Parallel Computing 113 (2022) 102955

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

SVM-SMO-SGD: A hybrid-parallel support vector machine algorithm using sequential minimal optimization with stochastic gradient descent

Gizen Mutlu *, Çiğdem İnan Acı

Mersin University, Department of Computer Engineering, 33343, Mersin, Türkiye

ARTICLE INFO

Keywords:
SVM
SGD
SMO
Hybrid classification
Parallel machine learning
Performance improvement

ABSTRACT

The Support Vector Machine (SVM) method is one of the popular machine learning algorithms as it gives high accuracy. However, like most machine learning algorithms, the resource consumption of the SVM algorithm in terms of time and memory increases linearly as the dataset grows. In this study, a parallel-hybrid algorithm that combines SVM, Sequential Minimal Optimization (SMO) with Stochastic Gradient Descent (SGD) methods have been proposed to optimize the calculation of the weight costs. The performance of the proposed SVM-SMO-SGD algorithm was compared with classical SMO and Compute Unified Device Architecture (CUDA) based approaches on the well-known datasets (i.e., Diabetes, Healthcare Stroke Prediction, Adults) with 520, 5110, and 32,560 samples, respectively. According to the results, Sequential SVM-SMO-SGD is 3.81 times faster in terms of time, and 1.04 times more efficient RAM consumption than the classical SMO algorithm. The parallel SVM-SMO-SGD algorithm, on the other hand, is 75.47 times faster than the classical SMO algorithm in terms of time. It is also 1.9 times more efficient in RAM consumption. The overall classification accuracy of all algorithms is 87% in the Diabetes dataset, 95% in the Healthcare Stroke Prediction dataset, and 82% in the Adults dataset.

1. Introduction

In recent years, performance improvements have been made in many learning algorithms, such as the Support Vector Machine (SVM) algorithm, frequently used in classification applications due to the increasing interest in machine learning applications and growing datasets [1–4].

SVM draws a straight line to separate points on a plane. Then, two more classifier lines are drawn at +1 and -1 from this line. The area between these lines is called the margin. The larger the margin, the better the classifier's chance to succeed [5].

Besides being a good classifier, SVM training requires solving the Quadratic Programming (QP) optimization problem which takes time as an inner loop. The Sequential Minimal Optimization (SMO) algorithm performs this solution at the optimal level. The SMO algorithm presented by Platt [6] iteratively solves a QP problem with two unknowns using analytical methods. In training set size, SVM scales linearly and cubically. In SMO, on the other hand, matrix calculation is not used, and there is an SVM evaluation in the background of the calculation time of SMO. Therefore, SMO is the fastest method used in linear SVM with sparse datasets. The memory size that SMO can run on is directly proportional to the training set and allows for working with large training sets [6].

Algorithms such as Stochastic Gradient Descent (SGD) also overcome performance issues and speed up convergence, especially in large datasets. It has been shown that these algorithms positively affect performance optimization [7].

The SGD algorithm randomly selects a sample from the training dataset at each step and calculates the gradient for that sample only. The algorithm is much faster as it requires very little data processing to work on a sample at each iteration. In addition, it becomes convenient to work with vast datasets since each iteration only has one sample in memory [8].

SMO allows faster optimization by breaking down QP problems into smaller pieces while performing SVM calculations. In other words, it is possible to speed up SVM calculations at a certain level using SMO. At the same time, since it can solve QP problems more easily, it also indirectly benefits time and RAM consumption. The amount of memory required for SMO is directly proportional to the training dataset size and is suitable for working with large datasets [6]. However, as the size of the dataset increases, the working time and resource consumption increase in direct proportion. So it is open to performance improvement. The SGD algorithm, on the other hand, is one of the popular algorithms used for large data sets, thanks to its ability to optimize resource consumption according to the periods given to the training set [9]. SGD is

* Corresponding author.

https://doi.org/10.1016/j.parco.2022.102955
Received 28 April 2022; Received in revised form 27 June 2022; Accepted 15 July 2022
Available online 16 July 2022

inherently suitable for large dataset instances and large feature datasets. It allows the algorithm to run faster as it has very little data to manipulate at each iteration by randomly choosing a sample at each epoch. However, only one sample per iteration resides in memory and is suitable for operation on large training sets [8]. Therefore, SGD helps the efficient use of resource consumption. In short, the main reason for combining SMO and SGD is to achieve both high accuracy and partial runtime savings by using SMO. This paper aims to improve the working time and resource consumption with a hybrid approach by updating the weight in the SMO model without disturbing the SGD's nature. Although various performance improvements have been made for the SVM algorithm before, as far as we know, the hybrid version of the SVM-SMO-SGD algorithms has not been proposed yet.

This study evaluated the SVM algorithm on the well-known large-scale datasets (i.e. Healthcare, Adults) with 5,110 and 32,560 samples, respectively, starting from medium-sized datasets with 520 samples (i.e. Diabetes) while calculating with SMO. As the data size increased, the algorithm's running time increased proportionally. The main reason for this was that the weight calculation cost of the algorithm increased proportionally as the data size increased. The proposed SVM-SMO-SGD hybrid algorithm fits the purpose of optimizing the calculation of the weight cost. The serial and parallel performance of the proposed algorithm was compared with classical SMO and the Parallel Cross-Validation (PCV) algorithm that was run on CUDA by Li et al. [10]. Encouraging results were obtained from the proposed SVM-SMO-SGD hybrid algorithm on the Central Processing Unit (CPU) and Graphics Processing Unit (GPU).

The paper is organized as follows: Section 2 presents the SVM improvements existing in the literature, the basic principles of SVM, SMO, SGD and the materials used in this study. In section 3, the results are obtained from various datasets of the algorithm. Section 4 gives the performance results and discussions.

2. Materials and method

This section briefly describes the basic principles of L1 soft margin linear and core SVM design for SMO, SGD, and binary classification.

2.1. Literature review

Previous studies on performance improvements using SVM are summarized in the following paragraphs:

Diaz et al. [11] applied their parallel algorithms by dividing the matrices into quadruple branches to share the matrix inversion process among the existing processors. They implemented this process using OpenMP and reduced the running time of an SVM algorithm by using a multi-core environment with shared memory, using a parallel semi-parametric support vector machines algorithm.

Gu et al. [12] proposed a generalized framework to accelerate SMO through Sub-Stochastic Gradient Descent (SSGD) for various SVMs. According to the obtained results, the method effectively accelerated SMO.

Li et al. [10] developed a new parallel SVM training application to speed up cross-validation by running multiple tasks simultaneously on the GPU. These tasks with different parameter values share the same cache that stores the kernel accounts of the support vectors. Therefore, it reduces unnecessary core values calculations in various training tasks.

Sopyła et al. [13] presented the Barzilai-Borwen algorithm in five different variants for the classical SGD algorithm in the update step for solving the optimization function that emerged in the primal form of L2-SVM. This method aims to check how effective it is in terms of execution time, convergence, and initial hyperparameters in large-scale scenarios. When they compared the results with the classical linear SVM algorithm results, they showed that the results were very similar to each other.

You et al. designed a new algorithm for researchers who suffer from low prediction accuracy and high runtime overhead in the high-performance computing area of the SVM algorithm. This algorithm designed MIC SVM, a highly efficient parallel SVM for x86-based multi-core and multi-core architectures such as Intel Ivy Bridge CPUs and Intel Xeon Phi coprocessor (MIC). MIC-SVM achieved 4.4–84 and 18–47 times accelerations compared to LIBSVM algorithm on MIC and Ivy Bridge CPUs, respectively, on various datasets [14].

