Evaluation of a Novel Parallelization Method in The Spark Framework

Lab Report: Lab Development and Application of Data Mining and Learning Systems 2016

Rufat Babayev

Abstract. In this paper, we evaluate predictive performance and runtime of a proposed parallelization method that can be applied to various machine learning algorithms. In comparable runtime, the hypothesis found by the parallelization method has the same or better quality as compared to state-of-the-art parallelization algorithms in the Spark framework. The method can be applied to massive datasets to obtain weak hypotheses from its samples and then can be aggregated to stronger one using convex optimization. Our empirical evaluation results prove the strength and optimality of this method on various classification datasets.

1 Introduction

We study the performance of the parallelization method for different machine learning techniques. Our goal is to analyze the proposed parallelization strategy in terms of predictive performance and runtime. For this purpose, a base serial learning algorithm is utilized inside the parallelization method. In general, parallel algorithm cannot use the same heuristics as a serial learning algorithm. In this case, the aim is to keep the quality of the output the same or to obtain better quality output as compared to a base learner. The base learning algorithm is considered as a black box through parallelization. We apply the method to small samples of given data to retrieve weak hypotheses, then through iterative process we obtain strong hypothesis from weak ones. Our evaluation results compare the parallelization method of base learning algorithm to its parallelized variants in Spark framework [Sparks et al., 2013, Zaharia et al., 2012].

Algorithms in machine learning aim at finding hypothesis by using joint probability distribution between data instances and their labels. The quality of hypothesis is then defined by the value of loss function between original and predicted labels. Optimal hypothesis is in general unreachable because joint probability distribution is unknown by default. Therefore, machine learning algorithms tries to optimize this hypothesis on randomly drawn samples of data instances and their labels. This optimization can be

achieved using optimization methods such as Stochastic Gradient Descent and quasi-Newton methods. Both of these methods are used in the evaluation.

For parallelization of such machine learning algorithms the base learning algorithm and merely its implementation are regarded as a black box learner. In this respect, we find weak hypotheses on small samples of data and aggregate those hypotheses to obtain stronger one. In the parallelization strategy we continue the aggregation process until single a strong hypothesis is obtained. We give the pseudo-code of the parallelization strategy in Section 4 by using the Radon point as an aggregation operator and show what is changed as compared to original [Kamp et al., 2015].

For one natural comparison, we compare the output of the parallelization strategy on a base learning algorithm to the parallelized variant in the Spark machine learning library. To do this, we compare the runtimes and accuracies using cross validation of both parallelized algorithms. We use parameter optimization to obtain optimal parameters on a separate part of the chosen dataset.

It is natural to compare our parallelization strategy on base learner to its parallelized variant in Spark on the same input. In this regard, we report the empirical study in Section 5 with various base learners, their parallelized variants in Spark and benchmark datasets. Our results show that a speed-up can be achieved as compared to Spark's parallelized algorithms. Compared to parallel machine learning algorithms in Spark, our general parallelization strategy achieves the same or better hypothesis in comparable runtime.

Most of parallel machine learning algorithms are based on stochastic gradient descent by optimizing expected loss. From moderate to big dataset sizes, it is widely recognized that stochastic gradient descent behaves poorly compared to, e.g., quasi-Newton methods such as L-BFGS [Nocedal, 1980] or support vector machines. It can be debated that the existence of stochastic gradient descent in this context is its ease of parallelization. We review the state of the art in parallel machine learning in the next section.

2 Related Work

In the last decade, data volumes have grown faster than computational power of processors. For big datasets the major concern is the time of computation, not the size of the data itself. For example, randomized first-order methods can show significant speed-up. However, they obtain low or medium accuracy solutions by using only first-order information from the objective, such as gradient estimates [Nesterov and Nemirovski, 2013]. The natural way to quickly process large datasets is a usage of parallelization. To process the data in such a distributed fashion, efficient algorithms are required. Key aspect of those parallelized algorithms is their performance. The performance metrics for parallel machine learning algorithm is basically based on (i) difference on predictive performance between obtained parallel hypothesis and obtained serial hypothesis, (ii) communication time between workers/processors to find parallel hypothesis, (iii) speed-up of the parallel version of the algorithm when running on some number of workers/processors. Heuristically, a parallel system uses more sophisticated data processing and hypothesis/solution

finding techniques than a serial system. Either a single solution is maintained throughout the optimization process or multiple solutions are generated and used for a joint solution. In this regard, hypothesis generation in parallel system is performed by using following communication routines between workers: (1) the meta-data is exchanged between workers during the optimization phase and at the end in aggregation phase, (2) hypothesis are generated in parallel without the exchange of meta-data during the optimization phase and exchange of meta-data at the end in aggregation phase.

Both types of meta-data exchange are exemplary in gradient descent (GD) and stochastic gradient descent (SGD). There are some algorithms proposed for parallelization of gradient descent. For example, Distributed subgradient [Chu et al., 2007, Teo et al., 2009], Distributed convex solver [Mcdonald et al., 2009] and Multicore stochastic gradient [Zinkevich et al., 2009]. Distributed subgradient is based on distributed computing of gradients in each worker that keeps partition of data and aggregation of those gradients for a global update step. On the other hand, Distributed convex solver tries different strategy by solving sub-problems in each worker/processor in parallel and then averaging the results to retrieve a final joint result. For the former, quality of the hypothesis is optimal because of meta-data exchange throughout the whole process. However, it requires aggregation of all gradients for a global update step which is prohibitive for communication. For the latter, communication costs are low, but convergence rate is not optimal. The convergence rate of parallel SGD can be improved by the method named stochastic average gradient SAG [Polyak and Juditsky, 1992]. In this case, rather than finding the average of hypotheses at the end, an iterative average of hypotheses are maintained in each iteration. In other words, for each instance, hypotheses are averaged which is computationally prohibitive. [Dekel et al., 2012] proposed mini-batch SGD which relaxes hypotheses averaging condition to mini-batch of data rather than single data instance. Despite of a runtime improvement, this method has slower convergence rates.

Another set of methods that can be parallelized are ensemble methods. Ensemble methods are learning algorithms that builds a set of hypotheses and then predict an output for unseen instances by taking a weighted or a majority vote of predictions of this set of hypotheses. Hypotheses are commonly generated from different data distributions. For example, an ensemble method bagging can be parallelized by computing hypotheses on samples from common big dataset [Breiman, 1996]. Parallelized bagging can be enhanced with distributed boosting. Distributed boosting is a technique for obtaining highly accurate hypothesis ensembles, where the hypotheses are generated in a distributed fashion from disjoint data samples, with the weights on the data instances adaptively set according to the performance of previous hypotheses [Lazarevic and Obradovic, 2002].

For the implementation of parallelized algorithms, special parallel machine learning frameworks are available such as Hadoop MapReduce [Dean and Ghemawat, 2008, Shvachko et al., 2010] and Spark [Sparks et al., 2013, Zaharia et al., 2012]. We used Spark framework and algorithms from its machine learning library to compare with our parallelization method. Chosen algorithms are SGD and LBFGS-based and performing periodic meta-data exchange and aggregation-at-end optimization strategy. We will talk more about these later in next sections.

