KU LEUVEN

Fixed Size Least Squares Support Vector Machines: A Scala based programming framework for Large Scale Classification

Mandar Chandorkar

Thesis submitted for the degree of Master of Science in Artificial Intelligence, option Engineering and Computer Science

Thesis supervisors:
Prof. dr. ir. Bart De Moor
Prof. dr. ir. Johan A.K Suykens

Assessor:
Dr. Raghvendra Mall

Mentors:
Oliver Lauwers
Dr. Raghvendra Mall

Academic year 2014 – 2015

Preface

I would like to thank everybody who kept me motivated the last year, especially my promoters and my supervisors. I would also like to thank the jury for reading the text. My sincere gratitude also goes to my family and friends without whose support none of this would be possible.

Mandar Chandorkar

Contents

Abstract

We propose *FS-Scala*, a flexible and modular *Scala* based implementation of the Fixed Size Least Squares Support Vector Machine (FS-LSSVM) for large data sets. The framework consists of a set of modules for (gradient and gradient free) optimization, model representation, kernel functions and evaluation of FS-LSSVM models. A kernel based *Fixed-Size Least Squares Support Vector Machine* (FS-LSSVM) model is implemented in the proposed framework, while heavily employing distributed *MapReduce* via the parallel computing capabilities of *Apache Spark*. Global optimization routines like *Coupled Simulated Annealing* (CSA) and *Grid Search* are implemented and used to tune the hyper-parameters of the FS-LSSVM model. Finally, we carry out experiments on benchmark data sets like *Magic Gamma* and *Adult* and *Forest Cover Type*, recording the performance and tuning time of various kernel based FS-LSSVM models.

List of Figures and Tables

List of Figures

List of Tables

List of Abbreviations and Symbols

Abbreviations

SVM	Support Vector Machine
FS-LSSVM	Fixed Size Least Squares Support Vector Machine
SPFS-LSSVM	Sparsified Primal FIxed Size Least Squares Support Vector Machine
SD-LSSVM	Sub-sampled Dual Least Squares Support Vector Machine
SSD-LSSVM	Sparsified Sub-sampled Dual Least Squares Support Vector Machine
CG	Conjugate Gradient
L-BFGS	Limited Memory Broyden-Fletcher-Goldfarb-Shanno
SGD	Stochastic Gradient Descent
CSA	Coupled Simulated Annealing
AFE	Automatic Feature Extraction
API	Application Programming Interface
KKT	Karush Kuhn Tucker
RHKS	Reproducing Kernel Hilbert Space
SVC	Support Vector Classification
SVR	Support Vector Regression
SMO	Sequential Minimal Optimization
QP	Quadratic Programming
ROC	Receiver Operating Characteristic
OOP	Object Oriented Programming
FP	Functional Programming
JVM	Java Virtual Machine

Symbols

n	Number of training instances
m	Number of prototypes selected
X	Training data $x_i \, \epsilon \, \mathbb{R}^d{}_{i=1...n}$
Y	Vector of labels $y_i \, \epsilon \, \{-1, 1\}_{i=1...n}$ for the training set X
$\mathcal{J}(w, b)$	Lagrangian objective function constructed to solve the FS-LSSVM optimization problem
$\mathfrak{D}_n$	The training data set consisting of features and labels (X, Y)
$\mathfrak{W}_m$	Working set of size m chosen from the training set $\mathfrak{D}_n$
$\mathcal{H}$	Hilbert space
$<, >_{\mathcal{H}}$	Inner product of the Hilbert space $\mathcal{H}$
$\mathcal{L}_p(C)$	Hilbert space of p-integrable functions defined on a compact set C, having the associated norm $\|\|f\|\|_{\mathcal{L}_p} = (\int_C f^p d\mu(C))^{1/p}$
γ	Regularization parameter used in the LSSVM formulations outlined in equations (2.2) and (2.4)
K	Symmetric positive semi-definite Kernel Function of the form $\mathbb{R}^d \times \mathbb{R}^d \longrightarrow \mathbb{R}$
$\mathcal{T}$	Integral operator of kernel function K
Ω	Kernel matrix constructed from $\mathfrak{W}_m$, entries given by $[K(x_i, x_j)]_{m \times m}$
$\hat{\phi}$	Approximate feature map $\mathbb{R}^d \longrightarrow \mathbb{R}^m$ obtained from AFE on a the kernel matrix Ω
$\hat{\Phi}_e$	The extended feature matrix $\begin{pmatrix} \hat{\phi}_1(x_1) & \cdots & \hat{\phi}_m(x_1) & 1 \\ \vdots & \ddots & \vdots & \vdots \\ \hat{\phi}_1(x_n) & \cdots & \hat{\phi}_m(x_n) & 1 \end{pmatrix}$
A	Matrix $\left(\hat{\Phi}_e^\intercal \hat{\Phi}_e + \frac{I_{m+1}}{\gamma}\right)$ required for CG based training of the FS-LSSVM
c	Vector $\hat{\Phi}_e^\intercal Y$ required for CG based training of the FS-LSSVM
$(\hat{w}, \hat{b})^T$	Parameters of an FS-LSSVM model estimated from A and c

Chapter 1

Introduction

The 21^{st} century stands out in how mankind learned the value of storing and making predictions/decisions from large volumes of data. A significant aspect of large scale data analysis is distributed computation frameworks like *High Performance Computing*, *Message Passing Interface* etc. Recently large scale commodity hardware clusters have replaced the two former frameworks as the most popular model for parallel data analysis. With this crucial change in hardware came a change in computational models as well. It is at this juncture that distributed *Map Reduce* became the de-facto computational philosophy for large scale data analysis and words such as *Hadoop* [2, 22, 18] and *Apache Spark* [53, 5] have become synonymous with large scale data analysis and machine learning.

Along with innovation in hardware design and distributed computing models, there came a need for good programming libraries and frameworks to work with various Machine Learning models on large data sets. It was demonstrated in [27] that a gigantic language corpus encapsulates almost all aspects of human language and speech. So far the prevalent 'motto' in the Internet industry has been "large data, simple models". Often, this is misunderstood as the Machine Learning translation of *Occam's Razor*. The bias-variance trade-off [49] (appendix A.1) is a far better mechanism to ensure that a model does not become overly complex, and this, rather than restricting the user to simple models, is the real Occam's razor in training a model.

Therefore, in order to extract maximum value from large scale data, it is important to have the flexibility to train and compare different model families before arriving at the one that fits the requirement of the user. Therefore one must be able to train general nonlinear models and tweak them by changing the various components which they employ to learn (i.e., a model may be linear or kernel based, it can be optimized by various methods like *Stochastic Gradient Descent*, *Conjugate Gradient*, etc.). This is not possible in a rigid, monolithic programming framework. Modularity, extensibility and ease of usage are of paramount importance while designing Machine Learning software for large scale data applications.

Scala [41], a multi-paradigm Java Virtual Machine (JVM) based programming language has gained popularity for its expressiveness and performance. It's power and easy interpolability with Java has contributed to its adoption in software architectures that heavily rely on Java and its ecosystem. The current state of the art in distributed Machine Learning in Scala is the *MLLib* module in *Apache Spark* [36]. It has implementations of Linear

SVM and Logistic Regression for solving binary classification problems. But a crucial component missing in *MLLib* and all distributed Machine Learning libraries is the ability to learn classification models with nonlinear decision boundaries. FS-Scala aims to solve the problem of scalable non-linear classification models by implementing the *Fixed-Size Least Squares Support Vector Machine* (FS-LSSVM) algorithm [24, 47] with model tuning capabilities.

In recent literature we find sparse reductions to FS-LSSVM methods [34, 33]. The authors in [34, 33] explored the sparsity vs error trade-off [1] for FS-LSSVM models. Even though they run experiments on large scale data sets like *Forest Cover*, the scalability of these methods are restricted to available memory on a single machine. Moreover, they don't exploit the possibility of parallelism available in several components of the FS-LSSVM model.

Another work [31] converts the Big Data into a Big Network and then uses a network based subset selection technique (*Fast and Unique Representative Subset selection* (FURS) [32]) to obtain a representative subset of the original data. It then builds a FS-LSSVM model using this subset. However, in this thesis we showcase that we can distribute the subset selection computation which maximizes the *Quadratic Rènyi Entropy* for Big data sets and use the generated subset as the set of prototype vectors (PV) essential for building the FS-LSSVM model.

The rest of this thesis is organized as follows.

- Chapter 2 briefly discusses the classical Support Vector Machine (SVM) as well as the LSSVM formulations. In the case of the LSSVM, the dual formulation is discussed motivating the introduction of the FS-LSSVM which solves the LSSVM in the primal for large data sets.

- Chapter 3 compares and contrasts various programming languages and paradigms, justifying the choice of Scala in the implementation of the FS-LSSVM. The rest of the chapter focuses on the salient features of the current SVM and LSSVM software implementations and motivates the issues tackled by FS-Scala. Chapter 3 concludes with the contributions of *FS-Scala* in LSSVM software design and implementation.

- Chapter 4 introduces distributed *MapReduce* and details the computational phases of the FS-LSSVM which are performed in a distributed manner in *FS-Scala*.

- Chapter 5 discusses experiments conducted using *FS-Scala* on the *Magic Gamma*, *Adult* and *Forest Cover Type* data and tabulates the performance of various kernel based as well as linear FS-LSSVM models on the selected data sets.

- The thesis concludes 6 with a discussion of the results of experiments of chapter 5 and further research directions that can be pursued with respect to the development of *FS-Scala*.

[1] Sparsity vs error trade off refers to how the model error increases when the number of prototype vectors is reduced via the L_0 regularization on model parameters.

