Making Kernel Machines Scalable Combining Matrix Approximations and Distributed Computing

Author
Dr. Jakob L.K. Gerstenlauer

Supervisor
Dr. Lluís A. Belanche Muñoz

April 8, 2018
Abstract

Kernelized machine learning methods are able to predict highly nonlinear relationships for both numeric input data and heterogeneous symbolic data types such as strings, protein and DNA sequences, images, and graphs. Established algorithms such as interior point algorithms and sequential minimal optimization are very accurate and fast, but they do not scale to big data situations where data processing has to take place in a distributed system. In this work, a stochastic gradient descent algorithm was developed that solves the kernelized support vector machine in the dual formulation. This core algorithm can be used in a local parallelized and two different distributed implementations. The parallelized version implements a lean sparse matrix representation using hash maps and is able to concurrently calculate partial gradients. The two distributed versions use distributed matrices provided by Apache Spark to scale out the evaluation of the gradient and the evaluation of the model coefficients on the training and validation set.

Apart from the algorithm that learns the support vector machine, additional tools are provided: First, a subsection selection heuristic that uses an efficient projection method to filter instances. Second, a set of representative kernels and a heuristic for finding an appropriate value for the Gaussian kernel. Third, a visualization tool that helps to evaluate the quality of the classifier and decide on a decision threshold based on a receiver operator characteristic (ROC) curve for the validation set. And finally, a general cross-validation scheme that i) implements an early stopping strategy by selecting the model coefficients from the optimal iteration, ii) finds the optimal level of sparsity, iii) and helps to decide on a useful decision threshold. Asynchronous programming principles are used, to accelerate the computations and facilitate parallelization.

Results on a data set with 120,000 instances show that accuracies close to the LibSVM benchmark library (63.54% versus 65.88%) can be achieved, while considerably reducing computation time from 1h and 46m for the evaluation of the training set alone to 40m for the complete data analysis process (evaluation of training, validation, and test set). On a smaller empirical data set with 19,020 instances, the accuracy was only slightly lower (82.95% versus 84.31%), but the new method offered the advantage of flexibly deciding on a decision threshold. Using the subset selection heuristic, it was possible to process a a synthetic data set with 300,000 instances in the local implementation. Our results indicate, that stochastic gradient descent can be successfully applied to kernelized support vector machines. Based on the open source Scala library developed in this work, local, parallelized, and distributed implementations of the kernelized support vector machine can be learned from a diverse range of data sets.
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td></td>
<td>i</td>
</tr>
<tr>
<td>List of Figures</td>
<td></td>
<td>iv</td>
</tr>
<tr>
<td>List of Tables</td>
<td></td>
<td>iv</td>
</tr>
<tr>
<td>1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>Objectives</td>
<td>2</td>
</tr>
<tr>
<td>1.3</td>
<td>Structure</td>
<td>3</td>
</tr>
<tr>
<td>1.4</td>
<td>Glossary</td>
<td>4</td>
</tr>
<tr>
<td>1.5</td>
<td>Notation</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>Machine Learning Concepts</td>
<td>6</td>
</tr>
<tr>
<td>2.1</td>
<td>Evaluation of Classifiers</td>
<td>6</td>
</tr>
<tr>
<td>2.1.1</td>
<td>How to learn useful Models?</td>
<td>6</td>
</tr>
<tr>
<td>2.1.2</td>
<td>The Bias-variance Trade-off</td>
<td>8</td>
</tr>
<tr>
<td>2.1.3</td>
<td>Regularization and Shrinkage</td>
<td>8</td>
</tr>
<tr>
<td>2.1.4</td>
<td>The Sensitivity-specificity Trade-off</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Kernel Methods</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>Support Vector Machines for Classification</td>
<td>12</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Hard Margin Support Vector Machines</td>
<td>12</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Soft Margin Support Vector Machines</td>
<td>12</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Duality</td>
<td>13</td>
</tr>
<tr>
<td>2.3.4</td>
<td>Kernelized Support Vector Machines</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>Related Work</td>
<td>15</td>
</tr>
<tr>
<td>3.1</td>
<td>Algorithms and Strategies for Support Vector Machines</td>
<td>15</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Overview of Algorithms</td>
<td>16</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Overview of Strategies</td>
<td>17</td>
</tr>
<tr>
<td>3.1.3</td>
<td>Gradient Descent</td>
<td>19</td>
</tr>
<tr>
<td>3.1.4</td>
<td>Conjugate Gradient Descent</td>
<td>19</td>
</tr>
<tr>
<td>3.1.5</td>
<td>Instance Selection Strategies</td>
<td>20</td>
</tr>
<tr>
<td>3.2</td>
<td>Distributed Algorithms for Support Vector Machines</td>
<td>21</td>
</tr>
<tr>
<td>3.2.1</td>
<td>The Challenges of exact distributed Algorithms</td>
<td>21</td>
</tr>
<tr>
<td>3.2.2</td>
<td>The Map-Reduce Programming Model</td>
<td>23</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Limitations of Map-Reduce</td>
<td>25</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Principles for efficient distributed Algorithms</td>
<td>25</td>
</tr>
<tr>
<td>3.2.5</td>
<td>Map-Reduce Formalism for Matrix Vector Multiplications</td>
<td>26</td>
</tr>
</tbody>
</table>
4 Materials and Methods
   4.1 Subset Selection Heuristic .............................................. 27
   4.2 Implemented Kernel Functions ........................................ 27
      4.2.1 Scaling of Data Matrices ........................................ 28
   4.3 Details of the algorithmic Implementation .......................... 28
      4.3.1 Gradient Descent and Refinements ............................... 29
   4.4 Cross-validation of Number of Iterations and Sparsity ............ 31
      4.4.1 Centralized Algorithm .......................................... 31
      4.4.2 Distributed Algorithms ........................................ 31
   4.5 Tuning the Decision Threshold based on Scores ..................... 31
   4.6 Computational Details ................................................ 32
      4.6.1 Hardware Specification ......................................... 32
      4.6.2 Class Hierarchy ................................................ 33
      4.6.3 Decoupling Data Size from Memory Footprint ................. 34
      4.6.4 Multithreading and asynchronous Programming ............... 35

5 Working Examples of How to use the Scala Library .................... 37
   5.1 Data Sets ............................................................... 37
      5.1.1 Synthetic Data Sets .............................................. 37
      5.1.2 Validation on Empirical Data Sets .............................. 39
   5.2 Local Implementation .................................................. 39
      5.2.1 Probing the Size of the Kernel Matrices ....................... 39
      5.2.2 Running a local Support Vector Machine on simulated Data .. 39
      5.2.3 Running a local Support Vector Machine on empirical Data Sets 40
   5.3 Distributed Implementations ........................................... 40

6 Results .............................................................................. 43
   6.1 Synthetic Data Set ....................................................... 43
   6.2 Magic Data Set .......................................................... 45
   6.3 HIGGS Data Set ........................................................ 47

7 Conclusions and Future Work ............................................... 50

Acknowledgments ..................................................................... 52

References ............................................................................. 52

8 Code Listings ...................................................................... 58
   8.1 Listing 1 ........................................................................ 58
   8.2 Listing 2 ........................................................................ 60
   8.3 Listing 3 ........................................................................ 61
List of Figures

1. Map-Reduce Programming Model. ................................................. 24
2. ROC Curve Magic Data Set ...................................................... 32
3. UML Class Diagram .............................................................. 33
4. Workflow Graph for distributed Algorithm ................................. 38
5. ROC Curve Synthetic Data Set .................................................. 44
6. ROC Curve Synthetic Data Set .................................................. 45
7.Profiler Screenshot for Magic Data Set ...................................... 47
8. ROC Curve Higgs Data Set ...................................................... 49
9. Profiler Screenshot for Higgs Data Set ...................................... 49

List of Tables

1. Glossary .................................................................................. 4
2. Table of Symbols. ................................................................... 5
3. Confusion Matrix for a Binary Classification Problem .................. 10
4. Overview of Classification Quality Indices ................................. 10
5. Confusion Matrix Magic Data Set using Kernlab ......................... 45
6. Confusion Matrix Magic Data Set Using the Scala Library .......... 46
7. Confusion Matrix Higgs Data Set using Kernlab ......................... 47
8. Confusion Matrix Higgs Data Set using the Scala Library .......... 48
1 Introduction

1.1 Motivation

Linear machine learning methods with an inner-product formulation can be combined with kernel functions describing the similarity between data points [60, 34, 45]. Support vector machines, relevance vector machines, ridge regression, principal component analysis, and Fisher Discriminants have all been successfully kernelized [53]. Such “kernelized” machine learning methods show high rates of predictive accuracy when combined with an appropriate kernel function. Kernel functions have been derived for diverse categories of data, such as numeric values, bit sets, bag of words, strings, protein and DNA sequences, images, and graphs.

There are two major obstacles to the application of kernelized machine learning methods (i.e. kernel machines) to today’s big data challenges. First, their association to huge kernel matrices. The kernel matrix grows quadratically with the number of observations, i.e. instances. As a result, kernel machines do not scale very well to data sets with many observations. Secondly, commonly used algorithms for kernelized support vector machines are efficient and very accurate for small to medium sized data sets but impossible to parallelize or distribute. The most popular optimization approach to kernelized support vector machines, sequential minimal optimization, is fast, accurate, and has good scaling properties, it is however intrinsically difficult to parallelize and distribute. In this work, we followed a different approach, developing gradient descent algorithms which have successfully been applied to a wide range of distributed machine learning applications such including deep neural networks and linear support vector machines. Although gradient descent algorithms are more difficult to tune, they facilitate parallelization and distribution by averaging partial gradients.

Conceptually, there are two main approaches to make an algorithm scalable: First, to parallelize it locally, which is usually referred to as horizontal scaling in the computer science literature and might involve several CPUs or GPUs with direct access to a common storage structure. Second, to distribute the algorithm over several computing nodes, which is usually referred to as vertical scaling and is associated with distributed file systems such as the Hadoop File System [56] or distributed data bases. In the case of kernel machines, most efforts have been focused on a third approach, which is to approximate the kernel matrix in order to reduce both its CPU and memory footprint in order to be able to process the data locally [58, 57]. Ideally, all three approaches should be combined: First of all, approximations of the kernel matrix help to reduce the size of the kernel matrix. Secondly, depending on the final size of the approximated matrix, the kernel matrix is either stored locally or in a distributed file system. Finally, the processing of the kernel matrix takes either place locally using a
parallelized approach or by way of a distributed algorithm. In order to implement scalable algorithms, the same code should allow for both locally parallelized and if necessary distributed execution.

Parallelization and distribution of computations are greatly facilitated by use of the functional programming style. Scala is a statically typed programming language which allows for both imperative and pure functional programming [48]. It is a Java Virtual Machine language with a performance similar to Java while, when used in combination with linear algebra libraries such as Breeze [31], allowing for a very concise and clean syntax similar to scientific programming languages such as R, Julia or Matlab. In Scala there is, furthermore, a rapidly expanding developer community around the cluster computing engine Spark [72], the associated distributed machine learning library MLlib [24], and other machine learning libraries with a Scala API such as Smile [39] and the Scala NLP project [31] which facilitates the development of new, distributed machine learning algorithms.

1.2 Objectives

The main objective of this work is to show that algorithms based on gradient descent can be used to solve the kernelized support vector machine in a parallelized and distributed way. We furthermore wanted to facilitate the use of kernelized support vector machines for binary classification providing a toolbox tailor-made for big data applications where the standard data science workflow of separately tuning hyper-parameters using k-fold cross-validation is not feasible any more because of computational limitations. We thus seek to achieve the following objectives:

- Develop a parallel implementation of stochastic gradient descent for kernelized support vector machines for binary classification.

- Use horizontal scaling and intelligent sampling of the kernel matrix to mitigate the CPU footprint of processing increasing matrix sizes.

- Provide a Receiver Operator Curve visualization and flexible threshold choice to allow tuning of the decision threshold.

- Provide automatic kernel parameter estimation for the Gaussian kernel to facilitate the use of binary support vector machines for big data sets.

- Develop automatic cross-validation to avoid overfitting of the gradient descent algorithm.

- Create a prototype, distributed version of the algorithm that uses distributed matrices of the Spark cluster computing library.

The final Scala library is publicly available on github[26].
1.3 Structure

**Section 1** The general motivation of the study, as well as the overall structure of the document and mathematical notation are presented.

**Section 2** Relevant machine learning concepts, such as cross-validation, the bias-variance trade-off, regularization and different ways of evaluating classifiers are introduced. After a short overview of Kernel methods in general, the support vector machine is introduced. This subsection starts with the primal formulation for linear support vector machines and ends with the mathematical derivation of the kernelized support vector machine which is the model of interest in this work.

**Section 3** Here, related work is presented. First, we give an overview of different algorithmic approaches to support vector machines. Given the diversity of algorithms and strategies employed, it is impossible to go in much detail here, except for conjugate gradient descent and feature selection, two of the approaches which have been used in this study. Next, distributed algorithms for support vector machines are discussed. This topic warrants a separate subsection because the reader has to be introduced to some generic problems and principles in distributed computing and because it includes a discussion of the problems of an efficient and non-approximate distributed implementation of kernelized support vector machines.

**Section 4** First, the details of a local and a distributed algorithm are given. Then, details of a subset selection heuristic, of cross-validation principles, and tuning of decision-threshold are given. Then, we present the kernel functions implemented and a tuning heuristic for the kernel parameter of the Gaussian kernel. In section 4.6, computational details, such as the class hierarchy and the implementation of multithreading and asynchronous programming are presented. At the end, details concerning the creation of a synthetic data set and the processing of empirical data sets are presented.

**Section 5** Contains working examples for the usage of the implemented Scala library for the local and the distributed implementation.

**Section 6** We present empirical results for synthetic and empirical data sets comparing the new Scala library with the R *kernlab* package (based on *SVMlib*).