The parallelization strategy *Parallel Radon Machine* is similar to aggregation at-the-end optimization strategies. With a help of Radon point as an aggregation operator it can be applied to various expected loss minimizations. However, it differs from parallel SGD where parallel SGD uses SGD update rule and meta-data exchange within the optimization process. Our strategy aggregates weak hypotheses to a single formed stronger one; it differs from boosting that previous weak hypotheses are not updated based on their predictive performance. Basically, found weak hypotheses are independent of each other and after generation, they are only used for aggregation. It is different from bagging in a sense that ensemble of weak hypotheses are not preserved for predictions such as the weighted or majority vote, they are only used for aggregation to single stronger hypothesis.

3 Preliminaries

In geometry, Radon's Theorem states that for any set $\mathbb{P}$ of $d+2$ points in d-dimensional space $\mathbb{R}^d$, one can partition $\mathbb{P}$ into two disjoint sets $\mathbb{P}_1$, $\mathbb{P}_2$, such that their convex hulls intersect [Radon, 1921]. Two disjoint sets $\mathbb{P}_1$ and $\mathbb{P}_2$ are called Radon partitions of a point set $\mathbb{P}$. A point in the intersection of convex hulls is called a Radon point of this point set. For instance, in Euclidean space where $d = 2$, any set of four points can be partitioned in one of three ways. In a first case, it can be a triangle of points and a single point, where the convex hull of that triangle contains that single point. The Radon point here is the single point itself which denotes the intersection of triangle of points and single point. In a second case, it can be two pairs of points which are the endpoints of two intersecting line segments. It is easily seen that the intersection is a Radon point. Finally, in a third case, the points can reside in one line, where line segment formed by one pair of points contains other line segment formed by second pair of points. Here any point from containing line segment can be chosen as a Radon point.

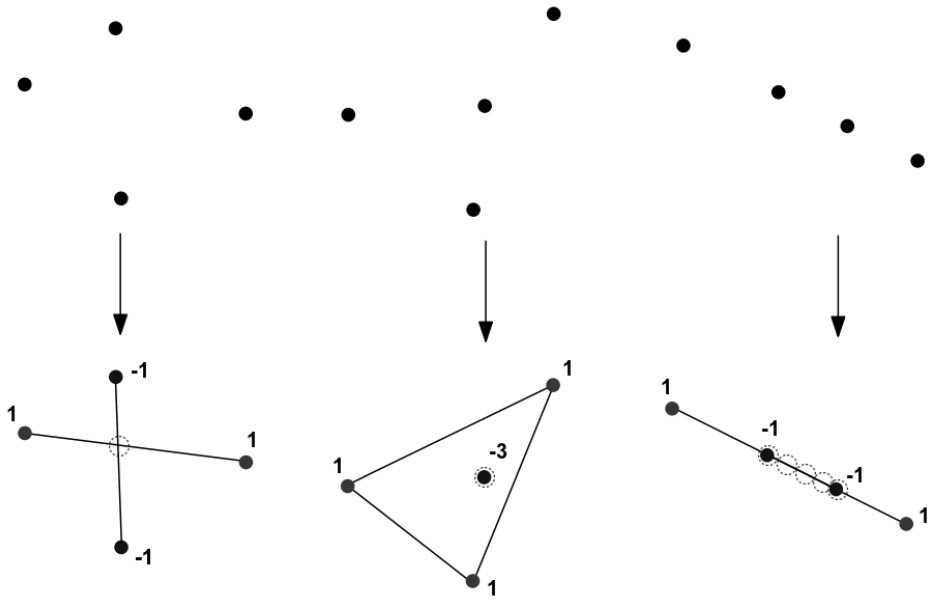


Figure 1: Three sets of four points in the plane, the multipliers solving the system of four linear equations for these points, the Radon partitions formed by separating the points with positive multipliers from the points with negative multipliers and dashed circles for denoting Radon points.

For a general case, consider any set $\mathbb{P}$ of points where $\mathbb{P} = \{z_1, z_2, \ldots, z_{d+2}\} \subset \mathbb{R}^d$ of $d+2$ points in d-dimensional space. Then there exists a set of multipliers $\alpha = \{\alpha_1, \alpha_2, \ldots, \alpha_{d+2}\}$, not all of which are equal to zero, which solves the system of linear equations

$$\sum_{i=1}^{d+2} \alpha_i \cdot z_i = 0, \quad \sum_{i=1}^{d+2} \alpha_i = 0,$$

where there are $d+2$ unknowns (multipliers). The set of multipliers is found by the original points and used later for calculating Radon point. Assume there is a nonzero solution $\alpha = \{\alpha_1, \alpha_2, \ldots, \alpha_{d+2}\}$. Then all *positive* entries in α form the weight set α^+. All *negative* ones form α^- . And I^+ represents the indices of multipliers in α which are positive; I^- represents the indices of multipliers in α which are negative. If we choose the points from $\mathbb{P}$ whose indices are present in I^+ we will obtain Radon partition $\mathbb{P}_1$ and if we choose the points from $\mathbb{P}$ whose indices are present in I^- we will obtain Radon partition $\mathbb{P}_2$ where convex hulls of $\mathbb{P}_1$ and $\mathbb{P}_2$ intersects. Then Radon Point p :

$$p = \sum_{i \in I^+} \frac{\alpha_i}{sum} z_i = \sum_{j \in I^-} \frac{-\alpha_j}{sum} z_j,$$

where

$$sum = \sum_{i=0}^{|\alpha^+|} \alpha_i = -\sum_{j=0}^{|\alpha^-|} \alpha_j.$$

The left hand side of the equation of p expresses this point as a convex composition of the points in $\mathbb{P}_1$, and the right hand side expresses it as a a convex composition of the points in $\mathbb{P}_2$. Therefore, p belongs to convex hulls of both $\mathbb{P}_1$ and $\mathbb{P}_2$. With this way, one can construct Radon point efficiently by solving the system of linear equations for the multipliers using Gaussian elimination or other efficient algorithms [Clarkson et al., 1996].

4 Radon Parallelization

Classification or regression on a dataset intends to find an unknown dependence between data instances and their labels. The data instances are elements of an *input space* $\mathbb{X}$, their labels are elements of *output space* $\mathbb{Y}$, and the model or hypothesis is chosen from *hypothesis space* $\mathbb{F}$ of hypotheses of the form $f : \mathbb{X} \to \mathbb{Y}$. Our parallelization method is suitable for both classification and regression. The method looks for hypotheses of the form $f(Z) = W^T Z$ which is equivalent to

$$f(z_1, z_2, z_3, \ldots, z_n) = w_0 + w_1 z_1 + w_2 z_2 + w_3 z_3 + \cdots + w_n z_n$$

where $Z = \{z_1, z_2, z_3, \ldots, z_n\}$ represents values for n number of attributes of data instance Z. And $W = \{w_0, w_1, w_2, w_3, \ldots, w_n\}$ represents weight set for the hypothesis. Here, $w_1, w_2, w_3, \ldots, w_n$ are normal weights and w_0 expresses an intercept or a bias for a hypothesis f.