Chapter 2

Least Squares Support Vector Machines

2.1 Support Vector Machines

The classical soft margin Support Vector Machine (SVM) model as proposed by Cortes et. al [23] relies on maximizing the margin between the separating hyper-plane while minimizing the mis-classification error on the input patterns. It is formulated in equation (2.1) below.

$$\begin{aligned} \min_{w,b,e} \quad & \mathcal{J}(w,e) = \frac{1}{2} w^\intercal w + \gamma \sum_{i=1}^{N} e_i \\ \text{s.t.} \quad & y_i[w^\intercal \phi(x_i) + b] \geq 1 - e_i, \; i = 1, \ldots, N. \\ & e_i \geq 0, \; i = 1, \ldots, N. \end{aligned} \tag{2.1}$$

The functions $\phi(x)$ are suitably chosen basis functions, which describe an inner product space called a *Reproducing Kernel Hilbert Space* (RKHS) [38]. The RKHS generated by $\phi(x)$ can be identified by a positive semi-definite Kernel function $K(x,y) = \phi(x)^T \phi(y)$, a class of kernel functions popularly known as *Mercer* kernels [37]. Generally the product $\phi(x)^T \phi(y)$ is replaced by the Kernel function $K(x,y)$ without explicitly calculating or solving for $\phi(x)$, this is known in SVM literature as the *kernel trick*.

It is well known that solving the optimization problem in equation (2.1) leads to a *sparse* representation of the best separating hyper-plane between the two classes in the data. This can be achieved by introducing Lagrange multipliers and applying the Karush Kuhn Tucker (KKT) conditions yielding a classifier of the form $w = \sum_{k=1}^{N} \alpha_k y_k \phi(x_i)$. Most of the multipliers α_k are zero due to the KKT complementary slackness condition, while the non zero multipliers determine the *support vectors*. How ever the classical soft margin SVM formulation also leads to a *Quadratic Programming* (QP) problem in terms of the Lagrange multipliers which often leads to $O(N^3)$ time complexity in the model training.

2.2 Least Squares Support Vector Machines

Least Squares Support Vector Machines (LSSVM) proposed by Suykens et.al [47, 48] modify the classical soft margin SVM formulation above by replacing the *hinge* loss function in (2.1) with the *squared error* loss function and the inequality constraints (in terms of the error/slack variables e_i) with equality constraints, we obtain the following optimization problem (2.2).

$$\begin{aligned} \min_{w,b,e} \quad & \mathcal{J}(w,e) = \frac{1}{2} w^\intercal w + \frac{\gamma}{2} \sum_{i=1}^{N} e_i^2 \\ \text{s.t.} \quad & y_i[w^\intercal \phi(x_i) + b] = 1 - e_i, i = 1, \ldots, N. \end{aligned} \tag{2.2}$$

Introducing the Lagrangian and applying the KKT conditions gives us the solution of the problem in the dual (2.3).

$$\left[\begin{array}{c|c} 0 & y^\intercal \\ \hline y & \Omega + \gamma^{-1} I \end{array}\right] \left[\begin{array}{c} b \\ \hline \alpha \end{array}\right] = \left[\begin{array}{c} 0 \\ \hline 1_v \end{array}\right], \tag{2.3}$$

In the above equation, the quantities Ω_{kl}, α and $K(x_k, x_l)$ are given by the following expressions.

$$\begin{aligned} \Omega_{kl} &= y_k y_l K(x_k, x_l) \\ \alpha &= [\alpha_1; ...; \alpha_N] \\ K(x_k, x_l) &= \phi(x_k)^\intercal \phi(x_l) \end{aligned}$$

This solution implies a loss of sparsity as compared to the classical SVM since each point becomes a support vector. However, we gain linearity of the solution (i.e. we do not have to solve the *Quadratic Programming* problem as in the classical SVM). This affords us greater freedom in choosing optimization algorithms to solve the formulation in (2.2), apart from the standard *Stochastic Gradient Descent* we can also employ algorithms for linear systems such as *Conjugate Gradient*, *Gauss Seidel*, *Jacobi* etc.

2.3 FS-LSSVM: Tackling Large Scale Problems

It is observed that solving the problem (2.3) in the dual is not advantageous for large scale analysis as the size of the solution matrix is equal to the size of the original data. In order to make the training of kernel based SVM models for large scale data applications feasible, one must solve the optimization problem in the primal and make approximations to the computation of the kernel matrices. The Fixed-Size LSSVM (FS-LSSVM) as proposed by De Brabanter, Suykens et. al [24, 47] consists of solving the LSSVM problem in the primal as follows.

$$\min_{w,b} \frac{1}{2} w^\intercal w + \frac{\gamma}{2} \sum_{i=1}^{n} \left(y_i - w^\intercal \hat{\phi}(x_i) - b \right)^2. \tag{2.4}$$

The solution to equation 2.4 is given by:

$$\begin{pmatrix} \hat{w} \\ \hat{b} \end{pmatrix} = \left(\hat{\Phi}_e^\intercal \hat{\Phi}_e + \frac{I_{m+1}}{\gamma} \right)^{-1} \hat{\Phi}_e^\intercal y, \tag{2.5}$$

$$\text{where} \quad \hat{\Phi}_e = \begin{pmatrix} \hat{\phi}_1(x_1) & \cdots & \hat{\phi}_m(x_1) & 1 \\ \vdots & \ddots & \vdots & \vdots \\ \hat{\phi}_1(x_n) & \cdots & \hat{\phi}_m(x_n) & 1 \end{pmatrix}.$$

In the above formulation, $\hat{\phi}(x_k)$ is an approximation to the true feature map $\phi(x_k)$ which is related to the kernel $K(x_i, x_j) = \phi(x_i)^\intercal \phi(x_j)$ (Mercer's theorem). The approximate feature map $\hat{\phi}(x_k)$ is calculated using the Nyström method as outlined in [24, 34, 33]. A low rank approximation to the kernel matrix is constructed by iteratively calculating a subset of the original data which maximizes the *Quadratic Rènyi Entropy*. This procedure of extracting $\hat{\phi}(x_k)$ from a data set, given a kernel function, is called *Automatic Feature Extraction* (AFE) (see appendix A.2).

Algorithm 1: Tuning FS-LSSVM

1 **Data**: Data Set, Kernel, Global Optimization routine, grid parameters
2 **Result**: Proposed Tuned FS-LSSVM model
3 Pre-process the data by mean scaling.;
4 **Calculate the prototype set by maximizing the Quadratic Rènyi Entropy in parallel using MapReduce.**;
5 Initialize a grid for the hyper-parameters;
6 **while** *termination of global optimization routine* **do**
7 Initialize the kernel using the hyper-parameters. Do AFE on the kernel matrix constructed from the prototypes, using the Nystrom method;
8 **Train the resulting model**;
9 evaluate the **cross validation score** for the particular hyper-parameter values;
10 **end**

Kernel based models are sensitive to hyper-parameters. In the case of FS-LSSVM we have to tune the model with respect to γ the regularization parameter and the parameters of the kernel chosen. Models are generally compared with their cross-validation performance in which case the objective cost function with respect to the hyper-parameters is in general non-smooth and non-convex. Gradient free methods like Grid Search, Nelder Mead [39] and Coupled Simulated Annealing [52] are suitable to tackle the problem of model selection for FS-LSSVM based kernel models. Algorithm 1 explains the steps involved in tuning the FS-LSSVM model with the bold part representing our contributions in this thesis, which have been implemented in the distributed *MapReduce* environment of *Apache Spark*.

Chapter 3

FS-Scala

3.1 Scientific Computing: An Overview

The term *Scientific Programming* must be interpreted with proper context, as the very first applications of the primitive computational infrastructure developed during the second world war were conducting numerical simulations of various engineering problems of practical significance. It must thus be recognized that only after large scale adoption of computers for business applications became popular, that the term *Scientific Computing* became relevant as a sub-domain of Computer Science.

Nevertheless it is instructive to take a quick glance at the history of programming languages in order to motivate the design philosophy behind *FS-Scala* and the *MapReduce* computational paradigm which has been applied extensively in modern large scale data processing frameworks.

We classify programming languages on two broad issues, for an in-depth treatment of the subject one may refer to some of the canonical texts in the area [45], [12].

- Translation to Machine level code. Programs written in any language have to be translated to low level machine readable instructions or codes, this may be achieved in two ways.
  - Compilation: The high level program is "compiled" or translated into low level executable code, this is achieved in various ways on different platforms (i.e. .exe file on Windows based systems, .sh files on *nix systems). Note that this process has to be done once and the executable code can be run many times.
  - Interpretation: Instead of translating the high level code before hand, an interpreter translates all the content of a program into low level instructions every time it is executed.
- Paradigms.
  - Imperative Programming. Programs consist of instructions which modify a given 'state', which may refer to a memory location or variable.

- Object Oriented Programming (OOP). Programs consist of entities called 'objects' which contain data, called 'fields' and subroutines to modify their data.
- Functional Programming (FP). Programs consist of expressions which take input and output the results, its worth noting that the input and output data structures are immutable. It is worth noting that in Functional Programming one may pass a function itself as an argument to another function/subroutine, this behavior is colloquially referred to by the phrase "code as data".

There are advantages and limitations of each programming language or paradigm, which must be understood before choosing one for a particular application. It is well known that compiled languages have much better performance than interpreted languages, though one must keep in mind that being compiled or interpreted are not inherent characteristics of the languages themselves rather practical implementation details. This advantage that languages like C and Fortran possess is evidenced in the fact that much of the Python scientific programming frameworks use underlying *C* or *Fortran* primitives for fast computation, *scikit-learn* [43] being the best example.

So far *Python* and *MATLAB* have been the dominant languages used in computation for engineering and science applications, this is because of their ease of learning with respect to syntax and the ready availability of packages which can be used to extend their capability for specific applications like signal processing (in *MATLAB*) and Natural Language Processing (see *NLTK* [17]) in *python*.