**Section 7** The results and the contribution of this work are discussed. We suggest implications and insights for future work.
### 1.4 Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>The categorical output in a classification problem (synonym: labels).</td>
</tr>
<tr>
<td>CPU</td>
<td>The standard processor(s) of a computing device (central processing unit).</td>
</tr>
<tr>
<td>distributed</td>
<td>Any algorithm or data structure that is processed on separate but interconnected computing nodes.</td>
</tr>
<tr>
<td>distributed system</td>
<td>Any set of computing devices which are interconnected via a network.</td>
</tr>
<tr>
<td>dual</td>
<td>Formulation of an optimization problem in instance space (2.3.3).</td>
</tr>
<tr>
<td>feature space</td>
<td>A possibly infinite dimensional space created by the mapping ( \phi ).</td>
</tr>
<tr>
<td>hard margin SVM</td>
<td>A formulation of the SVM that does not allow for misclassifications in the training set (2.3.1).</td>
</tr>
<tr>
<td>hyperplane</td>
<td>A plane in ( n ) (linear SVM) or ( d ) (non-linear SVM) dimensional space that (partly) separates instances from two classes.</td>
</tr>
<tr>
<td>inputs</td>
<td>The original variables or features which can be used to predict the output.</td>
</tr>
<tr>
<td>instances</td>
<td>Observations, i.e. cases or data rows, in a data set.</td>
</tr>
<tr>
<td>kernel function</td>
<td>A function that describes the similarity between instances (2.2). More technically it computes an inner product in feature space.</td>
</tr>
<tr>
<td>labels</td>
<td>The categorical output in a classification problem (synonym: class).</td>
</tr>
<tr>
<td>output</td>
<td>The response variable in a data set.</td>
</tr>
<tr>
<td>parallelization</td>
<td>Concurrent processing of algorithms on separate threads of execution on a single computing device.</td>
</tr>
<tr>
<td>primal</td>
<td>Formulation of an optimization problem in terms of the inputs.</td>
</tr>
<tr>
<td>regularization</td>
<td>Statistical technique to reduce the dimensionality of models (2.1.3)</td>
</tr>
<tr>
<td>ROC curve</td>
<td>The receiver operator characteristic curve (2.1.4).</td>
</tr>
<tr>
<td>sensitivity</td>
<td>Ability of a binary classifier to detect positive cases (2.1.4).</td>
</tr>
<tr>
<td>shrinkage</td>
<td>A statistical technique to shrink the model coefficients in order to reduce the dimensionality of models (2.1.3)</td>
</tr>
<tr>
<td>soft margin</td>
<td>A formulation of the SVM balancing misclassifications in the training set with the width of the margin (2.3.2).</td>
</tr>
<tr>
<td>specificity</td>
<td>Ability of a binary classifier to detect negative cases (2.1.4).</td>
</tr>
<tr>
<td>support vectors</td>
<td>Training set instances which contribute to the definition of the hyperplane and are thus part of the final SVM model.</td>
</tr>
<tr>
<td>SVM</td>
<td>The support vector machine.</td>
</tr>
<tr>
<td>thread</td>
<td>Independent computing process with its own memory and computing resources.</td>
</tr>
</tbody>
</table>

Table 1: A summary of important terms used throughout the text.
### 1.5 Notation

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( b )</td>
<td>The intercept term of the SVM hyperplane.</td>
</tr>
<tr>
<td>( d )</td>
<td>Number of original features, i.e. inputs, variables.</td>
</tr>
<tr>
<td>( e )</td>
<td>A unit vector, i.e. vector of ones.</td>
</tr>
<tr>
<td>( i )</td>
<td>Index of the gradient descent iteration.</td>
</tr>
<tr>
<td>( l )</td>
<td>Length ( n ) vector of labels (+1 for signal and -1 for noise).</td>
</tr>
<tr>
<td>( n )</td>
<td>Number of observations, i.e. instances, cases.</td>
</tr>
<tr>
<td>( p )</td>
<td>Sampling probability (stochastic gradient descent, feature selection).</td>
</tr>
<tr>
<td>( s )</td>
<td>Sample size of a subsample of the training set.</td>
</tr>
<tr>
<td>( t )</td>
<td>A scalar regularization parameter used in ridge regression and the Lasso.</td>
</tr>
<tr>
<td>( x )</td>
<td>A single instance, i.e. observation or data point with dimension ( d ).</td>
</tr>
<tr>
<td>( y )</td>
<td>Length ( n ) vector of numeric outputs in regression problems.</td>
</tr>
<tr>
<td>( z )</td>
<td>Projection of instances based on the ( \alpha ) of a subset of training set instances.</td>
</tr>
<tr>
<td>( A )</td>
<td>The ( n \times d ) data matrix.</td>
</tr>
<tr>
<td>( B )</td>
<td>A ( n \times m ) matrix containing ( m ) columns of the kernel matrix ( K ) (Nyström method).</td>
</tr>
<tr>
<td>( C )</td>
<td>A scalar regularization hyper-parameter of the support vector machine.</td>
</tr>
<tr>
<td>( I )</td>
<td>The identity matrix (a diagonal matrix with ones in the diagonal).</td>
</tr>
<tr>
<td>( K )</td>
<td>The ( n \times n ) kernel matrix.</td>
</tr>
<tr>
<td>( L )</td>
<td>A ( n \times n ) diagonal matrix with the labels ( l ) as diagonal.</td>
</tr>
<tr>
<td>( M )</td>
<td>A kernel matrix of a subsample of ( m ) out of ( n ) training instances (Nyström method).</td>
</tr>
<tr>
<td>( Q )</td>
<td>An ( n \times n ) matrix that includes the labels into ( K ).</td>
</tr>
<tr>
<td>( Z )</td>
<td>A non-square ( n \times s ) kernel matrix used to calculate the projection ( z ).</td>
</tr>
<tr>
<td>( \alpha )</td>
<td>The length ( n ) vector of dual variables of the support vector machine.</td>
</tr>
<tr>
<td>( \beta )</td>
<td>A length ( d + 1 ) vector of model coefficients including the intercept ( \beta_0 ).</td>
</tr>
<tr>
<td>( \gamma )</td>
<td>The spread parameter of the Gaussian kernel.</td>
</tr>
<tr>
<td>( \delta )</td>
<td>The tolerance parameter in terms of the ( L_1 ) distance between ( \alpha_i ) and ( \alpha_{i+1} ).</td>
</tr>
<tr>
<td>( \epsilon )</td>
<td>The sparsity threshold for the processing of kernel functions (a scalar).</td>
</tr>
<tr>
<td>( \eta )</td>
<td>The step size of gradient descent (a scalar).</td>
</tr>
<tr>
<td>( \mu )</td>
<td>In the context of the Lagrangian function, it denotes a Lagrange multiplier.</td>
</tr>
<tr>
<td>( \tau )</td>
<td>The momentum which is newly estimated at each iteration in conjugate gradient descent.</td>
</tr>
<tr>
<td>( \phi )</td>
<td>Mapping from input to feature space.</td>
</tr>
<tr>
<td>( \psi )</td>
<td>Decay rate of the learning rate and shrinkage of the out-of-batch samples.</td>
</tr>
<tr>
<td>( f(x) )</td>
<td>The objective function of the dual support vector machine.</td>
</tr>
<tr>
<td>( k(x,y) )</td>
<td>A valid kernel function.</td>
</tr>
<tr>
<td>( m(x) )</td>
<td>The midpoint function.</td>
</tr>
</tbody>
</table>

Table 2: A summary of mathematical symbols used throughout the text.
2 Machine Learning Concepts

2.1 Evaluation of Classifiers

In this section, we will introduce theoretical concepts related to model selection and evaluation which are central to statistics and machine learning and thus form the theoretical foundation for this work.

2.1.1 How to learn useful Models?

Learning models from the training set that generalize well to unknown data is the fundamental objective in machine learning and statistics [33, p.219]. Here, learning is equivalent to fitting the parameters, or coefficients, of a model. But how can we assess the ability of a model to generalize well to new data? We can expect statistical models to generalize well to unknown data, if the error on an independent subset of the data is low. This error is called test error or generalization error [33, p.220]. In this context, independence means that the test data must not be used in any way to extract information about the underlying probabilistic distributions. The model must not be trained on this data set, but the requirements are even more strict: Data analysis normally includes several data processing steps prior to the training process of a given model, such as e.g. transformations of inputs, outlier analysis, feature selection, and dimensionality reduction. All these data processing steps must also be conducted separately and independently for the learning set (training and validation set) and the test set. If not, the true generalization error may be considerably higher than expected based on the model error on the test set [33, p.245–247].

Most statistical and machine learning methods contain both coefficients and hyperparameters. Model coefficients are fitted by the model when training on a given training set. They are the decision variables of the underlying optimization problem. Hyperparameters, however, are fixed parameters which have to be supplied by the user. They might be parameters of the optimization algorithm itself, such as the step size in gradient descent algorithms, or regularization parameters that alter the formulation of the optimization problem (compare 3.1.3). In the context of kernelized support vector machines, the regularization parameter \( C \) and the kernel parameters (if applicable) are hyper-parameters. In order to find suitable values for these hyper-parameters, it is recommended to use cross-validation procedures. In order to cross-validate, the learning set has to be split into a training set and a validation set. To summarize, there are three different categories of data sets:

1. The training set is used to fit the model coefficients.
2. The validation set is used for model selection (tune hyper-parameters and com-
3. The test set is used to estimate the error of the final model on unknown data. In the literature, the terminology is blurry, because the term training set is being used in a broad and in a strict sense. Here, we use the term learning set to describe the combined training and validation set. The recommended practice is k-fold cross-validation, where the learning set is split into $k$ equal sized folds and the model is independently evaluated in $k$ replications: In each replication, one fold is set aside as validation set and the other folds are used as training set. At the end, the arithmetic mean of the error on the validation set is compared between different values for the hyper-parameters [33, p.241–245]. Typical values for $k$ are five or ten. In the context of big data applications, where model evaluations are very expensive and the size of training and validation sets are huge, we might reduce $k$ even to $k = 1$, if we tested beforehand that the variability between different estimates of the validation error is negligible. A typical workflow of a statistical data analysis project can thus be summarized as:

1. Separate the test data from the rest of the data set (training and validation).
2. Independently pre-process the test and the training (broad) set.
3. Tune the hyper-parameters of your model using k-fold cross-validation.
4. Fit the model parameters (coefficients) using the best hyper-parameter settings.
5. Evaluate the final model on the test set.

If feasible, it is recommended to replicate the whole process in order to retrieve a more robust estimate of hyper-parameters, coefficients, and the generalization error of the final model. This is the established approach to model selection and evaluation. However, there are situations, especially when the cost of training the model is very high in terms of computational resources, when alternative procedures are appropriate. One major drawback of the standard approach based on k-fold cross-validation is, that the number of cross-validation runs grows exponentially with the number of hyper-parameters, especially if the hyper-parameters interact [7, p.33]. This is one of the reasons why it is so important to be able to estimate at least some of the hyper-parameters directly from the training data or based on experience and a deeper understanding of the underlying optimization algorithm [28, p.415–424]. This will become relevant later, when we explain how we estimate the Gaussian kernel parameter using an approximation method. An alternative approach, that estimates hyper-parameters directly from the training data and does thus not depend on many nested cross-validation runs, is Bayesian model comparison [7, p.161-165]. This approach is interesting in its own right, but also because it is the basis of the relevance vector machine, a Bayesian alternative to the support vector machine[7, p.345–356] [65].
2.1.2 The Bias-variance Trade-off

After describing the general approach in machine learning in terms of the practical workflow and the different data partitions involved, we will now highlight the theoretical underpinnings. In statistics, there is a general theory of a trade-off between optimal fit, or minimal training error, on the one hand, and low generalization error, on the other hand. The idea is, that training more complex models tends to increase the probability of over-fitting the data and, that the “complexity” of models has to be chosen in accordance with the size and signal-to-noise ratio of the training data [33, p.219–227]. The term bias-variance trade-off describes the dilemma of either training models with low bias or high variance. Models with high variance tend to be highly dependent on the peculiarities of the training set. Models with low bias have a low deviance between the expected value of the model prediction and the true value. Regularization (2.1.3) can be seen as a way of reducing model variance by introducing some bias [33, p.224–225].

Measuring complexity of models is a difficult and ambiguous task, and there have been several approaches to quantify it as e.g. the effective degrees of freedom in the context of smoothing splines [33, p.153–154], the effective number of parameters [33, p.158–160,232–233], or based on the very general theory of Vapnik-Chervonenksi dimensions [33, p.237–241]. The urge to use such measures of effective numbers of parameters is based on the reliance on model selection criteria such as the Akaike Information Criterion [33, p.230–232] or the Bayesian Information Criterion [33, p.233–235], which directly calculate a model quality measure solely based on the empirical loss on the training set. There are, however, classes of models, such as kernelized support vector machines and artificial neural network, which operate in such a high-dimensional space and combine different regularization principles, that the concept of effective numbers of parameters is not useful any more: There is simply no way of quantifying it. In consequence, the empirical evaluation of the test error in k-fold cross-validation or an alternative Bayesian model comparison approach, are the only feasible approaches for such highly nonlinear models.

2.1.3 Regularization and Shrinkage

Based on the bias-variance trade-off, machine learning models have to be able tune their degree of “complexity” or “effective number of parameters”. There are standard approaches to restrict model complexity in a discrete way, e.g. by feature selection or dimensionality reduction methods. An alternative and more gradual approach is to work in the original high dimensional problem space but to shrink the coefficients of the model using regularization. Standard shrinkage regularization approaches are ridge regression, the Lasso, and intermediate elastic net regularization [33, p.61–73].
Compared with subset selection and a selection of significant dimensions using e.g. principal component analysis, regularization methods that shrink the coefficients are expected to show lower variance [33, p.61–73]. Ridge regression and the Lasso differ only in the way the norm of the coefficients is calculated, but this difference in the penalization term has significant consequences [28, p.223–230]. Ridge regression which uses the $L^2$ norm can be solved analytically while the Lasso ($L^1$ norm) constitutes a quadratic programming problem. While ridge regression never shrinks coefficients to zero, the Lasso leads to sparse solutions and thus implicitly runs feature selection when training on a data set. The same differences exists between least squares support vector machines [62, 18, 12] and the standard support vector machines which are using the Hinge loss. For completeness, here is the optimization problems solved by ridge regression in a form that is comparable to the support vector machine formulation [33, p.63]:

$$\hat{\beta} = \minimize_{\beta} \sum_{i=1}^{n} (y_i - \beta_0 - \sum_{j=1}^{d} x_{ij} \beta_j)^2$$

subject to $\sum_{j=1}^{d} \beta_j^2 \leq t$.

The equivalent formulation for the Lasso is [33, p.68]:

$$\hat{\beta} = \minimize_{\beta} \sum_{i=1}^{n} (y_i - \beta_0 - \sum_{j=1}^{d} x_{ij} \beta_j)^2$$

subject to $\sum_{j=1}^{d} |\beta_j| \leq t$.

Here, $y_i$ denotes the numeric output, i.e. the response, $x_{ij}$ denotes input $j$ of instance $i$, and $\beta_i$ the model coefficients. The model coefficients include the intercept as $\beta_0$. Here parameter $t$ is a regularization parameter that restricts model complexity. Note that ridge regression and the Lasso differ only in the way the constraint is formulated.