Algorithm 1 Parallel Radon Machine

Input: algorithm $\mathbb{L}$, dataset $D \subseteq \mathbb{X} \times \mathbb{Y}$, Radon number $r \in \mathbb{N}$,
and number of iterations $h \in \mathbb{N}$

Output: hypothesis $f \in \mathbb{F}$

1: **divide** D into r^h subsets D_i
2: **run** $\mathbb{L}$ in parallel to obtain $f = \mathbb{L}(D_i)$
3: $S \leftarrow \{f_1, f_2 \ldots . f_{r^h}\}$
4: **for** $i = 1, \ldots, h$ **do**
5: **partition** S into substes $S_1, S_2, \ldots, S_{|S|/r}$ of size r
6: **calculate** $\tau(S_1), \tau(S_2), \ldots, \tau(S_{|S|/r})$ in parallel
7: $S \leftarrow \{\tau(S_1), \tau(S_2), \ldots, \tau(S_{|S|/r})\}$
8: **end for**
9: **return** one $f \in S$

The main point of the parallelization method is to divide the whole data to almost equally sized subsets and run the base learner on each of these subsets in parallel to

obtain weak hypotheses and then iteratively aggregate them to single robust one. For the aggregation of hypotheses we use *Gaussian* elimination method to find set of multipliers from weight vectors of weak hypotheses then use these multipliers to find the Radon point of them. As a base learner $\mathbb{L}$ we use *Logistic* (logistic regression) and *SMO* (John Platt's sequential minimal optimization algorithm for training a support vector classifier) from the WEKA machine learning workbench [Hall et al., 2009] which we discuss later on in this section. The number of iterations h has to be chosen in a way that $h \geq 1$ and $r^h \leq |D|$. Therefore, the algorithm divides whole dataset to r^h subsets $D_1, \ldots, D_{r^h}$ (line 1) of almost equal size and runs our chosen base learner in parallel to obtain weak hypotheses $f_1, \ldots, f_{r^h}$. These hypotheses are put together in a set S for iterative aggregation to form single robust one at the end (line 4-8). For iterative aggregation, set S is again divided into subsets of exact size r (line 5) and calculation of a Radon point is parallelized for each subset (line 6). The τ operator denotes the Radon point calculation. On line 7, formed hypotheses after the Radon point calculation is assigned to set S where previous hypotheses are not retained. This process continues until we end up with single strong hypothesis as compared to original algorithm on [Kamp et al., 2015] where S can contain multiple hypotheses at the end and one is returned randomly.

As described above, the parallelization method finds a hypothesis with linear weight set. In this regard, we limit our parallelization method to linear hypothesis spaces with common notion of convexity. According to [Kay and Womble, 1971], the calculation of a Radon point in d-dimensional space from set of $d+2$ points can be regarded as a convex optimization problem. However, the Radon number r is defined for the hypothesis space. For example, the weight set W which is defined above has cardinality of $n+1$ where n is the number of attributes of one data instance. It is $n+1$ because, we additionally consider intercept w_0 for hypothesis generation. In this case, the dimension d of the weight vector becomes $d = n+1$ where the Radon number is $r = d+2$ which is in turn $n+1+2$. So, the hypotheses generated by a base learner $\mathbb{L}$ can be represented as a element of the Euclidean space $\mathbb{R}^d$ such that convexity of the problem is maintained. Keeping that in mind, one can calculate Radon point of hypotheses by solving system of linear equations of size $r \times r$.

In the following, we give the details of the algorithms used to build and test our parallelization method. For base learners, we use **weka.classifiers.functions.Logistic**, **weka.classifiers.functions.SMO** from WEKA machine learning workbench. The former is a class for building and using a multinomial logistic regression model with a ridge estimator [Le Cessie and Van Houwelingen, 1992]. However, there are some changes in the implementation as compared to the original paper. If n data instances with m attributes have k class labels, the parameter matrix B will have dimensions $m \times (k-1)$. The (negative) multinomial log-likelihood L is minimized to build the matrix B during the optimization process. For this, a quasi-Newton method is utilized to search for the optimal values of the $m \times (k-1)$ variables [Xu]. The original API does not provide a method to build a classifier from initial weights. This is important for testing purposes in cross validation. We modified the API to make it possible. The latter implements John Platt's sequential minimal optimization algorithm for training a support vector classifier [Platt, 1999, Keerthi et al., 2001, Hastie et al., 1998]. The implementation replaces all

missing values in data instances and converts nominal attributes into binary attributes. All attributes are normalized by default. In that case obtained coefficients/weights are based on the normalized data instances, not the original dataset [Frank et al.]. In our parallelization method we disabled normalization and standardization to make it comparable to the Spark implementation. In the algorithm, multi-class problems are solved using pairwise classification. However, in our implementation we restrict it to binary classification problems. Again, the API is modified for building a classifier from initial weights which is just working for binary classification problems.

To compare predictive performance and runtime of our parallelization method we use Spark learners **LogisticRegressionWithSGD**, **LogisticRegressionWithLBFGS** and **SVMWithSGD** from **org.apache.spark.mllib.classification** package. The first and second one is compared to parallelized version of **weka.classifiers.functions.Logistic** and third one to **weka.classifiers.functions.SMO**. At the time of writing, Spark Mllib uses two optimization methods, *SGD* and *L-BFGS*. These methods can exhibit different convergence rates depending on the properties of the objective function. For the common standard and simplicity of the API, we prefer *SGD* over *L-BFGS* in early parameter selection. However, we also use *L-BFGS* version of logistic regression with default parameters for comparison. The API of *LogisticRegressionWithLBFGS* does not provide a way to obtain optimal parameters for learning except initialization of initial weights. The first class *LogisticRegressionWithSGD* trains a classification model for binary logistic regression using SGD. The second class *SVMWithSGD* trains an SVM using SGD and third class *LogisticRegressionWithLBFGS* trains a classification model for multinomial or binary logistic regression using Limited-memory BFGS. Standard feature scaling is applied in L-BFGS implementation by default. *L2* regularization is used for all three algorithms during learning.