While *python* and *MATLAB* are convenient for quick prototyping of new algorithms and models, for production machine learning systems their performance makes them unsuitable for those applications. The most common languages used for production systems are *C/C++* and *Java*.

The Java Virtual Machine (JVM) has become a standard platform for enterprise applications in the past two decades. The *Apache Hadoop* [2] and *Mahout* [4] big data and machine learning frameworks are written entirely in *Java*. *Apache Mahout* has been applied to large scale problems [42] of categorization, clustering [15] and in recommendation, [29].

Apart from *Hadoop* and *Mahout*, other prominent Java machine learning frameworks are *Mallet* [35], *Deeplearning4j* [7], *BoofCV* [11], *MOA* (Massive Online Analysis) [16] and *javaML* [10] among many others. As such, the JVM offers a sound platform for future efforts in large scale machine learning. One of the recent developments has been the development of JVM based languages which have embraced the FP paradigm along with the inherent Object Oriented (OO) nature of the *Java* language and the JVM.

Scala, MapReduce and Data Processing

Scala is a hybrid, multi-paradigm language developed at the EPFL, Lausanne in 2004 [41]. It is a JVM based language because all *Scala* code is translated to JVM byte code to be execute like any other Java program. This gives it seamless interpolability with Java and all the programming frameworks written in Java. It is primarily an OOP language which has many functional characteristics like support for *tail recursive* optimization, function composition and immutable data structures. Its is inspired by many prominent FP languages like *Lisp*, *Haskell*, *OCaml* etc.

An important concept or paradigm espoused in many FP languages is *MapReduce*, it can be readily observed that many programs/computations in languages like *Lisp*, *Scala* use *MapReduce* on immutable data structures like lists in order to obtain results. In chapter 4 we delve further into the *MapReduce* paradigm and why it is so central to modern big data architectures. It is instructive to note that in *Scala*, data structures like lists, maps and so on have in built `map` and `reduce` functions which enable the programmer to think and code in this paradigm.

It is for these reasons that *Scala* has become crucial to the implementation of modern big data frameworks like *Apache Spark* and the language of choice for us in the implementation of the FS-LSSVM.

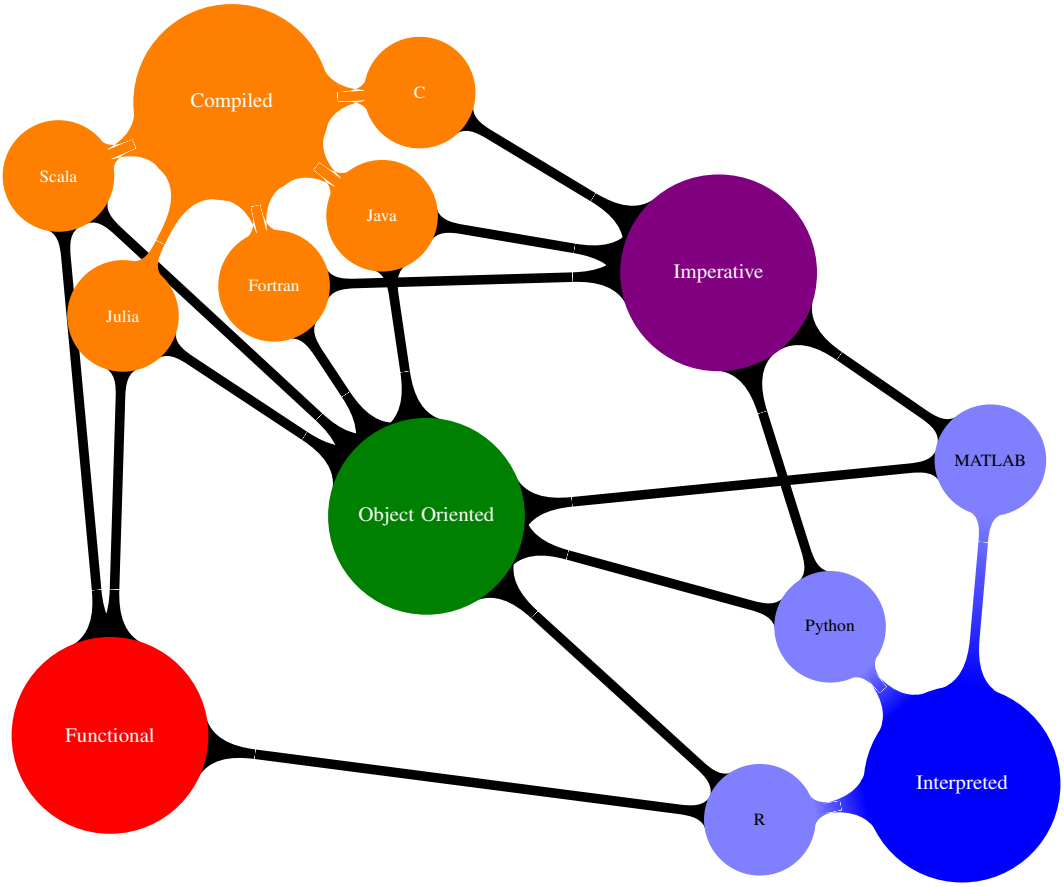

FIGURE 3.1: Overview of Scientific Computing Languages

3.2 SVM Software: The state of the art

Since its inception, there have been a number of implementations of the classical soft-margin SVM, one can find a diverse list of software at [1]. Notable implementations include *LibSVM* [21], *SVM*^{Light} [28] and *mySVM* [44] among others. With respect to the LSSVM there is

an implementation by Suykens et.al [25] called *LSSVMLab* written in *MATLAB*. Below we review salient features of a select subset of SVM software implementations.

- LibSVM: A C/C++ software package that supports Support Vector Classification (C-SVC and ν-SVC), Regression (ε-SVR and ν-SVR) and Density Estimation. It supports binary as well as multi-class classification. Weighted training for unbalanced data sets, along with support for kernels (precomputed and non precomputed kernel matrices). It also supports Sequential Minimal Optimization (SMO) based training of SVM models. One of the most popular SVM implementations with several language ports and multiple libraries (ex. Scikit Learn [43]) which use it as a back-end SVM model builder.

- $\text{SVM}^{\text{Light}}$: A C based software package that aims to reduce the training times for SVM models using faster optimization algorithms which are achieved by a combination of kernel evaluation caching, heuristics and support vector selection based on *steepest feasible descent*. Recently a new implementation SVM^{perf} has also been introduced to further speed up model training compared to SVM^{light} for large data sets and to optimize multivariate performance measures like F1 measure, ROC area, etc. For an implementation applicable to structures like trees, one can also use $\text{SVM}^{\text{struct}}$.

- LSSVMLab: A MATLAB software toolbox to train and test LSSVM models. It has support for kernels as well as Bayesian LSSVM formulations which calculate posterior probability of a model or set of hyper-parameters.

- Apache Spark SVM: This SVM model implementation exists as a part of the *MLLib* machine learning library in the *Apache Spark* big data and cluster computing platform. It is one of the several linear models that are offered in this libraries. *MLLib* comes bundled with optimization algorithms such as Stochastic Gradient Descent (SGD) [19] and Limited Memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) [14, 20], both methods being iterative require multiple passes through the training data to learn a SVM model instance.

3.3 FS-Scala: Motivation and Design

FS-Scala tackles three major issues w.r.t. the implementation of the FS-LSSVM:

- **Tuning Kernel Models**: Since the performance of kernel based models is sensitive with respect to the choice of hyper-parameters, one has to choose a mechanism of model selection or hyper-parameter optimization. In FS-Scala, we implement the Grid Search and Coupled Simulated Annealing global optimization algorithms for model tuning.

- **Parallel Computation**: Big Data analysis requires the distribution of computational work load, *MapReduce* is the dominant paradigm employed for writing distributed data processing programs. In FS-Scala we leverage *MapReduce* to distribute the computation in the pre-processing, training and cross-validation tasks.

- **Infrastructure Flexibility**: The big data landscape has many tools which enable the storage and analysis of large streams of data, they consist of technologies such as, but not limited to *Apache Spark*, *Hadoop*, Graph Databases like *Titan* [8], *OrientDB* [6], *Neo4j* [3]. Creating a powerful framework for model training and evaluation requires the decoupling of storage and processing infrastructure from the actual logic that implements the architecture of learning models.

Architecture

FIGURE 3.2: Schematic structure of FS-Scala

Figure 3.2 shows the organization of modules in FS-Scala. It can be decomposed into five principal modules:

- Model Classes: This is the core set of classes which form the heart of the library,

a number of abstract model categories are defined each with its own set of defined behaviours.

- Optimization application programming interface (API): A module which houses the implementation of common optimization methods (i.e. Gradient and Gradient free). Currently FS-Scala has implementations for Conjugate Gradient, Gradient Descent, Grid Search and Coupled Simulated Annealing [52] (CSA).

- Kernels: FS-Scala is equipped with a powerful abstract API for representing kernel functions. The module has two abstract classes to outline the behaviors of kernels used in SVM based applications as well as density estimation. The library comes bundled with an implementation for AFE as well as for common SVM kernels i.e. Linear, Radial Basis Function (RBF), Polynomial, Laplace, Exponential. New kernel functions can be easily added to the library by extending the base classes in this module.

- Evaluation Metrics: We have implemented evaluation metrics for Binary Classification and Regression problems. Further more, the implementation of binary classification performance expressed as the area under Receiver Operating Characteristic (ROC), is carried out using *MapReduce* in a *single pass* fashion through the evaluation data points, which can be seen in algorithm 5. Calculating the area under the ROC curve in a *single pass* fashion greatly increases the speed of the eventual FS-LSSVM source code.

- Miscellaneous Utilities: This module contains code to carry out auxiliary tasks for model learning and optimization. It contains the implementation of entropy calculation, summary statistics, prototype selection as well as a set of various functions which can be required for implementing new model classes using the library.