2.1.4 The Sensitivity-specificity Trade-off

Having described general machine learning concepts, we will now discuss possible ways to assess the classification error in binary classification problems. In classification problems with two classes, there are a number of indices describing model quality which can all be derived from the confusion matrix (compare Table 3 and 4). In many situations, evaluating models solely based on overall accuracy will result in poor model selection choices. In medicine, a false negative test result is in general more detrimental to a subject’s fate, than a false positive rate: Positive tests lead to a follow-up of additional tests and examinations which will with high likelihood reveal the test error. False negative test results for e.g. infectious diseases or terminal illnesses might,
on the other hand, have grave repercussions for the subject and the general population. Even if sensitivity and specificity (Table 4) have the same cost, in situations of highly unbalanced class distributions we might still want to tune the classification threshold or cutoff to balance off the false positive and false negative rates, because classifiers tend to have a higher accuracy for the over-represented class.

In order to be able to compare the prediction quality of classifiers over a range of possible decision thresholds, the technique of receiver operator characteristic (ROC) curves has been developed [33, p.316][32]. Based on the graphical representation of a ROC curve, or based on its numeric representation as the area under ROC curve, it is possible to make a general comparison of the ability of classifiers to classify well over all possible cutoff-values. By default, support vector machines for classification do not allow for such a flexible approach, that is why we developed a method to show ROC curves for the validation set and let the user choose a suitable cutoff value for the final model (compare 4.5).

<table>
<thead>
<tr>
<th>Predicted</th>
<th>True class</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>true positive(TP)</td>
</tr>
<tr>
<td>-</td>
<td>false positives(FP)</td>
</tr>
</tbody>
</table>

Table 3: Confusion Matrix for a Binary Classification Problem.

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>True positive rate</td>
<td>TP/(TP+FP)</td>
</tr>
<tr>
<td>False positive rate</td>
<td>FP/(TP+FP)</td>
</tr>
<tr>
<td>True negative rate</td>
<td>TN/(FN+TN)</td>
</tr>
<tr>
<td>False negative rate</td>
<td>FN/(FN+TN)</td>
</tr>
<tr>
<td>Sensitivity</td>
<td>TP/(TP+FN)</td>
</tr>
<tr>
<td>Specificity</td>
<td>TN/(FP+TN)</td>
</tr>
<tr>
<td>Accuracy</td>
<td>(TP+TN)/(TP+TN+FP+FN)</td>
</tr>
</tbody>
</table>

Table 4: Classification quality indices related to the confusion matrix (Table 3).

### 2.2 Kernel Methods

Although linear methods, which form the basis of classical statistics, have a long and successful history, there are many situations with non-linear relationships between inputs and outputs. As a consequence, numerous methods have been developed in statistics and machine learning that derive non-linear basis functions of the original
inputs (i.e. variables) and use these derived features to model non-linear relationships using linear methods [33, p. 139–141]. Using this trick, linear methods can be used to model non-linear relationships. Examples for this approach include non-linear transformations (square root, logarithm), (piecewise) polynomials, and splines of the original inputs. A major drawback of this approach is that an increase in dimensionality is always accompanied with an increased risk of overfitting. In order to mitigate overfitting, three different general strategies have been developed: First, restriction methods which limit the class of functions (e.g. only additive functions or only a certain number of basis functions. Second, selection methods which actively select a subset of basis functions. Third, regularization methods which use all basis functions but shrink the coefficients (e.g. ridge regression) [33, p. 140–141]. Kernel methods can be seen as a natural extension of the basis function approach: Instead of explicitly calculating the basis functions, kernel methods use the so-called kernel trick (or kernel property):

A kernel function $k(x,y)$ calculates the inner product in a potentially infinite feature space. The key advantage and strong point of kernel methods is, however, not the fact that they avoid explicitly operating in high dimensional feature space, but that the function space generated by the kernel function, a reproducing kernel Hilbert space, can be cast into a general regularization framework [33, p. 168–169]. Simply put this means, that a valid kernel function will implicitly regularize, when operating in the high-dimensional, and possibly infinite-dimensional, feature space. This point is explained in detail by Schölkopf and Smola (2002) by stressing that kernel functions are in fact regularization operators [53, p. 92–96]: “Thus Support Vector Machines are simply a very convenient way of specifying the regularization and a matching class of basis functions via one kernel function.”[53, p. 95].

Valid kernel functions are functions guaranteed to generate positive-semidefinite kernel matrices $K$ [54, p. 57–58]. Kernel functions can be combined to create new valid kernel functions following certain algebraic rules [54, p. 57–58][7, p. 296]. Linear machine learning methods can be kernelized if they can be expressed in inner product form. The transformation from a linear to a non-linear, kernelized, version of a given method involves replacing the standard linear inner product, or dot product, $x^\top x = \langle x, x \rangle$ by the inner product calculated in feature space based on the kernel function: $k(x,y) = \langle \phi(x), \phi(y) \rangle$. Here, $\phi(x)$ denotes the mapping from the original $d$ dimensional input space into the potentially infinite dimensional feature space. This inner product formulation of optimization problems naturally occurs in the Lagrangian which leads to the dual (compare 2.3.4). This has already been achieved for a long list of statistical methods, such as k-nearest neighbours, principal component, canonical correlation, Fisher discriminant, independent component, and linear discriminant analysis as well as clustering [54, 34]. The support vector machine is, however, the most popular kernel method and it was originally derived in its kernelized form.
2.3 Support Vector Machines for Classification

Given a binary classification problem with instances (i.e. data points) \(x_1, ..., x_n\), and a vector of class labels \(l\) with \(n\) elements \(l_i \in [-1, +1]\), we can find a hyperplane with dimension \(d\), weights \(w\), and intercept \(b\) that defines a classification model of the form: 
\[
\hat{l} = \langle w, x \rangle + b \quad \text{[53, p. 189][15, p.9–11].}
\]
The first model for computing such a separating hyperplane was Rosenblatt’s perceptron which is only guaranteed to converge if the data are linearly separable [15, p.11–19][52]. Other models for fitting such a hyperplane include Fisher’s discriminant [15, p.19] and the linear support vector machine [53, p. 190]. The linear support vector machine is constructing a canonical, i.e. scaled, hyperplane in the original input space with the additional objective of maximizing the geometric margin. Maximizing the geometric margin is key, because the margin is related to the ability of the model to generalize to new observations [53, p. 189–193]. There are a number of different formulations for linear support vector machines. Here, we will only list the formulations which are commonly referred to as C-SVM for classification.

2.3.1 Hard Margin Support Vector Machines

The hard margin version of the SVM does not allow any misclassification of instances from the training set. This model does only make sense for linearly separable data. The optimization problem for the hard margin SVM is formulated as:

\[
\begin{align*}
\text{maximize}_{w, b} & \quad \frac{2}{\|w\|^2} \\
\text{subject to} & \quad l_i (w^T x_i + b) \geq 1.
\end{align*}
\]

In words, the objective is to maximize the margin while correctly classifying all observations.

2.3.2 Soft Margin Support Vector Machines

The soft margin version of the SVM allows for misclassifications by introducing slack variables \(s\) and the crucial regularization hyper-parameter \(C\):

\[
\begin{align*}
\text{minimize}_{w, b, s} & \quad \frac{1}{2} w^T w + C \sum_{i=1}^{n} s_i \\
\text{subject to} & \quad l_i (w^T x_i + b) + s_i \geq 1.
\end{align*}
\]

In words, the objective is to maximize the margin while keeping the sum over all slack variables minimal. The hyper-parameter \(C\) controls the contribution of the penalty term that sums the values of the slacks. A small value of \(C\) is equivalent to a high degree of regularization because it leads to hyperplanes with a wider margin.
2.3.3 Duality

The above mentioned formulations for the linear SVM are primal optimization problems. Each optimization problem can be expressed in two complementary ways: as a primal and as a dual problem. Primal optimization problems are minimization problems formulated in the $d$ dimensional space of original inputs. Dual optimization problems are maximization problems formulated in the $n$ dimensional space of instances, i.e. observations. Dual variables, also known as Lagrange multipliers, describe how the constraints of the underlying quadratic optimization problem affect the objective function[15, p.88]. In statistics and machine learning, methods which are based on the solution of primal optimization problems are called parametric while methods based on the solution of dual optimization problems are called non-parametric. Primal and dual problem formulations are complementary, because they can be used to derive upper (primal) and lower (dual) bounds for the objective function. While primal and dual formulation are not in general equivalent, they lead to the same result for convex optimization problems with certain constraint qualifications (Slater constraints) [6, p.243,247]. This strong duality theorem is satisfied for the different primal and dual formulations of the support vector machine, it is therefore possible to work with the dual formulation in order to derive the solution for the primal optimization problem.

2.3.4 Kernelized Support Vector Machines

The kernelized SVM is a machine learning method based on the equivalent dual formulation of the primal SVM optimization problems we formulated above. This dual formulation is derived from the Karush-Kuhn-Tucker conditions for constrained nonlinear optimization problems [6, chapter 4][53, p.166]. We will now derive the dual formulation of the soft margin C-SVM based on lecture notes from a class on nonlinear optimization [11]. First, introducing a unit vector $e$, i.e. a vector of ones with length $n$, we replace the sum over all slack variables by an inner product formulation. Then, we introduce a $n \times d$ data matrix $A$ and a diagonal indicator matrix $L$ with $-1$ or $+1$ as diagonal elements, and $l = \text{Trace}(L)$. The primal constrained, nonlinear optimization problem can then be formulated in matrix form as:

\[
\begin{align*}
\text{minimize} & \quad \frac{1}{2} w^\top w + Ce^\top s \\
\text{subject to} & \quad -L(Aw - be) - s + e \geq 0 \quad s_i \geq 0, \forall i \in \{1, \ldots, n\}.
\end{align*}
\]

The Lagrangian $L$ with Lagrange multipliers $\alpha$ and $\mu$ of this constrained optimization problem is:

\[
L(w, b, s, \alpha, \mu) = \frac{1}{2} w^\top w + Ce^\top s + \alpha^\top (-L(Aw - be) - s + e) - \mu^\top s
\]
Using the KKT conditions based on the Lagrangian (equation 2), the Wolfe dual problem for the soft margin SVM is then given by:

\[
\begin{align*}
\text{maximize} & \quad \frac{1}{2}w^\top w + Ce^\top s + \alpha^\top (-L(Aw - be) - s + e) - \mu^\top s \\
\text{subject to} & \quad w - (\alpha^\top LA)^\top = 0 \\
& \quad \alpha^\top Le = 0 \\
& \quad Ce - \alpha - \mu = 0 \\
& \quad \alpha_i \geq 0, \mu_i \geq 0, \forall i \in \{1, \ldots, n\}. 
\end{align*}
\]

(3)

The Karush-Kuhn-Tucker optimality conditions state that all three partial derivatives of the Lagrangian have to be zero for the optimal solution [53, p.166]. From the first partial derivative, it follows that:

\[
w = (\alpha^\top LA)^\top
\]

(4)

Making use of this equivalence relation, we can eliminate \( w \) formulating the optimization problem only in terms of dual variables. We have, thus, successfully dualized the optimization problem [15, p.88].

\[
\begin{align*}
\text{maximize} & \quad \alpha^\top e - \frac{1}{2}\alpha^\top LAA^\top L\alpha \\
\text{subject to} & \quad \alpha^\top Le = 0 \\
& \quad Ce - \alpha - \mu = 0 \\
& \quad \alpha_i \geq 0, \mu_i \geq 0, \forall i \in \{1, \ldots, n\}.
\end{align*}
\]

Replacing the second Lagrange multiplier by \( \mu = Ce - \alpha \) and guaranteeing that \( \mu \geq 0 \), we can further simplify to:

\[
\begin{align*}
\text{maximize} & \quad \alpha^\top e - \frac{1}{2}\alpha^\top LAA^\top L\alpha \\
\text{subject to} & \quad \alpha^\top Le = 0 \\
& \quad 0 \leq \alpha_i \leq C, \forall i \in \{1, \ldots, n\}.
\end{align*}
\]

The matrix product \( AA^\top \) represents the gram matrix, i.e. kernel matrix, of the linear kernel. Using not the linear kernel \( k(x, y) = \langle x, y \rangle \), but any other valid kernel function \( k(x, y) \) [7, pp 294-299] and associated \( n \times n \) positive semidefinite kernel matrix \( K \) instead of \( AA^\top \), we have successfully kernelized the support vector machine. In order to simplify notation, we include the class labels in the kernel matrix and construct matrix \( Q = LAA^\top L \), or \( Q = LKL \) for any positive semidefinite kernel matrix \( K \).

\[
\begin{align*}
\text{maximize} & \quad \alpha^\top e - \frac{1}{2}\alpha^\top Q\alpha \\
\text{subject to} & \quad \alpha^\top Le = 0 \\
& \quad 0 \leq \alpha_i \leq C, \forall i \in \{1, \ldots, n\}.
\end{align*}
\]
In this matrix notation, it becomes clear that the objective function tries to maximize the $L^1$ norm of $\alpha$ while minimizing the $L^2$ norm in a metric defined by matrix $Q$. An analogous scalar formulation of this optimization problem is [16]:

$$\begin{align*}
\text{maximize} \quad & \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j l_i l_j k(x_i, x_j) \\
\text{subject to} \quad & \sum_{i=1}^{n} \alpha_i l_i = 0 \\
& 0 \leq \alpha_i \leq C, \forall i \in \{1, \ldots, n\}.
\end{align*}$$

Note the similarity and differences of this formulation to the optimization problem for the Lasso and ridge regression. Again, we have a regularization parameter that restricts the size of the decision variables. But this time, the size of individual decision variables $\alpha_i$ is restricted, while it was the $L^1$ or $L^2$ vector norm before. The main difference is, that the decision variables are now dual variables which represent the constraints of the underlying quadratic optimization problem. Based on the Karush-Kuhn-Tucker complementarity conditions, only active constraints have non-zero dual variables [15, p.88]. The associated instances are called support vectors in the context of support vector machines. All non-active constraints are associated with dual variables that are zero.

