Linear Support Vector Machines via Dual Cached Loops

Shin Matsushima*
Information Science and Technology
The University of Tokyo, Tokyo
masin@r.dl.itc.u-tokyo.ac.jp

S.V. N. Vishwanathan
Statistics & Computer Science
Purdue, West Lafayette IN
vishy@stat.purdue.edu

Alexander J. Smola
Yahoo! Research
Santa Clara CA
alex@smola.org

ABSTRACT

Modern computer hardware offers an elaborate hierarchy of storage subsystems with different speeds, capacities, and costs associated with them. Furthermore, processors are now inherently parallel offering the execution of several diverse threads simultaneously. This paper proposes StreamSVM, the first algorithm for training linear Support Vector Machines (SVMs) which takes advantage of these properties by integrating caching with optimization. StreamSVM works by performing updates in the dual, thus obviating the need to rebalance frequently visited examples. Furthermore we trade off file I/O with data expansion on the fly by generating features on demand. This significantly increases throughput. Experiments show that StreamSVM outperforms other linear SVM solvers, including the award winning work of [38], by orders of magnitude and produces more accurate solutions within a shorter amount of time.

1. INTRODUCTION

In this paper we propose StreamSVM an efficient method for large scale *single machine* linear Support Vector estimation, yielding the fastest currently available solver which runs on general purpose hardware by over an order of magnitude. To the best of our knowledge, this is the first work on training linear Support Vector Machines (SVMs) which takes explicit advantage of different storage subsystems on a computer to achieve very high data throughput. In particular, our optimization algorithm is co-designed to take advantage of the fact that memory access is much faster than disk access. Our approach is entirely generic and can be extended to a more fine-grained storage hierarchy and a larger class of convex optimization problems.

1.1 Solvers for Training SVMs

Support Vector Machines (SVMs) have arguably revolutionized machine learning over the past decade. Fueled by their impressive performance in a number of real-world applications, there have been numerous efforts to scale SVMs to large data sets. Some notable contributions include Sequential Minimal Optimization (SMO) [28, 10], SVMLight [19], LaSVM [4], and gradient projection based solvers [39].

Much of the initial success of SVMs was attributed to the so-called *kernel trick* wherein training data is implicitly mapped to a high dimensional feature space, and a margin maximizing linear classifier is learned in this mapped space. In contrast, *linear* SVMs do not employ the kernel trick explicitly.[1] As massive, high-dimensional data sets are becoming commonplace, there is a recent surge of interest in linear SVMs. Some recent papers [20, 18, 17, 35] tackle this problem with great success, and provide algorithms with convergence guarantees. The above solvers either assume that the data resides in memory or that a cluster of machines with appropriate communication and synchronization facilities are available.

In an award winning paper [38] revisited the problem of training linear SVMs when the data does not fit into memory [29, 27, 19]. In a nutshell, the key idea is to split the data into manageable blocks, compress and store each block on disk, and perform dual coordinate descent by loading each block sequentially. This basic idea was improved upon by [11] who observed that the block minimization (BM) algorithm of [38] does not retain important points before discarding each block. They therefore, propose to retain some important points from the previous blocks in the RAM. This simple idea leads to significant speed-ups, and [11] demonstrate that their selective block minimization (SBM) algorithm outperforms BM. Here we propose a new algorithm StreamSVM, and show that it outperforms both BM and SBM on a number of publicly available datasets.

1.2 Exploiting the Storage Hierarchy

StreamSVM takes advantage of the different characteristics inherent in the storage hierarchy of modern computers. That is, while hard disks excel at storing large amounts of data, they have typically mediocre data transfer rates and are outright slow at random access operations. Compared to that, main memory comes at a hundredfold premium in terms of space but offers two to three orders of magnitude faster data transfer rates. CPU caches are yet faster again. Similar considerations hold for solid state drives, PCI interconnects, and graphics subsystems.

This suggests that algorithms which require streaming

*The research was performed when visiting Purdue.

[1]They may, however, use the kernel trick implicitly by hashing and direct feature expansion as described in [31].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
KDD 2012 Beijing, China
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$10.00.

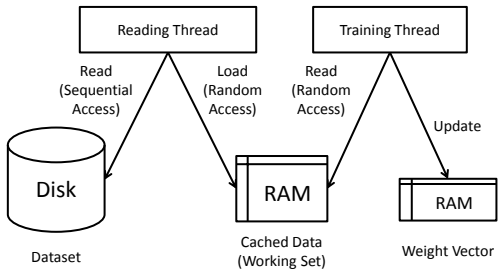


Figure 1: Basic data flow diagram of the optimization algorithm. A reader keeps on filling the main memory from disk while the optimization thread(s) perform updates on the data available in RAM. Both threads operate asynchronously.

through data from disk should take advantage of the data they already have in main memory while waiting for more data to arrive from disk. Obviously, the same rationale applies to a sequence of storage systems with different capacity/bandwidth characteristics. To make things somewhat more explicit we list a range of such systems below:

System	Capacity	Bandwidth	IOP/s
Disk	3TB	150MB/s	10^2
SSD	256GB	500MB/s	$5 \cdot 10^4$
RAM	16GB	30GB/s	10^8
Cache	16MB	100GB/s	10^9

In the present paper we focus on two parts of this hierarchy — disk and memory (see Figure 1). This already affords quite dramatic improvements in terms of speed relative to sequential algorithms. In a nutshell our algorithm does the following:

> *Iterate over the data in main memory while streaming data from disk. Evict primarily examples from main memory that are "uninformative".*

A naive approach which takes, e.g. stochastic gradient descent steps based on the importance of examples would likely run afoul of data weighting problems — informative examples need not have extended statistical weight but rather only a higher influence on the choice of objective function. One way of dealing with this issue is to resort to dual updates. That is, we consider the dual optimization problem to SVMs (and related problems) and judiciously update the associated Lagrange multipliers. This leads to the following algorithm:

Reader
while not converged **do**
  read example (x, y) from disk
  if buffer full **then** evict random (x', y') from memory
  insert new (x, y) into ring buffer in memory
end while

Trainer
while not converged **do**
  randomly pick example (x, y) from memory
  update dual parameter α
  update weight vector w
  if deemed to be uninformative **then** evict (x, y) from memory
end while

1.3 Enterprise Scale Solvers

To place our research in perspective note that industrial datasets regularly exceed the capacities offered by single computers in terms of both storage and computation. This means that distributed inference techniques are required. Unfortunately, large server centers often come with rather severe restrictions on reliability, inter-machine latency, communication trees, delays, etc. such that it is desirable to find algorithms which compute estimates using a bare minimum of communication. Note that not all estimation problems are amenable to efficient high-latency scenarios. For instance, latent variable models typically require excellent communication and great care needs to be taken to obtain fast estimates [2, 32]. A common attribute of these models is that they have a large degree of symmetry and non-convexity in their parametrization.

Fortunately much of what is commonly known as generalized linear models can be addressed with convex solvers. That is, the estimation problems can be decomposed efficiently into parts which are guaranteed to yield very similar solutions. In statistical learning terminology this is known as *stability* of the solution space and there exists a rich body of research [7, 36, 9, 16] extolling the desirable theoretical properties of convexly penalized estimation problems. It is therefore natural to take advantage of these properties in terms of implementations. For instance, [40] show that it is possible to perform stochastic gradient descent on individual processors *independently*, using random sub-samples of the data and to average the parameter estimates afterwards and *simultaneously* reaping the benefits of parallelization. Note that in previous work [25] suggested a similar averaging strategy, however their theoretical analysis only showed that averaging does not hurt, rather than actually accelerate convergence. It is in the spirit of [40] that we approach the problem of estimation:

1. Decompose (possibly with oversampling) the data for several machines.
2. *Solve the estimation problem per machine as efficiently as possible.*
3. Average the solutions between machines to obtain a final estimate.

Much work in the analysis of [40] was invested into proving that the stochastic gradient descent solutions on subsets are sufficiently independent for averaging to be actually beneficial. If we treat the optimization step as a standard batch problem this obstacle disappears. In this case we can appeal directly to the asymptotic analysis of [26] to see that averaging is beneficial: in particular, [26] show that the parameter distribution of a penalized empirical risk minimizer $w^*|X, Y$ conditioned on some data X, Y is asymptotically normal. This means that if we obtain such estimates based on various subsets of data via $w^*|X_i, Y_i$, we will be able to aggregate this to an improved joint estimate via $\frac{1}{n}\sum_{i=1}^{n} w^*|X_i, Y_i$.

Consequently, in the present paper we are primarily concerned with step 2 of the above approach — to find the most efficient way of solving a convex optimization problem on a single machine. While we primarily focus on linear SVMs in this paper, our ideas are fairly generic and can be applied to other convex losses including losses used in structured prediction [3].

Outline. We will briefly review dual descent algorithms for linear Support Vector Machines in Section 2. Subse-

quently Section 3 gives a detailed description of the nested loop used in traversing through data in core memory and streaming from disk. Experimental results are provided in Section 4 and we conclude with a discussion in Section 5.

2. DUAL COORDINATE DESCENT

2.1 Support Vector Classification

In the following we assume that we are given n examples $x_i \in \mathcal{X}$ and labels $y_i \in \{\pm 1\}$ drawn independently and identically from some distribution $(x_i, y_i) \sim p(x, y)$. It is our goal to find some function $f : \mathcal{X} \to \mathbb{R}$ which minimizes the misclassification error, e.g. by minimizing the probability that $yf(x) \leq 0$. This constitutes the most basic of all estimation problems, namely that of binary classification. We simplify things further by assuming that $\mathcal{X} = \mathbb{R}^d$. This assumption will be relaxed subsequently when we discuss how to expand features on the fly. The primal formulation of a linear SVM can be written as follows[2] [13]:

$$\underset{w \in \mathbb{R}^d}{\text{minimize}} \frac{1}{2} \|w\|^2 + C \sum_{i=1}^{n} \max\{0, 1 - w^\top y_i x_i\} \tag{1}$$

Using standard convex optimization tools [8] the above problem can be rewritten in its dual form

$$\underset{\alpha}{\text{minimize}}\ D(\alpha) := \frac{1}{2}\alpha^\top Q \alpha - \alpha^\top \mathbf{1} \tag{2a}$$

$$\text{subject to } \mathbf{0} \leq \alpha \leq C\mathbf{1}. \tag{2b}$$

Here, Q is an $n \times n$ matrix whose entries are given by $Q_{ij} = y_i y_j x_i^\top x_j$, and $\mathbf{1}$ is the vector of all ones. The minimizer w^* of (1) and the minimizer α^* of (2) are related by the primal dual connection: $w^* = \sum_{i=1}^{n} \alpha_i^* y_i x_i$. The dual problem (2) is a Quadratic Program (QP) with box constraints, and the i-th coordinate α_i corresponds to the i-th example (x_i, y_i).

2.2 Dual Updates

The following coordinate descent scheme can be used to minimize the dual [18]:

- Initialize $\alpha^1 = \mathbf{0}$.
- At iteration t select coordinate i_t
- Update α^t to α^{t+1} via

$$\alpha_{i_t}^{t+1} = \underset{0 \leq \alpha_{i_t} \leq C}{\operatorname{argmin}} D(\alpha^t + (\alpha_{i_t} - \alpha_{i_t}^t)\mathbf{e}_{i_t}) \tag{3a}$$

$$\alpha_i^{t+1} = \alpha_i^t \qquad \text{if } i \neq i_t \tag{3b}$$

Here, $\mathbf{e}_i$ denotes the i-th standard basis vector. Since $D(\alpha)$ is a QP, the one-variable subproblem (3a) can be solved exactly (see [18] for details):

$$\alpha_{i_t}^{t+1} = \min\left\{\max\left\{0, \alpha_{i_t}^t - \frac{\nabla_{i_t} D(\alpha^t)}{Q_{i_t i_t}}\right\}, C\right\}. \tag{4}$$

Here, $\nabla_i D(\alpha)$ denotes the i-th coordinate of the gradient. The above updates are also closely related to implicit updates of [21, 22, 12] and the passive aggressive updates of [14]. If we maintain $w^t := \sum_{i=1}^{n} \alpha_i^t y_i x_i$, then the gradient $\nabla_{i_t} D(\alpha)$ can be computed efficiently using

$$\nabla_{i_t} D(\alpha) = \mathbf{e}_{i_t}^\top (Q\alpha - \mathbf{1}) = w^t y_{i_t} x_{i_t} - 1. \tag{5}$$

w^{t+1} is kept related to α^{t+1} by computing

$$w^{t+1} = w^t + (\alpha_i^{t+1} - \alpha_i^t) y_i x_i. \tag{6}$$

A naive choice for i_t is to traverse examples periodically, which means that $i_t = t \bmod n$. We call an iteration from α^{nk} to $\alpha^{n(k+1)}$ an *outer iteration*, contrasted with an *inner iteration* from α^t to α^{t+1}. An alternative is to randomly permute the index and to access examples in arbitrary order. In that case, $i_t = \sigma^k(t \bmod n)$ where $\sigma^k \in \mathfrak{S}_n$ is a permutation.