Wen et al. [15] developed a new open-source SVM library called ThunderSVM that takes advantage of the high performance of GPUs and multi-core CPUs. ThunderSVM efficiently performs all functions of LIBSVM, such as classification and regression. ThunderSVM is used in multiple languages, including C/C++ and Python. Experimental results of this library revealed that it is generally faster than LIBSVM when generating the same SVMs.

Nandan et al. [16] designed a new algorithm called Approximate Endpoints SVM (AESVM) to overcome the high training time of nonlinear core SVMs on large datasets. This algorithm performs SVM optimization on a sub-dataset selected as the representation of the training dataset. It revealed that the training time of the AESVM algorithm is considerably shorter than the other SVM algorithms they have compared, and the accuracy rate is similar to LIBSVM.

Baldomero-Naranjo et al. [17] proposed a new classification SVM model based on simultaneous outlier detection and feature selection. This classification model considers ramp loss margin error and imposes cost constraints to limit the number of features selected. Two different approaches, precise and intuitive, are proposed in the model. The quality of the solutions provided in the heuristic approach was verified by comparing them with the exact approach. Their efficiency was tested by comparing them with existing SVM-based models.

Kashef [18] proposes an algorithm for accelerating the increasing training time for nonlinear SVM classifiers on large datasets. This algorithm uses the concept of increasing and decreasing learning. This algorithm's classification accuracy and speed, proposed according to the results obtained from the experiments carried out in different sizes, showed better results than the standard SVM classifiers.

Nevertheless, working with machine learning algorithms on distributed systems, they can be blocked when machines fail during objective function optimization in large-scale datasets. Wang et al. [19] developed a hybrid algorithm to balance performance and efficiency. In this algorithm, at each iteration, they eliminated the results of the failed machines and discussed the relationship between accuracy and abandonment rate. They discussed that the algorithm they developed is very effective in terms of speed and efficiency in a distributed system.

Tong et al. [20] proposed a new parallel algorithm that truly parallelizes one of the most widely used quasi-Newtonian methods, the Limited memory Broyden–Fletcher–Goldfarb–Shanno algorithm (L-BFGS) method, with a guarantee of convergence. Using the variance reduction technique, they showed that while achieving the same linear convergence rate with the non-parallel stochastic L-BFGS algorithm, it also provided a significant acceleration. In addition to speeding up, this algorithm also gave better results than first-order methods in solving bad conditional problems.

When the studies mentioned above are examined, various studies have been carried out for SVM, SGD, and SMO algorithms in parallel and serial environments. However, as far as we know, there is no study that proposes a hybrid version of these three algorithms which runs on both CPU and GPU. A comparison table (Table 1) is presented to show the place of the proposed algorithm among previous studies more clearly. As is seen in Table 1, the performance of the parallel SVM-SMO-SGD algorithm is quite promising regarding computational cost and acceleration rate. Although the serial SVM-SMO-SGD algorithm has the least number of cores compared to other algorithms, it can be considered successful in performance improvement.

Table 1
Performance comparison of the previous studies with the SVM-SMO-SGD algorithm.

Ref. No.	Algorithm	# of features	# of samples	# of cores(CPU/GPU)	Time(sec.)	Speed Up
[12]	SSGD-SMO	8	59,535	4	74.64	3.96x
[14]	MIC-SVM	123	32,561	122	2.1	4x
	Sequential SVM-SMO-SGD	**123**	**32,561**	**2**	**40.12**	**5.13x**
[13]	SGD with BB	784	70,000	4	5.6	5.42x
[20]	AsySQN	20,958	72,309	8	17	6x
[18]	BSVM	2	1,000	6	41.65	6.65x
[16]	AESVM	123	32,561	No information	1.2	14.52x
[15]	ThunderSVM	300	49,749	10 (CPU)	16.5 (CPU)	23x (CPU)
				3,584 (GPU)	4.66 (GPU)	483x (GPU)
[17]	RL-FS-M	109	8,124	6	529.65	121.2x
	Parallel SVM-SMO-SGD	**123**	**32,561**	**5120 (GPU)**	**1.09 (GPU)**	**188.8x (GPU)**

2.2. Computing unified device architecture

CUDA architecture executes multiple CUDA threads with the same functionality. These threads run in parallel in a group called blocks. Multiple threading blocks run concurrently, and the group of these threading blocks is also called a grid. Fig. 1 shows the relationships between these concepts [24]:

The number of executable thread blocks in parallel varies depending on the block's resources and the number of resources available. The streaming multiprocessor count is 80 for Tesla V100. CUDA streaming multiprocessors control threads in batches of 32. This group of 32 is called warp. In this way, one or more warps configure a thread block [24]. Fig. 2 shows the threads in 16 × 16 × 1 blocks.

2.3. Support vector machine

2.3.1. Linear SVM

In simplest terms, a linear SVM is a hyperplane that separates the positive samples from the negative examples with the maximum distance. The distance defines the margin between the closest positive and negative examples to the hyperplane [17]. The linear SVM's formula is given in Eq. (1):

$$\boldsymbol{\mu} = \vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}} \; -- \boldsymbol{b}, \tag{1}$$

w is the normal hyperplane vector, b is a scaler and x is the input vector [26]. The separation hyperplane is the plane $\mu = 0$, and the closest point is in the plane $\mu = \pm 1$.

The margin is given in Eq. (2) [6]:

$$\boldsymbol{m} = \frac{1}{\boldsymbol{w}_2}. \tag{2}$$

Using the Lagrangian, the optimization problem can be converted into binary form, the QP problem, in which the objective function φ depends only on a set of Lagrangian multipliers α_i,

$$\min_{\alpha_i} \varphi(\vec{\alpha}) = \min_{\alpha_i} \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} y_i y_j \left(\vec{x}_i \cdot \vec{x}_j \right) \alpha_i \alpha_j - \sum_{i=1}^{N} \alpha_i, \tag{3}$$

$$s.t\ \alpha_i \geq 0,\ \forall i,$$

$$\sum_{i=1}^{N} y_i \alpha_i = 0. \tag{4}$$

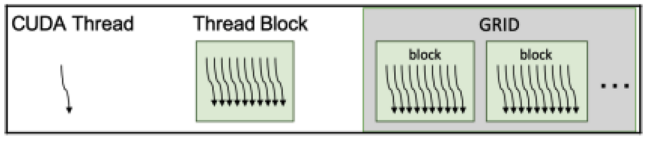

Fig. 1. Relationships between thread, block, and grid [24].

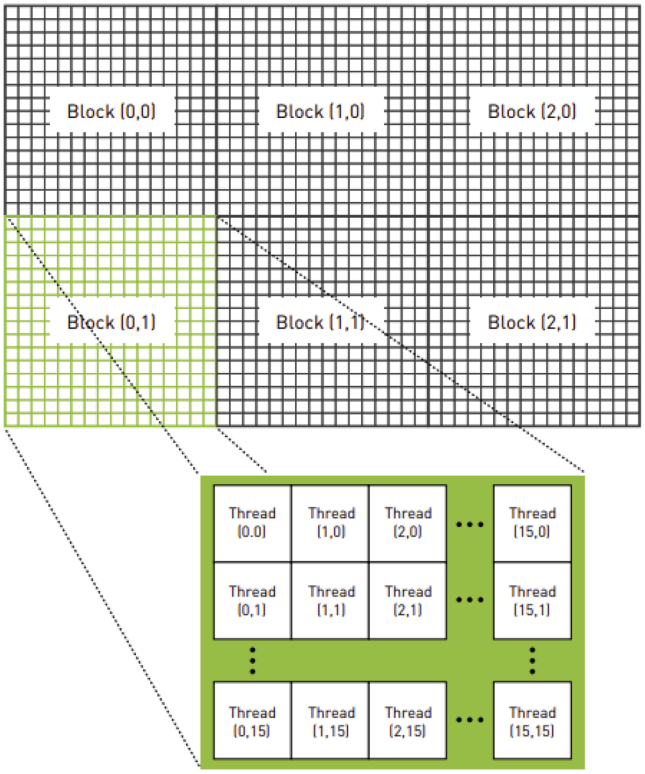

Fig. 2. 16 × 16 threads in a block in CUDA [25].