3 Related Work

3.1 Algorithms and Strategies for Support Vector Machines

There is a wide range of algorithms that have been developed to solve different formulations of the support vector machine [53, compare chapter 10] and these algorithms have been implemented in popular software packages such as SVMLib [13] and LIBLINEAR [21]. The two most popular families of algorithms, interior point algorithms [53, p.295–300] and sequential minimal optimization [53, p.305–314][27], are, however, intrinsically centralized algorithms which are difficult to parallelize and impossible to distribute on a computer network. Stochastic gradient descent [53, p.315–318], on the other hand, is difficult to tune and often less performing when executed on a single computing node. However, stochastic gradient descent algorithms can be successfully distributed over a network of computing nodes by averaging the gradient separately calculated on different batches of training instances which are located on different computing nodes. Recent advances in distributed machine learning algorithms for deep neural networks (e.g. TensorFlow [1, 4]) and general distributed computing cluster engines such as Apache Spark[72, 73] and associated machine learning libraries [43, 41] are all based on stochastic gradient descent.
3.1.1 Overview of Algorithms

Following the classification of Schölkopf and Smola (2002) and adding linear programming approximations, there are five categories of algorithms which can be used to solve support vector machine problems [53, p.279–281]:

1. **Interior point algorithms**: These are very reliable and accurate algorithms which have been successfully applied to small and moderately sized data sets. The main drawback of this class of algorithms is that they rely on storing and inverting a \( n \times n \) matrix \( Q \) which becomes computationally prohibitive with increasing \( n \). The only remedy for this problem can be found in low-rank approximations of \( Q \) for big data sets [53, p.295–300].

2. **Subset selection algorithms**: This is actually rather a meta-algorithm than a specific algorithm because the main idea is to break up the training set into many small subsets and train a given algorithm on each subset separately. In the chunking strategy, only the support vectors are kept after processing a given subset. This strategy is only successful if the number of support vectors is relatively small (the kernel matrix for all support vectors has to fit into memory). A more versatile approach is based on working set algorithms which select a small subset of instances and runs a given algorithm on this small subset. The key to the success of these algorithms are the selection strategies they employ which may take into account the constraints defined by the Karush-Kuhn-Tucker condition, the gradient at specific instances, or class balance [53, p.300–304].

3. **Sequential Minimal Optimization** [49]: This algorithm can be seen as the most extreme form of subset selection: only two instances are selected in each iteration. The basic idea is to find the two instances violating the Karush-Kuhn-Tucker conditions the most, correcting their dual variables \( \alpha_i \), and selecting the next subset of two. This algorithm is very popular because it has a very benign memory footprint, is easy to implement, and is quite fast [53, p.305–314]. As a result, most standard software packages (i.e. the R *kernlab* package or the Scala *Smile* library) use an implementation of this algorithm.

4. **Iterative methods**: Namely the Kernel Adatron algorithm and derived algorithms for online learning are iterative methods based on gradient descent [54, p. 241-250]. These algorithms can work both in the primal and the dual space (i.e. feature space) [53, p.315]. The advantage of gradient descent algorithms is that they only need a formulation to calculate the gradient and can then relatively easily be parallelized by calculating partial gradients on separate batches, i.e. subsets of the training set. Online learning methods [14, 36, 37] are particularly
interesting for streaming applications where new data is arriving continuously and the model itself is continuously updated.

5. **Linear programming approximations**: The support vector machine optimization problem as described above is a quadratic programming problem. It is however possible to approximate this problem by converting it into a linear program. Least-squares support vector machines replace the Hinge loss with quadratic loss and solve a linear programming problem analytically in the dual [62, 18, 12]. As a result of using the quadratic loss function, these models do not result in sparse solutions for $\alpha$.

### 3.1.2 Overview of Strategies

The above overview of four different classes of algorithms makes sense from an algorithmic point of view. There is, however, an alternative perspective which is less focused on the algorithmic details and more focused on the computational details. Although, we can link some of the above mentioned categories to certain strategies, there might not be such a clear match for others. It is even possible to combine several of the below mentioned strategies for a given algorithm. The general strategies are:

1. **Filtering training instances**: One of the key desirable properties of the original support vector machine formulation is that it leads to sparse solutions. A certain proportion of the training instances have associated dual variables $\alpha_i$ with value zero. The ratio of such training instances determines the degree of sparsity. If we knew the identity of the support vectors beforehand, we could omit all other training instances and thus significantly reduce the size of the optimization problem. However, we usually don’t know which instances will become support vectors, because this is an output of the support vector machine. There are, however, alternative methods, less costly in terms of memory and CPU footprint, to determine instances which are less likely to become support vectors. We will describe one particular method to filter training instances below.

2. **Matrix approximation techniques**: Algorithms that work directly with the kernel matrix, such as interior point algorithms, can be significantly accelerated if the kernel matrix is approximated. There is a range of approaches based on low-rank approximations of $K$ which use a random subsample of the training instances [53, p.288–295]. The most popular method is commonly referred to as the Nyström method [68] and there are a number of extensions and variations mainly differing in the way the subsample of instances is created [58]. In the basic Nyström method, a subsample of $m$ out of $n$ training instances are sampled in order to
create the $n \times m$ matrix $B$ which contains the columns of $K$ that correspond to selected training instances [68]. Then, a $m \times m$ kernel matrix $M$ is calculated for the selected instances. A rank $k$ approximation $M_k$ of $M$ is calculated using singular-value decomposition. Based on the pseudo-inverse $M_k^+$ of this matrix, the kernel matrix for all instances can be approximated as: $K = BM_k^+ B^\top$. While this method leads to a considerable reduction in memory and CPU footprint, depending on the scale of the kernel, low-rank approximations of $K$ may be associated with a considerable approximation error [58]. In order to take into account both block diagonal structure and low rank structure, Si et al. (2014) proposed a new kernel approximation technique called MEKA [58]. Based on this matrix approximation, huge data sets can be processed on a single computing node using kernel methods such as support vector machines or kernel ridge regression.

3. Dividing into independent subproblems: Whenever the kernel matrix shows a block diagonal structure, it makes sense to use the block kernel approximation [58]. This approximation basically consists in splitting the training set into a number of disjunct subsets, where each subset represent one diagonal block of $K$. Although it is not necessary to calculate and store $K$, it is necessary to have some intuition about a suitable number of clusters $k$. Then, we can use k-means in input space to create $k$ subsets and separately fit a support vector machine on each subset. The final model is created by concatenating the dual variables $\alpha_i$ of the subproblems [58]. This approach only works for a fixed $b = 0$. Note that this strategy is not equivalent to the subset selection algorithm, because the subset selection meta-algorithm works in an iterative way while the block kernel approximation can be completely parallelized.

4. Explicit approximations of the feature space: Instead of working with the dual formulation and operating implicitly in a potentially infinite dimensional feature space, it is also possible explicitly map the inputs into a low dimensional approximation of the feature space using non-linear transformations and then apply the primal formulation of the support vector machine[51]. We introduced kernel methods as generalizations of non-linear basis functions (compare 2.2). This approach takes exactly the opposite direction: Instead of removing the explicit non-linear transformation by introducing the dual formulation, explicit kernel methods approximate the implicit feature space by explicit non-linear basis functions.

After giving a general overview of algorithmic approaches and general strategies, we will highlight gradient descent and instance selection as a specific algorithm and a specific strategy used in this work.
3.1.3 Gradient Descent

Gradient descent is a very general optimization technique that only requires a formula to calculate the gradient, i.e. first derivative $\nabla f(x)$, of the objective function $f(x)$. In the case of primal optimization methods, this gradient is $d$ dimensional. In the case of dual optimization methods, this gradient is $n$ dimensional. The most basic version of gradient descent is called steepest descent, it consists of taking the negative of the gradient as descent direction in order to improve the estimate of variable $\alpha$ [28, p.79–82]:

$$\hat{\alpha} = \alpha + \eta(-\nabla f(x))$$ (5)

Despite being generally applicable there are two major drawbacks of this method: First, being only based on first-order information, it shows a low rate of convergence. Second, the learning rate (i.e. step size) $\eta$ is a very delicate parameter of the method: If $\eta$ is too small, the algorithm may take many iterations until convergence. If $\eta$ is too high, the algorithm may not converge at all because successive gradients vary strongly [28, p.288–289, 418–419]. While there is no general remedy, momentum methods such as conjugate gradient descent have been developed to decrease the variance of successive gradients [28, p.288–292, 304–307] (compare 3.1.4). In comparison to second order methods, such as the classical Newton method with quadratic convergence, or improved first-order methods such as Quasi-Newton methods (superlinear) and conjugate gradient methods (quadratic) the rate of convergence is only linear. In the context of big data applications, algorithms based on gradient descent have one key advantage: They are straightforward to parallelize and distribute. First, we can replace gradient descent by stochastic gradient descent, simply by taking random samples, i.e. mini-batches, from the training set [28, p.147–149]. Second, it is straightforward to calculate the gradient on separate local or distributed subsamples of the training set and average the gradient [64]. Being the optimal approach for parallel and distributed algorithms, we chose to use stochastic gradient descent to implement a kernelized support vector machine.

3.1.4 Conjugate Gradient Descent

For non-parametric methods, it is not possible to use second-order methods based on the Hessian in big data situations. Non-parametric methods are based on the dual formulation. As a result, the Hessian matrix grows quadratically with $n$ [7, p.247–256]. It is simply not possible to store the Hessian and calculate its inverse in big data situations. Conjugate gradient descent, on the other hand, is cheap in terms of storage and CPU load and can considerably improve the stability of consecutive descent directions. The problem with simple steepest descent is that consecutive gradients tend to be orthogonal which slows down convergence and makes the choice of an appro-
appropriate value for $\eta$ more difficult. Conjugate gradient descent mitigates this problem by averaging over successive gradients [28, p.304–307]. By taking the previous descent direction into account, we add momentum to the gradient descent. But instead of working with a fixed momentum parameter $\tau$ [50], we estimate the momentum at each iteration based on the covariation between the current and the previous gradient. There are two popular methods for computing the momentum variable $\tau$, the method of Fletcher-Reeves and the method of Polak-Ribière [28, p.306]. We used the Polak-Ribière method following a tutorial on the subject, which states that the Polak-Ribière method is guaranteed to converge only if negative estimates are set to zero [55, p.42].

3.1.5 Instance Selection Strategies

One of the attractive features of the support vector machine is sparsity, i.e. the fact that a subset of instances is sufficient to explain the differences between classes. This also means that, in hindsight, it would not have been necessary to train on non-support vector instances. The circularity of this argument is obvious, because we need to train the model in order to learn about the essential instances. But what if we could learn the support vectors using an alternative, computationally less demanding, method? Based on this line of reasoning, several approaches try to select instances which have a high probability of becoming support vectors and we will sketch the essential idea of a method recently developed by David Morán Pomés [44]: In order to find a projection of all training instances, we can train a support vector machine implementation on a relatively small subset of the training set with $s < n$ instances. Given the estimated $\alpha_i$ of dual variables we can calculate the projection $z$ for all instances $i$ as:

$$z_i = \sum_{j=1}^{s} \alpha_j k(x_j, x_i)$$

(6)

This is equivalent to formulating a $n \times s$ kernel matrix $Z$ with $Z_{i,j} = k(x_i, x_j)$ with $i \in \{1...n\}$ and $j \in \{1...s\}$ and calculating the projection as:

$$z = \alpha^T Z$$

(7)

The reliability of this projection can be improved by training not one, but several support vector machines on separate subsets of the training set and averaging over respective projections. Once a projection has been obtained, we can choose to classify instances according to the extremeness of the projection:

1. Instances on the extremes of the projection are less probable to become support vectors because they tend to lie in areas where there are only instances of their own class.
2. Instances close to the hyperplane are more probable to become support vectors because they tend to lie in areas where there instances of both classes are mixing. Based on this approach, it is possible to select instances with extreme projection values which can be ignored, thus substantially reducing the computational complexity of the final support vector machine.

3.2 Distributed Algorithms for Support Vector Machines

3.2.1 The Challenges of exact distributed Algorithms

Decentralized Algorithms When scaling out from a single computing node to a distributed system, several fundamental challenges arise. In general, communication costs increase significantly and messages might be lost. Most fundamentally, nodes may fail and nodes have only partial information. Given these fundamental differences between distributed systems and a single computing node, Tanenbaum (2007) demanded, that “only decentralized algorithms should be used. These algorithms generally have the following characteristics, which distinguish them from centralized algorithms:

1. No machine has complete information about the system state.
2. Machines make decisions based only on local information.
3. Failure of one machine does not ruin the algorithm.
4. There is no implicit assumption that a global clock exists.” [63].

Distributed file systems such as the Hadoop File System [56] or the Google File System [42] take care of synchronizing data nodes, replicating data, and continuously checking for node failure using heart beats. Cluster computing engines, such as Apache Spark [72, 73] or Apache Flink [3] are typically build on top of distributed file systems but they can also process data from distributed data bases or data streams [23]. These cluster computing engines take care of synchronizing the computing nodes which are typically identical to the data nodes. Whenever a node fails, the driver must be able to restore the data and restart the computation without corrupting the data or aborting the process. This restoration is possible based on lineage information which is stored by the cluster computing engine. As a result, the third and the fourth requirement for a truly decentralized algorithm can be met by current cluster computing engines, such as Apache Spark.
Decentralized Algorithms for Primal Support Vector Machines  While requirements three and four can be guaranteed by the collaboration of several middle-ware components, the first and the second requirement are much more difficult to put into practice. These requirements imply, that we have to be able to translate an existing centralized algorithm into an equivalent Map-Reduce formulation. The Map-Reduce programming model is a formal way of, first, specifying logically consistent and self-contained chunks of data that can be processed independently, and second, specifying how the outcome of this data processing has to be merged together to achieve the final result of a data operation. In the case of support vector machines, there is a straightforward way of processing distributed chunks of data for the primal formulation which can be formulated in terms of Map-Reduce [66, p.436–443]. For the dual formulation, processing chunks of data independently is impossible, because calculating the gradient for any instance \( x_i \) involves iterating over all other data points and evaluating the kernel function (compare equation 6). In consequence, only the linear but not the kernelized support vector machine is implemented in distributed machine learning libraries such as MLlib [24] and LIBLINEAR [21, 41] (both based on stochastic gradient descent).