3. PROPOSED METHOD

We assume that the entire dataset cannot fit in RAM, therefore there must exist an $\Omega < m$, which is an upper bound for the number of training data which can be stored in the RAM. Our algorithm, StreamSVM, maintains a working set A corresponding to the indices of the data points in memory and ensures that $|A| \leq \Omega$ at all times. The major challenge is to select the active set carefully. Towards this end, we use a strategy that is inspired by shrinking.

A remarkable property of the dual problem is that we can potentially reduce the size of optimization problem by carefully choosing the subset of the data. This property was first observed by [19] and is widely used in popular solvers such as SVMLight[19], LibSVM [10], and Liblinear [17]. We also aim to utilize the property and solve the entire problem efficiently. First we look at the following fact:

FACT 1. *Let α^* be the optimal solution of (2). Define A^{in} and A^{bo} as $\{i|\alpha_i^* = 0\}$ and $\{i|\alpha_i^* = C\}$ respectively. Any optimal solution of*

$$\underset{\alpha}{\text{minimize}}\ D(\alpha)$$

$$\text{subject to } \alpha_i = \begin{cases} 0 & \text{if } i \in A^{\text{in}} \\ C & \text{if } i \in A^{\text{bo}} \end{cases}$$

is also an optimal solution of (2).

A^{in} is called the *inactive set*, A^{bo} the *bound set*, and $A^{\text{ac}} := \{i|0 < \alpha_i^* < C\}$ the *active set*. If w^* denotes the optimal primal solution corresponding to α^*, then using the KKT conditions [8] it follows that [29] $i \in A^{\text{in}}$ implies that $w^{*\top} y_i x_i > 1$. Similarly, $w^{*\top} y_i x_i < 1$ for all $i \in A^{\text{bo}}$. Intuitively, what this means is that data which is in the inactive set are well classified and the points in the bound set are not. Usually A^{ac} is a small fraction of the original dataset, and therefore it is advantageous to focus most of the attention of the solver on identifying the coordinates in A^{ac} and optimizing them.

One way to identify the active set is to use projected gradients. For the bound constrained optimization problem (2) the i-th component of the projected gradient $\nabla_i^\pi D(\alpha)$ can be computed via

$$\nabla_i^\pi D(\alpha) := \begin{cases} 0 & \text{if } \alpha_i = 0 \text{ and } \nabla_i D(\alpha) > 0 \\ 0 & \text{if } \alpha_i = C \text{ and } \nabla_i D(\alpha) < 0 \\ \nabla_i D(\alpha) & \text{otherwise .} \end{cases} \tag{8}$$

[2]We omit an explicit bias term since this greatly simplifies the dual problem while remaining entirely general since the bias can be introduced easily as an additional coordinate in the data.

The shrinking heuristic then works as follows: At the end of each outer iteration we compute $M_+ = \max_i \nabla_i^\pi D(\alpha)$ and $M_- = \min_i \nabla_i^\pi D(\alpha)$. During the inner iteration, if a point is found to satisfy $\alpha_i = 0$ and $\nabla_i D(\alpha) > M_+$ or $\alpha_i = C$ and $\nabla_i D(\alpha) < M_-$ then it is shrunk, that is, the corresponding α_i is frozen. If needed, the shrunk variables are visited at a later stage of the optimization process.

Maintaining the Active Set.

Just like in shrinking, points which are very well classified or very badly classified with respect to the current solution are eliminated. However, unlike shrinking our criterion is adaptive and also takes into account the size of the RAM. Let $\alpha^1 \dots \alpha^n$ denote the intermediate solutions produced by the trainer in the last n steps. We compute

$$\varepsilon := \max_{t=1,\dots,n} |\nabla_{i_t}^\pi D(\alpha^t)|. \tag{9}$$

During the next n updates, a point is evicted from the RAM if $\alpha_i = 0$ and $\nabla_i D(\alpha) > \varepsilon$ or $\alpha_i = C$ and $\nabla_i D(\alpha) < -\varepsilon$. Furthermore, if the RAM is deemed to be full, that is, if $|A| > 0.9\Omega$ then we set $\varepsilon = \beta\varepsilon$ for some $\beta < 1$. We find that $\beta = 0.9$ works well in practice. In general, our criterion is more aggressive than shrinking, but this is not a problem; even if a point is evicted, it will subsequently be reloaded by the reader into the RAM, and the trainer will get a chance to update the corresponding α_i.

Synchronization Issues.

Detailed pseudo-code of our scheme can be found in Algorithms 1 and 2. A major advantage of our method is that there is very little need for synchronization between the reader and the trainer. The only shared object which needs to be accessed via a mutex is the set of active points A (we assume that the RAM is thread safe. see Section 4 for details). This also means that our framework is fairly modular. For instance, the reader can read data from the disk, and perform pre-processing such as computing a feature vector $\phi(x)$ without affecting the trainer.

Stopping Criterion.

If the data resides in main memory then one can use the infinity norm of the projected gradient as an effective stopping criterion. However, in our case this is not possible. Therefore, we use the following approximate scheme: Suppose the trainer performs T updates in the time that it takes the reader to perform one scan through the data. With some abuse of notation let $\alpha^1, \dots, \alpha^T$ denote the intermediate solutions produced by the trainer. Then we compute

$$M_+ := \max_{t=1,\dots,T} \nabla_{i_t}^\pi D(\alpha^t) \text{ and } M_- := \min_{t=1,\dots,T} \nabla_{i_t}^\pi D(\alpha^t), \tag{10}$$

and stop when $M_+ - m_-$ falls below a pre-specified threshold, which is set to a default value of 0.0001. Note that since T is usually much larger than n (10) is usually a more stringent stopping criterion than that used by either LibLinear, BM, or SBM.

3.1 Proof of Convergence (sketch)

Our analysis builds upon the results of [24] and [18].

DEFINITION 1 (LUO AND TSENG PROBLEM). *Consider the*

Algorithm 1 Reader

for $k = 1, \dots, \text{max\_iter}$ **do**
  for $i = 1, \dots, n$ **do**
    if $|A| = \Omega$ **then**
      randomly select $i' \in A$
      `lock` **mutex**
      $A = A \setminus \{i'\}$
      `unlock` **mutex**
      delete $y_{i'}, Q_{i'i'}, x_{i'}$ from the `RAM`
    end if
    read y_i, x_i from the `Disk`
    calculate $Q_{ii} = x_i^\top x_i$
    store y_i, Q_{ii}, x_i in the `RAM`
    `lock` **mutex**
    $A = A \cup \{i\}$
    `unlock` **mutex**
  end for
  if stopping criterion is met **then**
    exit
  end if
end for

Algorithm 2 Trainer

$\alpha^1 = \mathbf{0}, w^1 = \mathbf{0}, \varepsilon = 9, \varepsilon^{\text{new}} = 0, \beta = 0.9$
while stopping criterion is not met **do**
  for $t = 1, \dots, n$ **do**
    while $|A| = 0$ **do**
      sleep
    end while
    if $|A| > 0.9 \times \Omega$ **then**
      $\varepsilon = \beta\varepsilon$ // aggressively shrink ε
    end if
    randomly select $i \in A$
    read y_i, Q_{ii}, x_i from the `RAM`
    compute $\nabla_i D := w^{t\top} y_i x_i - 1$
    if ($\alpha_i^t = 0$ and $\nabla_i D > \varepsilon$) or ($\alpha_i^t = C$ and $\nabla_i D < -\varepsilon$) **then**
      `lock` **mutex**
      $A = A \setminus \{i\}$
      `unlock` **mutex**
      delete y_i, Q_{ii}, x_i from the `RAM`
      continue
    end if
    $\alpha_i^{t+1} = \min\{\max\{0, \alpha_i^t - \frac{\nabla_i D}{Q_{ii}}\}, C\}$
    $w^{t+1} = w^t + (\alpha_i^{t+1} - \alpha_i^t) y_i x_i$
    if $\varepsilon^{\text{new}} < |\nabla_i D|$ **then**
      $\varepsilon^{\text{new}} = |\nabla_i D|$
    end if
  end for
  Update stopping criterion
  $\varepsilon = \varepsilon^{\text{new}}$
end while

following minimization problem

$$\underset{\alpha}{\text{minimize}} \;\; g(E\alpha) + b^\top \alpha \tag{11}$$

$$\text{subject to} \;\; L_i \leq \alpha_i \leq U_i \tag{12}$$

where α and b are n-dimensional vectors, E is a $d \times n$ dimensional fixed matrix, $L_i \in [-\infty, \infty)$ and $U_i \in (-\infty, \infty]$ are upper and lower bounds respectively. The above optimization problem is a Luo and Tseng problem if the following conditions hold:

1. *E has no zero columns*
2. *the set of optimal solutions $\mathcal{A}$ is non-empty*
3. *the function g is strictly convex and twice continuously differentiable*
4. *for all optimal solutions $\alpha^* \in \mathcal{A}$, $\nabla^2 g(E\alpha^*)$ is positive definite.*

The following result was implicitly shown in the proof of Theorem 1 in [18].

LEMMA 1. *If no data $x_i = \mathbf{0}$, then the SVM dual (2) is a Luo and Tseng problem.*

PROOF. Set $E_{ij} = y_i x_i^\top \mathbf{e}_j, b = \mathbf{1}, L_i = 0, U_i = C$ and $g(\cdot) = \frac{1}{2}\|\cdot\|^2$. E has no zero columns because of our assumption that $x_i \neq \mathbf{0}$. Since the constraint set is bounded, and g is a strictly convex function, the set of optimal solutions $\mathcal{A}$ is non-empty. Since g is twice differentiable and strongly convex, $\nabla^2 g$ is positive definite everywhere. □

DEFINITION 2 (ALMOST CYCLIC RULE). *There exists an integer $B \geq n$ such that every coordinate is iterated upon at least once every B successive iterations.*

THEOREM 2 (THEOREM 2.1 OF [24]). *Let $\{\alpha^t\}$ be a sequence of iterates generated by a coordinate descent method (3) using the almost cyclic rule. The α^t converges linearly to an element of $\mathcal{A}$.*

All that remains to establish the convergence our coordinate descent scheme is to show it satisfies the almost cyclic rule. If the trainer accesses the cached training data sequentially, and the following conditions hold:

- The trainer is at most $\kappa \geq 1$ times faster than the reading thread, that is, the trainer performs at most κ coordinate updates in the time that it takes the reader to read one training data from disk.
- A point is never evicted from the RAM unless the α_i corresponding to that point has not been updated.

Then it is easy to see that this corresponds to the almost cyclic rule with $B = \kappa n$. In practice, the trainer randomly accesses training points from the RAM, and if the RAM is full then the reader evicts a random point. Therefore, there is a very small probability that a point is evicted before the corresponding α_i is updated. However, in practice we find that this does not affect convergence.

4. EXPERIMENTS

We performed experiments on some of the largest publicly available data sets for binary classification to evaluate the performance of our algorithm, and to compare it with two existing methods namely SBM and BM. Since SBM was shown to be competitive with other methods such as Vowpal Wabbit (VW), Stochastic Gradient Descent (SGD), or Pegasos we do not explicitly compare against them here.

Datasets.

Table 1 summarizes the datasets used in our experiments. The webspam trigram dataset `webspam-t`[3], as well as the the KDD cup 2010 dataset `kddb`[4] are from the LibSVM binary data collection[5]. The `dna` and `ocr` datasets were obtained from the Pascal Large Scale Learning Workshop website [34]. For all the datasets we randomly selected 80% of the labeled data for training and the remaining 20% for testing. Besides being massive, note that our datasets cover a variety of real-life scenarios. The `dna` dataset is dense, but is heavily imbalanced in terms of the number of positive vs negative examples. The `ocr` dataset is dense and balanced while both `kddb` and `webspam-t` are very sparse but very high dimensional.

Implementation Details.