3.4 Contributions

3.4.1 Hyper-parameter tuning

The modular design approach of FS-Scala provides the user flexibility to apply different tuning algorithms on kernel as well as linear LSSVM models. Kernel based models in FS-Scala all implement the interface *GloballyOptimizable* contained in the optimization module (see Figure B.2 in appendix B). *GlobalOptimizer* and its subclasses (i.e. *GridSearch* and *CoupledSimulatedAnnealing*) all optimize models which implement the GloballyOptimizable interface.

3.4.2 Speed Improvements

- **Fast Entropy Calculation**: Rènyi Entropy based iterative prototype selection is of complexity $O(m^2)$ where m is the number of prototypes to be selected. In FS-Scala after the initial entropy calculation, subsequent computations are only calculated by the entropy difference due to the exchange of one element between the prototype set

and rest of the training data. Calculating the entropy differences each iteration is of the order $O(m)$.

- **Single Pass Model Training**: FS-Scala contains an implementation of CG (algorithm 3) which requires only a single pass through the training data to calculate the matrices $A = \left(\hat{\Phi}_e^\intercal \hat{\Phi}_e + \frac{I_{m+1}}{\gamma}\right)$ and $c = \hat{\Phi}_e^\intercal y$ which are then used to carry out local CG iterations.

- **Fast Cross Validation**: By implementing the fast v-fold cross validation as outlined in [24], FS-Scala eliminates the need to go through the entire training data to calculate the matrices A_v and c_v required for each fold.

- **Caching**: The matrices A and c require a complete pass through the training data to be calculated, but their calculation requires only the original data (X, Y) and the approximate feature map $\hat{\phi}$ which in turn depends on a precomputed kernel matrix with respect to a kernel parameter. By caching the matrices A and c for a particular value of the kernel parameter, FS-Scala can reuse them for multiple values of the regularization parameter γ. This greatly reduces the total number of passes through the data set the FS-LSSVM algorithm has to carry out, a factor which crucial for tuning of large scale kernel based SVM models.

The implementation of the FS-LSSVM in FS-Scala, outlined in algorithm 1, chapter 2 is as described in in De Brabanter et al. [24].The FS-Scala software is available at [9].

Chapter 4

Distributed Computation Support

4.1 Map Reduce

Although *MapReduce* [26] has been the focal point of the large scale machine learning surge, as a computational paradigm it is hardly new. Programming languages with FP capabilities (ex. Haskell, Scheme, Scala) have operations `map` and `reduce` on various data immutable (and sometimes mutable) data structures like Maps, Lists, Arrays, etc. One can argue that *MapReduce* which relies on the "code as data" philosophy is quite inherent in the FP paradigm.

It is worthwhile to note that although *MapReduce* is supported in several programming languages, the current distributed implementations of *MapReduce* like *Apache Hadoop* [2] and *Apache Spark* have gained popularity because of their ability to distribute these computations across massive clusters. We inspect *Map Reduce* in more detail to understand how this capability is achieved. Figure 4.1 below shows a schematic picture of distributed `MapReduce`.

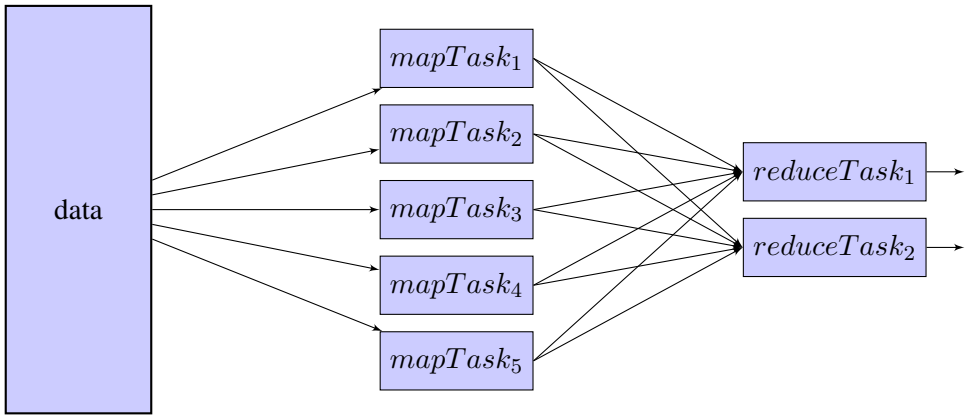

FIGURE 4.1: Schematic Diagram of Distributed Map Reduce

It can be broadly divided into two major phases or steps, although there are more phases that can be optionally applied, the reader is urged to refer to references in this area for a

more in-depth explanation [46], [51] [50].

- *Map*: Each file is divided into splits, on each node/split a `map` function (of the form $V \longrightarrow W$) which is self contained (it is assumed to be independent of data on other splits) is applied on each data split. Due to this inter-partition `map` task independence (also referred to as locality), this phase can be carried out in a (massively) parallel fashion.

- *Reduce*: This consists of application of the so called `reduce` function which is of the form $W \times W \longrightarrow W$, it "combines" or "merges" two quantities/entities which are outputs of the `map` function in some meaningful way. In order for this phase to be implemented in parallel, it is required that this `reduce` function be associative in the algebraic sense, although commutativity is not a requirement of this function, it allows for further distributed optimization of this computation.

- *Combine* (Optional): This is an optional phase which may be applied before the *Reduce* phase, it is a function called `combine` whose functional signature is identical to that of the `reduce` function, its utility is primarily reducing the overhead on the *Reduce* phase by preprocessing the intermediate output of the `map` functions.

4.2 Application in FS-Scala

FS-Scala leverages *MapReduce* in two separate stages as outlined below, using *Apache Spark* to distribute the workload amongst a number of processes/threads.

Model Training

Algorithm 2: Calculate feature matrices from data using MapReduce: $FeatureMat$

Data: $X = [x^i],\ x^i \,\epsilon\, \mathbf{R}^n, \hat{\phi} : \mathbf{R}^n \longrightarrow \mathbf{R}^m, Y = [y^i], y^i \epsilon \mathbf{R}$
Result: $\left(\hat{\Phi}_e^\intercal \hat{\Phi}_e\right),\ \hat{\Phi}_e^\intercal Y$
begin
  $MapFn(x, y)$:
    $M \longleftarrow \hat{\phi}(x)\hat{\phi}(x)^T$
    $v \longleftarrow \hat{\phi}(x)\, y$
  $emit(M, v)$
begin
  $RedFn((M, v), (M', v'))$:
  $emit(M + M', v + v')$
begin
  $(F, v) \longleftarrow MapReduce(X, MapFn, RedFn)$
  return (F, v)

For training a single FS-LSSVM model instance (given a fixed value of hyper-parameters), one must apply the feature map $\hat{\phi}$ on each training example and generate the extended feature matrix $\hat{\Phi}_e$ in order to solve the linear system in equation (2.5) chapter 2, this is achieved by algorithm 2. As noted earlier, distributed *MapReduce* relies on the independence and locality of data processing operations. In the map phase of algorithm 2, locality is achieved by emitting $\hat{\phi}(x)\hat{\phi}(x)^T$ and $\hat{\phi}(x)y$ for each data point $(x, y) \in \mathfrak{D}_n$ in a particular partition.

Algorithm 3: Conjugate Gradient: CG

Data: $X = [x^i],\ x^i \,\epsilon\, \mathbf{R}^n,\ \hat{\phi} : \mathbf{R}^n \longrightarrow \mathbf{R}^m,\ Y = [y^i], y^i \epsilon \mathbf{R},\ \gamma,\ \epsilon$

Result: $\begin{pmatrix} \hat{w} \\ \hat{b} \end{pmatrix} = \left(\hat{\Phi}_e^\intercal \hat{\Phi}_e + \frac{I_{m+1}}{\gamma}\right)^{-1} \hat{\Phi}_e^\intercal Y$

begin

  $(F, v) \longleftarrow FeatureMat(X, Y, \hat{\phi}, \gamma)$

  $A \longleftarrow F + \frac{1}{\gamma}\mathbf{I}_{m\times m}$

  while *not* $maxiterations$ *and* $\Delta(\hat{w}, \hat{b}) \geq \epsilon$ **do**

    $(\hat{w}_{i+1}, \hat{b}_{i+1}) \longleftarrow CGUpdate(\hat{w}_i, \hat{b}_i, A, v)$

    $\Delta(\hat{w}, \hat{b}) \longleftarrow \|\hat{w}_{i+1} - \hat{w}_i\|^2 + \left\|\hat{b}_{i+1} - \hat{b}_i\right\|^2$

Once matrices $\hat{\Phi}_e^\intercal \hat{\Phi}_e$ and $c = \hat{\Phi}_e^\intercal Y$ are computed from 2, they are used to carry out CG iterations to estimate model parameters $\hat{w},\ \hat{b}$ as shown in algorithm 3. In 3 the matrix received as input $\hat{\Phi}_e^\intercal \hat{\Phi}_e$ is first regularized by adding $\frac{I_{m+1}}{\gamma}$, leading to a linear system of the form $Ax = c$.

Due to the primal formulation of the FS-LSSVM in section 2.3, the size of matrix $A = \hat{\Phi}_e^\intercal \hat{\Phi}_e + \frac{I_{m+1}}{\gamma}$ which specifies the linear system in (2.5) is $(m + 1) \times (m + 1)$. Due to the fact that $m \ll n$, the CG iterations can be carried out locally once the matrices A and c are computed.

Model Evaluation

In the model evaluation process, there are two tasks which need to be solved with respect to calculating the cost of performance of a particular model instance.

v-fold Cross Validation

The fast v-fold cross-validation outlined by De Brabanter et.al. [24] is implemented in algorithm 4 to reduce the execution time for calculating the feature matrices.