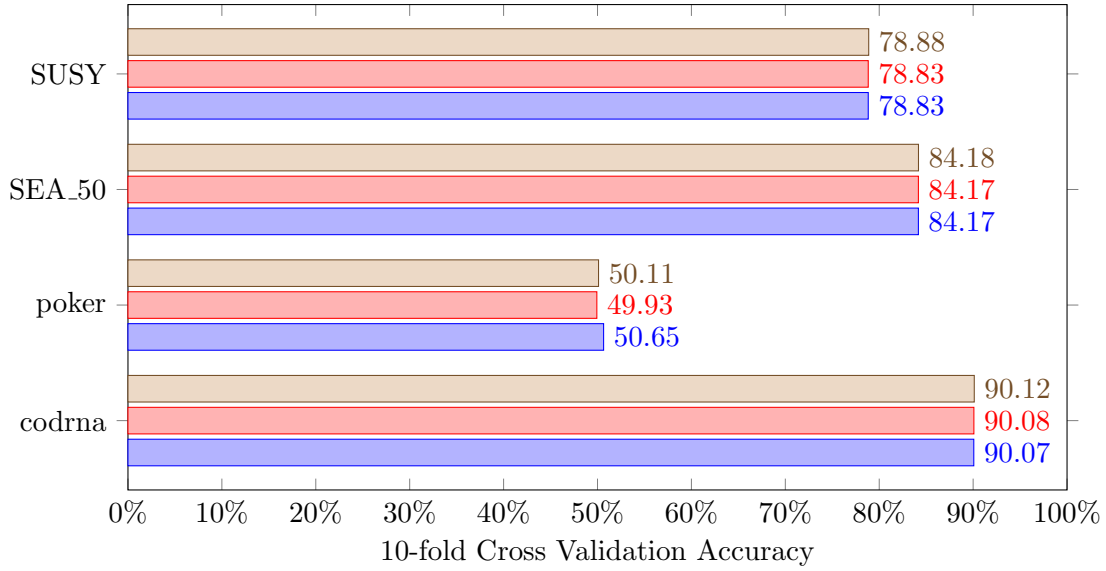
Why do not we use all three Spark Mllib algorithms inside Parallel Radon Machine? Spark holds input data as *RDD* (resilient distributed datasets). It is a main data structure of the Spark framework. RDDs are logically partitioned, such that each partition holds a subset of the data. Partitions are assigned to workers/executors in a cluster mode or to worker threads in a local mode. For fast in-memory processing each partition/split will be in RAM. In our parallelization method we map each partition to one model vector obtained by running base learning algorithm inside map function. Then, later on we aggregate those vectors together. *MapReduce* is a functional programming paradigm, such that functions are objects; each map and reduce function is serialized and then deserialized at workers to achieve parallel processing. Convention of serialization states that every object/reference inside a serializable object has to be serializable. Spark Mllib algorithms are working on RDDs and to create an RDD, single *SparkContext* object is required. In our case, we need to create an RDD inside the map function to use Spark's algorithms. However, it is not possible, because the SparkContext object is not serializable. Then the alternative is to use algorithms from WEKA machine learning workbench inside map functions and we try to use comparable algorithms there.

5 Empirical Evaluation

In our empirical evaluation we compare the *Parallel Radon machine* to an execution of Spark learner with the same amount of data. All experiments are performed on the same computer with **Intel(R) Core(TM) i7-3610QM** processor where we cannot test with big Radon number r and large number of iterations h. For the distributed processing on worker threads, environment provides 8GB of RAM with 8 CPUs at the frequency of 2.3 GHz.

What is the performance of our parallelization method in practice? We performed experiments on 4 classification datasets where we compared the predictive performance and runtime of the parallelization method on chosen base learner with a chosen Spark learner. The parellelized version of **weka.classifiers.functions.Logistic** is compared with **LogisticRegressionWithSGD** and **LogisticRegressionWithLBFGS**; the parellelized **weka.classifiers.functions.SMO** is compared with **SVMWithSGD**. For both comparisons the performance of hypothesis is evaluated by using *10*-fold cross validation and by producing average test accuracy over all folds. By utilizing *10*-fold cross validation we obtain the statistical stability that single train-test split does not provide. The datasets are taken from openML [Vanschoren et al., 2014] and UCI Machine Learning Repository [Lichman, 2013] namely *codrna*, *poker*, *SEA_50* and *SUSY*. They all are binary classifications datasets with real valued attributes where *codrna* has 488565 data instances, one data instance has 8 attributes, *poker* has 1025010 instances with 10 attributes, *SEA_50* has 10^6 instances with 3 attributes and *SUSY* has 5×10^6 instances with 18 attributes. Radon number r is detected automatically by number of attributes and h is restricted to dataset size where r^h cannot exceed it, since the dataset is divided to r^h subsets of size n and we request $n > 1$. In Figure 2, we show predictive performance of *Parallel Radon Machine* for $h = 1, 2, 3$ on chosen datasets. For each value of h and for each dataset with the corresponding Radon number r we obtain polynomial r^h number of splits. Figure 2(a) depicts average cross validation accuracy of our parallelization method with base learner *Logistic*. Predictive performance of *Logistic* is decent on *codrna*, *SEA_50* and *SUSY* datasets. *Poker* is found to be a challenging dataset for classification algorithms. For this dataset, obtaining 50% accuracy with linear hypothesis spaces is reasonable, because for better accuracy artifical neural network is utilized [Ivanic]. However, for *codrna* dataset it falls behind the base learner *SMO* in Figure 2(b). For all comparable Spark algorithms the same number of worker threads is used. For instance, if *Parallel Radon Machine* with the base learner *Logistic* uses 11^3 threads for *codrna* dataset with $h = 3$, then *LogisticRegressionWithSGD* and *LogisticRegressionWithLBFGS* do the same.

(a) Base learner: *Logistic*



(b) Base learner: *SMO*

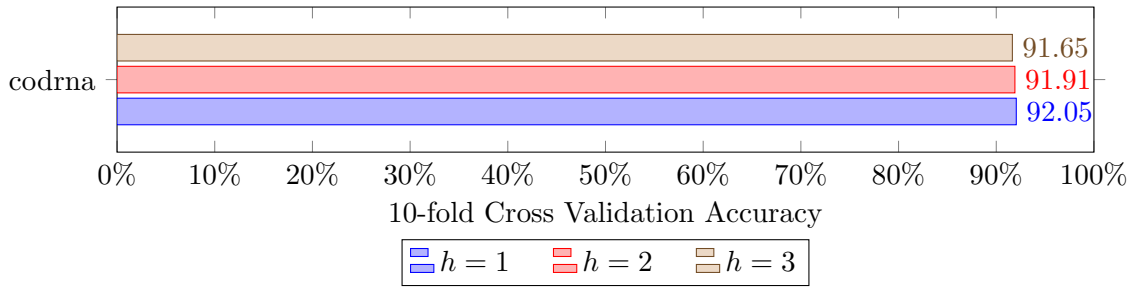
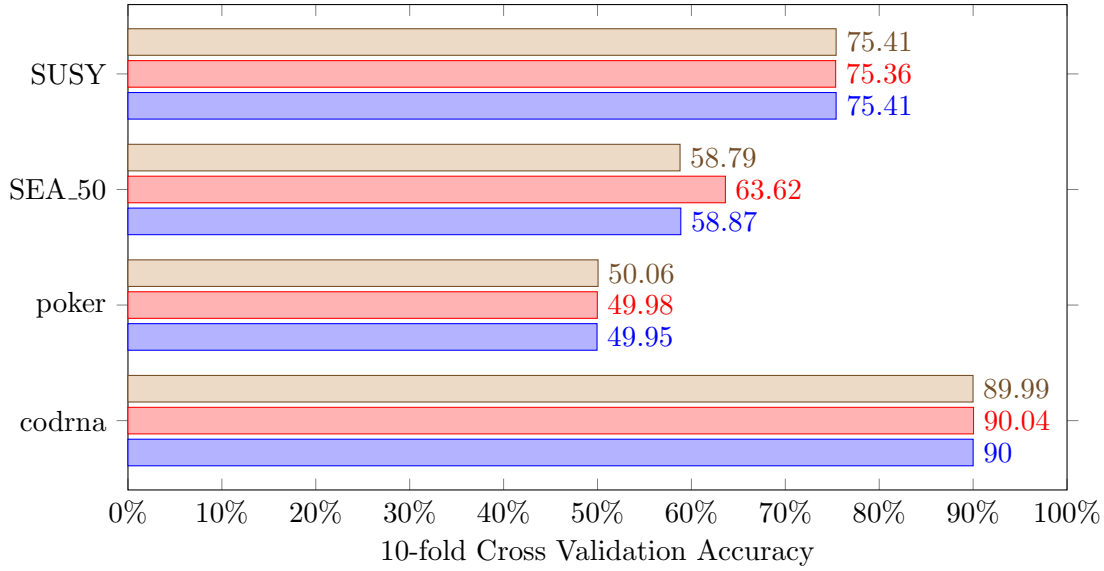