Our code is implemented in portable C++ and uses `POSIX` threads (pthreads) for multi-threaded programming. In order to cache the data efficiently we use a thread safe in-memory hash table (StashDB) provided by Kyoto Cabinet[6]. We empirically find that the use of Kyoto Cabinet reduces our memory footprint by approximately 25% compared to SBM and BM. Our program utilizes two threads namely the `reader` and `trainer` (see Algorithms 1 and 2). The reader thread, as the name implies, is tasked with reading the data from disk and storing it into the RAM. If the RAM is full then the reader randomly replaces a cached point with the new data. The trainer thread randomly selects training points in the RAM and performs updates. The trainer also deletes points from the RAM based on the criterion we discussed in Section 3. It is also responsible for computing M_+ and M_- (10). When the gap $M_+ - M_-$ is lower than the tolerance threshold or the number of iterations exceeds the maximum, then both the threads exit. Open-source code as well as all the scripts needed to reproduce our experimental results will be made available for download from `http://www.stat.purdue.edu/~vishy`.

Hardware.

All experiments were conducted on the `Rossmann` computing cluster at Purdue University[7], where each node has two 2.1 GHz 12-core AMD 6172 processors with 48 GB physical memory per node.

Experimental Setup.

We focused our study on the following two aspects: How does StreamSVM compare with SBM and BM for different values of C? How does the size of RAM Ω affect the performance of StreamSVM? We also conducted an experiment where we expanded the features on the fly to demonstrate the scalability and flexibility of our framework. Since the objective functions we are minimizing are strongly convex, all optimizers will converge the same solution (within numerical precision) and produce the same generalization performance

[3]Originally from `http://www.cc.gatech.edu/projects/doi/WebbSpamCorpus.html`.

[4]This dataset was derived from KDD CUP 2010 second problem bridge_to_algebra_2008_2009.

[5]`http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html`.

[6]`http://fallabs.com/kyotocabinet/`

[7]`http://www.rcac.purdue.edu/userinfo/resources/rossmann`.

Table 1: Summary of the datasets used in our experiments. n is the total # of examples, d is the # of features, s is the feature density (% of features that are non-zero), $n_+ : n_-$ is the ratio of the number of positive vs negative examples, Datasize is the size of the data file on disk, Ω is the number of points in the RAM, and each Blocks is the data split used for the corresponding method. M denotes a million.

dataset	n	d	$s(\%)$	$n_+ : n_-$	Datasize	Ω	SBM Blocks	BM Blocks
`ocr`	3.5 M	1156	100	0.96	45.28 GB	150,000	40	20
`dna`	50 M	800	25	3e−3	63.04 GB	700,000	60	30
`webspam-t`	0.35 M	16.61 M	0.022	1.54	20.03 GB	15,000	20	10
`kddb`	20.01 M	29.89 M	1e-4	6.18	4.75 GB	2,000,000	6	3

eventually. Therefore, what we are specifically interested in is the rate at which the objective function decreases.

4.1 Results

Varying C.

As C increases the effect of the regularizer decreases and the non-smooth hinge loss dominates the primal objective. Intuitively, this means that the problem becomes harder to solve for large values of C. Therefore, it is important to test an optimization algorithm across a range of C values.

We used $C \in \{0.0001, 0.001, \ldots, 1000.0\}$ to test the performance of StreamSVM and contrast it with SBM and BM. For this experiment we set the maximum number of iterations for all algorithms to be 100; for StreamSVM an iteration is defined as one complete pass through the data by the reader thread. The stopping tolerance was set to be 10^{-3}. A job is killed by the queue manager on the cluster if it does not finish within 48hrs. Besides the space required to store the weight vector w and the coefficients α, both algorithms were allowed to use up to 2GB of extra RAM.

In Figure 2 we used the `dna` dataset and plot the relative function value difference as function of wall clock time for SBM, BM, and StreamSVM[8] The relative function value difference is defined as $(D^* - D^t)/D^*$ where D^t is the dual objective function output by the optimizer at the t-th iteration, and D^* is the largest dual objective function value produced by either SBM, BM, or StreamSVM for the same parameter settings. Note that since we are plotting the y-axis on a log scale some points with relative function value difference of 0 are not displayed.

In the top three plots in Figure 3 we used the same value of $C = 1.0$, and plot the convergence behavior of SBM, BM, and StreamSVM on `kddb`, `ocr`, and `webspam-t` datasets respectively.

Note that in almost all cases, StreamSVM outperforms SBM and BM comprehensively. In particular, on the `dna`, `ocr` and `webspam-t` datasets, StreamSVM has converged to a high accuracy solution even before BM and SBM have managed to read and compress the data during the first pass. Also, both SBM and BM do not converge for large values of C on the `dna` dataset.

On the `kddb` dataset, StreamSVM does not exhibit any particular advantage over SBM. We investigated this further and found that the major bottleneck here is the need to frequently access random elements of w (29.89 million entries) and α (20.01 million entries), both of which are stored as dense vectors. This causes a number of cache misses and consequently slows down the trainer. On the other hand, since the data is very sparse (maximum 200 nonzero entries per training example) the reader thread is very fast. Consequently StreamSVM is not faster than SBM, although it is very competitive as compared to BM.

Varying Ω.

Next we study the effect of varying the RAM size Ω on the performance of StreamSVM. The same setup as the previous experiment was used here but with two notable changes. First, we fixed the value of C to be 1.0 for all datasets. Second, we set the RAM size to 256MB, 1GB, 4GB, and 16GB respectively.

On all datasets, the performance with 256MB RAM was inferior compared to higher RAM sizes. However, increasing the RAM size beyond 4GB does not help much. In fact, on the `kddb` dataset, increasing the size of RAM actually reduces performance. The reasons are as the same as above. Even though we can cache a large number of training points, the bottleneck here is in updating w and α. Therefore, the increase in Ω does not significantly improve convergence speed. For the `ocr` and `webspam-t` datasets there is marginal improvement when moving from 4GB to 16GB. This could potentially be explained by the fact that at the final solution there were relatively few training examples in the active set (675 for `ocr` and 1509 for `webspam-t`). We conjecture that large RAM sizes help only if the number of non-zero entries per training example are large, or of the dataset is very high dimensional and noisy because of which a large number of training examples are in the active set.

Expanding Features on the Fly.

In our final experiment, our aim is to show that explicit feature expansion can be easily incorporated into our framework. We used the `dna` dataset, and following [33] we computed a feature vector $\phi(x)$ corresponding to weighted degree kernel of degree 8. Variants of this kernel have been successfully used for various sequence analysis tasks in bioinformatics [33]. In a nutshell, to compute this kernel each training point in the `dna` dataset is represented as a string of length 200. For each position $i = 1, \ldots, 200$ we represent the sub-strings of length $j = 1, \ldots, 8$ that occur at position i. This results in a sparse feature vector of 17,476,000 dimensions.

We plot the relative function value difference vs wall clock time in the bottom plot of Figure 3. The same figure also shows how the gap is decreasing as a function of wall clock time. In less than 75 hrs StreamSVM is able to reduce the gap to approximately 0.47. To put our results in perspective, contrast this which the recent work of [1] who use a very

[8]Due to lack of space we plot only a subset of our results here. Detailed results including the generalization error and evolution of the gap can be found at `http://www.r.dl.itc.u-tokyo.ac.jp/~masin/Appendix.pdf`.

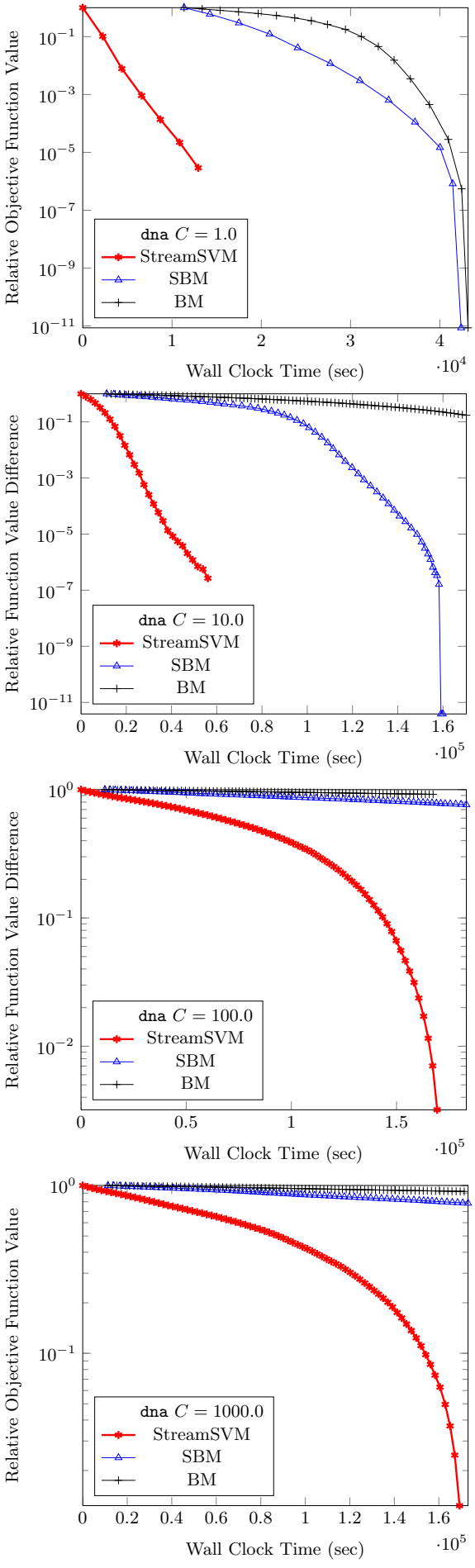


Figure 2: Relative function value difference vs wall clock time on the `dna` dataset for various values of C

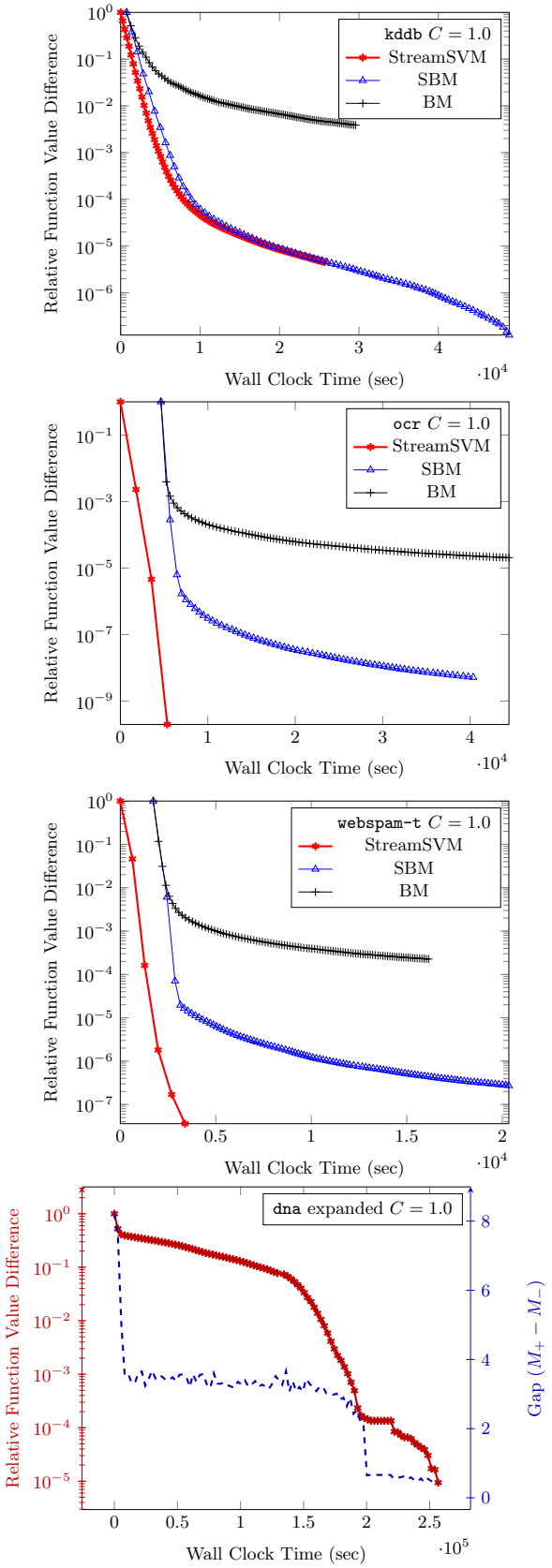


Figure 3: Top three figures: Relative function value vs wall clock time on the `kddb`, `ocr`, and `webspam-t` datasets for $C = 1.0$. The red solid curve in the bottom figure is the relative function value vs wall clock time and the blue dashed curve is the gap vs wall clock time on the `dna` dataset expanded using the weighted degree kernel of degree 8 with $C = 1.0$. Note that some points on the plot are interpolated because the server disk quota exceeded when running this experiment.