Performance Metrics

In order to compare different model instances, performance metrics must be computed during each v-fold cross validation task for each fold of the data. These performance metrics are then averaged over all folds to yield an average performance rating for a model instance with specified values of hyper-parameters and regularization. In *FS-Scala* the classification

Algorithm 4: Distributed v-Fold Cross-Validation

Data: $X = [x^i],\ x^i\ \epsilon\ \mathbf{R}^n$, $\hat{\phi} : \mathbf{R}^n \longrightarrow \mathbf{R}^m$, $Y = [y^i], y^i \epsilon \mathbf{R}$, γ, folds
Result: Cross Validation Performance
begin
    $(A, v) \longleftarrow FeatureMat(X, Y, \hat{\phi}, \gamma)$
    $score \longleftarrow 0$
    for $i \longleftarrow 1\ to\ folds$ **do**
        $(X_i, Y_i) \longleftarrow$ fold i
        $(A_i, v_i) \longleftarrow FeatureMat(X_i, Y_i, \hat{\phi}, \gamma)$
        $(\hat{w}, \hat{b}) \longleftarrow CG(A - A_i + \frac{1}{\gamma}\mathbf{I}_{m\times m}, v - v_i)$
        $score \longleftarrow score + evaluateFold(\hat{w}, \hat{b}, X_i, Y_i)$
    return $score/folds$

accuracy, F1 score and ROC area are recorded during cross validation. Although only the value of ROC area is used to drive the global optimization procedure.

Algorithm 5 contains the implementation for calculating test set performance for each fold of test data. The true positive versus false positive curve is calculated on the test/validation fold provided and used to calculate the ROC area.

Algorithm 5: Evaluate performance for fold: $evaluateFold$

Data: $X_f = [x^i],\ x^i \in \mathbf{R}^n$, $\hat{\phi} : \mathbf{R}^n \longrightarrow \mathbf{R}^m$, $Y_f = [y^i], y^i \in \mathbf{R}$, $\hat{w}$, $\hat{b}$.
Result: score for given fold
begin
  $predictLabel(\hat{w}, \hat{b})(x, y)$:
  $emit(\hat{w} \cdot x + \hat{b}, y)$
begin
  $Vector.fill(length)(IndicatorFn)$:
  $vec \longleftarrow (0, ..., 0)_{length}\ map(IndicatorFn)$
  $return(vec)$
begin
  $MapScore(score, label)$:
  if *label = 1.0* **then**
    $Pos \longleftarrow Pos + 1$
    $tpv \longleftarrow Vector.fill(l)(IndicatorFn(sign(score - thresholds(i)) == 1.0))$
    $fpv \longleftarrow Vector.fill(l)(IndicatorFn(false))$
  else
    $Neg \longleftarrow Neg + 1$
    $tpv \longleftarrow Vector.fill(l)(IndicatorFn(false))$
    $fpv \longleftarrow Vector.fill(l)(IndicatorFn(sign(score - thresholds(i)) == 1.0))$
  $emit(tpv, fpv)$
begin
  $RedScore((u, v), (u', v'))$:
  $emit(u + u', v + v')$
begin
  $thresholds \longleftarrow List(t_1, t_2, \ldots t_l)$
  $Pos \longleftarrow 0$
  $Neg \longleftarrow 0$
  $scoresLabels \longleftarrow (X_f, Y_f)\ map\ predictLabel(\hat{w}, \hat{b})$
  $(tp, fp) \longleftarrow scoresLabels\ map(MapScore)\ reduce(RedScore)$
  $tp \longleftarrow tp/Pos$
  $fp \longleftarrow fp/Neg$
  $roc \longleftarrow thresholds\ zip(tp\ zip\ fp)$
  return $1 - area(roc)$

Chapter 5

Experiments

The following hardware mentioned in table 5.1 is employed to execute *FS-Scala* on a set of benchmark data sets. Machine number 1 is at the Department of Electrical Engineering, KU Leuven. The distributed FS-LSSVM implementation in *FS-Scala* inside a local *Apache Spark* application running concurrently on each core of the host machine.

TABLE 5.1: Hardware Specifications

Machine No.	Machine	Configuration	Data Sets
1	sista-nc-3	40 core 64GB RAM	*Adult* and *Forest Cover Type*
2	Laptop PC	4 core 8GB RAM	*Magic Gamma*

5.1 Data Sets

Experiments are carried out on the *Magic Gamma* Telescope, *Adult* and *Forest Cover Type* data sets available from the UCI Machine Learning Repository [30]. Below we give a short description of each.

- Magic Gamma: The data is generated by the registration of high speed gamma particles measured by a ground based atmospheric Cherenkov gamma telescope. Each entry consists of 10 numerical attributes and a binary class attribute.

TABLE 5.2: Magic Gamma Metadata

Name	Value
Training samples	18792
Test Samples	228
Features	10

- Adult: This is based on a a census study carried out in 1994, the data consists of 6 numerical attributes and 8 categorical attributes. The target attribute is binary class value, which indicates if the given individual has an annual income more than 50000$.

TABLE 5.3: Adult Metadata

Name	Value
Training samples	29310
Test Samples	3251
Features	13

- Forest Cover Type: This consists of cartographic data collected by the US Forest Service (USFS) on 30×30 metre cells which have can have seven different forest cover types. For the purposes of the experiments below, a binary classification problem is constructed which consists of recognizing cover type class two from the other cover types.

TABLE 5.4: Cover Type Metadata

Name	Value
Training samples	523076
Test Samples	57936
Features	53

5.2 Methodology

The performance of various kernel based FS-LSSVM models is carried out for various values of the experimental parameters listed in table 5.5. Performance metrics given in table 5.6 are computed for each cross validation and used to choose the best performing model during model tuning. To calculate the final performance estimate of each model given the number of prototypes, kernel and grid parameters, each experiment is repeated four times, the mean and standard deviation (in brackets), of the classification accuracy, area under ROC curve and execution time are recorded and presented in tables 5.7, 5.8 and 5.9 for the *Magic Gamma*, *Adult* and *Forest Cover Type* data sets respectively.

TABLE 5.5: Experiment Parameters

Name	Meaning	Values
Kernel	Kernel Type	Linear, RBF, Polynomial, Laplacian
Prototypes	No. of prototypes	50, 100, 200
Global Opt.	Global optimization	gs: Grid Search, csa: Coupled Simulated Annealing
Grid Size	Number of grid points	2,3,4
$nTrials$	Number of times each experiment is carried out	4
$CGIterations$	Number of local CG iterations	35
$nFolds$	Number of folds: fast CV 4	
$CSAIterations$	Number of CSA iterations when used as Global Opt.	5

TABLE 5.6: Metrics

Name	Meaning
Accuracy	Classification accuracy on test set averaged over
ROC area	avg. area under the ROC curve
Execution Time	avg. execution time of FS-LSSVM model tuning in seconds

5.3 Results

Magic Gamma

The performance of binary FS-LSSVM classifiers on the *MAGIC Gamma* Telescope Data Set obtained from the UCI Machine Learning Repository, are summarized in Table 5.7. FS-LSSVM models trained with polynomial kernels give better classification performance than the RBF and Linear counterparts, on the *MAGIC Gamma* data. Judging from the difference in the performance results between polynomial and linear LSSVM models, it can be said that the *Magic Gamma* data set has weak non linear behavior.

TABLE 5.7: Magic Gamma Test Results

Kernel	Prototypes	Global Optimization	Grid Size	Accuracy	ROC area	Execution Time
Linear	N.A.	gs	2	0.716(0.0)	0.412(0.0)	6.037(0.606)
Linear	N.A.	gs	3	0.716(0.0)	0.412(0.0)	7.750(0.809)
Linear	N.A.	gs	4	0.716(0.0)	0.412(0.0)	10.164(0.857)
RBF	50	gs	2	0.737(0.015)	0.572(0.124)	56.697(8.472)
RBF	50	gs	3	0.736(0.016)	0.516(0.125)	93.265(2.685)
RBF	50	gs	4	0.731(0.0210)	0.483(0.225)	153.472(3.270)
RBF	100	gs	2	0.721(0.007)	0.500(0.0718)	196.819(5.706)
RBF	100	gs	3	0.717(0.017)	0.472(0.188)	346.722(14.628)
RBF	100	gs	4	0.718(0.007)	0.557(0.096)	549.974(2.158)
RBF	200	gs	2	0.724(0.013)	0.598(0.127)	734.623(1.986)
RBF	200	gs	3	0.719(0.016)	0.535(0.063)	1339.845(1.602)
Polynomial	50	gs	2	0.717(0.013)	0.515(0.138)	53.954(4.470)
Polynomial	50	gs	3	0.712(0.004)	0.577(0.099)	100.611(1.870)
Polynomial	50	gs	4	0.716(0.005)	0.408(0.118)	183.198(2.923)
Polynomial	100	gs	2	0.716(0.008)	0.547(0.044)	163.274(1.009)
Polynomial	100	gs	3	0.711(0.005)	0.477(0.119)	320.734(0.521)
Polynomial	100	gs	4	0.720(0.008)	0.527(0.086)	602.008(1.628)
Laplacian	50	gs	2	0.727(0.029)	0.585(0.203)	63.497(2.506)
Laplacian	50	gs	3	0.715(0.0157)	0.405(0.258)	115.859(2.676)
Laplacian	50	gs	4	0.724(0.033)	0.574(0.122)	200.471(23.115)
Laplacian	100	gs	2	0.720(0.018)	0.584(0.137)	215.171(0.497)
Laplacian	100	gs	3	0.733(0.027)	0.697(0.111)	390.498(0.406)
Laplacian	100	gs	4	0.733(0.064)	0.459(0.219)	635.436(1.037)
Exponential	50	gs	2	0.719(0.023)	0.448(0.217)	44.403(6.738)
Exponential	50	gs	3	0.716(0.017)	0.522(0.041)	81.63(5.897)
Exponential	50	csa	2	0.726(0.018)	0.481(0.089)	71.798(3.346)
Exponential	50	csa	3	0.732(0.037)	0.607(0.128)	99.723(0.652)
Exponential	100	gs	2	0.73(0.026)	0.585(0.147)	124.612(0.567)

Adult

The performance of binary FS-LSSVM classifiers on the *Adult* Data Set, are summarized in Table 5.8. FS-LSSVM models trained with exponential kernels give approximately equivalent classification performance to the RBF and Linear counterparts, on the *Adult* data. Thus it can be asserted that the *Adult* data set exhibits strong linear behavior.

