systems 13*, pages 682–688. MIT Press, 2001.

[52] Jing Yang. An improved cascade svm training algorithm with crossed feedbacks. In *Computer and Computational Sciences, 2006. IMSCCS'06. First International Multi-Symposiums on*, volume 2, pages 735–738. IEEE, 2006.

[53] Tianbao Yang, Yu-feng Li, Mehrdad Mahdavi, Rong Jin, and Zhi-Hua Zhou. Nyström method vs random fourier features: A theoretical and empirical comparison. In F. Pereira, C.J.C. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 476–484. Curran Associates, Inc., 2012.

[54] Hao Yu, Ramendra K Sahoo, C Howson, G Almasi, JG Castaňos, Manish Gupta, José E Moreira, JJ Parker, TE Engelsiepen, Robert B Ross, et al. High performance file i/o for the blue gene/l supercomputer. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 187–196. IEEE, 2006.

[55] 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*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[56] Jian-Pei Zhang, Zhong-Wei Li, and Jing Yang. A parallel svm training algorithm on large-scale classification problems. In *Machine Learning and Cybernetics, 2005. Proceedings of 2005 International Conference on*, volume 3, pages 1637–1641. IEEE, 2005.

Appendix A

A Evaluation Tables

A.1 Tables for Data Set A

# Cascade Iterations	# processes	mean training time [s]	max training time [s]	min training time [s]
1	1	321.97	329.87	318.39
1	2	86.53	108.87	77.81
1	4	51.71	67.16	47.65
1	8	34.18	45.17	29.94
1	16	28.68	33.41	26.12
1	32	19.98	20.09	19.78
2	1	668.27	670.14	666.15
2	2	382.27	407.41	286.64
2	4	256.56	300.84	245.26
2	8	254.69	299.03	243.51
2	16	375.09	394.99	365.87
2	32	329.75	333.14	328.58
3	1	987.47	993.38	983.25
3	2	558.11	663.78	454.94
3	4	427.11	457.97	419.28
3	8	439.03	469.81	431.17
3	16	545.19	629.65	467.69
3	32	300.69	315.64	296.76

Table A.1: Training Time on data set A

A.2 Tables For Data Set B

# Cascade Iterations	# processes	test time [s]	accuracy
1	1	83.76	0.925
1	2	31.45	0.869
1	4	15.14	0.884
1	8	6.79	0.853
1	16	5.06	0.764
1	32	1.82	0.707
2	1	96.26	0.915
2	2	32.67	0.913
2	4	16.23	0.912
2	8	8.03	0.906
2	16	5.87	0.910
2	32	2.91	0.859
3	1	95.71	0.915
3	2	32.93	0.914
3	4	17.31	0.915
3	8	8.22	0.913
3	16	4.18	0.913
3	32	3.64	0.859

Table A.2: Accuracy on data set A

# Cascade Iterations	# processes	mean training time [s]	max training time [s]	min training time [s]
1	1	179.10	201.07	161.80
1	2	72.49	74.64	71.61
1	4	61.58	61.80	61.38
1	8	19.33	19.46	19.28
1	16	11.29	11.48	11.21
1	32	5.91	6.17	5.78
2	1	372.06	397.15	294.72
2	2	265.93	267.37	265.05
2	4	153.15	153.40	153.05
2	8	124.81	132.80	119.93
2	16	132.47	134.65	131.81
2	32	16.78	16.97	16.55
3	1	493.82	600.04	390.62
3	2	476.28	477.82	475.03
3	4	344.48	373.51	308.06
3	8	256.24	271.74	252.16
3	16	320.49	360.75	294.85
3	32	27.61	28.50	26.26

Table A.3: Training Time on Set B

# Cascade Iterations	# processes	test_time	accuracy
1	1	35.68	0.973
1	2	20.30	0.953
1	4	8.23	0.931
1	8	2.34	0.892
1	16	3.68	0.850
1	32	0.73	0.770
2	1	28.90	0.973
2	2	19.76	0.971
2	4	7.12	0.955
2	8	2.84	0.936
2	16	1.78	0.935
2	32	2.77	0.770
3	1	29.15	0.973
3	2	21.25	0.973
3	4	7.05	0.971
3	8	3.25	0.965
3	16	4.49	0.959
3	32	2.53	0.770

Table A.4: Accuracy on Set B

# Cascade Iterations	# processes	training time [s]	test time [s]	accuracy
1	1	21257.62	19.67	0.985
1	2	2816.49	8.25	0.967
1	4	967.18	3.65	0.930
1	8	352.64	2.18	0.888
1	16	133.64	0.71	0.834
1	32	90.39	0.24	0.814
2	2	-	-	-
2	4	-	-	-
2	8	-	-	-
2	16	2047.51	1.10	0.946
2	32	1925.96	0.57	0.925
3	2	-	-	-
3	4	-	-	-
3	8	-	-	-
3	16	-	-	-
3	32	5422.09	0.65	0.953

Table A.5: Training and Testing Time on Set B^*. Missing entries terminated after 12 hours.

tasks	training time	test_time	accuracy
4	2043.62	3.64	0.935
8	699.21	1.47	0.903
16	426.96	0.82	0.863
32	136.83	0.26	0.843

Table A.6: Training and Testing Time with Feedback on Set B^*.

Abbreviations

CSV	Comma-separated values
FSD	Federated Systems and Data
HDF5	Hierarchical Data Format
HPC	High Performance Computing
HTC	High Throughput Computing
JSC	Juelich Supercomputing Centre
MPI	Message Passing Interface
PBS	Portable Batch Scheduler
PCA	Principle Component Analysis
rbf	radial basis function
SMO	Sequential Minimal Optimization
SPMD	Single Program Multiple Data
SV	support sector
SVM	Support Vector Machine
XML	Extensible Markup Language