The one-to-one relationship between α_i and $\vec{x}_i$.

$$\vec{w} = \sum_{i=1}^{N} y_i \alpha_i \vec{x}_i, \; b = \vec{w} \cdot \vec{x}_k - y_k,$$

$$\alpha_k > 0. \tag{5}$$

The calculation required to evaluate a linear SVM is equal to the number of non-zero support vectors since it can be calculated via the above Eq. (5) with the training data before using $\vec{w}$.

2.3.2. Non-linear SVM

SVMs may not always use linear classifiers. A nonlinear SVM is computed with Lagrange multipliers [6],

$$\mu = \sum_{i=1}^{N} y_j \alpha_j K\left(\vec{x}_j, \vec{x} \right) \; -- b. \tag{6}$$

The kernel function designated K measures the similarity or distance between the input vector $\vec{x}$ and the training vector $\vec{x}_j$. There are various kernels, such as Gaussians and polynomials. If the kernel is

linear, the SVM calculation turns into Eq. (1). Objective function according to Non-Linear Kernel

$$\min_{\vec{\alpha}_i}\varphi(\vec{\alpha}) = \min_{\vec{\alpha}_i}\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} y_i y_j K\left(\vec{x}_i, \vec{x}_j\right)\alpha_i\alpha_j - \sum_{i=1}^{N}\alpha_i \tag{7}$$

$$0 \le \alpha_i \le C, \ \forall i,$$

$$\sum_{i=1}^{N} y_i\alpha_i = 0. \tag{8}$$

Karush-Kuhn-Tucker (KKT) for the QP problem in Eq. (8).

$$\left.\begin{array}{l} \alpha_i = 0 \Leftrightarrow \ y_i\mu_i \ge 1, \\ 0 < \alpha_i < C \ \Leftrightarrow \ y_i\mu_i = 1, \\ \alpha_i = \ C \ \Leftrightarrow y_i\mu_i \le 1. \end{array}\right\} \tag{9}$$

2.4. Sequential minimal optimization (SMO) with two Lagrange multipliers

SMO calculates the bounds on the two Lagrange multipliers and solves these factors for the constrained minimum point. Due to their limited constraints, Lagrange multipliers stay in a box. With the linear equality constraint, these multipliers cause them to lie on the diagonal line. Therefore, the minimum point of the objective function must be on a diagonal. The minimum number of Lagrange multipliers to optimize is two because if the SMO algorithm only optimizes one factor, it cannot satisfy the linear equality constraint at every step [6]. The pseudocode of the SMO algorithm is given in Algorithm 1.

If y_1 is not equal to y_2, the ends of this diagonal line can be expressed as

$$y_1 \ne y_2 \left\{\begin{array}{l} L = \max(0, \ \alpha_2 - \alpha_1) \\ H = \min(C, \ C + \alpha_2 - \alpha_1). \end{array}\right. \tag{10}$$

If y_1 equals y_2,

$$y_1 = y_2 \left\{\begin{array}{l} L = \max(0, \ \alpha_2 + \alpha_1 - C), \\ H = \min(C, \ \ \alpha_2 + \alpha_1). \end{array}\right. \tag{11}$$

At each iteration while training the model, the α values are updated as follows:

$$\eta = K\left(\vec{x}_1, \vec{x}_1\right) + K\left(\vec{x}_2, \vec{x}_2\right) - 2K\left(\vec{x}_1, \vec{x}_2\right). \tag{12}$$

E_i is the error on the ith training sample.

$$s = y_1 y_2$$

$$\alpha_2^{new} = \alpha_2 + \frac{y_2(E_1 - E_2)}{\eta},$$

$$\alpha_2^{new,clipped} = \left\{\begin{array}{c} H \ if \ \alpha_2^{new} \ge H; \\ \alpha_2^{new} \ if \ L < \alpha_2^{new} < H; \\ L \ if \ \alpha_2^{new} \le L \end{array}\right.$$

$$\alpha_1^{new} = \ \alpha_1 + s\left(\alpha_2 - \alpha_2^{new,clipped}\right). \tag{13}$$

Under ordinary conditions, η should be positive, but a negative η can cause the objective function to be ambiguous. Even with the correct kernel, zero η can occur if the training set has the exact input vector x.

For the b threshold, the KKT conditions are recalculated for both optimized samples after each step [6].

$$b_1 = E_1 + y_1\left(\alpha_1^{new} - \alpha_1\right)K\left(\vec{x}_1, \vec{x}_1\right) + \ y_2\left(\alpha_2^{new,clipped} - \alpha_2\right)K\left(\vec{x}_1, \vec{x}_2\right) + b. \tag{14}$$

$$b_2 = E_2 + y_1\left(\alpha_1^{new} - \alpha_1\right)K\left(\vec{x}_1, \vec{x}_2\right) + \ y_2\left(\alpha_2^{new,clipped} - \alpha_2\right)K\left(\vec{x}_2, \vec{x}_2\right) + b \ . \tag{15}$$

2.5. Stochastic gradient descent (SGD)

The SGD algorithm aims to solve the high computational cost by having much faster convergence. However, as the number of data increases, the time required for weight updates increases [27].

In this algorithm, instead of the update direction is precisely based on the gradient, the update direction is chosen as a random vector, and it is ensured that the expected value is equal to the gradient direction at each iteration [28]. The pseudocode of the SGD algorithm is given in Algorithm 2.

The loss function in the SVM algorithm is as follows,

$$\lambda w^2 + \max\left\{0, \ 1 - y w^T\phi(x)\right\}. \tag{16}$$

And to get the weight vector with SGD [29]

$$w \leftarrow w - \gamma_t \left\{\begin{array}{l} \lambda_w \ \ if \ y_t w^T\phi(x_t) > 1, \\ \lambda_w \ - y_t\phi(x_t) \ otherwise. \end{array}\right. \tag{17}$$

2.6. Parallel cross-validation (PCV) algorithm

Li et al. [10] developed a new parallel SVM algorithm to speed up the cross-validation procedure in training tasks with multiple and different hyperparameters on CUDA simultaneously. The pseudo-code of the PCV algorithm is given in Algorithm 3 and we refer to it as PCV later in the article. In cross-validation, training time becomes very long due to unnecessary calculations of kernel values during multitasking. Therefore, they have cached the core matrix values of the support vectors obtained at the end of each task, significantly reducing unnecessary computations. If the relevant kernel matrix value exists in the cache, it calls this result from the cache. If not, the relevant calculation is done on CUDA. Thus, unnecessary process redundancy is prevented, and since the value encountered for the first time is calculated on CUDA, it has become a very efficient algorithm in terms of time consumption. As a result of their experiments, they proved that the cost of multitasking cross-validation training in terms of time is relatively low. The b_{up} and b_{lo} values given in Algorithm 3 are obtained using Eq. (18) and Eq. (19).