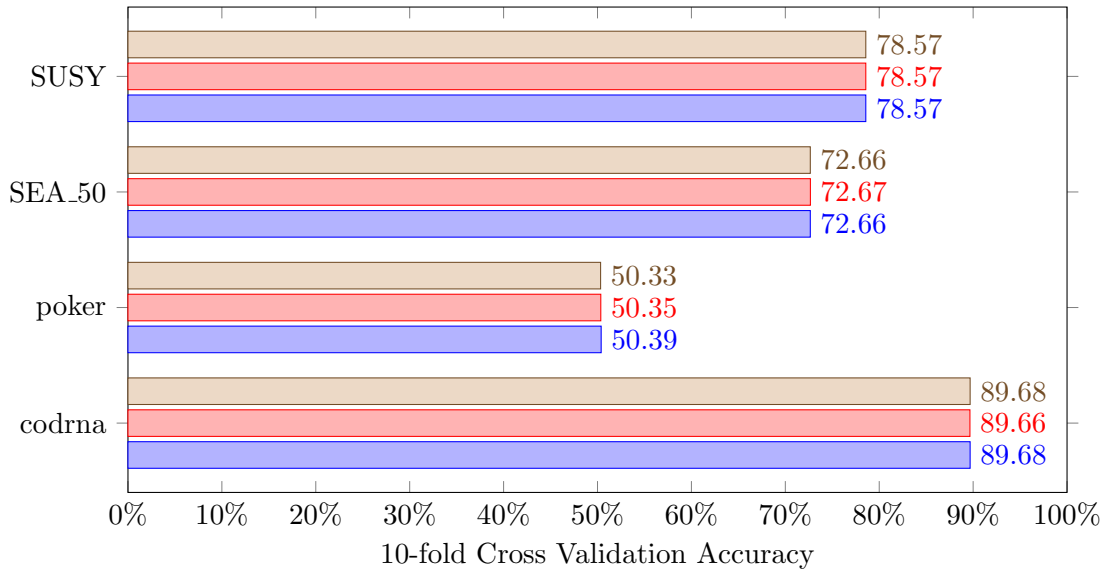
Figure 2: Predictive performance of *Parallel Radon Machine* for $h = 1, 2, 3$ on *codrna*, *poker*, *SEA_50* and *SUSY* datasets. Number of threads r^h based on Radon number r of a dataset and h iteration parameter: {codrna, 11^h}, {poker, 13^h}, {SEA_50, 6^h}, {SUSY, 21^h}. In (b) base learner *SMO* only tested on the *codrna* dataset.

The results, shown in Figure 3, indicate predictive performance of Spark learners on the same datasets for comparison. On some datasets *Parallel Radon Machine* even achieved higher accuracy than the comparable Spark learner. For instance, in Figure 3(a), *LogisticRegressionWithSGD* is outperformed by *Parallel Radon Machine* with the base learner *Logistic* in the SEA_50 dataset. It also slightly falls behind in SUSY dataset. This also applies to LBFGS version in the SEA_50 dataset. In Figure 3(b), it achieves almost the same accuracy for the other datasets without performing parameter selection.

(a) Spark learner: *LogisticRegressionWithSGD*



(b) Spark learner: *LogisticRegressionWithLBFGS*



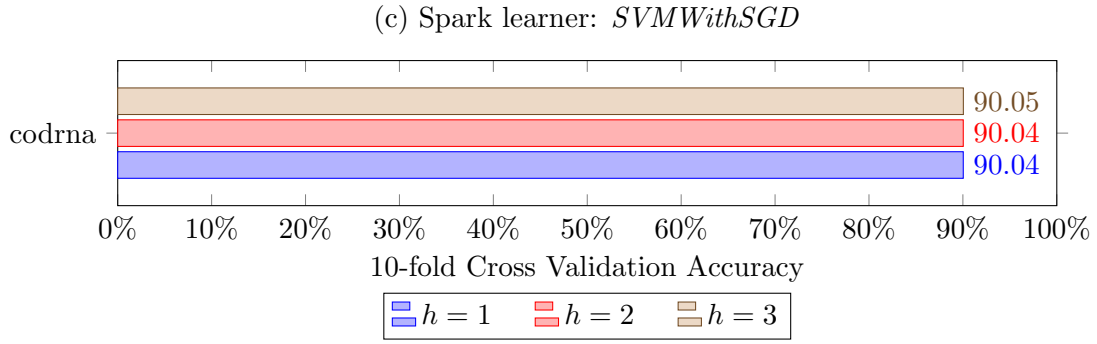


Figure 3: Predictive performance of Spark learners on the same datasets for comparison. Number of threads r^h based on Radon number r of a dataset and h iteration parameter: {codrna, 11^h}, {poker, 13^h}, {SEA_50, 6^h}, {SUSY, 21^h}. In (c) Spark learner *SVMWithSGD* only tested on the *codrna* dataset. In (b) for Spark learner *LogisticRegressionWithLBFGS*, parameter selection is not performed.

Figure 3(c) shows the predictive performance of Spark counterpart of the parallelized version of *SMO* algorithm. On *codrna* dataset, it is worse up to 1.5% - 2%.