TABLE 5.8: Adult Data Set Test Results

Kernel	**Prototypes**	**Global Optimization**	**Grid Size**	**Accuracy**	**ROC area**	**Execution Time**
Linear	N.A.	gs	2	0.764(0.0)	0.689(0.0)	9.023(0.557)
Linear	N.A.	gs	3	0.764(0.0)	0.689(0.0)	9.933(0.592)
Linear	N.A.	gs	4	0.764(0.0)	0.689(0.0)	12.187(0.298)
RBF	50	gs	2	0.756(0.002)	0.596(0.023)	41.082(14.059)
RBF	50	gs	3	0.759(0.008)	0.617(0.184)	53.779(3.565)
RBF	50	gs	4	0.758(0.002)	0.618(0.094)	86.915(11.217)
RBF	50	csa	2	0.754(0.011)	0.429(0.115)	155.509(7.287)
RBF	50	csa	3	0.760(0.0193)	0.489(0.264)	219.662(3.118)
RBF	100	gs	2	0.757(0.001)	0.567(0.12)	73.78(9.813)
RBF	100	gs	3	0.757(0.003)	0.531(0.099)	106.391(4.403)
RBF	100	gs	4	0.755(0.005)	0.5(0.194)	150.163(13.694)
RBF	100	csa	2	0.755(0.004)	0.438(0.081)	432.430(7.632)
RBF	100	csa	3	0.763(0.004)	0.616(0.124)	638.854(18.368)
RBF	200	gs	2	0.755(0.003)	0.447(0.337)	214.831(12.035)
RBF	200	gs	3	0.758(0.004)	0.224(0.372)	321.246(22.744)
RBF	200	gs	4	0.756(0.0)	0.052(0.325)	479.994(14.111)
Polynomial	50	gs	2	0.756(0.001)	0.505(0.056)	36.064(3.478)
Polynomial	50	gs	3	0.753(0.004)	0.452(0.117)	54.271(3.08)
Polynomial	50	gs	4	0.747(0.013)	0.42(0.075)	98.455(10.061)
Polynomial	100	gs	2	0.756(0.0)	0.0(0.0)	82.131(8.588)
Polynomial	100	gs	3	0.756(0.0)	0.0(0.0)	119.084(6.842)
Polynomial	100	gs	4	0.757(0.004)	0.486(0.096)	203.252(13.439)
Polynomial	200	gs	2	0.756(0.0)	0.0(0.0)	242.182(12.325)
Polynomial	200	gs	3	0.756(0.0)	0.0(0.0)	381.291(16.416)
Polynomial	200	gs	4	0.756(0.0)	0.0(0.0)	641.61(27.328)
Laplacian	50	gs	2	0.757(0.001)	0.481(0.16)	39.318(2.667)
Laplacian	50	gs	3	0.761(0.013)	0.637(0.08)	60.076(5.784)
Laplacian	50	gs	4	0.761(0.005)	0.526(0.158)	103.05(12.915)
Laplacian	100	gs	2	0.757(0.001)	0.575(0.058)	99.229(11.334)
Laplacian	100	gs	3	0.755(0.001)	0.529(0.113)	145.03(12.354)
Laplacian	100	gs	4	0.762(0.005)	0.561(0.167)	221.745(13.37)
Laplacian	200	gs	2	0.752(0.002)	0.602(0.059)	317.955(10.67)
Laplacian	200	gs	3	0.754(0.002)	0.469(0.094)	481.726(18.635)
Laplacian	200	gs	4	0.756(0.0)	0.451(0.079)	746.019(28.615)

Forest Cover

Due to the computational overhead, only CSA based model tuning is employed in the experiments on the *Forest Cover Type* data, yet some clear trends can be observed in table 5.9. It is observed that the Laplacian kernel based models outperform the RBF as well as the linear LSSVMs. This is because the Laplacian kernel relying on the L_1 norm is more robust

to outliers and leads to a matrix (A) with a better condition number, leading CG iterations that converge in fewer iterations.

TABLE 5.9: Forest Cover Type Data Set Test Results

Kernel	**Prototypes**	**Global Optimization**	**Grid Size**	**Accuracy**	**ROC area**	**Execution Time**
Linear	N.A.	gs	2	0.624(0.0)	0.653(0.0)	945.084(20.839)
Linear	N.A.	gs	3	0.624(0.0)	0.653(0.0)	1284.482(27.841)
Linear	N.A.	gs	4	0.624(0.0)	0.653(0.0)	1507.708(108.976)
RBF	50	gs	2	0.514(0.009)	0.438(0.047)	2352.569(152.165)
RBF	50	gs	3	0.511(0.006)	0.467(0.016)	4215.124(92.704)
RBF	50	gs	4	0.513(0.004)	0.454(0.012)	7306.696(206.339)
RBF	100	gs	2	0.516(0.004)	0.446(0.012)	2736.229(51.411)
RBF	100	gs	3	0.513(0.001)	0.438(0.025)	5162.514(77.945)
RBF	100	gs	4	0.51(0.005)	0.435(0.024)	9707.263(914.778)
RBF	200	gs	2	0.519(0.006)	0.46(0.015)	8242.96(386.373)
Polynomial	50	gs	2	0.521(0.007)	0.447(0.032)	2272.617(95.663)
Polynomial	50	gs	3	0.516(0.004)	0.442(0.025)	4151.522(299.824)
Polynomial	50	gs	4	0.521(0.007)	0.458(0.036)	6243.599(553.171)
Polynomial	100	gs	2	0.52(0.003)	0.473(0.018)	2991.99(46.103)
Polynomial	100	gs	3	0.53(0.016)	0.484(0.02)	5271.503(312.424)
Polynomial	100	gs	4	0.521(0.005)	0.474(0.039)	10520.785(1200.765)
Laplacian	50	gs	2	0.682(0.028)	0.755(0.031)	2144.282(29.124)
Laplacian	50	gs	3	0.693(0.012)	0.747(0.02)	4029.492(43.372)
Laplacian	50	gs	4	0.692(0.021)	0.76(0.016)	6905.343(116.567)
Laplacian	100	gs	2	0.699(0.004)	0.768(0.016)	4535.487(525.434)
Laplacian	100	gs	3	0.694(0.021)	0.756(0.032)	8416.604(1125.397)
Laplacian	100	gs	4	0.703(0.014)	0.77(0.021)	13652.18(1507.549)

Chapter 6

Conclusions and Future Work

In this thesis, *FS-Scala* a Scala-based implementation of the FS-LSSVM is proposed as a programming framework for working with small and large scale FS-LSSVM models. The current SVM and LSSVM software do not leverage distributed *MapReduce* to implement kernel based SVM/LSSVM models but rather provide support for only linear models in the case of large data sets (*Apache Spark* [5]).

The proposed library *FS-Scala* takes advantage of the latest developments in distributed data processing as well as functional and object oriented programming to create a robust and performance oriented LSSVM research and development framework that can be used to train advanced LSSVM models on large and small data sets. Emphasis on modularity of design helps *FS-Scala* towards becoming an attractive option for cutting edge Machine Learning research in the area of kernel based models.

FS-Scala includes global optimization algorithms like *Grid Search* (GS) and *Coupled Simulated Annealing* (CSA) used to tune FS-LSSVM models. In order to make the tuning of non linear classification models possible on large data sets, a number of performance enhancements are proposed in section 3.4, which include but not limited to fast v-fold cross validation and feature matrix caching.

In chapter 5, *FS-Scala* is applied on a number of benchmark data sets (i.e. *Magic Gamma*, *Adult* and *Forest Cover Type*) and the performance and model tuning time are recorded. It is observed that our implementation enables scalable training, tuning and evaluation of LSSVM models, while still providing flexibility to tweak various underlying data processing infrastructure. We conclude by discussing ideas for future research work and further improvements to *FS-Scala*.

Future Work

Further research/improvements on *FS-Scala* can be carried out in a number of directions.

- Experiments of large scale commodity clusters: Simulations 5 were carried on a *Apache Spark* standalone cluster which runs tasks concurrently over the different cores of the parent machine. But it is also possible to scale this process even further for greater computational benefits by using large commodity clusters as provided by cloud computing providers.

- Simultaneous evaluation of multiple model instances: During a particular v-fold cross validation operation only a single model instance (configuration of hyper-parameters) is trained and evaluated, therefore to evaluate a $k \times k$ grid, k^2 v-fold cross validations must be carried out. Substantial improvements can be achieved if we can train and do v-fold CV on multiple model instances using only a single pass through the training data, leading to a new and exciting area of research in large scale FS-LSSVM model tuning.

- Distributed FS-LSSVM committee models: An exciting area of further research is application of distributed programming methodology to the implementation of FS-LSSVM committee models. *Apache Spark* inherently represents large data sets as sets distributed partitions, this can be used to train a FS-LSSVM committee model where each partition trains a local LSSVM model using only its local data. The models trained on each partition can now be used to train a weighted FS-LSSVM committee network.