$$I_0 = \{i : 0 < \ \alpha_i < C \ \},$$

$$I_1 = \{i : \ y_i = 1, \ \alpha_i = 0 \ \},$$

$$I_2 = \{i : \ y_i = -1, \ \alpha_i = C \ \},$$

$$I_3 = \{i : \ y_i = 1, \ \alpha_i = C \ \},$$

$$I_4 = \{i : \ y_i = -1, \ \alpha_i = 0 \ \},$$

$$I_{up} = I_0 \cup I_1 \cup \ I_2,$$

$$I_{lo} = I_0 \cup I_3 \cup \ I_4. \tag{18}$$

$$e_i = \sum_{j:\vec{x_j}\in S} \alpha_j y_j K\left(\vec{x_j}, \vec{x_i}\right) - y_i,$$

$$b_{up} = \min\left\{e_i : i \in I_{up}\right\},$$

$$b_{lo} = \max\left\{e_i : i \in I_{lo}\right\},$$

$$b_{lo} \le b_{up}. \tag{19}$$

2.7. The proposed SVM-SMO-SGD algorithm

As it is mentioned before, the SVM algorithm is very costly in terms of time, CPU, and RAM consumption in large datasets due to the weight calculation process. Platt's SMO algorithm is very convenient for optimizing the weight calculation cost. At this point, we know that the SGD algorithm accelerates the weight calculation process accurately, especially in large datasets. In this study, our motivation and starting point is to optimize the time, CPU, and RAM usage by using SGD while calculating the SVM algorithm with SMO in large datasets.

In the SVM-SMO-SGD algorithm (Fig. 3), an empty array of training set sizes is created to store alpha values. Likewise, a copy of the alpha array is created after alpha updates are made that stores the old alpha values. Following the SMO rules, a random sample (j) is selected together with the training dataset sample (i) for kernel calculations, and the kernel calculation is made for these two samples. These kernel calculations can be of Gaussian, Polynomial, or Linear type. We adopted the Gaussian Radial Basis Function kernel calculation in this study. If the i and j values in the target dataset are not equal, the ninth and tenth steps given in Algorithm 4 are applied for L and H calculations. If they are equal, steps 13 and 14 are performed. Then, according to the obtained values, weight calculation is made using the standard SGD [30] algorithm presented in Algorithm 2. In this study, the number of iterations for the parameters of the SGD algorithm were selected as 4 and the learning rate was 0.45. A bias (b) calculation is made with the weight value obtained. To calculate the error rate of the i^{th} sample in the training set, the i^{th} training sample is dotted with the weight vector. Then, it is added to the sign function by adding the b value we obtained in the previous step. The sign function marks $+1$ if the result is greater than*0, - 1* otherwise. Accordingly, when the i^{th} in the target set is

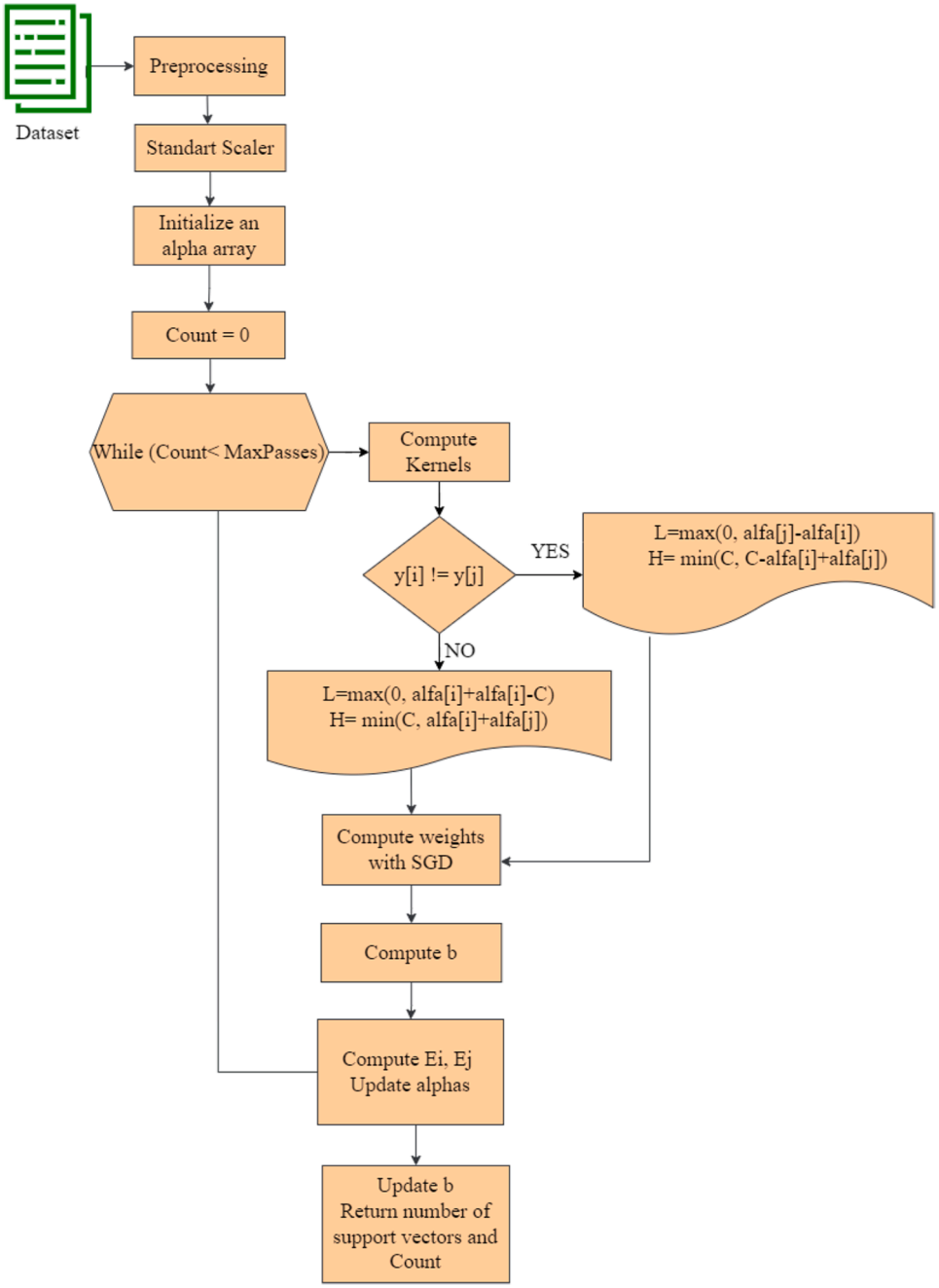

Fig. 3. Flowchart of the proposed SVM-SGD-SMO algorithm.

removed from the sample, the error value E_i is obtained for the i^{th} sample. The same operations are performed for the $E_j$value. Alpha updates for the i and j values, and finally, the b update is performed. While running the algorithm, no hyper-parameters were involved other than the standard SMO and SGD parameters. The pseudo-code of the SVM-SMO-SGD algorithm is given in Algorithm 4.

$$sign(w, x_i + b). \tag{20}$$

In the prediction array obtained from Eq. (20), data labeled as 1 were classified as True Positive (TP), and data labeled as - *1* were classified as True Negative (TN). Then, as in Eq. (21), the accuracy rate was obtained by dividing the sum of TN and TP by the length of the test dataset.

$$\frac{TP + TN}{Lenght\ of\ Test\ set} \tag{21}$$

The CUDA version of our hybrid algorithm we explained above is given in Algorithm 5. The alpha array of the same size as the generated training set size is sent to the GPU to store the alpha values. SMO rules are enforced on the GPU unchanged. The standard SGD algorithm given in Algorithm 2 runs on the GPU during the weight calculation. Hyper-parameters are implemented in the same way as in the CPU. The resulting number of support vectors and the variable value is marked as the count sent from the GPU to the CPU. The CUDA-based SVM-SMO-SGD algorithm is run on a 32 × 32 grid size and 32 × 32 × 1 block.