Decentralized Algorithms for Dual Support Vector Machines  The crux is, that all nodes need access to the kernel matrix \( Q \) when evaluating the gradient. The same is true for kernel matrices describing the similarity between the instances in the training set and the instances in the validation or test set (which are needed when evaluating the model on these other data sets). These matrices have to be set-up once, by a centralized algorithm that has access to the complete data. This scenario does not fit into the data model of a distributed computing platform such as Apache Spark: The distributed model assumes that the distributed chunks of data can be processed independently and without moving the raw data itself. To add a finer point, it is in fact technically possible to implement my Scala library on top of RDD objects and we implemented this option adding a special class for RDD data. However, when creating the distributed kernel on top of a RDD, this leads to a shuffling of data rows for each individual matrix elements, which does not make any sense because the computational overhead is prohibitive. In consequence, it is not possible to implement a kernelized support vector machine on top of a distributed file system or higher level abstractions such as the Spark RDD abstraction. It is, however, possible for a data node that has access to all data, to create distributed data structures, such as the distributed sparse matrices of Apache Spark[10]. Once distributed kernel matrices have been created, the computing nodes need to exchange information concerning these distributed matrices at each gradient descent step using matrix-matrix multiplications on top of these distributed matrices. In order to elaborate on this point, we will give a short introduction to the Map-Reduce programming model and associated cluster computing engines.
3.2.2 The Map-Reduce Programming Model

While parallel algorithms have a long history in computer science, processing data that is stored in distributed systems [63] has become a hot topic in machine learning in the last decade. This is due to the fact that data is increasingly stored in distributed file systems such as the Google File System [42], the Hadoop File System [56], or in popular distributed data bases such as e.g. Cassandra [38] and MongoDB [2]. When processing this data, it is not feasible to download it to a single node. Instead, it is processed in cluster computing systems such as Apache Spark [72], Apache Flink [3] or commercial cloud platforms which try to avoid moving the data itself and instead ship the processing code to the data using the Map-Reduce programming model [19, 25]. We will now describe the different stages of the Map-Reduce formalism:

Data Nodes and Compute Nodes  Data locality is a key concept associated to Map-Reduce. It means that instead of moving data, we should ship code to the nodes where data is located, run the computations locally, and finally communicate the results in hopefully compact form back to the driver or else locally transform the data. The central idea is to move data as little as possible and formulate algorithms in a way that code can be executed independently on distributed chunks of data. The starting point of any Map-Reduce algorithm is, that data is already distributed in several chunks on separate data nodes. The same nodes should be used as compute nodes to transform the data and calculate summary information from the data. The mechanism of splitting data into chunks must not be random. Indeed, for many algorithms it makes sense to use specific rules to create either groups of observations or groups of features that are then sent to the same data node.
Figure 1: Schematic example of the Map-Reduce programming model with four map tasks located at four different data nodes and two reduce tasks located at two separate compute nodes. In this example, the upper reduce node is responsible for key k1 and the lower node for k2. Data flow from the second mapping node from the top is highlighted in blue, because data has to be send to both reduce nodes, given that key-value pairs with both key 1 and key 2 are generated.

**The Map Task** At the map step, data nodes transform local data into key-value pairs. The types of keys and values are arbitrary and keys are not necessarily unique. A given input element can be divided into several key-value pairs and some of these key-value pairs may have the same key [66, p.23-25]. The map task occurs on the data nodes (compare Fig. 1 where the mapping is referenced with a,b -> k,v).

**Grouping by Key** First, the local key-value pairs are locally sorted according to the keys (Fig. 1). In the next step, the sorted key-value pairs have to be sent to the reduce nodes responsible for the respective keys. This is the first step that involves data communication. The mapping between keys and computing nodes is the responsibility of the master, or driver node, who orchestrates the data flow. Once they have been received at the reduce node, all key-value pairs sharing the same key are merged into a single key-value list (Fig. 1)[66, p.24].

**The Reduce Task** At this step, each reduce node applies the reduce function to one or many key-list of values-pairs which were generated in the previous step (Fig. 1). This must result in another list of key-value pairs. Typical examples for the reduce function are aggregation functions such as sum, count, or mean but it can be any object specific function that summarizes the type of values that were created in the previous step. In the original Map-Reduce implementation, each node writes his output into a single file which is stored in the master node [66, p.24-25]. Newer implementations (or extensions) of Map-Reduce such as Apache Spark, store the results of the reduce task...
in the distributed file system itself, thus avoiding a potential bottle-neck and allowing for efficient recursive algorithms.

### 3.2.3 Limitations of Map-Reduce

There is some confusion about the difference between the logical and the physical model, i.e. if the Map-Reduce definition includes the fact that intermediate and end results are written to disc, because that is how it was implemented in Apache Hadoop and Google MapReduce. Spark \[72, 23, 73\], which is based on so-called resilient distributed datasets (RDDs), allows in-memory computations based on the Map-Reduce logical model. Iterative algorithms which form the basis on machine learning methods can be massively accelerated by keeping the results of Map-Reduce operations in memory \[29\]. While the Map-Reduce programming model is able to describe many processing steps which are common in data mining and machine learning, some iterative algorithms cannot be translated into the rigid structure of Map-Reduce. As a result more general programming models have been developed that extend the Map-Reduce model \[66, p.38-43\]. Spark is based on a programming model which generalizes Map-Reduce operations to acyclical directed graphs. As a result, the associated library for machine learning MLlib \[43\] implements a range of popular machine learning algorithms such as ridge regression, principal component analysis, and linear support vector machines. However, it does not support kernelized support vector machines.

### 3.2.4 Principles for efficient distributed Algorithms

Based on the increased cost of communication in a distributed system, three general rules for efficient distributed algorithms have evolved \[17\]:

1. Memory and CPU footprint should scale linearly with $n$ and $d$.
2. Computations should be run in parallel and in memory.
3. Network communication should be minimized by limiting the number of iterations and batching communication.

Based on these rules, there is a tendency to decrease the number of effective iterations of iterative algorithms by batching iterations and not exchanging information after each iteration. In the most extreme case, local algorithms do not exchange information at all, but only report the final result to the master node, i.e. driver. Here is a rough categorization \[17\]:

1. Command and conquer strategy: all iterations are run in parallel and without coordination among compute nodes. Compute nodes do not communicate among each other.
2. Batching strategy: some iterations are run together before exchanging data with other compute nodes.

3. Complete coordination strategy: Compute nodes exchange results after each single iteration.

In the context of support vector machines for classification, the command and conquer strategy could consist in independently training support vector machines on several computing nodes with their own subset of instances. After training these local models, a global model could be created by, i) averaging the estimated hyperplanes or, ii) by a model averaging strategy that might involve a majority vote rule. Some authors have argued that such an approach might be appropriate, whenever there are data privacy concerns which do not allow for data exchange between nodes [22, 46]. The intermediate batching strategy is not an option for an exact implementation of kernelized support vector machines, because the gradient has to be evaluated after each iteration. As was outlined above, this evaluation of the gradient involves all training instances. It seems therefore that only the two extreme options, training several separate models in isolation on their own private data set or coordinating between separate nodes after each iteration by calculating the global gradient are useful strategies. In the case of the latter option, which is the strategy employed in this work, it is necessary to calculate the gradient using a Map-Reduce formalism, as will be detailed below.

3.2.5 Map-Reduce Formalism for Matrix Vector Multiplications

Linear algebra operations that involve matrices and vectors can be translated into the the Map-Reduce formalism. In the specific case of distributed stochastic gradient descent for support vector machines, we will now describe how the gradient can be evaluated by multiplying a distributed matrix $Q$ with a local vector $\alpha$ [66, compare p.28–29]. Given a distributed matrix $Q$ with row index $i$ and column index $j$ and a local vector $\alpha$ the product $g = Q\alpha$ can be calculated as:

$$g_i = \sum_{j=1}^{n} Q_{ij}\alpha_j$$

(8)

**The Map Task**  In the map task, all matrix elements $Q_{ij}$ which are at the disposal of this node and vector $\alpha$, which has to be available in its entirety, are mapped into the key value pair $(i, Q_{ij}\alpha_j)$.

**The Reduce Task**  In the reduce task, all key-value pairs with identical key $i$ are grouped together. Next, the list of values is aggregated using the sum function resulting in key-value pairs $(i, g_i)$. Note that, the local vector $\alpha$ has to be communicated in its entirety to each compute node.
4 Materials and Methods

4.1 Subset Selection Heuristic

Based on the general ideas of David Morán Pomés [44], we implemented a heuristic for subset selection before processing the training set. With a given sampling probability \( p \), \( s \) training instances were randomly selected. This random subset of the training set was used to fit a least-squares support vector machine based on the linear kernel matrix \( K = XX^\top \):

\[
\alpha = (K + \lambda I)^{+} l
\]  

Here, \( I \) denotes a \( s \times s \) identity matrix, \( X^{+} \) denotes the pseudo-inverse of matrix \( X \), and \( \lambda \) is a regularization parameter with a default value of 0.5. Note that this is exactly the formula for kernel ridge-regression. Based on the analytically derived vector \( \alpha \), we then calculated a projection vector \( z \) for all instances in the training set (equation 6). In order to make the subset selection method scalable, we introduced a rule to create several random subsets and average the projections calculated on different subsets (also based on ideas of David Morán Pomés). After having calculated the final projection \( z \), we then sorted \( l \) according to \( z \) and decided on a lower and upper threshold based on a simple error tolerance rule: The user has to define a maximum error probability, which can then be translated into an upper number of cases which are misclassified. We defined misclassifications as all instances with class +1 at the lower range and class -1 vice versa. The lower and upper thresholds were then chosen in such a way as to allow no more than half of the total number of acceptable misclassifications at the lower and upper range of the projection.

4.2 Implemented Kernel Functions

The linear kernel, the Gaussian kernel, and the Polynomial kernel were implemented. Here, we will refer only to the Gaussian kernel, because it is a very versatile and thus popular kernel function. There are two alternative but mathematically equivalent formulations of the Gaussian kernel. The first formulation is based on \( \sigma \) [54, p.296], while the second is based on \( \gamma \):

\[
k(x, z) = \exp\left(-\frac{\|x - z\|^2}{2\sigma^2}\right)
\]

\[
k(x, z) = \exp\left(-\gamma\|x - z\|^2\right)
\]

The relationship between the two parameters is \( \gamma = \frac{1}{2\sigma^2} \). We used the second formulation throughout this text. Users of support vector machine software are typically expected to run a grid-search on possible values of the kernel parameter using cross-validation. Given that this implementation is focused on big data situations where such a grid-search would be computationally demanding, we developed a method
that estimates a suitable value for the kernel parameter of the Gaussian kernel: In order to estimate a suitable $\gamma$, it was recommended to sample the distribution of the squared Euclidean distance between data points and use any value between the 10% and 90% percentile as an estimate for $\sigma$ [35]. We followed a similar, but slightly different approach to estimate a suitable value for $\gamma$: First, the 1% percentile of the Euclidean distance $q$ between instances (data points) was estimated. Next, an estimate of the optimal $\sigma$ was calculated as: $\sigma = q^e_4$, where $e$ is the base of the natural logarithm. The estimate for $\gamma$ was calculated as $\gamma = \frac{1}{2\sigma^2}$.

4.2.1 Scaling of Data Matrices

Although, there might be situations where all original features are measured in the same measurement unit and on the same scale, it is in general more safe to center and scale the data before evaluating the kernel function. If not, features with high variance dominate the kernel function. This is a standard practice in support vector machine implementations. My Scala implementation accordingly z-transforms all data sets based on the arithmetic means and standard deviations of the respective columns of the training set.

4.3 Details of the algorithmic Implementation

We implemented three different algorithms (Fig. 3), all based on the same stochastic gradient descent algorithm. First, a local version based on a sparse matrix implementation that uses multithreading in combination with asynchronous programming (class NoMatrices). Second, a distributed version based on distributed Spark matrices [10] using a sequential programming approach in the master thread (class SG). Third, a distributed version based on distributed Spark matrices [10] which uses multithreading in combination with asynchronous programming in the master thread (class SGwithFutures). Here, master thread refers to the main thread (i.e. driver) which runs on the Spark master node and is responsible for the setup of all distributed matrices (compare Fig. 4 where this master thread is depicted by spheres). In the following, we will detail the inner workings of the general stochastic gradient descent algorithm, which is implemented in the hasGradientDescent trait. For the sake of readability, we will first describe the core gradient descent algorithm, before we give details on some refinements, and conclude with a pseudocode description (Algorithm 1).
4.3.1 Gradient Descent and Refinements

**Gradient Descent** First, the vector $\alpha$ of dual variables, the regularization parameter $C$, and the step size, i.e. learning rate, $\eta$ were initialized. The elements of $\alpha$ were initialized with random uniform numbers between 0 and $C$. Parameters $C$ and $\eta$ are parameters that have to be specified by the user of the software. Then, the gradient was calculated as:

$$\nabla f(\alpha) = Q\alpha - e$$  \hspace{1cm} (12)

The first, tentative, estimate of the updated parameters was:

$$\hat{\alpha} = \alpha^i + \eta(-\nabla f(x))$$ \hspace{1cm} (13)

Then, $\hat{\alpha}$ was projected onto the feasible region defined by the first constraint:

$$\alpha^{i+1} = \hat{\alpha} - \frac{l(l, \hat{\alpha})}{\langle l, l \rangle}$$ \hspace{1cm} (14)

Finally, in order to enforce the second constraint, each $\alpha_i$ was transformed by the midpoint function into the middle value between 0, $\alpha_i$, and $C$. Denoting the midpoint function as $m(x_1, x_2, x_3)$, we have:

$$\alpha^{i+1}_j = m(0, \alpha^{i+1}_j, C)$$ \hspace{1cm} (15)

**Stochastic Gradient Descent** The gradient descent algorithm described above was transformed into a stochastic gradient descent algorithm by randomly selecting a subset of observations in each iteration [15, p.129]. In consequence, the probability $p$, of selecting a given instance and including it into the current batch, is a new hyper-parameter of the algorithm that has to be specified by the user.

**Learning Rate Decline** For convex problems, i.e. optimization problems with convex objective function and convex feasible set, gradient descent works best if the learning rate decreases over time. This is not true for non-convex optimization problems, where it might be necessary to both decrease and increase the learning rate using adaptive algorithms [20]. Given that the support vector machine formulation as detailed above is a convex problem, we implemented an continuously decreasing learning rate using a constant decay rate $\psi$.

**Parameter Shrinkage** In the context of online learning, Kivinen et al. suggested continuous shrinkage of the dual variables which are not included in the current batch [36, 37]. Using this approach, the above algorithm can also be applied to online learning problems such as data streams. In order to avoid an additional hyper-parameter, we used a single decline rate $\psi$ to shrink both the learning rate $C$ and the elements of vector $\alpha$ that correspond to out-of-batch samples.
Conjugate Gradient Descent  Simple, first-order, gradient descent is difficult to tune because the rate of convergence may critically depend on the chosen value of the learning rate: Descent directions of successive iterations tend to be orthogonal which leads to slow convergence. If the learning rate $\eta$ is high, the algorithm is not guaranteed to converge. To mitigate this issue, moment methods have been developed which calculate a moving average over successive descent directions [28, p.288-292][50]. Instead of using a fixed value for the moment, we used the Polak-Ribière method to estimate an appropriate value for the moment at each iteration [28, p.304-306]. We followed a tutorial on the subject, which states that the Polak-Ribière method is guaranteed to converge only if negative estimates are set to zero [55, p.42].