- L_0 sparse extensions [34, 33] to the FS-LSSVM namely the SPFS-LSSVM and SSD-LSSVM may be considered for implementation in *FS-Scala*, moreover the Sub-sampled Dual Least Squares Support Vector Machine (SD-LSSVM) is an attractive candidate for inclusion in *FS-Scala* due to two reasons, firstly because it utilizes Quadratic Rényi Entropy driven subset selection (leading to re-use of existing implementation) and secondly because it is solved in the dual and does not require Automated Feature Extraction (AFE) using the Nyström method.

Appendices

Appendix A

Statistical Learning/Machine Learning reference

A.1 Bias-Variance Trade off

In order to choose better performing models for prediction or classification, one must construct an expression of their error given a data set from which they are built. Suppose that we are approximating a function $y_i = f(x_i) + \epsilon$ using data x_i drawn from a domain X given by a distribution $F(x)$, ϵ is assumed to be noise with zero mean and variance σ^2. An estimate $\hat{f}(x)$ is constructed via the learning process. The bias variance trade off refers to the implication of decomposing the error of predictive models into three components A.1.

- Bias: The error of the model with respect to the function that is being approximated.
- Variance: Sensitivity of the model with respect to the training data provided to it.
- Irreducible error σ^2

$$\mathrm{E}\Big[(y - \hat{f}(x))^2\Big] = \mathrm{Bias}[\hat{f}(x)]^2 + \mathrm{Var}[\hat{f}(x)] + \sigma^2 \tag{A.1}$$

$$\mathrm{Bias}[\hat{f}(x)] = \mathrm{E}[\hat{f}(x)] - f(x) \tag{A.2}$$

$$\mathrm{Var}[\hat{f}(x)] = \mathrm{E}\Big[(\hat{f}(x) - \mathrm{E}[\hat{f}(x)])^2\Big] \tag{A.3}$$

This leads to the observation that when we decrease the bias of an estimator $\hat{f}(x)$ (ex. Use a polynomial estimator instead of a linear one), we increase the sensitivity (or *variance*) of the estimator to unseen data. While it may predict with a high accuracy points in the training it has a tendency to *over fit* the training set and hence have poor out of sample accuracy. This is precisely the *bias-variance* trade off.

A.2 AFE: Nyström Method

Let $X_k \in \mathbb{R}^d$, $k = 1, \ldots, n$ be a random sample drawn from a distribution $F(x)$. Let $C \in \mathbb{R}^d$ be a compact set such that, $\mathcal{H} = \mathcal{L}^2(C)$ be a Hilbert space of functions given by the inner product A.4. Further let $M(\mathcal{H}, \mathcal{H})$ be a class of linear operators from $\mathcal{H}$ to $\mathcal{H}$.

$$< f, g >_{\mathcal{H}} = \int f(x)g(x)dF(x) \tag{A.4}$$

Automatic Feature Extraction (AFE) using the Nyström method [40] aims at finding a finite dimensional approximation to the kernel eigenfunction expansion of Mercer kernels [37], as shown below A.5. It is well known that Mercer kernels form a *RHKS* of functions. Every Mercer kernel defines a unique RHKS of functions as shown by the Moore-Aronszajn theorem [38]. For a more involved treatment of *RHKS* and their applications the reader may refer to the book written by Bertinet et.al [13].

$$K(x, t) = \sum_i \lambda_i \phi(x)\phi(t) \tag{A.5}$$

Mercer's theorem [37] states that the spectral decomposition of integral operator of K, $\mathcal{T} \in M(\mathcal{H}, \mathcal{H})$ defined as A.6 yields the eigenfunctions which span the RHKS generated by K and having an inner product defined as A.4.

$$(\mathcal{T}\phi_i)(t) = \int K(x, t)\phi(x)dF(x) \tag{A.6}$$

Equation A.6 above is more commonly also known as the Fredholm integral equation of the first kind. Nyström's method approximates this integral using the quadrature constructed by considering a finite kernel matrix constructed out of a prototype set X_k $k = 1, \ldots, m$ and calculating its spectral decomposition consisting of eigenvalues λ_k and eigen-vectors u_k. This yields an expression for the approximate non-linear feature map $\hat{\phi} : \mathbb{R}^d \longrightarrow \mathbb{R}^m$.

$$\hat{\phi}_i(t) = \frac{\sqrt{m}}{\lambda_i} \sum_{k=1}^{m} K(X_k, t)u_{k,i} \tag{A.7}$$

Appendix B

FS-Scala Class Hierarchies

Figures B.1 and B.2 depict the class hierarchy structures of the core and optimization modules respectively, we can discuss their role in depth.

B.1 Core Components

The core module consists of a set of classes which all originate from a base abstraction called *Model*. It also has classes *ParameterizedLearner* and *LinearModel* representing the core behaviors of parameter based Machine Learning models which form a large chunk of current learning techniques. By using generic typing inherent in the Scala language, one is able to separate the logic of a model from the details of the underlying data infrastructure. This is particularly relevant due to the explosion of data processing frameworks and systems like graph databases, Apache Spark, key value stores, column oriented databases, etc.

Kernel Functions and Kernel Based Models

The kernel module contains implementations of SVM kernels and AFE. The class *KernelizedModel* defines kernel based linear models which use the *GlobalOptimizer* class to tune values of hyper-parameters. The actual implementation of the *Automatic Feature Extraction* is abstracted out in the parent classes, thereby reducing the effort of writing new SVM kernels to merely expressing their evaluation functions.

B.2 Optimization Methods

Parametric models in FS-Scala have an embedded optimization object which is inherits from the *Optimizer* interface. Implementations of Conjugate Gradient and Gradient Descent are provided in the optimization module. New optimization algorithms can be added by inheriting from the top level *Optimizer* interface or the *RegularizedOptimizer* abstract class in case one is working with parametric models which involve regularization. Another important component of the optimization module is the *GlobalOptimizer* interface which acts as a skeleton for implementing gradient free global optimization algorithms.