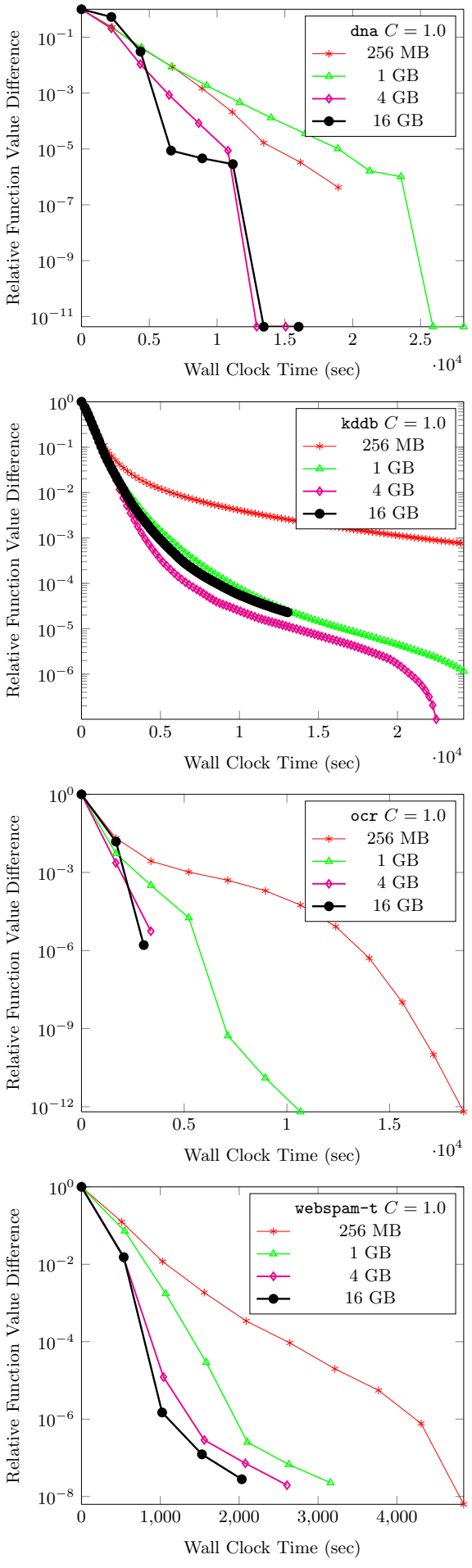


Figure 4: Relative objective function value as a function of wall clock time for various datasets as the RAM size Ω is varied.

similar (but not the same) feature representation as ours and train using LBFGS. Their algorithm requires around 30 iterations with each iteration requiring approximately 1 hr when distributed across 100 machines (total of 3000 CPU hours).

5. DISCUSSION

To conclude we would like to discuss some related work and place our contributions in perspective. Perhaps the closest in spirit to our work is the online version of SBM described by [11]. Here, data is assumed to arrive in a streaming fashion, and a dual coordinate descent procedure is used on blocks of data. However, there are some important differences between the two methods. First, online SBM only looks at the data once, while our algorithm StreamSVM performs multiple passes through the data. Second, the reading and the training in SBM happen in a synchronous fashion, while our reader thread asynchronously reads and caches data. In our experiments, StreamSVM also achieves near-optimal generalization performance after just one or two passes through the data.

In order to speed up linear VMs where the feature mapping can be computed explicitly, [33] introduced a computational framework for linear SVMs (COFFIN). Along the same lines, one can use hash functions to map sparse high-dimensional features into dense low-dimensional features. This idea, first described in [31] and improved upon in [37] is one of the important reasons why the Vowpal Wabbit learning framework[9] is fast. These techniques are very naturally incorporated into our framework as demonstrated in our experiments.

A closely related alternative to dual coordinate descent is stochastic gradient descent (SGD). Recently there have been numerous variants of which have been studied both theoretically [6, 30] as well as empirically [5]. However, SGD in its most basic form is inherently serial, and therefore there is a recent flurry of activity on developing parallel stochastic gradient descent solvers [23, 40, 15]. As explained in Section 1.3 we view our research as complimentary.

Our long term research goal is to design, analyze, and implement novel optimization algorithms that take advantage of modern hardware to enable learning on and mining of massive datasets. With the increasing availability of many-core processors, general-purpose graphics processing units, and solid-state drives, we are witnessing a hardware revolution. The next generation of systems-aware, efficient, scalable machine learning algorithms need to take advantage of these emerging computing paradigms. We view this paper as a step towards that direction.

6. REFERENCES

[1] A. Agarwal, O. Chapelle, M. Dudík, and J. Langford. A reliable effective terascale linear learning system. Technical report, 2011. `http://arxiv.org/abs/1110.4198`.

[2] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola. Scalable inference in latent variable models. In *WSDM*, 2012.

[3] G. Bakir, T. Hofmann, B. Schölkopf, A. Smola, B. Taskar, and S. V. N. Vishwanathan. *Predicting Structured Data*. 2007.

[9] `http://hunch.net/~vw/`

[4] A. Bordes, S. Ertekin, J. Weston, and L. Bottou. Fast kernel classifiers with online and active learning. *JMLR*, 6:1579–1619, September 2005.

[5] L. Bottou. Stochastic gradient SVMs. `http://leon.bottou.org/projects/sgd`, 2008.

[6] L. Bottou and O. Bousquet. The tradeoffs of large scale learning. In *NIPS 20*, 2007.

[7] O. Bousquet and A. Elisseeff. Algorithmic stability and generalization performance. In *NIPS 12*, pages 196–202, 2001.

[8] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[9] E. Candes and T. Tao. Decoding by linear programming. *IEEE Trans. Information Theory*, 51(12):4203–4215, 2005.

[10] C. C. Chang and C. J. Lin. *LIBSVM: a library for support vector machines*, 2001. `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[11] K. W. Chang and D. Roth. Selective block minimization for faster convergence of limited memory large-scale linear models. In *KDD*, pages 699–707, 2011.

[12] L. Cheng, S. V. N. Vishwanathan, D. Schuurmans, S. Wang, and T. Caelli. Implicit online learning with kernels. In *NIPS 19*, 2006.

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

[14] K. Crammer, Ofer Dekel, Joseph Keshet, Shai Shalev-Shwartz, and Y. Singer. Online passive-aggressive algorithms. *JMLR*, 7:551–585, March 2006.

[15] O. Dekel, R. Gilad-Bachrach, O. Shamir, and L. Xiao. Optimal distributed online prediction using mini-batches. Technical report, 2010. `http://arxiv.org/abs/1012.1367`.

[16] D. L. Donoho, A. Maleki, and A. Montanari. Message-passing algorithms for compressed sensing. *PNAS*, 106, 2009.

[17] R.-E. Fan, J.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. LIBLINEAR: A library for large linear classification. *JMLR*, 9:1871–1874, August 2008.

[18] C. J. Hsieh, K. W. Chang, C. J. Lin, S. S. Keerthi, and S. Sundararajan. A dual coordinate descent method for large-scale linear SVM. In *ICML*, pages 408–415, 2008.

[19] T. Joachims. Making large-scale SVM learning practical. In B. Schölkopf, C. J. C. Burges, and A. J. Smola, editors, *Advances in Kernel Methods — Support Vector Learning*, pages 169–184, 1999.

[20] T. Joachims. Training linear SVMs in linear time. In *KDD*, 2006.

[21] J. Kivinen and M. K. Warmuth. Exponentiated gradient versus gradient descent for linear predictors. *Information and Computation*, 132(1):1–64, January 1997.

[22] J. Kivinen, M. K. Warmuth, and B. Hassibi. The p-norm generaliziation of the LMS algorithm for adaptive filtering. *IEEE Trans. Signal Processing*, 54(5):1782–1793, May 2006.

[23] J. Langford, A. J. Smola, and M. Zinkevich. Slow learners are fast. `http://arxiv.org/abs/0911.0491`, 2009.

[24] Z. Q. Luo and P. Tseng. On the convergence of coordinate descent method for convex differentiable minimization. *Journal of Optimization Theory and Applications*, 72(1):7–35, 1992.

[25] G. Mann, R. McDonald, M. Mohri, N. Silberman, and D. Walker. Efficient large-scale distributed training of conditional maximum entropy models. In *NIPS 22*, pages 1231–1239, 2009.

[26] N. Murata, S. Yoshizawa, and S. Amari. Network information criterion — determining the number of hidden units for artificial neural network models. *IEEE Trans. Neural Networks*, 5:865–872, 1994.

[27] E. Osuna and F. Girosi. Reducing the run-time complexity in support vector regression. In B. Schölkopf, C. J. C. Burges, and A. J. Smola, editors, *Advances in Kernel Methods — Support Vector Learning*, pages 271–284, 1999.

[28] J. Platt. Fast training of support vector machines using sequential minimal optimization. In B. Schölkopf, C. J. C. Burges, and A. J. Smola, editors, *Advances in Kernel Methods — Support Vector Learning*, pages 185–208, 1999.

[29] B. Schölkopf and A. J. Smola. *Learning with Kernels*. MIT Press, 2002.

[30] S. Shalev-Shwartz, Y. Singer, and N. Srebro. Pegasos: Primal estimated sub-gradient solver for SVM. In *ICML*, 2007.

[31] Q. Shi, J. Petterson, G. Dror, J. Langford, A. Smola, A. Strehl, and S. V. N. Vishwanathan. Hash kernels. In *AISTATS*, 2009.

[32] A. J. Smola and S. Narayanamurthy. An architecture for parallel topic models. In *VLDB*, 2010.

[33] S. Sonnenburg and V. Franc. COFFIN: A computational framework for linear SVMs. In *ICML*, 2010.

[34] S. Sonnenburg, V. Franc, E. Yom-Tov, and M. Sebag. Pascal large scale learning challenge. 2008. `http://largescale.ml.tu-berlin.de/workshop/`.

[35] C. H. Teo, S. V. N. Vishwanthan, A. J. Smola, and Q. V. Le. Bundle methods for regularized risk minimization. *JMLR*, 11:311–365, January 2010.

[36] M. Wainwright. Sharp thresholds for noisy and high-dimensional recovery of sparsity. Technical report, Department of Statistics, UC Berkeley, 2006.

[37] K. Weinberger, A. Dasgupta, J. Attenberg, J. Langford, and A. J. Smola. Feature hashing for large scale multitask learning. In *ICML*, 2009.

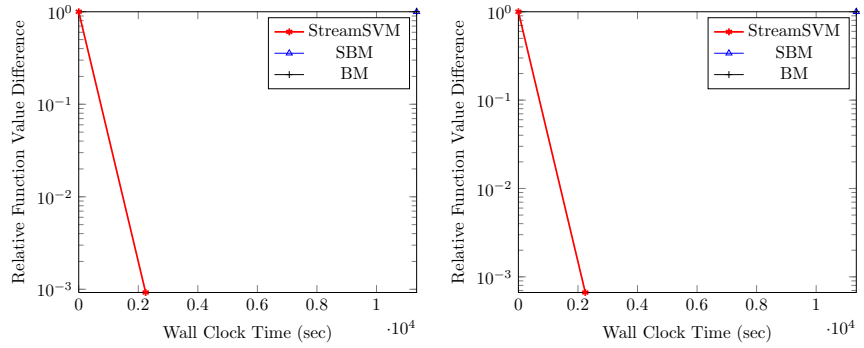
[38] H. F. Yu, C. J. Hsieh, K. W. Chang, and C. J. Lin. Large linear classification when data cannot fit in memory. In *KDD*, pages 833–842, 2010.

[39] L. Zanni, T. Serafini, and G. Zanghirati. Parallel software for training large scale support vector machines on multiprocessor systems. *JMLR*, 7:1467–1492, July 2006.

[40] M. Zinkevich, A. Smola, M. Weimer, and L. Li. Parallelized stochastic gradient descent. In *NIPS 23*, pages 2595–2603, 2010.