3. Results and discussions

3.1. Datasets

This section presents datasets used in the study for performance comparison. All of the datasets have been utilized for classification problems. Therefore, the workload of the algorithms only depends on the size and the number of features. Although the feature numbers of the datasets are close, the target sets have two classes. It is important to fix other properties in order to get a sound comparison. The main reason for choosing these datasets is that the size increases proportionally when comparing performance (Table 2).

3.1.1. Diabetes dataset

This dataset obtained from the data of people with diabetes is very suitable for classification applications as it contains 520 samples, 16 features, and a target class. Since it includes textual expressions such as Yes/No, Male/Female, and Positive/Negative, we changed these values to 0 and 1, making it easier for the algorithms to read the data [21].

3.1.2. Healthcare stroke prediction dataset

This dataset consists of 12 features and 5110 numerical samples and is used to predict or classify the probability of having a stroke depending on gender, age, various diseases, and smoking [22].

3.1.3. Adults dataset

This dataset, consisting of 32,560 samples and 14 features, was used to determine whether a person earned more than 50 thousand dollars per year and applied to classification problems [23].

Table 2
Number of training and test samples and hyperparameter of datasets

Dataset	#of training samples	# of test samples	#of features	C	γ
Diabetes	390	130	16	1	0.0625
Healthcare	3,832	1,278	12	1	0.0625
Adults	24,420	8,140	14	1	0.0625

3.2. Discussions

All algorithm experiments were performed on an NVIDIA Tesla V100 SXM2 5120 core graphics card with 16 GB of memory, using the epsilon parameter value of 0.001. Hardware specifications are given in Table 3.

The classical SMO algorithm is first applied to the datasets. As shown in Fig. 4, it has a computation time of 1,63 sec. in the Diabetes dataset, 17,24 sec. in the Healthcare Stroke Prediction dataset, and 205,79 sec. in the Adults dataset. In the PCV algorithm, the PyCUDA [31] framework is used to perform calculations on CUDA. It has a run time of 0.1 sec. in the PCV Diabetes dataset, 13,28 sec. in the Healthcare Stroke Prediction dataset, and 17 sec. in the Adults dataset. The SVM-SMO-SGD algorithm runs on the CPU and runs 0,93 sec. in the Diabetes dataset, 3,78 sec. in the Healthcare Stroke Prediction dataset, and 40,12 sec. in the Adults dataset. The SVM-SMO-SGD algorithm runs on the GPU in 0,26 seconds on the Diabetes dataset. It runs at 0,55 seconds on the Healthcare Stroke Prediction dataset and 1,09 seconds on the Adults dataset.

While the classical SMO algorithm consumes 10.1% of RAM in the Diabetes dataset, this value is seen as 5.87%, 9.7% and 5.1% in the PCV, Sequential and Parallel SVM-SMO-SGD algorithms, respectively. In the Healthcare Stroke Prediction dataset, the SMO algorithm consumed 10.4%, PCV 5.97%, sequential and parallel SVM-SMO-SGD 9.8% and 5.5% RAM. Finally, the SMO algorithm consumed 10.9%, PCV 7.4%, sequential and parallel SVM-SMO-SGD algorithm 10.5% and 5.9% RAM in the Adults dataset. The results are presented in Fig. 5.

As is seen in Fig. 6, while the classical SMO algorithm gave 87% accuracy in the Diabetes dataset, this value was observed as 87% in PCV, SVM-SMO-SGD, and parallel SVM-SMO-SGD algorithms. The SMO algorithm achieved 95% correct results on the Healthcare Stroke Prediction dataset, and PCV, SVM-SMO-SGD, and parallel SVM-SMO-SGD were 95% correct. Finally, the SMO algorithm produced 82% accuracy on the PCV, SVM-SMO-SGD, and parallel SVM-SMO-SGD Adults dataset. Accuracy rates did not change in the three different SVM calculations.

As is shown in Fig. 7, 231 support vectors are generated in the Diabetes dataset in the classical SMO algorithm, 51 in the PCV algorithm, 217 support vectors in the SVM-SMO-SGD, and parallel SVM-SMO-SGD algorithm. In the Healthcare Stroke Prediction dataset, 537 support vectors are produced with the SMO algorithm, 497 support vectors in PCV, 546 support vectors in SVM-SMO-SGD, and parallel SVM-SMO-SGD algorithms, respectively. 21,518 support vectors in SMO algorithm, 3256 support vectors in PCV algorithm, SVM-SMO-SGD, and 21,621 support vectors in parallel SVM-SMO-SGD algorithm were produced in the Adults dataset.

As is seen from Fig. 4 to Fig. 7, the proposed SVM-SMO-SGD algorithm, PCV algorithm, and Platt's SMO algorithms were compared in terms of time, CPU, RAM consumption, and accuracy rates. The proposed SVM-SMO-SGD algorithm is considered on both CPU and GPU, and both hardware results are included in the comparison. Before going through the performance analysis of each dataset, an overall analysis of the proposed algorithm is given below:

- In the classical SMO algorithm, as the data size increases, the time, the number of support vectors and the RAM consumption increase.

Table 3
Hardware Features

Hardware	
Graphics	NVIDIA Tesla V100 SXM2
Memory	16 GB
Processor	Intel Xeon CPU @ 2.20 GHz
Graphic Cores	5120
Graphics Speed	1312-1530 MHz
CPU Cores	2
CPU Speed	2199.998 MHz
Cache Size	56320 KB
RAM	12 GB

Algorithm 1
Pseudocode of the SMO Algorithm [6]

$y_1 = target[i_1]$
$E_1 = SVM\ output[1] - y_1$
$s = y_1 y_2$
If (L=H) return 0
k_{11} = kernel(point[i_1], point[i_1])
k_{12} = kernel(point[i_1], point[i_2])
k_{22} = kernel(point[i_2], point[i_2])
eta = k_{11}+k_{22} − 2·k_{12}
If (eta>0)
$\alpha_2 = alph_2 + y_2 \cdot (E_1 - 2)$/eta
  If (α_2< L) α_2= L
  Else If (α_2> H) α_2= H
  Else
  L_{obj} = objective function at α_2=L
  H_{obj} = objective function at α_2=H
  If ($L_{obj} < H_{obj} - eps$)
  α_2= L
  Else If ($L_{obj} > H_{obj} + eps$)
  α_2= H
  Else
  $\alpha_2 = alph_2$
  If ($|\alpha_2 - alph_2| <$ eps*(α_2+$alph_2$+eps))
  return 0
  $\alpha_1 = alph_1 + s \cdot (alph_2 - \alpha_2)$
  Update b to reflect change in alphas
  Update E_1 using new alphas
  Store α_1 in the alpha's array
  Store α_2 in the alpha's array
Return 1

Algorithm 2
Pseudocode of the SGD Algorithm [30]

n=0
While (convergence)
  Randomly shuffle training set
  For each (u, i) in the training set
  $e_{u,i} = (r_{u,i} - \sum_{k=1}^{K} p_{uk} \cdot q_{ki})$
  For each $q \in \{1 \ldots k\}$ ***do***
  $\widehat{p}_{uk} = p_{uk} + \eta \cdot (e_{u,i} \cdot q_{ki} - \lambda \cdot p_{uk})$
  For each $q \in \{1 \ldots k\}$ ***do***
  $\widehat{q}_{ki} = q_{ki} + \eta \cdot (e_{u,i} \cdot p_{uk} - \lambda \cdot q_{ki})$
  For each $q \in \{1 \ldots k\}$ ***do***
  $p_{uk} = \widehat{p}_{uk}$ *and* $q_{ki} = \widehat{q}_{ki}$)
  End for
  n=n+1
End while