How does the runtime depend on iteration parameter *h*? Figure 4 shows runtime of Parallel Radon Machine on different datasets for different choices of h. Total runtime is calculated by averaging runtimes of all folds in *10*-fold cross validation. With the bigger value of h, runtime of parallelization method decreases. Because with bigger h, number of splits of dataset increases through split parameter r^h. However, we choose h in a way that number of threads r^h does not become polynomially big. For example, for codrna dataset whose Radon number $r = 8+1+2 = 11$ and for number of iterations $h = 3$, number of data partitions will be 1331 and after performing weak hypotheses generation we will have 1331 partitions each having just *one* hypothesis. Then the following flow occurs through iterative Radon point calculation ($\sim$ denotes repartitioning, and $\Rightarrow$ denotes Radon point calculation) until we end up with one robust hypothesis:

$$1331 : 1 \sim 121 : 11 \Rightarrow 121 : 1 \sim 11 : 11 \Rightarrow 11 : 1 \sim 1 : 11 \Rightarrow 1 : 1$$

On the other hand, Spark learners choose their own best splitting strategy on a dataset determining their number of partitions. While Spark learners do not perform additional Radon point calculation, results shows that Parallel Radon machine achieves better runtime in comparison with some Spark learners for different choices of h. Figure 4(a) demonstrates the average cross validation runtime of *Parallel Radon Machine* with the base learner *Logistic*. While h gets bigger, number of splits gets polynomially bigger. In this case, the parallelizm level ensures speed-up. This is obvious for *codrna* and *poker* datasets for $h = 1, 2, 3$. In fact, *SMO* algorithm shows drastical speed-up in Figure 4(b). Our findings shows that this algorithm is not suitable for parallelization when data is not normalized. As we mentioned before, we disabled normalization for this algorithm to make it comparable to *SVMWithSGD*. While *SVMWithSGD* works well

without normalization, for partitions/splits with a lot of data instances *SMO* drastically falls behinds it in terms of runtime as depicted in Figure 4(b) and Figure 5(c). Therefore, we did not test that algorithm on the other datasets. However, Figure 2(b) shows that *SMO* can display good convergence rates.

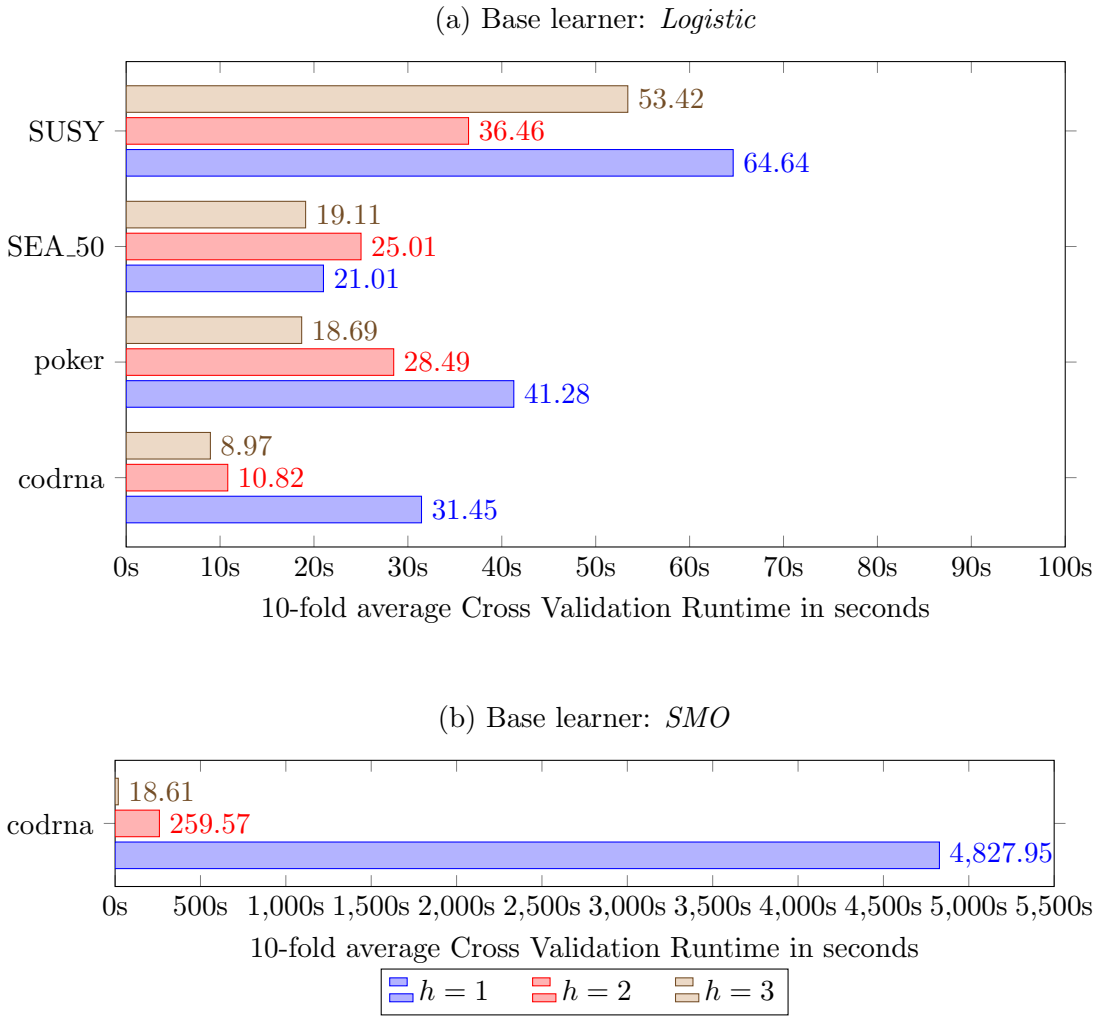
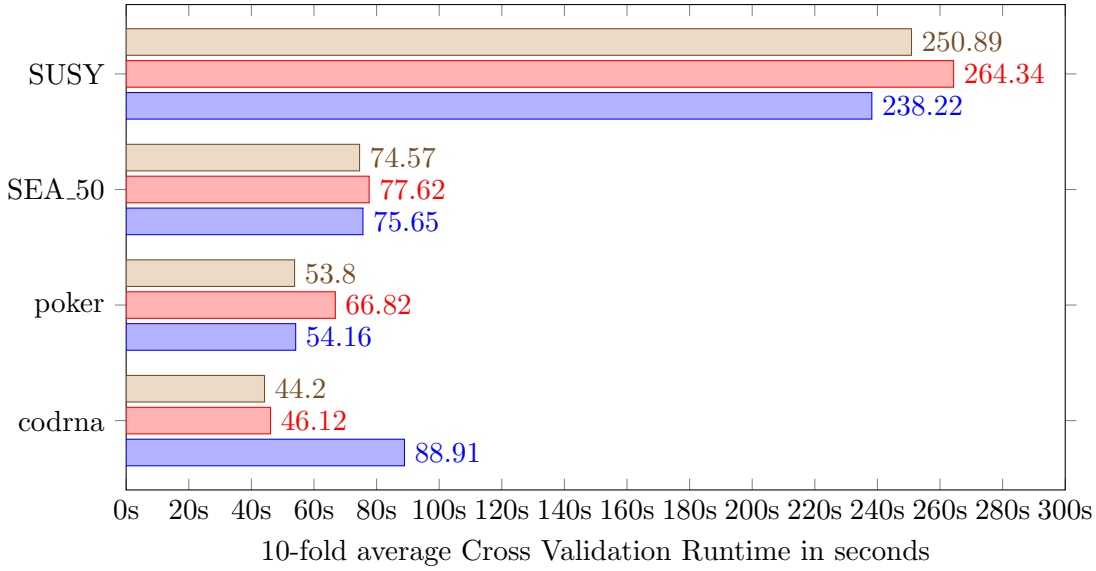