Algorithm 1  Pseudocode for the general stochastic gradient descent algorithm followed by all algorithms. The local and the two distributed implementations only differ in the way the gradient is calculated (the function calculateGradient in the pseudocode). Here $r$ denotes a random uniform number between 0 and 1 and $u$ and $v$ are auxiliary variables which are needed in order to implement the Polak-Ribiere formula for conjugate gradient descent. Note that the final, updated values of $\alpha$ are stored for each iteration (the output). This is necessary in order to be able to later choose the $\alpha$ associated to the lowest validation error for the final classification model.

Input: $Q,l,\eta,\alpha,\alpha^{old},\psi,\delta,p.$

while $|\alpha - \alpha^{old}|_1 \leq \delta$ do

$\eta \leftarrow \psi\eta$

$\alpha^s \leftarrow \psi\alpha$

$g \leftarrow \text{calculateGradient}(Q,l,\alpha)$

$\alpha \leftarrow \alpha - \eta g$

$\alpha \leftarrow \alpha - l\langle l, \alpha \rangle / \langle l, l \rangle$

$\alpha \leftarrow m(0, \alpha, C)$

for all $i \in \{1, \ldots, n\}$ do

$r \leftarrow \text{generateRandomUniformNumber}()$

if ($r > p$) then $\alpha^{new}_i \leftarrow \alpha^s_i$

end for

$u \leftarrow \langle \alpha, \alpha - \alpha^{old} \rangle$

$v \leftarrow \langle \alpha^{old}, \alpha^{old} \rangle$

$\tau \leftarrow \text{Max}(u/v, 0)$

$\alpha^{updated} \leftarrow \alpha + \tau \alpha^{old}$

Output: $\alpha^{updated}$

$\alpha^{old} \leftarrow \alpha$

$\alpha \leftarrow \alpha^{updated}$

end while
4.4 Cross-validation of Number of Iterations and Sparsity

While the three classes of algorithms all use the same trait class `hasGradientDescent` to implement the core stochastic gradient descent algorithm, they vary in the way they calculate the gradient (compare section 4.6.3) and how they cross-validate. Here, we will describe the details of cross-validation for the different algorithms.

4.4.1 Centralized Algorithm

In order to avoid overfitting, we used a validation set to evaluate both the optimal number of gradient descent iterations and optimal sparsity. After each iteration, we stored the final values of $\alpha$. Then, we performed a clipping operation on all possible quantiles from 0% to 99% of the distribution of $\alpha$: We truncated all vector elements to zero which were smaller or equal to the respective quantile. Then, we evaluated the accuracy of the resulting sparse vector on the validation set. For each iteration, we returned both the optimal quantile and the associated optimal validation set accuracy. When all $\alpha$ had been evaluated, the iteration with highest accuracy and the respective quantile were chosen as the final model parameters and could later be used to evaluate the test set.

4.4.2 Distributed Algorithms

For both distributed algorithms, only the iteration but not sparsity itself was cross-validated. After each iteration, the final values of $\alpha$ were stored and they were evaluated on the validation set. The difference between the classes $SG$ and $SG$withFutures is that the former works in a sequential way, while the latter uses Scala Futures to implement an asynchronous work flow (Fig. 4). While the latter version accelerates computations, it only works on Spark clusters with a multi-CPU driver node (we were not able to test the asynchronous version on a Spark cluster). Once, $\alpha$ has been evaluated on the validation set, the iteration with the highest accuracy on the validation set is chosen as the final model and can be evaluated on the test set.

4.5 Tuning the Decision Threshold based on Scores

Binary classification problems typically involve a trade-off between true and false positive rates (i.e sensitivity and specificity). As a result, any binary classifier should be evaluated for a range of possible cutoff values using Receiver Operating Characteristic curves. The standard implementation of SVMs is based on a rigid threshold value: The final class predictions are calculated by simply taking the signum function on the SVM scores. Although there have been probabilistic interpretations of the SVM scores, these approaches involve fitting additional parametric or non-parametric models on
top of the SVM model [61, 70, 30, 59]. Instead of using additional probabilistic models, we implemented a simple quantile based approach: Based on the empirical distribution of the final SVM scores, we calculated the percentiles from 1% to 99%. Then, we evaluated true and false positive rate and overall accuracy on the validation set for all percentiles. The user of the software receives the results of this evaluation both graphically, as a figure of the ROC curve, and numerically, in the form of a text output of the underlying values. Based on this information and the particular requirements of his data set and the domain, the user can decide on an appropriate cutoff value. Finally, he can calculate predictions for a new data set based on this cutoff value. By deciding on the optimal cut-off value, the user decides on the intercept terms of the final support vector machine decision function. While this intercept term is typically a model coefficient which is fitted during the training phase, in my implementation, the intercept term is missing, which considerably simplifies the training process.

![ROC curve](image)

Figure 2: Receiver Operating Characteristic curve for the Magic data set. The consecutive numeric labels indicate the decision quantile and help the user to directly infer the optimal decision threshold.

4.6 Computational Details

4.6.1 Hardware Specification

All experiments based on the local implementation were run on a machine with a Intel x86_64 processor, four 2242.5 MHz CPUs and 16GB RAM. The experiments based on
distributed versions of the algorithm were run on the Databricks Community Cloud[71] which offers a Spark driver node with 0.88 cores (CPU virtualization) and 6.0 GB RAM Memory.

4.6.2 Class Hierarchy

![Simplified UML Diagram of the Scala support vector machine implementation.](image)

Both algorithms for local computation (class NoMatrices) and for distributed computations are implemented (Fig. 3). For distributed computations there are two variants, with (class SG) and without asynchronous parallelized program flow in the driver (class SGwithFutures). Data can be of class LocalData, which describes empirical data in csv format, or SimData, which describes synthetic data sets. The details of the data classes are hidden from the algorithms by the Data interface.
4.6.3 Decoupling Data Size from Memory Footprint

One of the key computational problems of kernelized support vector machines is how to scale with increasing size of the data set. This is due to the fact, that the kernelized support vector machine operates in dual space which is $n$ dimensional. In practical terms, this means that the kernel matrices for training, validation, and test set grow quadratically with $n$. As a result, it is not feasible to physically store these matrices for big data sets. Typically, we expect these matrices to be sparse, i.e. that many elements are close to zero and can be ignored. It is thus key to find lean implementations of sparse matrices, or even better, a proxy for these sparse matrices that alleviates the memory footprint while reducing CPU demands based on continuous iterations over the entire matrix at the same time.

Local Implementation  The size of kernel matrices rises quadratically with $n$. As a result, it is not feasible to store the entire kernel matrices for big data applications. Instead of storing the matrix itself, we created hash maps with integer values as keys (the row index) and sets of integers as values (the column indices). These auxiliary multimap data structures can be used to drastically reduce the CPU footprint, especially for sparse kernel matrices. Every operation involving the respective matrix, can be translated into an iteration over the respective hash map, which only includes entries for non-sparse matrix elements though not the values of the kernel function itself. This approach represents a compromise between memory and CPU footprint. In order to probe even extremely high dimensional data sets with a high ratio of non-sparse matrix elements, the user can tune a tolerance parameter $\epsilon$ that defines the threshold value defining sparsity. The advantage of this approach is, that no assumptions are made concerning the overall structure of the kernel matrix (i.e. block-diagonal). The disadvantage is, that memory and CPU footprint ,while considerably reduced, still grow quadratically with the size of the training set. The local implementation is thus limited by total RAM availability and the sparsity of the kernel matrix.

Distributed Implementation  In the distributed implementation, kernel matrices are implemented by the Apache Spark implementation for sparse distributed matrices CoordinateMatrix [10]. Each matrix element is represented by a (Long,Long,Double) tuple for row index, column index, and the floating point value of the kernel function. As for the local implementation, a tolerance parameter $\epsilon$ defines the threshold value defining sparsity. The advantage of the distributed implementation is that the number of elements of the CoordinateMatrix is only limited by the number of available data nodes. It is, however, still necessary to create the distributed matrix on the master node (or driver) who has access to the complete data set. This represents a bottle neck of the
implementation. It is, in other words, not possible to follow the standard approach of Spark and run a parallel and distributed operation on top of the RDD abstraction of a distributed data set (compare section 3.2.1).

4.6.4 Multithreading and asynchronous Programming

Apart from the impact of data set size $n$ on the memory footprint, we also have to consider its CPU footprint and computation time. Here, we can make use of Map-Reduce principles and asynchronous programming to accelerate the computation. We used Scala Promises and Futures which provide a relatively simple interface to asynchronous thread management. In contrast to the standard concurrency programming model of Java, which is based on locks, asynchronous programming based on Promises and Futures is much safer because it can be implemented in a non-blocking way. As a result, deadlocks and race conditions can not occur [48, chapter 32]. In Listing 8.1 a scala function is shown that uses asynchronous programming and promises to parallelize code while avoiding deadlocks. Note that this function does not return a concrete integer value but an object of type Future[Int] which is a promise to return an integer value in the future. In order to be able to predict on either the validation or the test set, the accuracy of all gradient descent steps and all possible sparsity levels of $\alpha$ must already have been evaluated. In order to ascertain that these results are available, instead of blocking the thread, a callback on futureList is used:

```scala
futureList onComplete {
  case Success(list) => {
    ...
  }
  case Failure(ex) => println(ex)
}
```

The effect of returning a promise and using a callback block is that the function does not block the main thread. Once all results have been successfully calculated, the inner block is executed in a parallel thread. First, the optimal iteration and the associated optimal level of sparsity are determined:

```scala
val (optSparsity, maxAccuracy, optIteration) = list.foldRight((0,0,0))((a,b) =>
  if(a._2 <= b._2) b else a)
println("Based on cross-validation, the optimal sparsity of: "+ optSparsity +
  with max correct predictions: "+ maxAccuracy+" was achieved in iteration:
  "+ optIteration)
```

Then, the respective values of $\alpha$ are retrieved from a map and clipped according to the sparsity level:
val optAlphas : Alphas = alphaMap.getOrElse(optIteration, alphas)
if(optSparsity > 0.0) optAlphas.clipAlphas(0.01 * optSparsity)

Then, depending on the prediction method, the respective prediction function is called and the promise object is completed once this function, which also returns a future, has successfully completed:

predictionMethod match {
  case PredictionMethod.STANDARD => {
    val promisedTestResults : Future[Int] = kmf.predictOn(dataType, optAlphas)
    promise.completeWith(promisedTestResults)
  }
  case PredictionMethod.THRESHOLD => {
    val promisedTestResults : Future[Int] = kmf.predictOn(dataType, optAlphas, threshold)
    promise.completeWith(promisedTestResults)
  }
  case PredictionMethod.AUC => {
    val promisedTestResults : Future[Int] = kmf.predictOnAUC(dataType, optAlphas)
    promise.completeWith(promisedTestResults)
  }
}

Exploring the Matrix Space  When setting up the hash map associated with the kernel matrices for the different types of data sets (training, validation, test), we partitioned the matrices into four regions (or submatrices) and processed each region in a separate thread. After completion of the respective hash maps, the maps were merged into a single unified hash map for the training set. For the validation and the test set, the four maps were kept separately for later parallel evaluation. The exploration of the huge $n \times n$ matrix spaces for the different data sets (training, validation, test) is a bottleneck in terms of CPU time. As a result, parallelizing this step significantly reduced overall computation time.

Evaluating the Gradient  When evaluating the gradient, the keys of the hash map for the training set were split into four equal sized groups and processed in parallel by four threads. At the end, the partial gradients were added together with the partial gradient
that is due to the diagonal which was separately calculated in the caller thread.

**Evaluating the Accuracy on the Validation and Test Set** Both the validation and the test set were split into four subsets. For each subset a separate hash map was created which stored the information about relevant, non-sparse, matrix elements. When evaluating the accuracy of a given model for the validation or test set, this evaluation was run in parallel on the four different subsets using the associated hash maps. The evaluation of a given model on the validation set, was immediately triggered after each gradient descent iteration. The gradient descent algorithm did, however, not wait for the results of this evaluation. The evaluation of test set accuracy was triggered once the $\alpha_i$ of all iterations had been evaluated for all possible quantiles on the validation sets. The test set was evaluated with the $\alpha_i$ and the sparsity that had shown the highest accuracy on the validation set.

**Multithreading and asynchronous Programming in the distributed Implementation** In the algorithm class `SGwithFutures`, the same general asynchronous program flow is followed as in the local implementation: After each successful gradient descent step, the evaluation of the resulting vector of dual variables $\alpha$ on the validation set is triggered by the driver (Fig. 4). Both the gradient descent steps and the evaluation of the resulting $\alpha$ is implemented using distributed matrix vector multiplications. The main thread of the driver node creates new futures, each time a new evaluation is triggered. After the stochastic gradient descent algorithm has converged, the main thread waits until the resulting $\alpha$ of all $i$ iterations have been evaluated on the validation set, to choose the iteration with the lowest error on the validation set for the final model. Using this algorithm class does only make sense if the Spark driver node has more than one CPU, else the class `SG` should be used which sequentially executes the algorithm.

5 Working Examples of How to use the Scala Library

In this section, we will illustrate the practical usage of my software package using small code snippets for a synthetic data set and two empirical data sets.

5.1 Data Sets.

5.1.1 Synthetic Data Sets

In order to be able to try out the methods with data sets varying in size and classification difficulty, we created a generic data generator implementing a mixture of two multivariate Gaussians. The probabilistic data model was as follows: First, two random vectors $\mu_1$ and $\mu_2$ with dimension $d$ (number of features), were drawn from a
Figure 4: Distributed asynchronous workflow using the SGwithFutures class. Clouds indicate distributed matrices for the $Q$ matrix and kernel matrices for the validation set (V) and the test set (T). After convergence of the gradient descent algorithm, the driver process has to wait for the results of the evaluation of all $\alpha$ on the validation set. Note that the user has to interact with the program by deciding on a final decision threshold based on the ROC curve visualization and associated log data.
random uniform distribution:
\[ \mu_i \sim U(0, 1) \]  
(16)

These random vectors were used as multivariate means of two multivariate Gaussian distributions \( N_1 \) and \( N_2 \) with diagonal covariance matrix \( \Sigma \) to produce two samples \( S_i \) with class labels +1 and -1:

\[ S_i \sim N_i(\mu_i, \Sigma) \]  
(17)

The covariance matrix \( \Sigma \) was an identity matrix of dimension \( d \), thus there was no correlation between the original features.