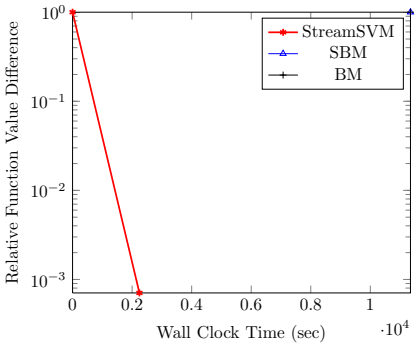
Additional Plots: Not Included in 9 Page Limit

1 Objective Function vs Wall Clock Time

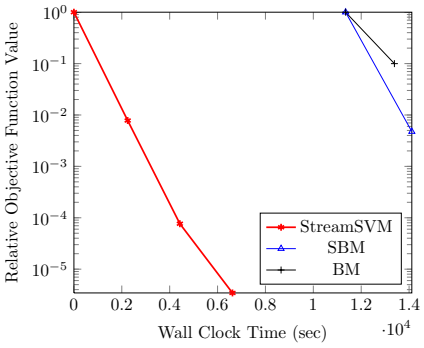


(a) $C = 0.0001$

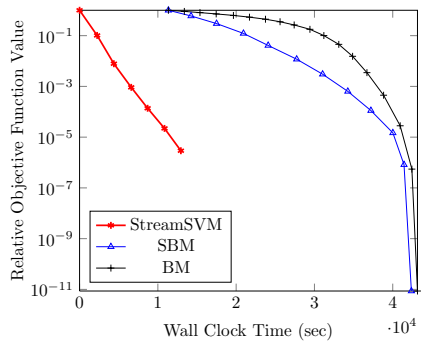
(b) $C = 0.001$



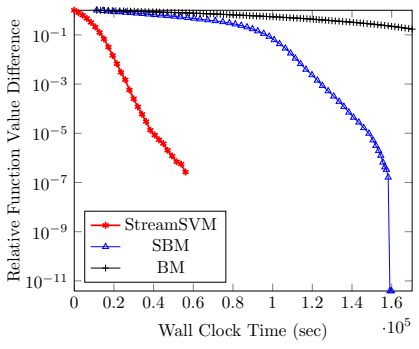
(c) $C = 0.01$



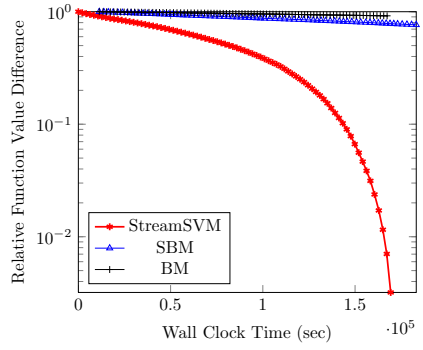
(d) $C = 0.1$



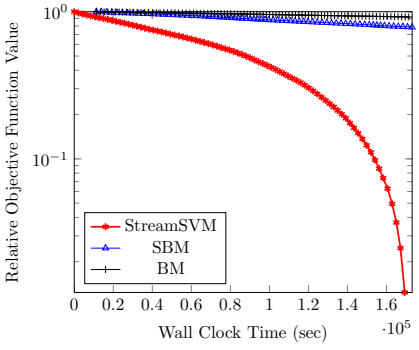
(e) $C = 1.0$



(f) $C = 10.0$

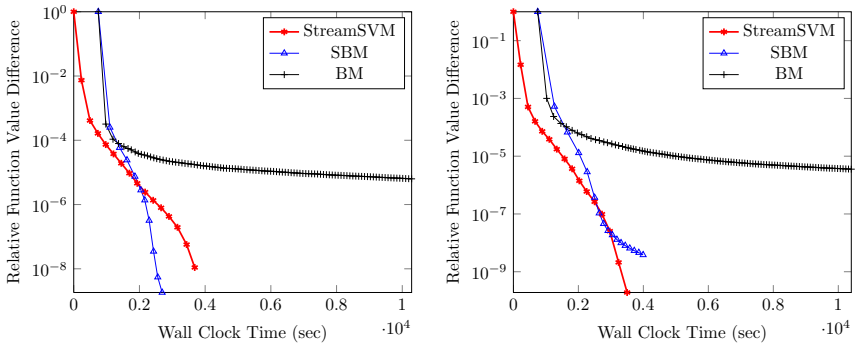


(g) $C = 100.0$

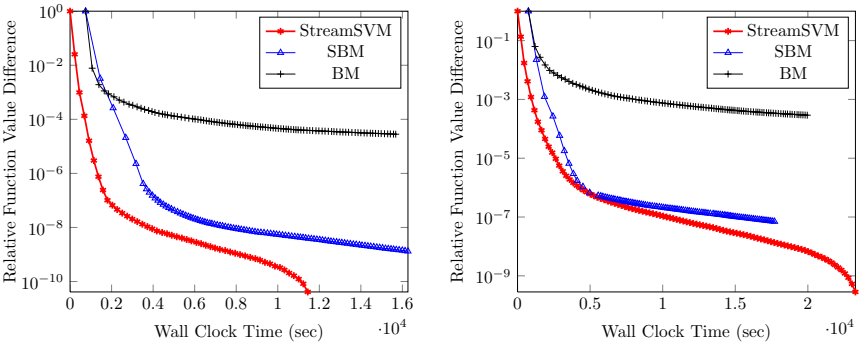


(h) $C = 1000.0$

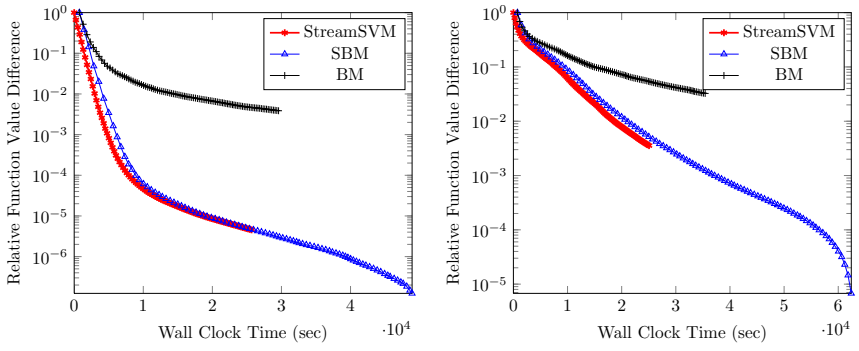
Figure 1: Relative function value difference vs wall clock time on the `dna` dataset for various values of C



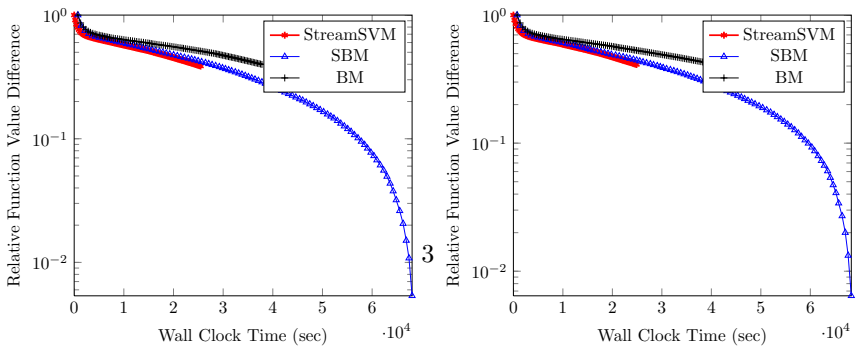
(a) $C = 0.0001$ (b) $C = 0.001$



(c) $C = 0.01$ (d) $C = 0.1$

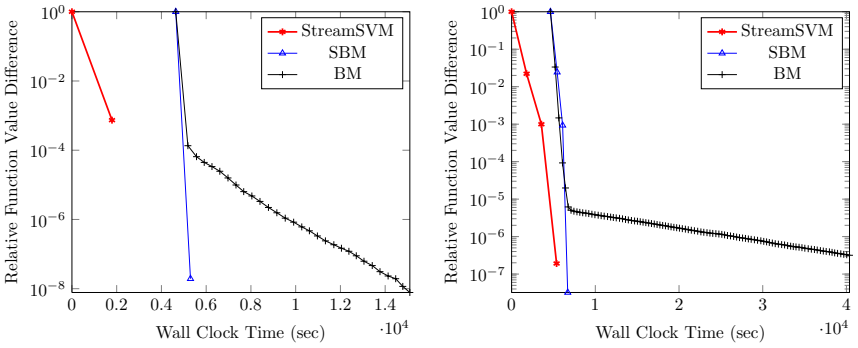


(e) $C = 1.0$ (f) $C = 10.0$



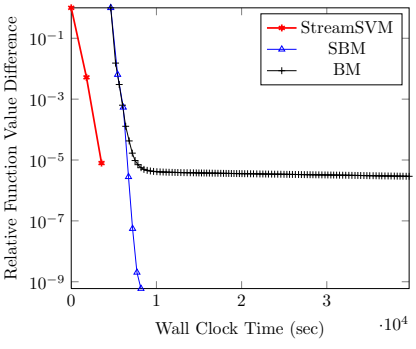
(g) $C = 100.0$ (h) $C = 1000.0$

Figure 2: Relative function value Value vs wall clock time on the `kddb` dataset for various values of C

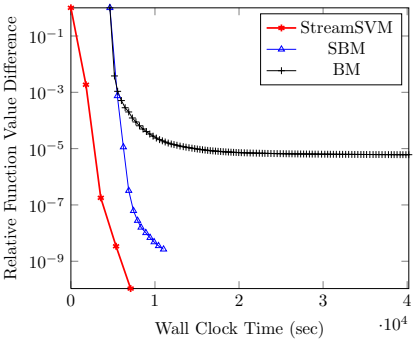


(a) $C = 0.0001$

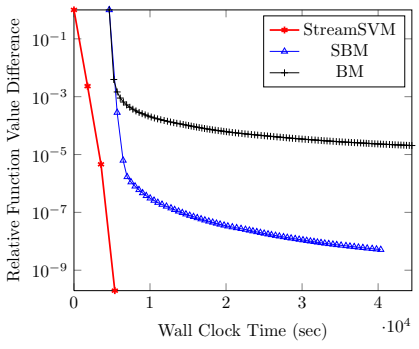
(b) $C = 0.001$



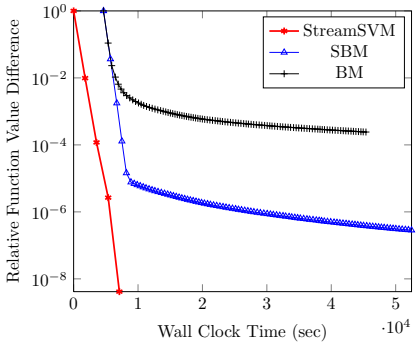
(c) $C = 0.01$



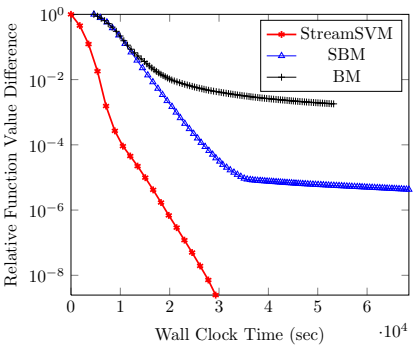
(d) $C = 0.1$



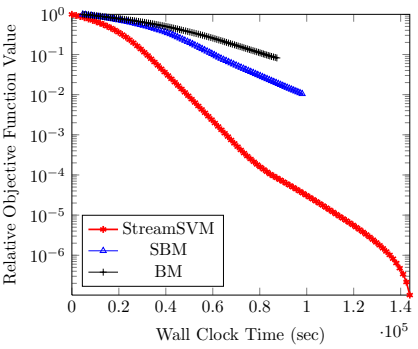
(e) $C = 1.0$



(f) $C = 10.0$

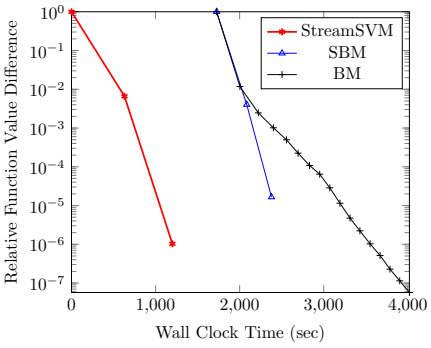


(g) $C = 100.0$

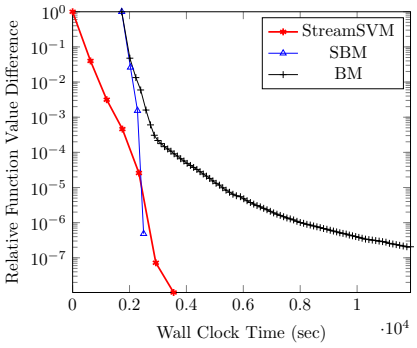


(h) $C = 1000.0$

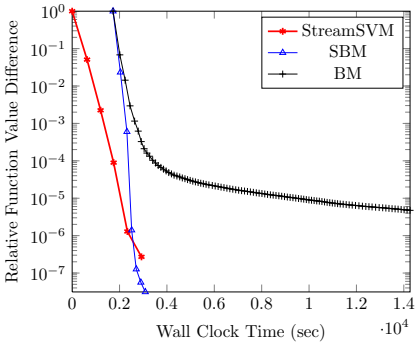
Figure 3: Relative function value difference vs wall clock time on the `ocr` dataset for various values of C