Algorithm 3
Pseudocode of the PCV Algorithm [10]

α_i^k=0, $e_i^{k,p} = -y_i^k$ (device)
Compute $b_{up}^{k,p}$, $b_{lo}^{k,p}$, $i_{up}^{k,p}$, $i_{lo}^{k,p}$ (device)
Compute $b_{up}^{k,p}$, $b_{lo}^{k,p}$, $i_{up}^{k,p}$, $i_{lo}^{k,p}$ (host)
While $b_{lo}^k > b_{up}^k + 2\tau$ **do**
  $\forall p, q \in [1, k]$ and $p, q \in *Z*$
  If $i_{up}^p = i_{up}^q \,||\, i_{up}^p = i_{lo}^q \,||\, i_{lo}^p = i_{up}^q \,\|\, i_{lo}^p = i_{lo}^q$
  Fetch kernel values from GPU memory
  Else
  Compute $K_{i_{lo}^k, i_{lo}^k}$, $K_{i_{up}^k, i_{up}^k}$, $K_{i_{up}^k, i_{lo}^k}$ (device)
  End if
  Update $\alpha_{i_{up}^k}$, $\alpha_{i_{lo}^k}$ (device)
  Compute $b_{up}^{k,p}$, $b_{lo}^{k,p}$, $i_{up}^{k,p}$, $i_{lo}^{k,p}$ (device)
  Compute b_{up}^k, b_{lo}^k, i_{up}^k, i_{lo}^k (host)
End While
Use α_i^k to test the testing set
Return $C = C[C_1, \ldots, C_k]$

Algorithm 4
Pseudocode of the proposed SVM-SMO-SGD algorithm

N = training set size
MaxPasses: max number to iterate without changing alphas
Alpha = an array size of the training set
Count=0
While (Count < MaxPasses):
  Count += 1
  $alpha_{old}$ = copy alpha's elements
  $K_{x_i} K_{x_i} = kernel(point[x_i], point[x_i])$
  $K_{x_j} K_{x_j} = kernel(point[x_j], point[x_j])$
  $K_{x_i} K_{x_j} = kernel(point[x_i], point[x_j])$
  $K_{i,j}$ = compute $K_{x_i} K_{x_i}$, $K_{x_j} K_{x_j}$, $K_{x_i} K_{x_j}$
  If y_i ! = y_j then
  L=max(0, $alpha_j - alpha_i$)
  H=min(C, $C - alpha_i + alpha_j$)
  Else
  L=max(0, $alpha_i + alpha_j - C$)
  H=min(C, $alpha_i + alpha_j$)
  End if
  w = compute weights via SGD in Algorithm 2
  $b = \overline{y - w, x}$
  $E_i = sign(w, x_i + b) - y_i$
  $E_j = sign(w, x_j + b) - y_j$
  Update $alpha_j$
  Update $alpha_i$
End While
Update b
Return number of support vectors, count

Algorithm 5
Pseudocode of the proposed SVM-SMO-SGD algorithm on CUDA

N = training set size
MaxPasses: max number to iterate without changing alphas
Alpha = an array size of the training set (send to device)
Count=0 (device)
While (Count < MaxPasses): (this loop evaluated on device)
Count += 1
  $alpha_{old}$ = copy alpha's elements
  $K_{x_i} K_{x_i} = kernel(point[x_i], point[x_i])$
  $K_{x_j} K_{x_j} = kernel(point[x_j], point[x_j])$
  $K_{x_i} K_{x_j} = kernel(point[x_i], point[x_j])$
  $K_{i,j}$ = compute $K_{x_i} K_{x_i}$, $K_{x_j} K_{x_j}$, $K_{x_i} K_{x_j}$
  If y_i ! = y_j then
  L=max(0, $alpha_j - alpha_i$)
  H=min(C, $C - alpha_i + alpha_j$)
  Else
  L=max(0, $alpha_i + alpha_j - C$)
  H=min(C, $alpha_i + alpha_j$)
  End if
  w = compute weights via SGD in Algorithm 2
  $b = \overline{y - w, x}$
  $E_i = sign(w, x_i + b) - y_i$
  $E_j = sign(w, x_j + b) - y_j$
  Update $alpha_j$
  Update $alpha_i$
End While
Update b (device)
Return number of support vectors, count (send support vectors and number of count to host from the device)

- Since the PCV algorithm was run on CUDA, better results are obtained in terms of time and RAM. As the PCV algorithm's data size increased, the GPU's RAM consumption also increased proportionally, but there was no significant change in the RAM consumption on the CPU. The PCV algorithm kept the CPU consumption at the optimum level due to the usage of the LRU cache.
- In the proposed SVM-SMO-SGD algorithm, the results are promising in both sequential and parallel versions. The parallel SVM-SMO-SGD algorithm consumed the same amount of RAM on the GPU in each dataset. Because the SGD algorithm only operates on one sample in each iteration, the RAM consumption on the GPU remained stable

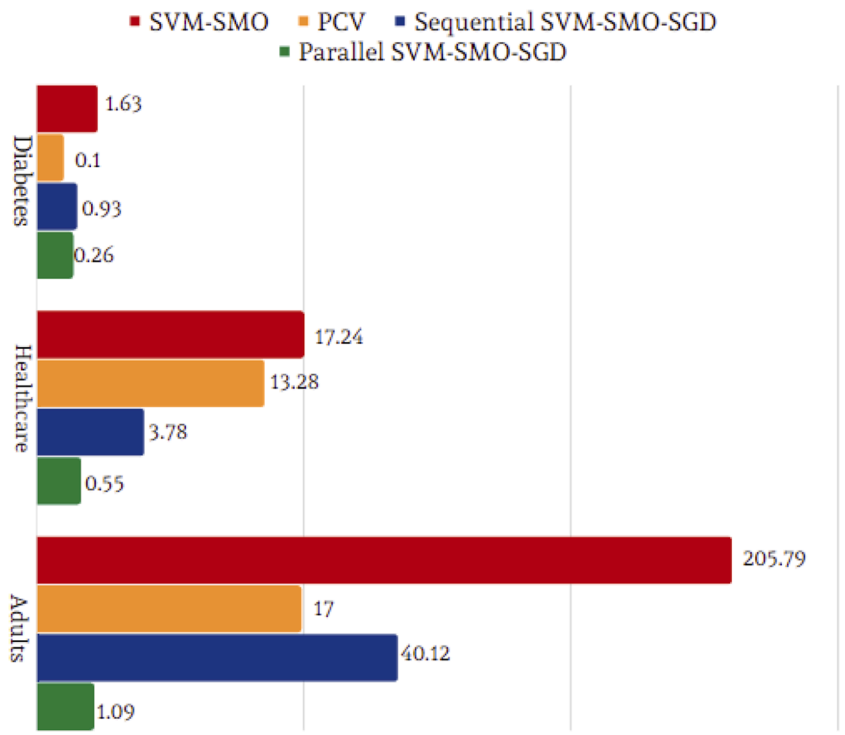

Fig. 4. Time Consumption of the Algorithms (sec).