5.1.2 Validation on Empirical Data Sets

In order to validate the method with empirical data sets, we used two public data sets with binary response variables available at the UCI machine learning repository [40]: The MAGIC Gamma Telescope Data Set contains 10 input features for 19,020 observations [8, 9]. The HIGGS Data Set contains 28 input features for 11 Mio observations [67, 5]. In order to compare the accuracy of the method, we compared it with the kernlab R package: We used 10-fold cross-validation, to find the optimal value for \( C \) (which is called \( C \) in kernlab) using the \texttt{ksvm} function. Then we used this optimal value of \( C \) in both software packages. For the kernel spread parameter \( \gamma \), the \texttt{sigest} function was used to automatically estimate a suitable value based on the empirical distribution of the squared euclidean norm of pairs of observations [35]. In the Scala implementation, the new function \texttt{probeKernelScale()} was used to estimate the kernel parameter \( \gamma \).

5.2 Local Implementation

5.2.1 Probing the Size of the Kernel Matrices

In a first step, it is recommended to probe the sparsity of the matrix to get an estimate of the memory footprint of the matrices. In Listing 8.2, a synthetic data set with 40,000 instances was created and separated into a training (\( n=20,000 \)), a validation (\( n=16,000 \)), and a test set (\( n=4,000 \)). Then, the size of each matrix was estimated by probing the sparsity of each matrix.

5.2.2 Running a local Support Vector Machine on simulated Data

Having assessed the approximate memory requirements for a given \( \epsilon \), the support vector machine can be trained on the simulated data set (Listing 8.3). After evaluating the ROC curve on the validation set, the user should decide on a suitable decision threshold. Then, this threshold can be applied on the test set, in order to gain an estimate of the generalization error of the final model. In the code example above, the default
decision threshold of 0.5 was used to evaluate the test set because the above program is not interactive. Instead of running a complete Scala application the user can use a scala shell, import the SVM package and other necessary imports and execute the above code interactively.

5.2.3 Running a local Support Vector Machine on empirical Data Sets

When fitting a support vector machine locally on a data set, it is assumed that the user can separate the data set into three separate csv text files for the training, validation and test set. The input files should not have a header and consist only of numeric data types. Any separator can be specified via the separator argument. The user additionally has to specify the complete path to the input files as string and the index of class labels and any column that should be ignored using zero-based indexing. It is implicitly assumed that the class labels are +1 for the signal and -1 for the background class. If the class labels follow a different code, a function has to be specified which transforms the labels into the correct code. In the example code a function $\texttt{(x:Double) => if(x<=0) -1 else +1}$ is used. After having read the input files, the user may want to print the class distribution for all data sets. The function $\texttt{probeKernelScale()}$ can be used to determine a useful estimate for the kernel parameter $\gamma$ of the Gaussian kernel function. After having constructed a kernel function with a sparsity threshold $\epsilon$, the user can initiate a new instance of class $\texttt{LeanMatrixFactory}$ for a given data instance, a kernel function object, and a value of $\epsilon$. The algorithm itself is initiated given this matrix factory object, a new $\texttt{Alphas}$ object and a $\texttt{ModelParams}$ object which bundles the parameters $C$ and the learning rate $\eta$ (the argument is called delta). The algorithm is iterated using a loop construct. At the end of the loop, a blocking $\texttt{Await.result()}$ is needed to keep the main thread from shutting down before the parallel evaluation of the final model on the test set has finalized (Listing 8.4).

5.3 Distributed Implementations

In order to use the distributed algorithms, the user needs access to a local Spark cluster or a commercial Spark cloud solution (e.g. Databricks). Depending on the number of central processing units (CPUs) available to the driver node, it is recommended to use the sequential $\texttt{SG}$ or the parallelized $\texttt{SGwithFutures}$ algorithm. Given that we had only access to the Databricks Community Edition with a 1 CPU driver node and 6GB random access memory(RAM), we were not able to able to test the $\texttt{SGwithFutures}$ algorithm. Some code snippets for the sequential algorithm are shown below. After uploading the $\texttt{SVM}$ library as a jar file, the following commands on the Spark shell create a synthetic data set:
import SVM._
val N = 40000
val kernelPar = GaussianKernelParameter(1.0)
val gaussianKernel = GaussianKernel(kernelPar)
val ratioTrainingObservations=0.5
val dataProperties = DataParams(N=N, d=10, ratioTrain=ratioTrainingObservations)
val d = new SimData(dataProperties)
d.simulate()

The corresponding console output was:

import SVM._
N: Int = 40000
kernelPar: SVM.GaussianKernelParameter =
Gaussian kernel parameter sigma 1.0.

gaussianKernel: SVM.GaussianKernel =
Gaussian kernel:
Gamma: 1.0

ratioTrainingObservations: Double = 0.5
dataProperties: SVM.DataParams =
Data parameters:
Total number of observations: 40000
Observations training set: 20000
Observations validation set: 16000
Observations test set: 4000
Number of features: 10

d: SVM.SimData =
Synthetic dataset with 10 variables.
Observations: 20000 (training), 16000(validation), 4000(test)
Data was already generated.
Command took 7.62 seconds

The distributed matrices for the training set, validation set, and test set are created with
the following commands:

val epsilon = 0.001
//returns the underlying SparkContext
val sc = spark.sparkContext
val kmf = new KernelMatrixFactory(d, gaussianKernel, epsilon, sc)
The console output logs the size of the distributed matrices.

Q matrix has rows: 20000 and columns: 20000
Kernel matrix of type Validation has rows: 20000 and columns: 16000
Kernel matrix of type Test has rows: 20000 and columns: 4000
epsilon: Double = 0.001
sc: org.apache.spark.SparkContext = org.apache.spark.SparkContext@961a966
kmf: SVM.\KernelMatrixFactory = KernelMatrixFactory(Synthetic dataset with 10 variables.
Observations: 20000 (training), 16000(validation), 4000(test)
Data was already generated.
\Gaussian kernel:
\Gamma: 1.0
0.001,org.apache.spark.SparkContext@961a966)
Command took 8.90 minutes

Now, the model itself and the algorithm is created:

```scala
import scala.collection.mutable.ListBuffer
import scala.concurrent.{Await, Future}
val mp = ModelParams(C = 0.5, delta = 0.01)
val alphas = new Alphas(N=N/2, mp)
val ap = AlgoParams(maxIter = 30, batchProb = 0.99, learningRateDecline = 0.8,
                   epsilon = epsilon, quantileAlphaClipping=0.0)
var algo1 = new SG(alphas, ap, mp, kmf, sc, new ListBuffer[(Int,Int)])
```

The console output is:

mp: SVM.ModelParams = Model parameters: C: 0.5
delta: 0.01 alphas: SVM.Alphas = Alphas(20000,DenseVector(0.3063, ...)
Command took 0.55 seconds

The iterative gradient descent optimization is run with:

```scala
var numIt = 0
while(numIt < 5){
    algo1 = algo1.iterate(numIt)
    numIt += 1
}
```

The accuracy of successive iterations is printed to the console, so that the user can decide about the optimal iteration.
Here, we choose iteration nr. 4 (0 based counting) and use the respective $\alpha$ to calculate the accuracy on the test set:

```scala
import SVM.DataSetType.{Test, Train, Validation}
algo1.predictOn[Test, PredictionMethod.STANDARD, 0.5, 3]
```

The resulting accuracy is $2,841/4,000 = 71.03\%$.

Command took 27.74 seconds

## 6 Results

In this section, we will present results in terms of different measures of model quality and metrics related to the local and distributed algorithms.

### 6.1 Synthetic Data Set

**Data Analysis with the new Scala Library** We used a Gaussian mixture model to generate a synthetic data set with 200,000 instances and 10 inputs (see details above). The 200,000 instances were divided into a training ($n = 100,000$), validation ($n = 80,000$), and test set ($n = 20,000$). The Scala implementation achieved an overall maximal accuracy of 63.4\% at decision threshold 50\% (true positive rate 63.3\%, false positive rate 36.5\%) with a sparsity of 0\%. Given that we knew the true data generation mechanism, we used a value of $\gamma = 1.0$. Parameter $\epsilon$ was 0.001, and the optimization hyper-parameters $\psi=0.8$, $\eta = 0.01$, $C = 0.4$, and $p = 0.99$ were used. The algorithm needed 6 gradient descent iterations and finished after 33 minutes using less than 4GB Ram. Increasing the precision by decreasing epsilon from 0.001 to 0.0001, overall maximal accuracy increased to 67.5\% at decision threshold 50\% (true positive rate 67.7\%, false positive rate 33.0\%) with a sparsity of 1\%. The run time increased from 33 minutes to 1h 30 minutes and the maximal RAM increased from 4 to 10 GB. Analyzing this synthetic data set thus shows that it is possible to analyze huge data sets starting with an approximation of the kernel matrix and increasing the precision in order to improve accuracy.
Subset Selection  In order to evaluate the subset selection heuristic, a new synthetic data set with 300,000 instances was created (training: 150,000; validation: 120,000; test: 30,000). Using the default settings for subset selection (sampling probability 0.1 and maximum error probability 0.01), 126,146 out of 150,000 instances were selected from the training set. Again a value of $\gamma = 1.0$, $\epsilon = 0.001$, and optimization hyper-parameters $\psi=0.8$, $\eta = 0.01$, $C = 0.4$, and $p = 0.99$ were used. The gradient descent algorithm converged after six iterations. The model resulting from the first iteration already showed the lowest error on the validation set (87,189/120,000=72.66% accuracy) with 1% sparsity (on top of the subset selection). After evaluating the ROC curve a cutoff value of 49% was chosen (overall accuracy on the validation set was 72.53%, with 71.61% sensitivity and 73.44% specificity) (Figure 5). The resulting test set accuracy was 21036/30000=70.12% with sensitivity 10517/15026=69.99% and specificity 10508/14974=70.17% (Figure 6).

![Figure 5: Receiver Operating Characteristic curve for the synthetic data set and the validation data.](image-url)
6.2 Magic Data Set

**Data Analysis with Kernlab** Setting aside a test data set with 4,756 observations and training on a combined data set with 14,264 observations, the support vector machine fitted with the *kernlab* package achieved a final accuracy of 84.31% on the test set (Table 5). This R SVM implementation found 5,119 support vectors (64.11% sparsity). Training the model on the training set took between 30 and 35s.

<table>
<thead>
<tr>
<th>True class</th>
<th>Predicted</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>2,914</td>
<td>578</td>
</tr>
<tr>
<td>-</td>
<td>168</td>
<td>1096</td>
</tr>
</tbody>
</table>

Table 5: Test Confusion matrix for the Magic data set using the kernlab library.

**Data Analysis with the new Scala Library** Training on a training set with 9,508 observations, using a validation set \(n=4,756\) for hyper-parameter tuning only, and predicting on the same test set \(n=4,756\), the new implementation achieved an overall maximal accuracy of 82.8% at decision threshold 71% for the validation set (with true positive rate \(2887/3082=83.36\%\), and false positive rate 16.64%, sensitivity 91.66%,

![Receiver Operating Characteristic (ROC) curve for the synthetic data set and the test data.](image)
specificity 66.65% and a sparsity of 13%). The estimated optimal value of $\gamma$ was 0.9970, $\epsilon$ was 0.001, and the optimization hyper-parameters $\psi = 0.8$, $\eta = 0.01$, and $p = 0.99$ were used. The algorithm needed four gradient descent iterations and finished after 70 seconds, running on four parallel threads for most of the time (Fig. 7). Based on the ROC curve (Fig. 2), the user can decide the optimal cutoff value. In this case, the ROC curve indicated that a cutoff at threshold quantile 71% or 72% made sense because that was where the gradient of the curve shallows out and maximum overall accuracies were reached (Figure 3). Using the 71% cutoff, the resulting overall accuracy on the test set was 82.95%, with $2887/3082 = 93.67\%$ sensitivity and $972/1674 = 58.06\%$ specificity (Table 6). The fact that the test error was so close to the validation error indicated that the model did not overfit.

<table>
<thead>
<tr>
<th>True class</th>
<th>Predicted</th>
<th>+</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>2,887</td>
<td>702</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>195</td>
<td>972</td>
<td></td>
</tr>
</tbody>
</table>

Table 6: Confusion matrix for the test set of the Magic data set using the Scala library and decision threshold 71%.

**Subset Selection** Using the subset selection method based on the linear kernel approximation with a maximal error rate of 1% and sampling probability 10%, 8,067 out of 9,508 instances were selected based on the linear kernel projection. The gradient descent algorithm converged after five iterations with a model of 17% sparsity (on top of the subset selection). After evaluating the ROC curve a cutoff value of 67% was chosen (overall accuracy on the validation set: 82.53%, with 88.34% sensitivity and 71.93% specificity). The resulting test set accuracy was $3936/4756 = 82.76\%$ with sensitivity $2782/3082 = 90.27\%$ and specificity $988/1674 = 59.02\%$. 


6.3 HIGGS Data Set

Data analysis with Kernlab Setting aside a test set with 40,000 observations for training, validation, and test, the support vector machine fitted with the kernlab package \( (\sigma = 0.0299, \gamma = 559.69, C = 100) \), achieved a final accuracy of 65.88% on the test set (Table 7). This R SVM implementation found 27,781 support vectors (30.5% sparsity). Fitting the model on the training set took 1 hour and 46 minutes.

<table>
<thead>
<tr>
<th>True class</th>
<th>Predicted</th>
<th>+</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>11,355</td>
<td>6,257</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>7,391</td>
<td>14,997</td>
<td></td>
</tr>
</tbody>
</table>

Table 7: Confusion matrix for the test set of the Higgs data set using kernlab.

Data Analysis with the new Scala Library Training on the same test set with 40,000 observations, using a validation set \( (n = 40,000) \) for hyper-parameter tuning only, and predicting on the same test set \( (n = 40,000) \), the Scala implementation achieved an overall maximal accuracy of 63.54% at decision threshold 64% (true positive rate 63.02%, false positive rate 36.98%, sensitivity 75.91%, specificity 49.50%) with a sparsity of 8%. The estimated optimal value of \( \gamma \) was 0.2456, \( \epsilon \) was 0.0001, and the optimization hyper-parameters \( \psi = 0.8, \eta = 0.01, C = 80, \) and \( p = 0.99 \) were used. The algorithm needed 6 gradient descent iterations and finished after 40 minutes, running on four parallel threads for most of the time (Fig. 9). Based on the ROC curve (Fig. 8), there is no clear indication for an optimal cutoff threshold, although the slope of the curve drops...
at quantile 64% where the overall accuracy is highest (Fig. 8). The final overall accuracy on the test set was \( \frac{25414}{40000} = 63.54\% \) (sensitivity: \( \frac{16162}{21254} = 76.04\% \), specificity: \( \frac{9013}{18746} = 48.08\% \)). All computations (reading in the data sets, subset selection, training on the training set, validation on the validation set with ROC curve and the final test evaluation) took 40 minutes. Note that this model has a higher sensitivity than the model fit by *kernlab*, but on the downside has a lower specificity (compare Table 7 with Table 8).