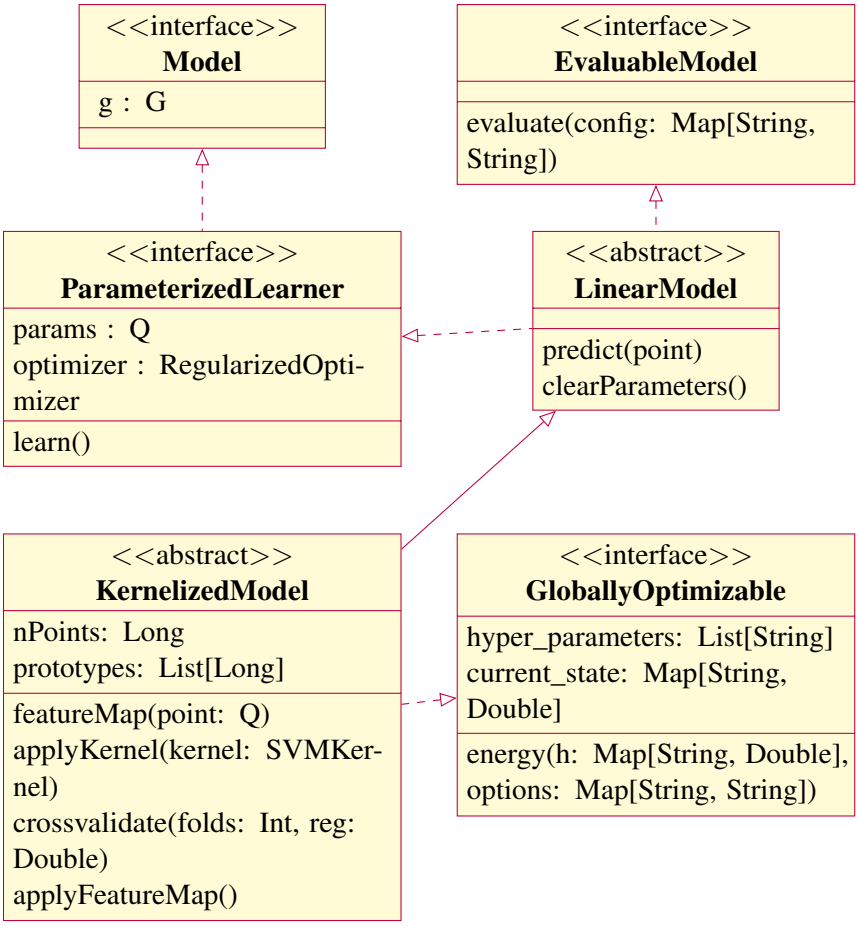

FIGURE B.1: Class Hierarchy of Core Models API

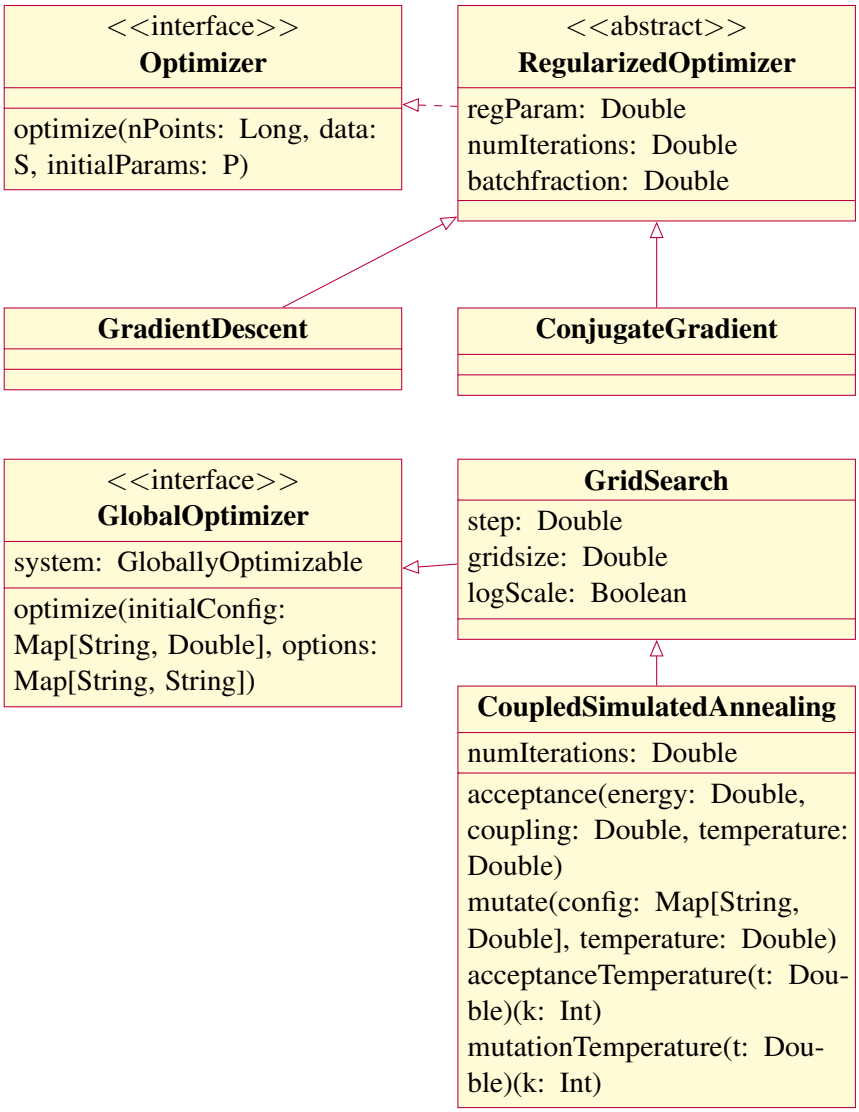

FIGURE B.2: Class Hierarchy of Optimization API

Bibliography

[1] Svm - support vector machines software, 2005 (accessed August 1, 2015).

[2] Apache hadoop: Lightning-fast cluster computing, 2005 (accessed July 6, 2015).

[3] Neo4j: The worlds leading graph database, 2007 (accessed July 6, 2015).

[4] Apache mahout: Scalable machine learning and data mining, 2008 (accessed August 12, 2015).

[5] Apache spark: Lightning-fast cluster computing, 2010 (accessed July 6, 2015).

[6] Orientdb, 2010 (accessed July 6, 2015).

[7] Deeplearning4j: Open-source, distributed deep learning for the jvm, 2014 (accessed August 12, 2015).

[8] Titan: Distributed graph database, 2014 (accessed July 6, 2015).

[9] Fs-scala: Apache spark implementation of fixed size least squares support vector machines, 2015 (accessed July 12, 2015).

[10] T. Abeel, Y. V. de Peer, and Y. Saeys. Java-ml: A machine learning library(machine learning open source software paper). *Journal of Machine Learning Research*, 10.

[11] P. Abeles. Boofcv. http://boofcv.org/, 2012.

[12] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition, 1996.

[13] C. T.-A. Alain Berlinet. *Reproducing Kernel Hilbert Spaces in Probability and Statistics*. Springer, 2004.

[14] G. Andrew and J. Gao. Scalable training of l1-regularized log-linear models. In *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, pages 33–40, New York, NY, USA, 2007. ACM.

[15] F. Aronsson. Large scale cluster analysis with hadoop and mahout, 2015. Student Paper.

[16] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer. Moa: Massive online analysis. *J. Mach. Learn. Res.*, 11:1601–1604, Aug. 2010.

[17] S. Bird, E. Klein, and E. Loper. *Natural Language Processing with Python*. O'Reilly Media, Inc., 1st edition, 2009.

[18] D. Borthakur, S. Rash, R. Schmidt, A. Aiyer, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, and A. Menon. Apache hadoop goes realtime at Facebook. *SIGMOD '11 - Proceedings of the 2011 international conference on Management of data*, page 1071, 2011.

[19] L. Bottou. Large-scale machine learning with stochastic gradient descent. In Y. Lechevallier and G. Saporta, editors, *Proceedings of the 19th International Conference on Computational Statistics (COMPSTAT'2010)*, pages 177–187, Paris, France, August 2010. Springer.

[20] C. G. BROYDEN. The convergence of a class of double-rank minimization algorithms 1. general considerations. *IMA Journal of Applied Mathematics*, 6(1):76–90, 1970.

[21] C.-C. Chang and C.-J. Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[22] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[23] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[24] K. De Brabanter, J. De Brabanter, J. A. K. Suykens, and B. De Moor. Optimized fixed-size kernel models for large data sets. *Computational Statistics and Data Analysis*, 54(6):1484–1504, June 2010.

[25] O. F. A. C. D. B. J. P. K. D. M. B. V. J. S. J. De Brabanter K., Karsmakers P. Ls-svmlab toolbox user's guide version 1.8.

[26] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[27] A. Halevy, P. Norvig, and F. Pereira. The unreasonable effectiveness of data. *IEEE Intelligent Systems*, 24(2):8–12, 2009.

[28] T. Joachims. Making large-scale svm learning practical. LS8-Report 24, Universität Dortmund, LS VIII-Report, 1998.

[29] T. Kumar and S. Pandey. Customization of recommendation system using collaborative filtering algorithm on cloud using mahout. In R. Buyya and S. M. Thampi, editors, *Intelligent Distributed Computing*, volume 321 of *Advances in Intelligent Systems and Computing*, pages 1–10. Springer International Publishing, 2015.

[30] M. Lichman. UCI machine learning repository, 2013.

[31] R. Mall, V. Jumutc, R. Langone, and J. A. K. Suykens. Representative subsets for big data learning using k-NN graphs. In *Proc. of IEEE BigData*, pages 37–42, 2014.

[32] R. Mall, R. Langone, and J. A. K. Suykens. FURS: Fast and Unique Representative Subset selection retaining large scale community structure. *Social Network Analysis and Mining*, 3(4):1075–1095, 2013.

[33] R. Mall and J. A. K. Suykens. Sparse Reductions for Fixed-Size Least Squares Support Vector Machines on Large Scale Data. In *Proc. of 17th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD 2013)*, pages 161–173, 2013.

[34] R. Mall and J. A. K. Suykens. Very Sparse LSSVM Reductions for Large-Scale Data. *IEEE Transactions on Neural Networks and Learning Systems*, 26(5):1086–1097, 2015.

[35] A. K. McCallum. Mallet: A machine learning for language toolkit. http://mallet.cs.umass.edu, 2002.

[36] X. Meng, J. K. Bradley, B. Yavuz, E. R. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. B. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. MLlib : Machine Learning in Apache Spark. *CoRR*, abs/1505.06807, 2015.

[37] J. Mercer. Functions of positive and negative type, and their connection with the theory of integral equations. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 209(441-458):415–446, 1909.

[38] N. Aronszjan. Theory of Reproducing Kernels.pdf. *Transactions of the American Mathematical Society*, 68(3):337–404, 1950.

[39] J. A. Nelder and R. Mead. A Simplex Method for Function Minimization. *The Computer Journal*, 7(4):308–313, Jan. 1965.

[40] E. Nyström. über die praktische auflösung von integralgleichungen mit anwendungen auf randwertaufgaben. *Acta Mathematica*, 54(1):185–204, 1930.

[41] M. Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[42] S. Owen, R. Anil, T. Dunning, and E. Friedman. *Mahout in Action*. Manning Publications Co., Greenwich, CT, USA, 2011.

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

[44] S. Rüping. mysvm-manual, university of dortmund, lehrstuhl informatik 8.

[45] R. Sethi. *Programming Languages: Concepts and Constructs*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

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

[47] J. Suykens, T. Van Gestel, J. De Brabanter, B. De Moor, and J. Vandewalle. *Least Squares Support Vector Machines*. World Scientific, 2002.

[48] J. A. K. Suykens and J. Vandewalle. Least Squares Support Vector Machine Classifiers. *Neural processing letters*, 9(3):293–300, 1999.

[49] G. Valentini, D. S. I. Dipartimento, and T. G. Dietterich. Bias-Variance Analysis of Support Vector Machines for the Development of SVM-Based Ensemble Methods. *Journal of Machine Learning Research*, 5:725–775, 2004.

[50] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[51] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[52] S. Xavier-De-Souza, J. A. K. Suykens, J. Vandewalle, and D. Bolle. Coupled simulated annealing. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 40(2):320–335, 2010.

[53] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark : Cluster Computing with Working Sets. *HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, page 10, 2010.

Master thesis filing card

Student: Mandar Chandorkar

Title: Fixed Size Least Squares Support Vector Machines: A Scala based programming framework for Large Scale Classification

Dutch title: Fixed Size Least Squares Support Vector Machines: Een Scala-gebaseerd programmeerkader voor classificatie op grote schaal.

UDC: 621.3

Abstract:

We propose *FS-Scala*, a flexible and modular *Scala* based implementation of the Fixed Size Least Squares Support Vector Machine (FS-LSSVM) for large data sets. The framework consists of a set of modules for (gradient and gradient free) optimization, model representation, kernel functions and evaluation of FS-LSSVM models. A kernel based *Fixed-Size Least Squares Support Vector Machine* (FS-LSSVM) model is implemented in the proposed framework, while heavily leveraging the parallel computing capabilities of *Apache Spark*. Global optimization routines like *Coupled Simulated Annealing* (CSA) and *Grid Search* are implemented and used to tune the hyper-parameters of the FS-LSSVM model. Finally, we carry out experiments on benchmark data sets like *Magic Gamma* and *Adult* and evaluate the performance of various kernel based FS-LSSVM models.

Thesis submitted for the degree of Master of Science in Artificial Intelligence, option Engineering and Computer Science

Thesis supervisors: Prof. dr. ir. Bart De Moor
Prof. dr. ir. Johan A.K Suykens

Assessor: Dr. Raghvendra Mall

Mentors: Oliver Lauwers
Dr. Raghvendra Mall