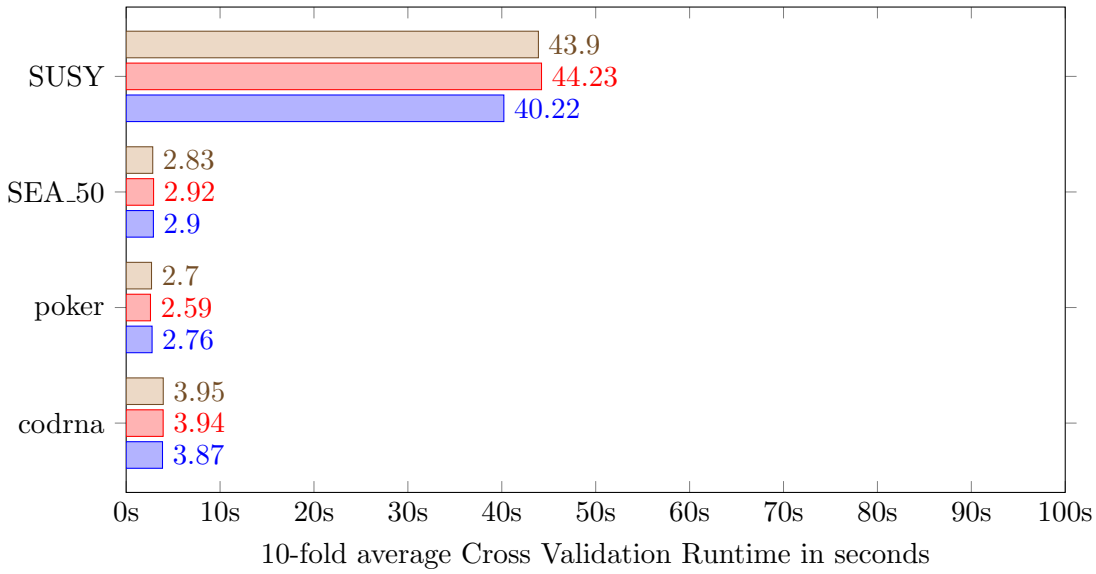
Figure 4: Runtime of Parallel Radon Machine on the same datasets for $h = 1, 2, 3$. Number of threads r^h based on Radon number r of a dataset and h iteration parameter: {codrna, 11^h}, {poker, 13^h}, {SEA_50, 6^h}, {SUSY, 21^h}. In (b) base learner *SMO* only tested on the *codrna* dataset.

The runtime of Spark learners is shown in Figure 5. *Parallel Radon Machine* with the base learner *Logistic* outperforms *LogisticRegressionWithSGD* in all chosen datasets with $h = 1, 2, 3$. However, it falls behind LBFGS implementation in most cases as shown in Figure 5(b).

(a) Spark learner: *LogisticRegressionWithSGD*



(b) Spark learner: *LogisticRegressionWithLBFGS*



(c) Spark learner: *SVMWithSGD*

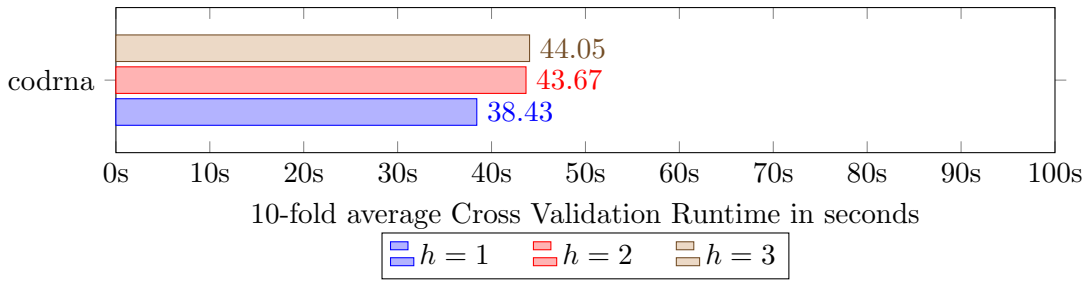


Figure 5: Runtime of Spark Learners on different datasets for comparison. Number of threads r^h based on Radon number r of a dataset and h iteration parameter: {codrna, 11^h}, {poker, 13^h}, {SEA_50, 6^h}, {SUSY, 21^h}. In (c) Spark learner *SVMWithSGD* only tested on the *codrna* dataset. In (b) for Spark learner *LogisticRegressionWithLBFGS*, parameter selection is not performed.

How do we improve the performance metrics of learning? Parameter selection is the problem of selecting a set of parameters for a learning algorithm with the aim of optimizing the algorithm's performance. For this purpose, an unseen independent data sample is chosen and then cross-validation and grid search is applied to this sample to obtain best parameters. Those are two independent techniques. After parameter selection chosen sample is discarded and normal learning is performed on the rest of the data. For our parallelization method and comparable learners, we use parameter selection via *10*-fold cross validation. For the base learner **weka.classifiers.functions.Logistic** C confidence parameter, R ridge parameter which sets the ridge in log-likelihood are optimized. Intervals are [0.001, 0.01] and [1.0E-14, 1.0E-8] respectively. Moreover, for **weka.classifiers.functions.SMO** C the complexity constant, L tolerance parameter and P the epsilon for round of error are adjusted throughout 10-fold cross validation. Respective intervals are [2.0, 3.0], [1.0E-3, 1.0E-2], and [1.0E-13, 1.0E-12]. By optimizing these parameters, cross validation accuracy of *Parallel Radon Machine* is improved for up to 2%.

For SGD-based spark learners, we optimize different set of parameters. While there are also other parameters that can be adjusted, for the sake of simplicity we choose the following four:

1. *numIterations* is the number of iterations for execution.
2. *stepSize* is a scalar value indicating initial step size for GD.
3. *regParam* is the regularization parameter in L2 regularization.
4. *miniBatchFraction* is the portion of the total data that is sampled in each iteration, to calculate the GD.

For **LogisticRegressionWithSGD** we refine $1^{st}, 2^{nd}$ and 4^{th} parameters where the grids are {150, 200}, {20.0, 40.0, 80.0} and {0.1, 0.2} respectively. But, for **SVMWith-**

SGD we utilize $1^{st}, 2^{nd}$ and 3^{rd} parameters where the corresponding grids are {150, 200}, {20.0, 40.0, 80.0} and {1.0/8192.0, 1.0/4096.0}. By performing parameter search through these values, we improve cross validation accuracy of both algorithms up to significant 12%. **LogisticRegressionWithLBFGS** is executed with default parameters in all experiments. However, it produces decent results even with default learning parameters as shown in Figure 3(b) and Figure 5(b).

6 Conclusion

Since, we obtain Radon number r from the dimension of dataset and use it along with iteration parameter h for splitting it, in our experiments we used datasets where dimension is much smaller than the dataset size. In this case, we cannot test our parallelization method for high-dimensional spaces unless the dataset size is at least a multiple of the Radon number r of the hypothesis space. Moreover, our experiments only consider linear hypothesis spaces. Learning on non-linear hypothesis spaces cause the same problems as in high-dimensional spaces. In the case of non-parametric machine learning methods such as kernel methods, the dimension of optimization problem can be equal to dataset size, since they often require far more parameters to train.