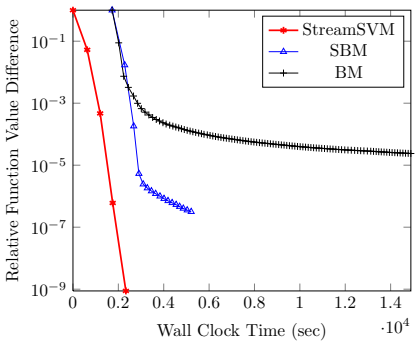
(a) $C = 0.0001$



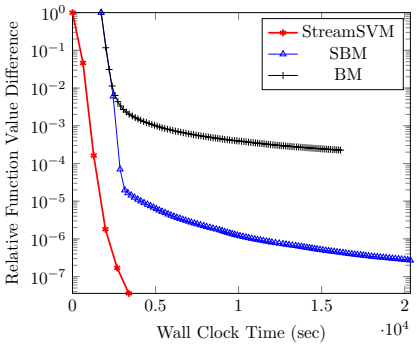
(b) $C = 0.001$



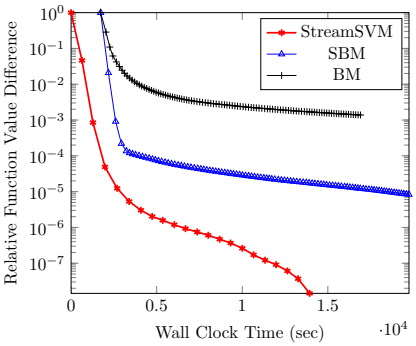
(c) $C = 0.01$



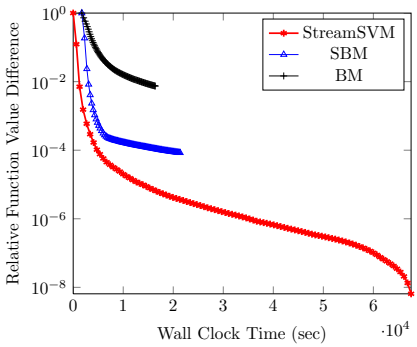
(d) $C = 0.1$



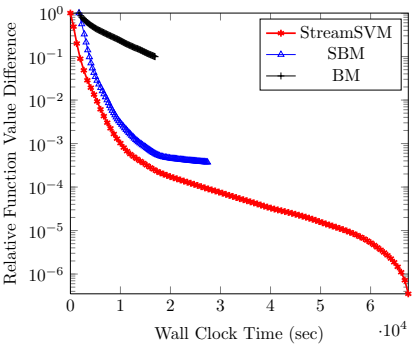
(e) $C = 1.0$



(f) $C = 10.0$



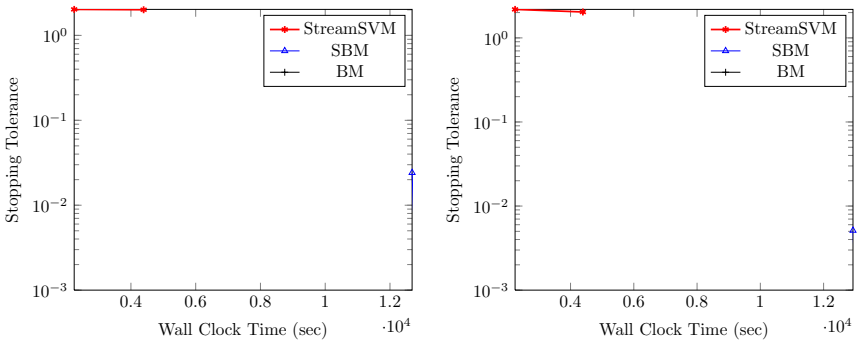
(g) $C = 100.0$



(h) $C = 1000.0$

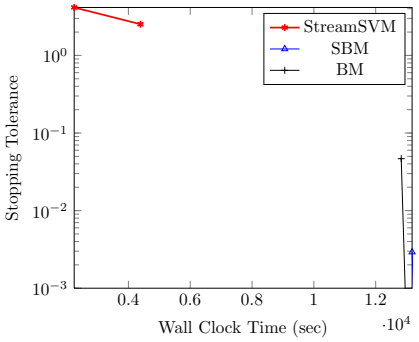
Figure 4: Relative function value difference vs wall clock time on the `webspam-t` dataset for various values of C

2 Stopping Tolerance vs Wall Clock Time

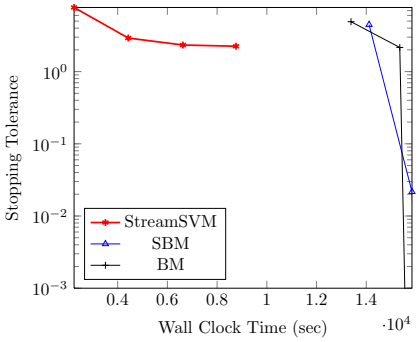


(a) $C = 0.0001$

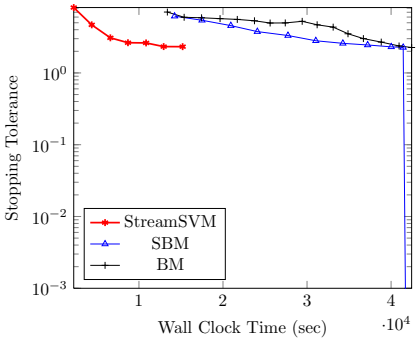
(b) $C = 0.001$



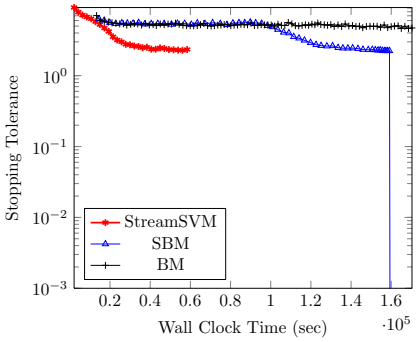
(c) $C = 0.01$



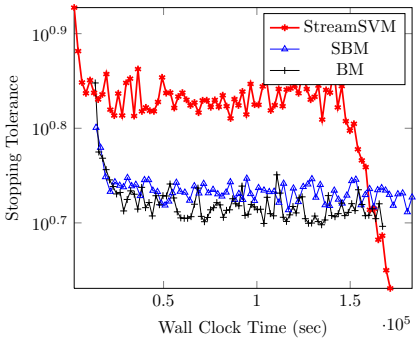
(d) $C = 0.1$



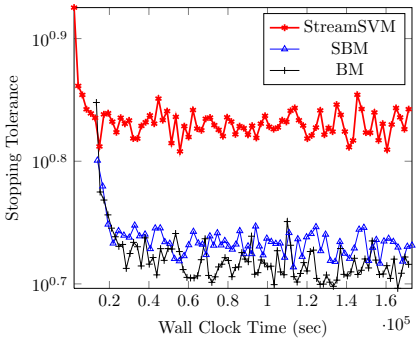
(e) $C = 1.0$



(f) $C = 10.0$

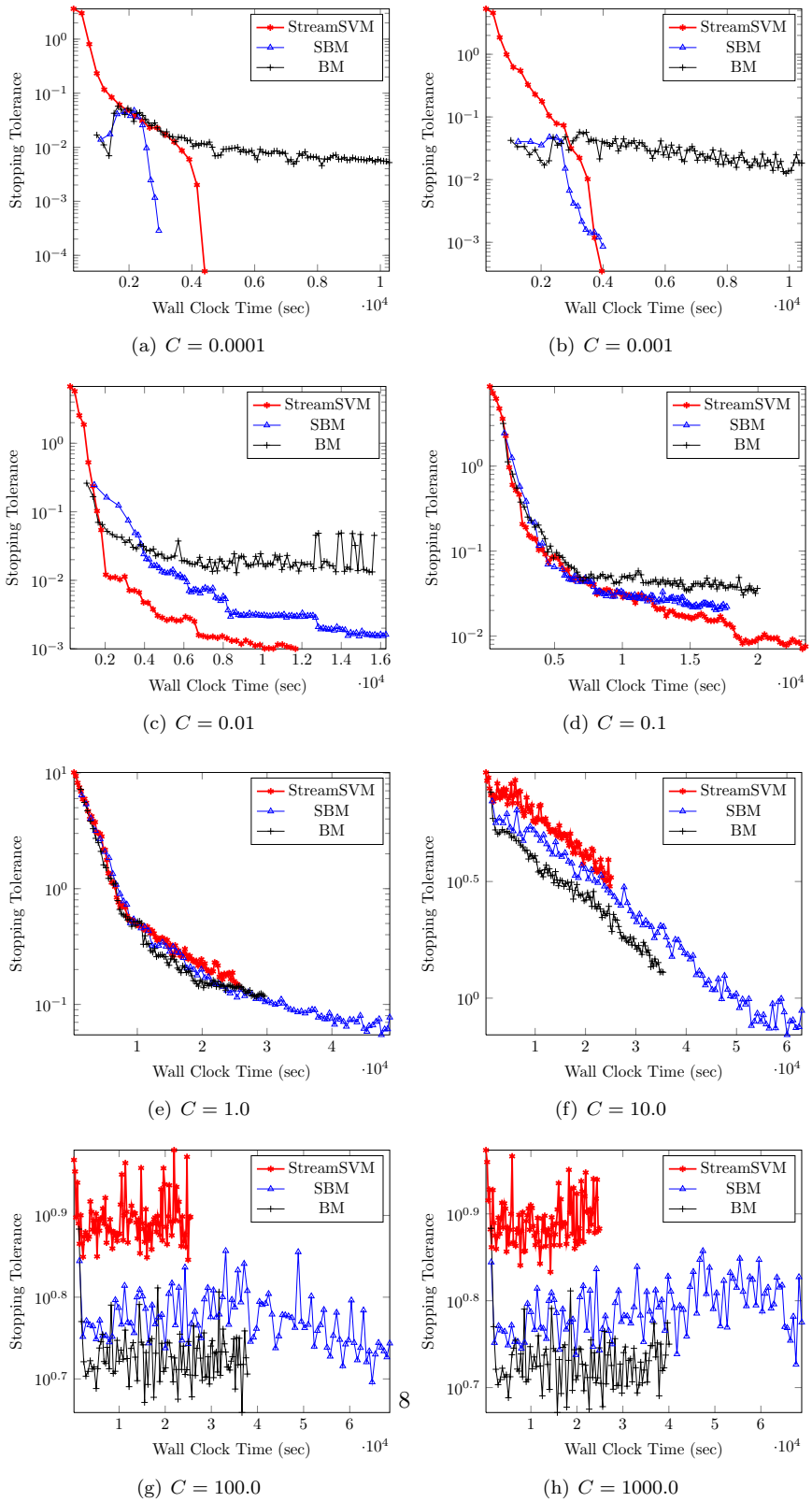


(g) $C = 100.0$



(h) $C = 1000.0$

Figure 5: Duality gap as a function of wall clock time on the `dna` dataset for various values of C



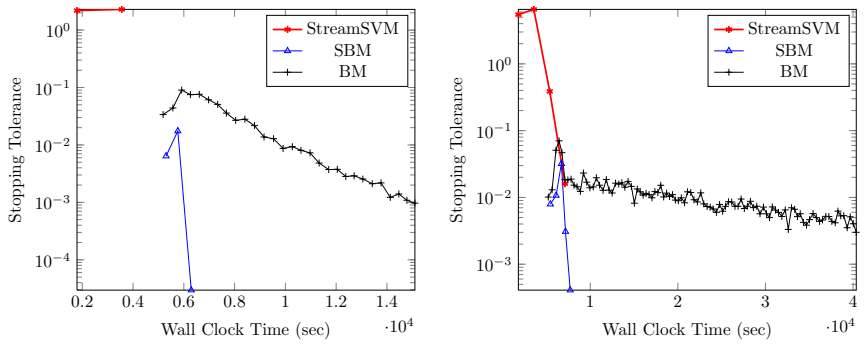
(a) $C = 0.0001$ (b) $C = 0.001$

(c) $C = 0.01$ (d) $C = 0.1$

(e) $C = 1.0$ (f) $C = 10.0$

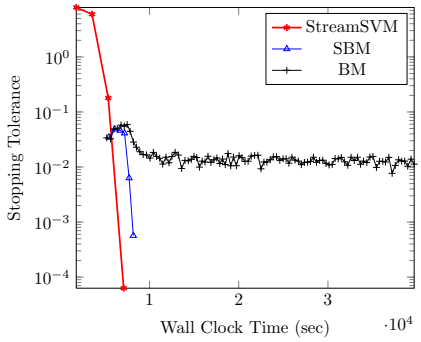
(g) $C = 100.0$ (h) $C = 1000.0$

Figure 6: Duality gap as a function of wall clock time on the `kddb` dataset for various values of C