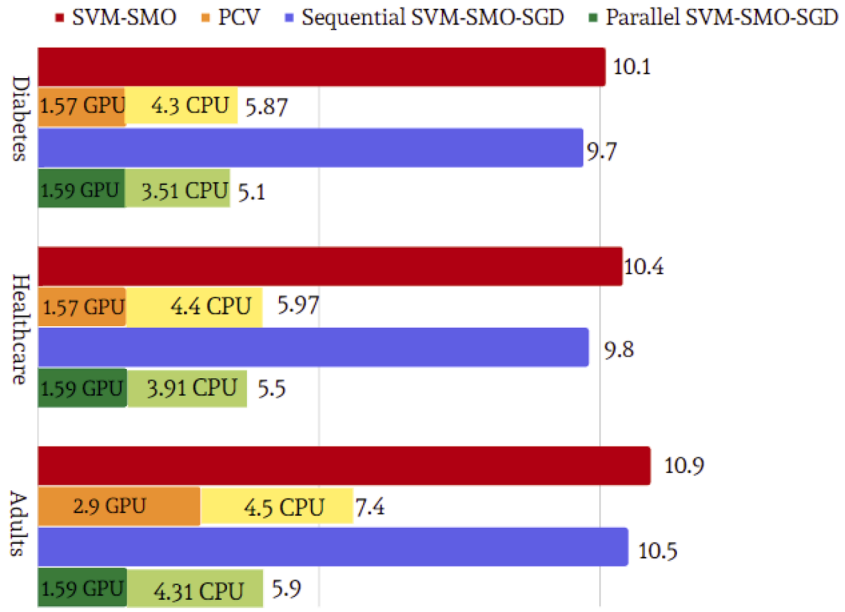

Fig. 5. RAM Consumption of the Algorithms (%).

even if the dataset grows. The increase in the amount of CPU-RAM usage resulted from the sending operation of calculated values to the CPU for accuracy calculation.

- While sequential SVM-SMO-SGD and PCV algorithms yielded very close results in terms of time consumption, close results were obtained with Platt's SMO algorithm in support vector generation. In the hybrid algorithm, slightly more support vectors are produced than the standard SMO due to the wheel of the SGD algorithm until it reaches the global minimum. The SVM-SMO-SGD algorithm's main contribution is that it produces the same accuracy value in a much shorter time than other algorithms with which it is compared.

Compared with the SMO algorithm in the diabetes dataset,

- The PCV algorithm is 16.3 times more efficient in time, and 1.72 times in RAM consumption. While the PCV algorithm produced 510 support vectors, the SMO algorithm produced 231 support vectors.
- The Sequential SVM-SMO-SGD algorithm is 1.75 times more efficient in time, and 1.04 times in RAM consumption. While 217 support vectors are produced in this hybrid algorithm, SMO produced 231. Since it is an SMO-based algorithm, we accept this similarity as usual.

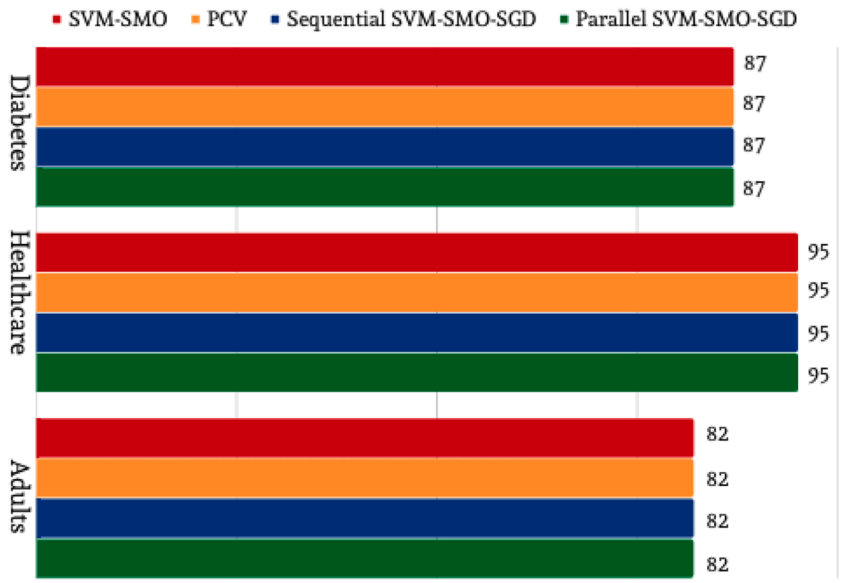

Fig. 6. Accuracy Scores of the Algorithms

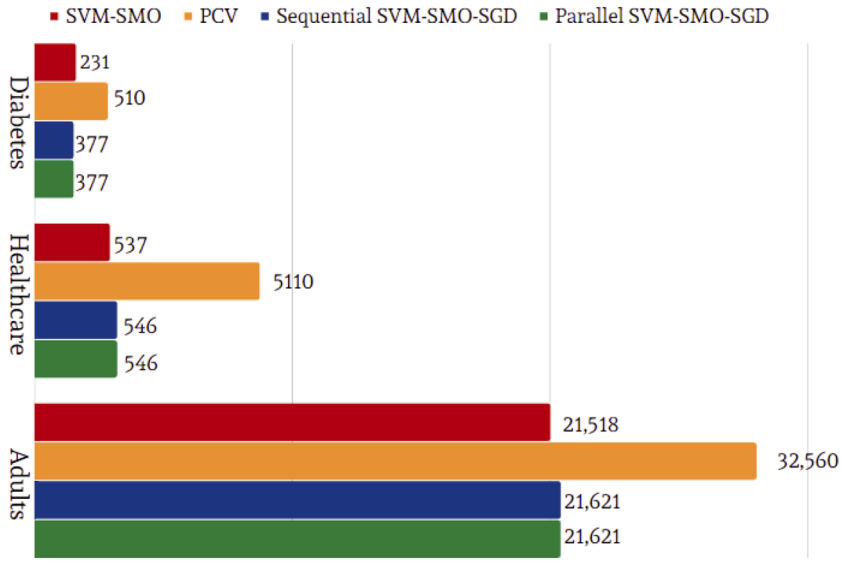

Fig. 7. Number of support vectors produced by algorithms.

- The Parallel SVM-SMO-SGD algorithm is 6.27 times more efficient in time, and 1.98 times in RAM consumption. Running sequentially or in parallel did not affect the number of support vectors produced.

Compared to the SMO algorithm in the Healthcare Stroke Prediction dataset,

- The PCV algorithm is 1.28 times more efficient in time, and 1.74 times in RAM consumption. While the PCV algorithm produced 5,110 support vectors, the SMO algorithm produced 537 support vectors.
- The Sequential SVM-SMO-SGD algorithm is 4.56 times more efficient in time, and 1.06 times in RAM consumption. While 546 support vectors are produced in this hybrid algorithm, SMO produced 537.
- The Parallel SVM-SMO-SGD algorithm is 31.35 times more efficient in time, and 1.89 times more efficient in RAM consumption. Running sequentially or in parallel did not affect the number of support vectors produced.

Compared to the SMO algorithm in the Adults dataset,

- The PCV algorithm is 12.10 times more efficient in time, and 1.47 times more efficient in RAM consumption. While the PCV algorithm produced 32,560 support vectors, the SMO algorithm produced 21,518 support vectors.
- The Sequential SVM-SMO-SGD algorithm is 5.12 times more efficient in time, and 1.03 times in RAM consumption. While 21,621 support vectors are produced in this hybrid algorithm, SMO produced 21,518.
- The Parallel SVM-SMO-SGD algorithm is 188.80 times more efficient in time, and 1.84 times in RAM consumption. Running sequentially or in parallel did not affect the number of support vectors produced.

As a result, the algorithms used in our study yielded the same accuracy in classifying the datasets.

4. Conclusion

As the data set grows, RAM, and especially time consumption of the SVM algorithm increases in direct proportion. The proposed SMO algorithm alone is not sufficient to reduce this workload. This study aims to reduce time, and RAM consumption by hybridizing SVM and existing SMO and SGD algorithms in two different versions, sequential and parallel, to reduce the workload of the SVM algorithm. As far as we know, there is no serial and parallel hybrid version of these three algorithms. The hybrid SVM-SMO-SGD algorithm we designed performs similarly to the PCV algorithm we compared in terms of time and RAM consumption, even in its serial version. However, according to classical serial algorithms, it is very promising in terms of time, and RAM consumption as the data set grows. The performance of the hybrid SVM-SMO-SGD algorithm is also very talented in terms of RAM, and especially time consumption as the data set grows.