<table>
<thead>
<tr>
<th>True class</th>
<th>Predicted +</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>16,162</td>
<td>9,733</td>
</tr>
<tr>
<td>-</td>
<td>5,092</td>
<td>9,013</td>
</tr>
</tbody>
</table>

Table 8: Confusion matrix for the test set of the Higgs data set using the Scala library.

**Subset Selection** Using the subset selection method based on the linear kernel approximation with maximum error rate 5% and 10% sampling probability, 31,549 out of 40,000 instances from the training set were selected. The gradient descent algorithm converged after six iterations with a model of 2% sparsity. After evaluating the ROC curve a cutoff value of 60% was chosen (overall accuracy on the validation set: 61.65%, with 70.47% sensitivity and 51.75% specificity). Based on a threshold of 60%, the resulting test set accuracy was 61.45% with sensitivity 70.80% and specificity 49.15%. All computations (reading in the data sets, subset selection, training on the training set, validation on the validation set with ROC curve and the final test evaluation) took 33 minutes.
Figure 8: Receiver Operating Characteristic curve for the Higgs data set.

Figure 9: Screenshot of the Java Visual Virtual Machine jvisualvm Monitor window during a run of the Scala SVM model for the Higgs data set. Note that all four available CPUs are used and that 6GB of RAM was necessary to fit the model.
7 Conclusions and Future Work

In this work, we implemented a stochastic gradient descent algorithm for kernelized support vector machines both for local parallelized applications and a distributed setting with access to an Apache Spark cluster. Stochastic gradient descent is the building block of all successfully distributed, large scale machine learning applications such as Spark MLlib [24] and TensorFlow [1], because it is straightforward to average over partial gradients [64]. There are distributed approaches to the support vector machine based on linear programming approximations [12] and the linear support vector machine [24, 41]. In TensorFlow, there is also a distributed implementation of non-linear support vector machines combining an explicit non-linear transformation with the primal support vector machine formulation [4]: The idea here is to approximate the infinite dimensional feature space of the Gaussian kernel by a lower dimensional representation and then operate directly in this space using the primal formulation [51]. There is, however, to my knowledge, no comparable implementation of the kernelized support vector machine based on distributed stochastic gradient descent for the standard formulation of the support vector machine.

Although there are different ways to generalize to multi-class situations [53, p.211-214][13, p.29-30], the support vector machine is a binary classifier, and for a binary classifier it is imperative to provide ways to tune the classification threshold. This is of paramount importance if the risks or costs associated to the different types of misclassification differ markedly or if the training set is highly unbalanced. In this work, a tool is provided that offers an intuitive way of finding the optimal classification threshold based on the validation set.

The standard approach to cross-validation is not well suited to big data situations: Tuning hyper-parameters using k-fold cross-validation schemes requires numerous model runs on slightly differing training sets. Especially for models with several interacting hyper-parameters, the computational costs are prohibitive in big data situations. Here, a tool was developed that automatically estimates the variance parameter of the Gaussian kernel function (4.2), hereby significantly reducing the need to tune hyper-parameters (only the hyper-parameter C has to be estimated). In another contribution to tackle this issue, a cross-validation scheme was developed (4.4) that, i) operates in an asynchronous way, ii) runs in parallel on different subsets of the validation set, iii) helps to find the optimal degree of sparsity, and iv) helps to avoid overfitting of the gradient descent algorithm using an early stopping strategy. The proposed 1-fold cross-validation scheme is appropriate for big data situations, where data set sizes of both the training and the validation set are substantial. In such a situation, the variance of the estimated classification error can be expected to be very low. As a result, it is not necessary to run that many replications using k-fold cross-validation. The key
advantage of this cross-validation approach is, furthermore, that it is integrated into the overall method and that the optimal sparsity and the optimal number of iterations are chosen automatically and independently of the gradient descent algorithm. It can thus be considered a meta-algorithm that runs on top of the stochastic gradient descent algorithm.

Processing big data sets with kernelized support vector machines requires a combination of strategies such as the approximation of kernel matrices and subset selection (compare 3.1.2 and [53, chapter 10]). In this implementation, a flexible threshold parameter $\epsilon$ is used, that controls the degree of sparsity of kernel matrices. Increasing $\epsilon$ can help to process big data sets by using an intuitive approximation of the kernel matrix. Tools are also provided to assess the final memory footprint of kernel matrices before deciding on a suitable value of $\epsilon$ (compare 5.2.1). While this approach can not compare with more elaborate matrix approximation methods [57], it is intuitive, fast, and easy to use.

Subset selection is also implemented in a very efficient way. Based on a number of random subsets, the analytically solvable linear support vector machine based on kernelized ridge regression is used to predict which instances have a low probability of entering the final model as support vectors. This approach provides a simple and computationally cheap way of filtering instances and thus decreasing the burden on the final model. The advantage of this approach is that it automatically adapts to the geometry of the problem. Using a low error probability, the risk of excluding future support vectors is very low. Although, a full implementation of the original method of David Morán Pomés which acts in feature space would be desirable[44], this must be left for future work. In the combination of matrix approximation and subset selection, the library can be used to explore data sets with up to 300,000 instances (compare 6.1) using commodity hardware (all experiments were run on a machine with a Intel x86_64 processor with four 2242.5 MHz CPUs and 16GB RAM).

Future Work  A number of extensions are possible based on the Scala library in its current form: First, the subset selection heuristic can be extended to non-linear, kernelized support vector machines as the model that generates the projection. This would include the power of the method but it would also be related to a higher computational burden. Secondly, functions can be arguments to other functions in the functional programming language Scala. Based on this powerful language trait, it should be possible to implement higher order kernel functions that are the combination of several separate kernel functions [47]. Thirdly, it would be handy to implement the standard Nystöm matrix approximation approach (compare 3.1.2). One of the key obstacles to the analysis of data sets with high $n$ are data situations that lead to kernel matrices with low sparsity. One possible approach could be to pre-process the inputs implement-
ing partial least-squares type 2 regression based on the NIPALS algorithm [69] after one hot encoding of the labels into two binary dummy variables. The resulting factors could then be used as inputs to the support vector machine. Reducing variance that is unrelated to the labels and working on orthogonal inputs should help to increase the sparsity of the kernel matrix and improve overall results. Apart from extensions to the library itself, the distributed algorithms should be tested more thoroughly on differently sized clusters to evaluate scalability.

Acknowledgments

I am very thankful to Jordi Castro for his excellent course on nonlinear optimization and his helpful comments regarding the formulation of the dual support vector machine optimization problem. I acknowledge the insights and suggestions of David Morán Pomés with whom I was discussing aspects of this work and who explained me his instance selection method. Josep Lluís Berral García introduced me to Spark in a course organized by the Barcelona Supercomputing Center and commented on sections of this work related to the Spark programming model. Luis Garrido Platero proofread part of the manuscript and provided me with helpful suggestions. Part of the cross-validation in the \texttt{R kernlab} code was based on a script of Martí Zamora i Casals. Last but not least, I acknowledge the invaluable support and council of Lluis A. Belanche Muñoz my supervisor.

References


8 Code Listings

8.1 Listing 1

```scala
def predictOn(dataType: SVM.DataSetType.Value, predictionMethod: PredictionMethod.Value, threshold: Double = 0.5) : Future[Int] = {
    assert(threshold>0.0 && threshold<1.0, "Invalid value for threshold! Must be between 0.0 and 1.0!"
    val promise = Promise[Int]
    //Turn the ListBuffer into a List
    val listOfFutures : List[Future[(Int,Int,Int)]] =
        optimalSparsityFuture.toList
    //List[Future[Int]] => Future[List[Int]]
    val futureList : Future[List[(Int,Int,Int)]] =
        Future.sequence(listOfFutures)
    //Wait for cross-validation results to choose the optimal level of sparsity:
    futureList onComplete {
        case Success(list) => {
            //Find the optimal iteration and associated sparsity and accuracy
            val (optSparsity, maxAccuracy, optIteration) =
                list.foldRight((0,0,0))((a,b) => if(a._2 <= b._2) b else a)
            println("Based on cross-validation, the optimal sparsity of: " +
                optSparsity + " with max correct predictions: " + maxAccuracy + " was
                achieved in iteration: " + optIteration)
            //Get the alphas for this optimal iteration
            val optAlphas : Alphas = alphaMap.getOrElse(optIteration, alphas)
            if(optSparsity > 0.0) optAlphas.clipAlphas(0.01 * optSparsity)
            println("Predict on the " +dataType.toString+ " set with prediction method
                " +predictionMethod.toString+ ".")
            predictionMethod match {
                case PredictionMethod.STANDARD => {
                    val promisedTestResults : Future[Int] = kmf.predictOn(dataType,
                        optAlphas)
                    promise.completeWith(promisedTestResults)
                }
                case PredictionMethod.THRESHOLD => {
                    val promisedTestResults : Future[Int] = kmf.predictOn(dataType,
                        optAlphas, threshold)
                    promise.completeWith(promisedTestResults)
                }
                case PredictionMethod.AUC => {
```
val promisedTestResults : Future[Int] = kmf.predictOnAUC(dataType,
  optAlphas)
promise.completeWith(promisedTestResults)

}
package test
import SVM._

object TestKernelMatrixWithoutSpark extends App {
    val gaussianKernel = GaussianKernel(GaussianKernelParameter(1.0))
    println(gaussianKernel)
    val N = 40000
    val dataProperties = DataParams(N = N, d = 10, ratioTrain = 0.5, ratioTest = 0.1)
    println(dataProperties)
    val d = new SimData(dataProperties)
    println(d.simulate())
    val epsilon = 0.001; val sampleProb = 0.01
    val ratioNonSparseElementsTrain = d.probeSparsity(epsilon, Train,
        gaussianKernel,sampleProb)
    val ratioNonSparseElementsValidation = d.probeSparsity(epsilon, Validation,
        gaussianKernel,sampleProb)
    val ratioNonSparseElementsTest = d.probeSparsity(epsilon, Test,
        gaussianKernel,sampleProb)
    println("Projected memory requirements for epsilon ="+epsilon+:")
    //There is also some overhead in the hash map, I assume one matrix element
    => 2 Int
    val intsPerKB = 256 * 0.5 //Integer = 32 bits = 4 Byte
    val N_train = N * 0.5; val N_val = N * 0.4; val N_test = N * 0.1
    println("Training matrix: "+ (ratioNonSparseElementsTrain * 
        N_train*N_train)/intsPerKB+"kB:")
    println("Validation matrix: "+ (ratioNonSparseElementsValidation * 
        N_val*N_val)/intsPerKB+"kB:")
    println("Test matrix: "+(ratioNonSparseElementsTrain * 
        N_test*N_test)/intsPerKB+"kB:")
}

8.3 Listing 3

```scala
import SVM._
import SVM.DataSetType.{Test, Train, Validation}
import scala.collection.mutable.ListBuffer
import scala.concurrent.{Await, Future, Promise}

object TestKernelMatrixWithoutSpark extends App {

  val gaussianKernel = GaussianKernel(GaussianKernelParameter(1.0))
  val N = 200000
  val dataProperties = DataParams(N = N, d = 10)
  val d = new SimData(dataProperties)
  d.simulate()
  val epsilon = 0.001
  val lmf = LeanMatrixFactory(d, gaussianKernel, epsilon)
  val mp = ModelParams(C = 0.4, delta = 0.01)
  val alphas = new Alphas(N = N / 2, mp)
  val ap = AlgoParams(maxIter = 30, batchProb = 0.99, learningRateDecline = 0.8, epsilon = epsilon, quantileAlphaClipping = 0.0)
  var algo = NoMatrices(alphas, ap, mp, lmf, new ListBuffer[Future[(Int, Int, Int)]])

  var numInt = 0
  while (numInt < ap.maxIter) {
    algo = algo.iterate(numInt)
    numInt += 1
  }

  val future = algo.predictOn(Validation, PredictionMethod.AUC)
  Await.result(future, LeanMatrixFactory.maxDuration)

  val future3 = algo.predictOn(Test, PredictionMethod.THRESHOLD)
  Await.result(future3, LeanMatrixFactory.maxDuration)
}
```
package SVM
import scala.collection.mutable.ListBuffer
import scala.concurrent.{Await, Future}
import SVM.DataSetType.{Test, Train, Validation}

object TestMAGIC extends App {
  val workingDir = "/home/user/data/"
  val pathTrain = workingDir + "magicTrain.csv"
  val pathValidation = workingDir + "magicValidation.csv"
  val pathTest = workingDir + "magicTest.csv"
  val indexLabel = 11
  val indexSkip = 0 //The first column has to be skipped (line nr!)
  val transLabel = (x:Double) => if(x<=0) -1 else +1
  val d = new LocalData()
  d.readTrainingDataSet (pathTrain, ',', indexLabel, transLabel, indexSkip)
  d.readTestDataSet (pathTest, ',', indexLabel, transLabel, indexSkip)
  d.readValidationDataSet(pathValidation, ',', indexLabel, transLabel, indexSkip)
  d.tableLabels()
  val medianScale = d.probeKernelScale()
  println("Estimated kernel scale parameter: "+medianScale)
  val epsilon = 0.0001
  val kernelPar = GaussianKernelParameter(medianScale)
  val gaussianKernel = GaussianKernel(kernelPar)
  val kmf = LeanMatrixFactory(d, gaussianKernel, epsilon)
  val mp = ModelParams(C = 100, delta = 0.01)
  val alphas = new Alphas(N=d.N_train, mp)
  val ap = AlgoParams(maxIter=10,batchProb = 0.99,learningRateDecline = 0.8,epsilon = epsilon)
  var algo = NoMatrices(alphas, ap, mp, kmf, new ListBuffer[Future[(Int,Int,Int)]]
  var numInt = 0
  while(numInt < ap.maxIter){
    algo = algo.iterate(numInt); numInt += 1
  }
  Await.result(algo.predictOn(Test, PredictionMethod.STANDARD), LeanMatrixFactory.maxDuration)
}