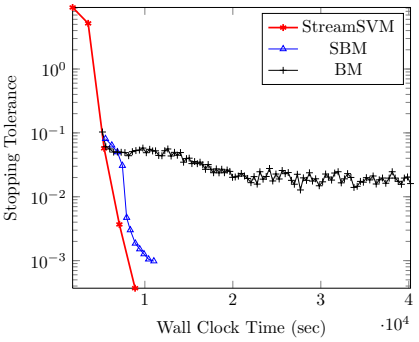


(a) $C = 0.0001$

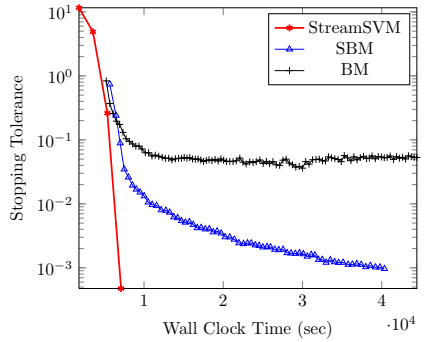
(b) $C = 0.001$



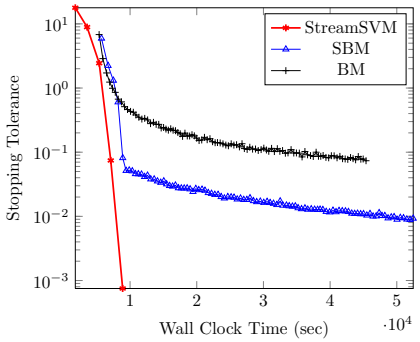
(c) $C = 0.01$



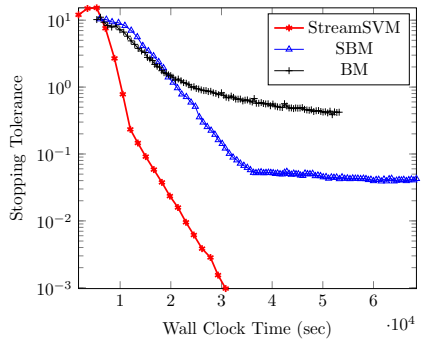
(d) $C = 0.1$



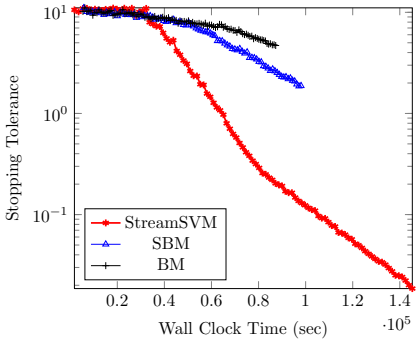
(e) $C = 1.0$



(f) $C = 10.0$

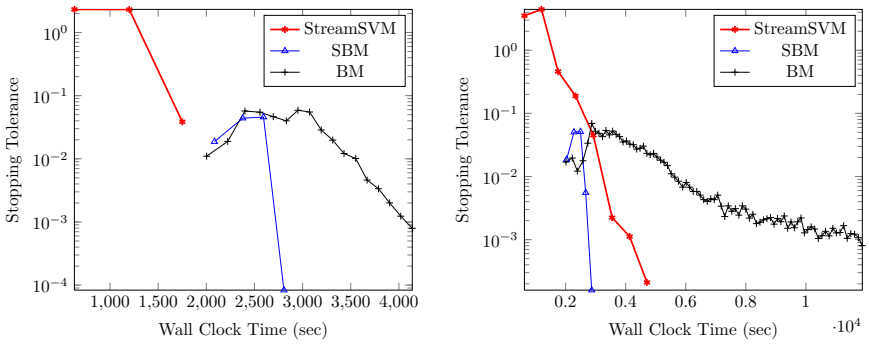


(g) $C = 100.0$



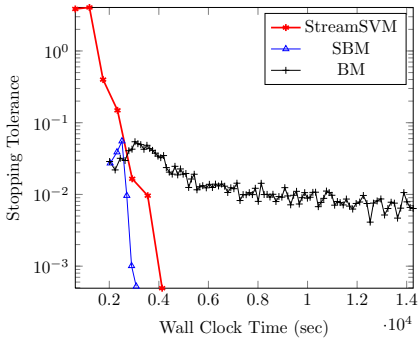
(h) $C = 1000.0$

Figure 7: Duality gap as a function of wall clock time on the `ocr` dataset for various values of C

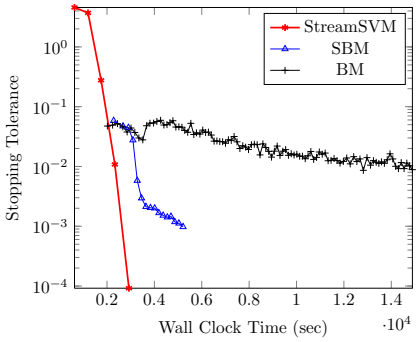


(a) $C = 0.0001$

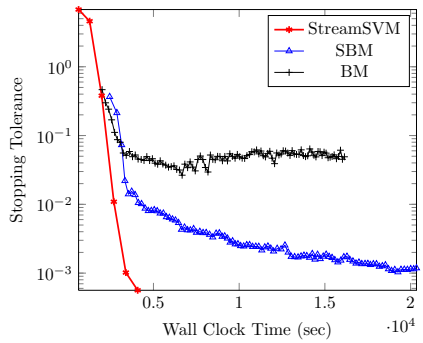
(b) $C = 0.001$



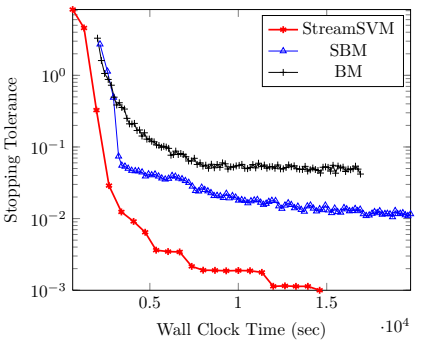
(c) $C = 0.01$



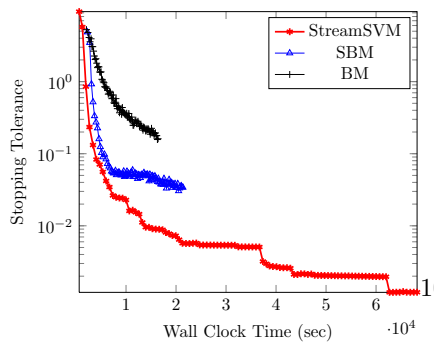
(d) $C = 0.1$



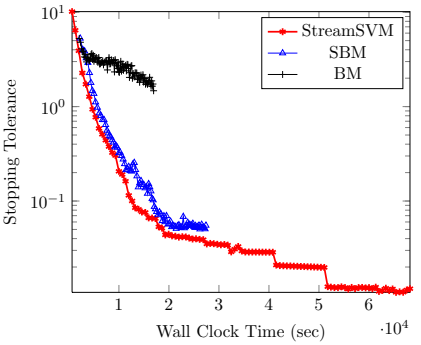
(e) $C = 1.0$



(f) $C = 10.0$



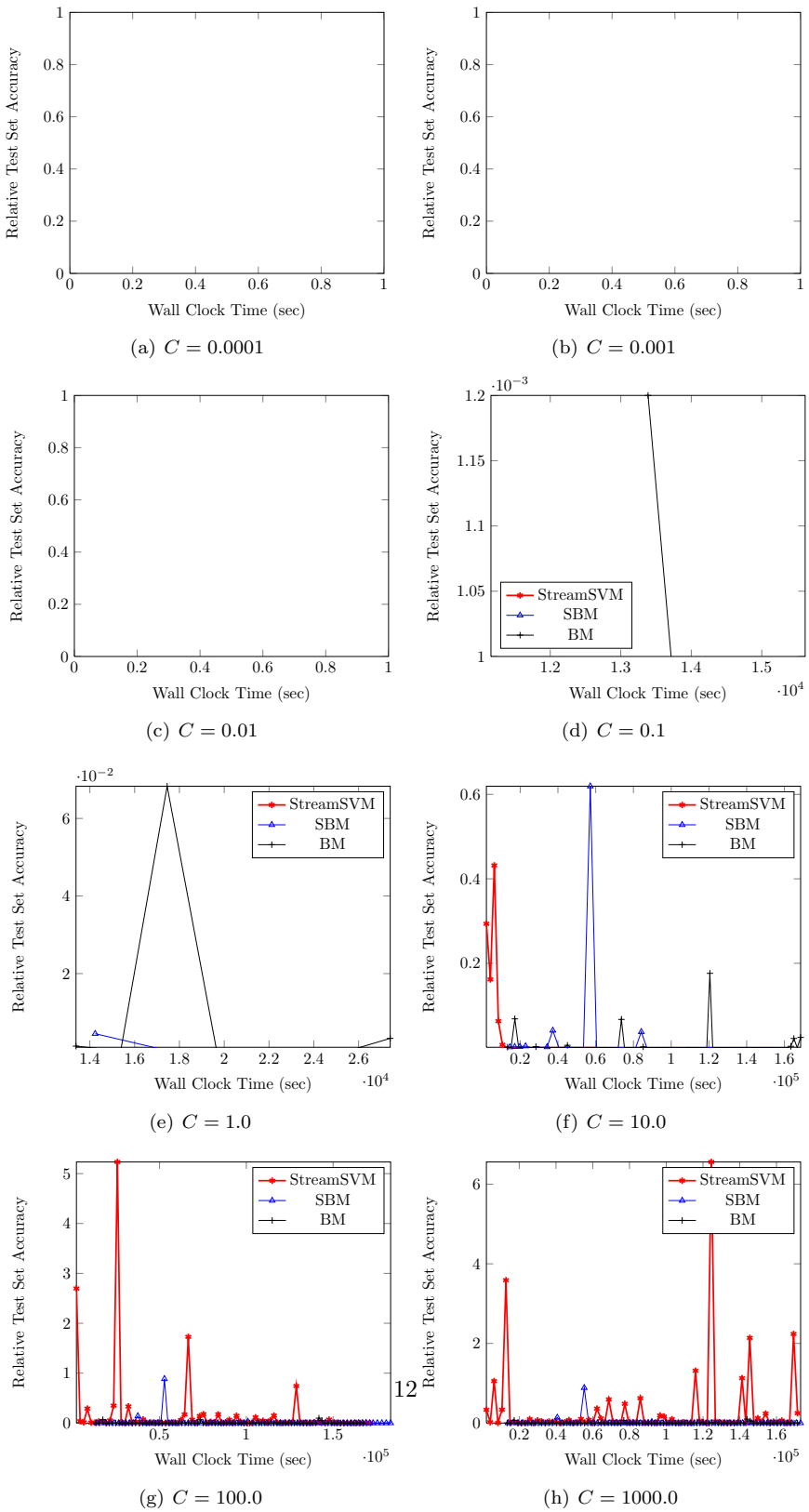
(g) $C = 100.0$



(h) $C = 1000.0$

Figure 8: Duality gap as a function of wall clock time on the `webspam-t` dataset for various values of C