As a result, while the SMO algorithm can be helpful for medium-sized datasets, the SVM-SMO-SGD algorithm can be used in both serial and parallel versions, even if the dataset grows. Our future study will focus on improving performance on large datasets using the mini-batch gradient descent algorithm with SVM-SMO.

Data availability

The data used in the article were obtained from well-known data sites such as UCI/Kaggle.

CRediT authorship contribution statement

Gizen Mutlu: Conceptualization, Methodology, Software, Writing – original draft, Visualization. **Çiğdem İnan Acı:** Writing – review & editing, Supervision.

Declaration of Competing Interest

The authors declare no conflicts of interest.

Data Availability

The data used in the article were obtained from well-known data sites such as UCI/Kaggle.

References

[1] M. Aci, M. Avci, Reducing simulation duration of carbon nanotube using support vector regression method, J. Faculty Eng. Arch. Gazi Univ. 3 (2010) 901–907, https://doi.org/10.17341/gazimmfd.337642.

[2] Y. Duan, B. Zou, J. Xu, F. Chen, J. Wei, Y.Y. Tang, OAA-SVM-MS: A fast and efficient multi-class classification algorithm, Neurocomputing 454 (2021) 448–460, https://doi.org/10.1016/j.neucom.2021.04.115.

[3] R. Strack, V. Kecman, B. Strack, Q. Li, Sphere support vector machines for large classification tasks, Neurocomputing 101 (4) (2013) 59–67, https://doi.org/10.1016/j.neucom.2012.07.025.

[4] J. Liu, Y. Hu, S. Yang, A SVM-based framework for fault detection in high-speed trains, Measurement 172 (2021), 108779, https://doi.org/10.1016/j.measurement.2020.108779.

[5] M. Kubat, an Introduction to Machine Learning, first ed., Springer, Cham, 2015.

[6] J. Platt, Sequential minimal optimization: a fast algorithm for training support vector machines, Microsoft Research Technical Report, (1998).

[7] I. Goodfellow, Y. Bengio, A. Courville, illustrated ed., MIT Press, 2016.

[8] A. Géron, Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems, second ed., O'Reilly Media, Inc., 2019.

[9] I. Chakroun, T. Haber, T.J. Ashby, SW-SGD: the sliding window stochastic gradient descent algorithm, Procedia Comput. Sci. 108 (2017) 2318–2322, https://doi.org/10.1016/j.procs.2017.05.082.

[10] Q. Li, R. Salman, E. Test, R. Strack, V. Kecman, Parallel multitask cross validation for support vector machine using GPU, J. Parallel Distrib. Comput. 73 (3) (2013) 293–302, https://doi.org/10.1016/j.jpdc.2012.02.011.

[11] R. Díaz-Morales, A. Navia-Vázquez, Efficient parallel implementation of kernel methods, Neurocomputing 191 (2016) 175–186, https://doi.org/10.1016/j.neucom.2015.11.097.

[12] B. Gu, Y. Shan, X. Quan, G. Zheng, Accelerating Sequential Minimal Optimization via Stochastic Subgradient Descent, IEEE Trans. Cybern. 51 (4) (2021) 2215–2223, https://doi.org/10.1109/TCYB.2019.2893289.

[13] K. Sopyła, P. Drozda, Stochastic gradient descent with Barzilai–Borwein update step for SVM, Inf. Sci. 316 (2015) 218–233, https://doi.org/10.1016/j.ins.2015.03.073.

[14] Y. You, H. Fu, S.L. Song, A. Randles, D. Kerbyson, A. Marquez, G. Yang, A. Hoisie, Scaling Support Vector Machines on modern HPC platforms, J. Parallel Distrib. Comput. 76 (2015) 16–31, https://doi.org/10.1016/J.JPDC.2014.09.005.

[15] Z. Wen, J. Shi, Q. Li, B. He, J. Chen, ThunderSVM: a fast SVM Library on GPUs and CPUs, J. Mach. Learn. Res. 19 (1) (2018) 1–5, https://doi.org/10.5555/3291125.3291146.

[16] M. Nandan, P.P. Khargonekar, S.S. Talathi, Fast SVM Training Using Approximate Extreme Points, J. Mach. Learn. Res. 15 (1) (2014) 59–98, https://doi.org/10.5555/2627435.2627437.

[17] M. Baldomero-Naranjo, L.I. Martínez-Merino, A.M. Rodríguez-Chía, A robust SVM-based approach with feature selection and outliers detection for classification problems, Expert Syst. Appl. 178 (2021), 115017, https://doi.org/10.1016/J.ESWA.2021.115017.

[18] R. Kashef, A boosted SVM classifier trained by incremental learning and decremental unlearning approach, Expert Syst. Appl. 167 (2021), 114154, https://doi.org/10.1016/J.ESWA.2020.114154.

[19] J. Wang, H. Wang, C. Zhao, J. Li, H. Gao, Iteration acceleration for distributed learning systems, Parallel Comput. 72 (2018) 29–41, https://doi.org/10.1016/J.PARCO.2018.01.001.

[20] Q. Tong, G. Liang, X. Cai, C. Zhu, J. Bi, Asynchronous parallel stochastic Quasi-Newton methods, Parallel Comput. 101 (2021), 102721, https://doi.org/10.1016/J.PARCO.2020.102721.

[21] C. Blake, E. Keogh, and C.J. Merz, UCI machine learning repository, 1998, [Online]. Available: https://archive.ics.uci.edu/ml.

[22] F.S.P., Kaggle, 2021, [Online]. Available: https://www.kaggle.com/datasets.

[23] R. Kohavi, B. Becker, UCI machine learning repository, 1996, [Online]. Available: https://archive.ics.uci.edu/ml.

[24] J. Han, B. Sharma, Learn CUDA Programming: a Beginner's Guide to GPU Programming and Parallel Computing with CUDA 10. x and C/C++, first ed., Packt Publishing, 2019.

[25] J. Sanders, E. Kandrot, CUDA by Example: An Introduction to General-Purpose GPU Programming, first ed., Addison-Wesley Professional, 2010.

[26] C. Cortes, V. Vapnik, Support-vector networks, Mach. Learn. 20 (3) (1995) 273–297, https://doi.org/10.1007/BF00994018.

[27] A. Sharma, Guided stochastic gradient descent algorithm for inconsistent datasets, Appl. Soft Comput. 73 (2018) 1068–1080, https://doi.org/10.1016/j.asoc.2018.09.038.

[28] S. Shalev-Shwartz, S. Ben-David, Understanding Machine Learning: From Theory to Algorithms, Cambridge University Press, Cambridge, 2014.

[29] L. Bottou, Stochastic gradient descent tricks. Neural networks: Tricks of the trade, second ed., Springer, Berlin, Heidelberg, 2012, pp. 421–436.

[30] B.G. Galuzzi, I. Giordani, A. Candelieri, R. Perego, F. Archetti, Hyperparameter optimization for recommender systems through Bayesian optimization, Comput. Manag. Sci. 17 (4) (2020) 495–515, https://doi.org/10.1007/s10287-020-00376-3.

[31] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, A. Fasih, PyCuda and PyOpenCL: A scripting-based approach to GPU run-time code generation, Parallel Comput. 38 (3) (2012) 157–174, https://doi.org/10.1016/j.parco.2011.09.001.