For base learners in Parallel Radon Machine and corresponding Spark learners we considered only binary classification tasks. However, regression and multi-class classification problems can be considered in this context while it is also possible to obtain linear models there.

How does our parallelization method impact communication? *Parallel Radon Machine* is aggregation at the end strategy, where the communication overhead is low. In a basic sense, hypotheses are found in parallel and the only communication need arises in an aggregation phase at the end. While in Parallel SGD and other parallelizable methods, either information is exchanged for a global update step or intermediate hypotheses are updated based on their predictive performance.

For the parameter selection we sample 10000 instances. This is because; parameter selection happens in driver node in cluster mode or main/driver thread in local mode and while the size of grid of parameters is bigger, one can face performance issues. As a rule of thumb, sampled data for parameter selection should be at least $\#attributes \times 20$. We explicitly check this and dataset size versus sample size for consistency in our implementation. We specifically chose those 4 datasets with a suitable class imbalance. Class imbalance for *codrna*, *poker*, *SEA_50* and *SUSY* datasets is around 33.3% - 66.6%, 50% - 50%, 61.5% - 38.5% and 50% - 50% respectively. In this case, we only use random sampling both in parameter optimization and in normal cross-validation. For datasets in which class imbalance differing significantly one can utilize stratified sampling for statistical stability.

During the cross validation of *Parallel Radon Machine* on base learners, the runtime not only depends the RDD manipulation for obtaining training and test sets and normal cross validation time, but also transformation from *LabeledPoint* to *Instance* to perform learning on WEKA algorithms. *LabeledPoint* object describes one data instance in the

Spark framework. *Instance* object is the same for WEKA machine learning workbench. This step is necessary, because WEKA does not work with Spark's data structures. Otherwise one need to implement his/her own algorithm to work on *LabeledPoint*. This transformation takes $\mathcal{O}(|D_i|)$ time for $i = 1, 2, \ldots, r^h$. With the careful choice of iteration parameter h, this can be minimized significantly. Since, the transformation is performed for each partition.

While we considered careful comparison of our method with state-of-art parallel algorithms in Spark framework, it can also be interesting to evaluate the speed-up per worker/processor of our method. In other words, *Parallel Radon Machine* can be compared against base learner itself when running both on the same dataset. With the usage $c = r^h$ workers/processors where each subset running in each worker/processor significant speed up can be made [Kamp et al., 2015]. On the other hand, running experiments in a local mode sometimes does not provide needed heuristics. It can be interesting to run the same experiments on the cluster to see the effects.

7 References

Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

Cheng Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. *Advances in neural information processing systems*, 19:281, 2007.

Kenneth L Clarkson, David Eppstein, Gary L Miller, Carl Sturtivant, and Shang-Hua Teng. Approximating center points with iterative radon points. *International Journal of Computational Geometry & Applications*, 6(03):357–377, 1996.

Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

Ofer Dekel, Ran Gilad-Bachrach, Ohad Shamir, and Lin Xiao. Optimal distributed online prediction using mini-batches. *Journal of Machine Learning Research*, 13(Jan): 165–202, 2012.

Eibe Frank, Stuart Inglis, and Shane Legg. Smo api. URL http://weka.sourceforge.net/doc.stable/weka/classifiers/functions/SMO.html.

Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

Trevor Hastie, Robert Tibshirani, et al. Classification by pairwise coupling. *The annals of statistics*, 26(2):451–471, 1998.

Nikola Ivanic. Predicting poker hands with neural networks. URL http://neuroph.sourceforge.net/tutorials/PredictingPokerhands/Predicting%20poker%20hands%20with%20neural%20networks.htm.

Michael Kamp, Mario Boley, and Thomas Gartner. Effective parallelization of machine learning algorithms. Preliminary work. Under review by ICML., 2015.

David Kay and Eugene W Womble. Axiomatic convexity theory and relationships between the carathéodory, helly, and radon numbers. *Pacific Journal of Mathematics*, 38(2):471–485, 1971.

S. Sathiya Keerthi, Shirish Krishnaj Shevade, Chiranjib Bhattacharyya, and Karuturi Radha Krishna Murthy. Improvements to platt's smo algorithm for svm classifier design. *Neural Computation*, 13(3):637–649, 2001.

Aleksandar Lazarevic and Zoran Obradovic. Boosting algorithms for parallel and distributed learning. *Distributed and Parallel Databases*, 11(2):203–229, 2002.

Saskia Le Cessie and Johannes C Van Houwelingen. Ridge estimators in logistic regression. *Applied statistics*, pages 191–201, 1992.

Moshe Lichman. Uci machine learning repository, 2013. URL http://archive.ics.uci.edu/ml.

Ryan Mcdonald, Mehryar Mohri, Nathan Silberman, Dan Walker, and Gideon S Mann. Efficient large-scale distributed training of conditional maximum entropy models. In *Advances in Neural Information Processing Systems*, pages 1231–1239, 2009.

Yurii Nesterov and Arkadi Nemirovski. On first-order algorithms for l 1/nuclear norm minimization. *Acta Numerica*, 22:509–575, 2013.

Jorge Nocedal. Updating quasi-newton matrices with limited storage. *Mathematics of computation*, 35(151):773–782, 1980.

John C Platt. Fast training of support vector machines using sequential minimal optimization. *Advances in kernel methods*, pages 185–208, 1999.

Boris T Polyak and Anatoli B Juditsky. Acceleration of stochastic approximation by averaging. *SIAM Journal on Control and Optimization*, 30(4):838–855, 1992.

Johann Radon. Mengen konvexer körper, die einen gemeinsamen punkt enthalten. *Mathematische Annalen*, 83(1):113–115, 1921.

Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. IEEE, 2010.

Evan R Sparks, Ameet Talwalkar, Virginia Smith, Jey Kottalam, Xinghao Pan, Joseph Gonzalez, Michael J Franklin, Michael I Jordan, and Tim Kraska. Mli: An api for distributed machine learning. In *2013 IEEE 13th International Conference on Data Mining*, pages 1187–1192. IEEE, 2013.

Choon Hui Teo, S Vishwanathan, Alex Smola, and Quoc V Le. Bundle methods for regularized risk minimization. *Journal of Machine Learning Research*, 1:55, 2009.

Joaquin Vanschoren, Jan N Van Rijn, Bernd Bischl, and Luis Torgo. Openml: networked science in machine learning. *ACM SIGKDD Explorations Newsletter*, 15(2):49–60, 2014.

Xin Xu. Logistic api. URL http://weka.sourceforge.net/doc.stable/weka/classifiers/functions/Logistic.html.

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy Mccauley, M Franklin, Scott Shenker, and Ion Stoica. Fast and interactive analytics over hadoop data with spark. *USENIX Login*, 37(4):45–51, 2012.

Martin Zinkevich, John Langford, and Alex J Smola. Slow learners are fast. In *Advances in Neural Information Processing Systems*, pages 2331–2339, 2009.