3 Test Set Accuracy vs Wall Clock Time



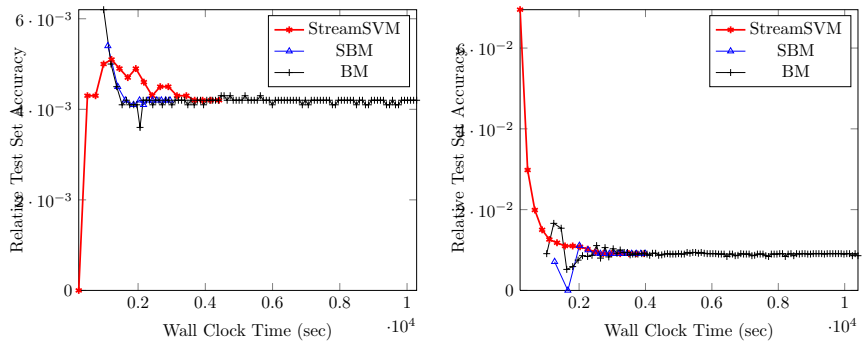
(a) $C = 0.0001$ (b) $C = 0.001$

(c) $C = 0.01$ (d) $C = 0.1$

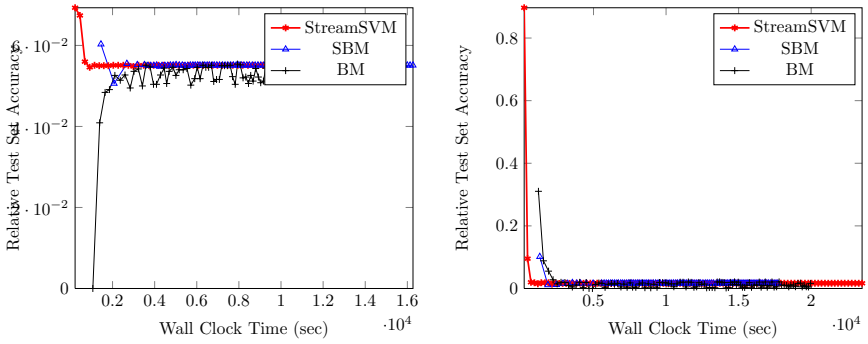
(e) $C = 1.0$ (f) $C = 10.0$

(g) $C = 100.0$ (h) $C = 1000.0$

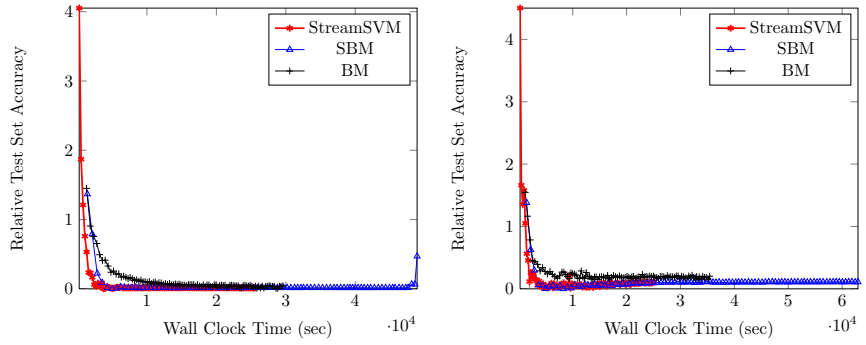
Figure 9: Relative test set accuracy as a function of wall clock time on the `dna` dataset for various values of C



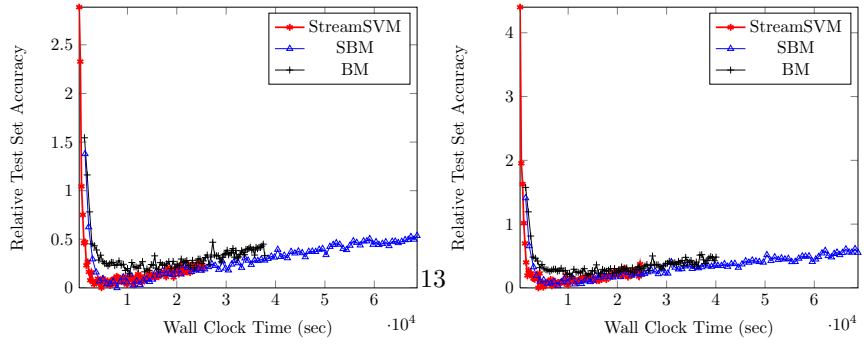
(a) $C = 0.0001$ (b) $C = 0.001$



(c) $C = 0.01$ (d) $C = 0.1$

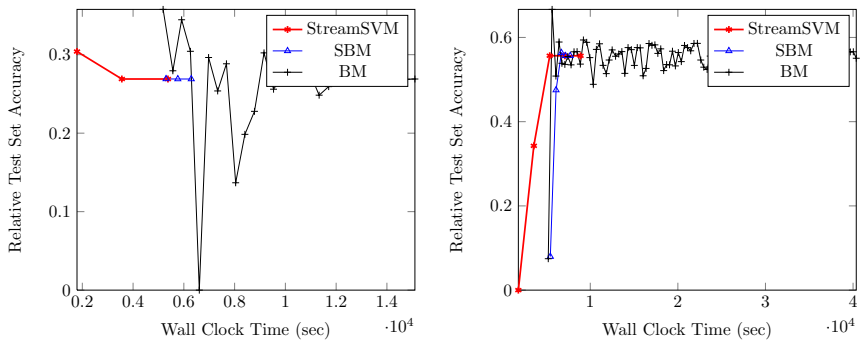


(e) $C = 1.0$ (f) $C = 10.0$



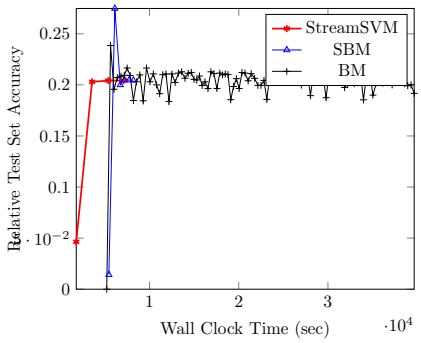
(g) $C = 100.0$ (h) $C = 1000.0$

Figure 10: Relative test set accuracy as a function of wall clock time on the `kddb` dataset for various values of C

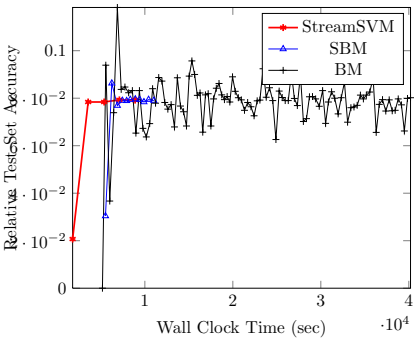


(a) $C = 0.0001$

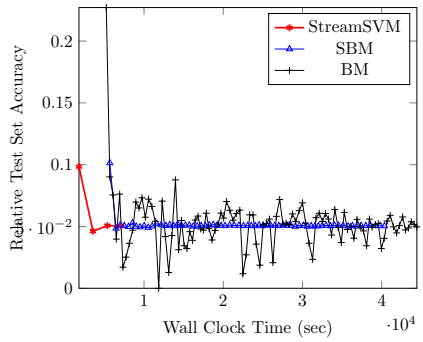
(b) $C = 0.001$



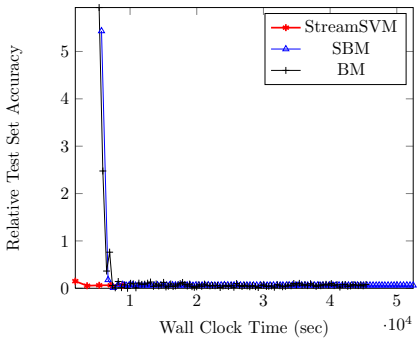
(c) $C = 0.01$



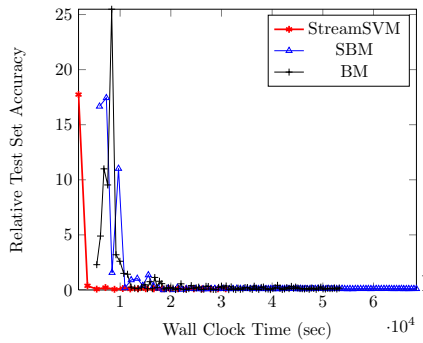
(d) $C = 0.1$



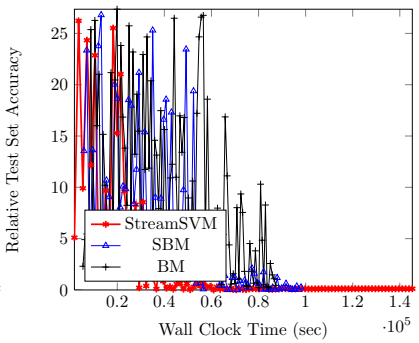
(e) $C = 1.0$



(f) $C = 10.0$

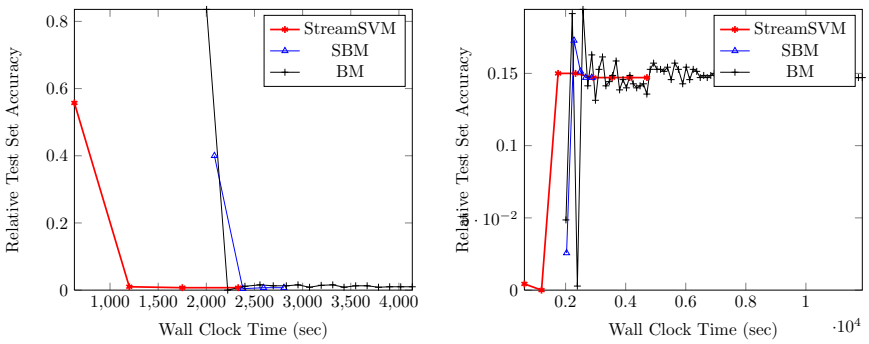


(g) $C = 100.0$



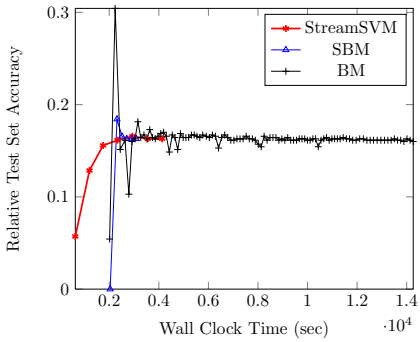
(h) $C = 1000.0$

Figure 11: Relative test set accuracy as a function of wall clock time on the `ocr` dataset for various values of C

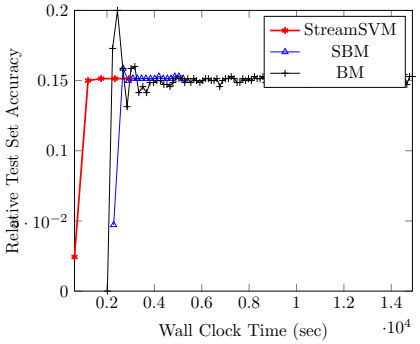


(a) $C = 0.0001$

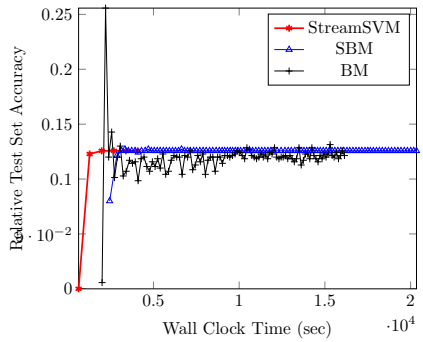
(b) $C = 0.001$



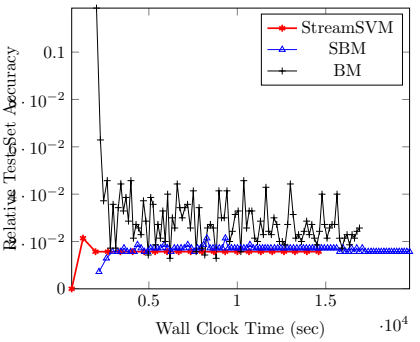
(c) $C = 0.01$



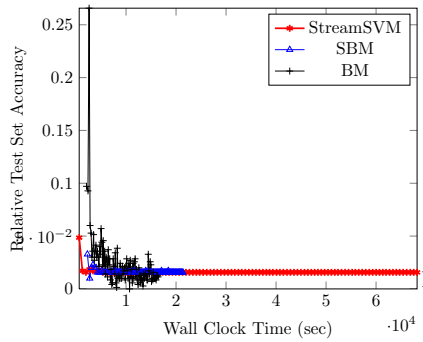
(d) $C = 0.1$



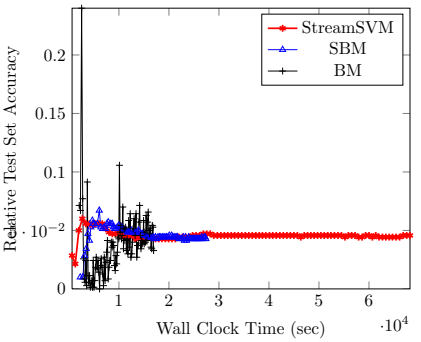
(e) $C = 1.0$



(f) $C = 10.0$



(g) $C = 100.0$



(h) $C = 1000.0$

Figure 12: Relative test set accuracy as a function of wall clock time on the `webspam-t` dataset for various values of C