Online boosting algorithm based on two-phase SVM training

Vsevolod Yugov*, Itsuo Kumazawa†

May 18, 2012

Abstract

We describe and analyze a simple and effective two-step online boosting algorithm that allows us to utilize highly effective gradient descent-based methods developed for online SVM training without the need to fine-tune the kernel parameters, and show its efficiency by several experiments. Our method is similar to AdaBoost in that it trains additional classifiers according to the weights provided by previously trained classifiers, but unlike AdaBoost, we utilize hinge-loss rather than exponential loss, and modify algorithm for the online setting, allowing for varying number of classifiers. We show that our theoretical convergence bounds are similar to those of earlier algorithms, while allowing for greater flexibility. Our approach may also easily incorporate additional nonlinearity in form of Mercer kernels, although our experiments show that this is not necessary for most situations. The pre-training of the additional classifiers in our algorithms allows for greater accuracy while reducing the times associated with usual kernel-based approaches. We compare our algorithm to other online training algorithms,and show, that for most cases with unknown kernel parameters, our algorithm outperforms other algorithms both in runtime and convergence speed.

1 Introduction

1.1 Online training methods

During the recent years there has been an increasing amount of interest in the area of online learning methods. Such methods are useful not only for setting where a limited amount of samples in being fed sequentially into a training system, but also for system, where the amount of training data is too large to fit into memory. In particular, several methods for online boosting and online support vector machine (SVM) training has been proposed. However, those methods have several limitations. The online boosting algorithms, such as [5] or [7], are usually limited to a fixed number of classifiers, while the online SVM training methods employing Mercer kernels ([6], [11]) may not converge well in the cases where the inappropriate kernel was chosen. Furthermore, kernel-using SVM usually have significant storage and computational requirements due to the large amount of kernel expansion terms. In this paper, we exploit the similarity between the mathematical description of boosting and support vector machines to create a two-stage online boosting algorithm with variable number of classifiers, that can exhibit greater flexibility than kernel -based SVM while having smaller computational costs.

Our method uses Pegasos, Stochastic Gradient Descent (SGD)-based SVM training method introduced in [11] to produce both a set of weak classifiers and boosting weights for combining them into strong classifier. Using this kind of training algorithm allows us to utilize solid theoretical background and well-defined convergence rate of SGD algorithms, as well as increased accuracy of classifier produced by boosting. We utilize a simplified variant of Pegasos with parameter k set to 1 on both stages. Our algorithm uses three parameters: λ_H and λ, corresponding to the regularization parameters in the SGD algorithm, and additional parameter r that defines the length each additional classifier is trained, and may be defined in several ways.

The resulting algorithm is simple and has low computational and storage costs, which allow it to be easily incorporated into real-time application.

1.2 Related Work

Our algorithm is closely related to algorithms used for SVM training and boosting. These algorithms, taken together, can be separated into several classes:

1.2.1 Support Vector Machines

The support vector machines are a class of linear or kernel-based binary classifiers that attempt to maximize the minimal distance (margin) between each member of the class and separating surface. Such maximization was shown to give near-optimal levels of generalization error ([12]). In most cases when dealing with kernels, the task of learning a support vector machine is cast as a constrained quadratic programming problem. Methods that deal with such problem usually need access to all labeled samples at once, and require about $O(m^2)$ operations, where m is the number of samples. Several approaches to the solution exist, such as interior point ([2]) methods and the decomposition methods, such as SMO ([8]). Also, recently, interest in gradient-based methods to solving primal SVM problem has risen drastically. These methods exhibit convergence rate independent of the number of samples, which is particularly useful for large datasets, and only need one or several samples for a single iteration, which lends itself well to the online setting.

*Tokyo Institute Of Technology, Dept. of Information Processing
†Dr. of Engineering, Imaging Science and Engineering Laboratory Tokyo Institute of Technology

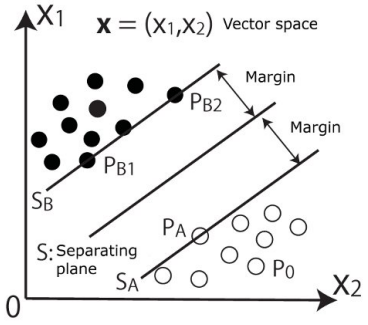


Figure 1: Illustration for Support Vector Machine

Main method that is used as a basis for our algorithm is Pegasos [11], which is a modified SGD method with an added projection step, although more generic algorithms such as NORMA ([6]) can be easily used in its place.

1.2.2 Boosting algorithms

Boosting is a meta-algorithm for supervised learning that combines several weak classifiers, i.e. classifiers that can label examples only slightly better than random guessing, into a single strong classifier with arbitrary classification accuracy. One of the most successful and well known examples is AdaBoost [4] and its variants, like LPBoost. The convergence properties of AdaBoost has been carefully analyzed ([10]), and it was used for such problems as text recognition, filtering and face recognition and feature selection. Boosting algorithms that employ linear combination of weak classifiers to form confidence function for strong classifier were shown to be closely related to the primal formulation for support vector machines ([9]). As in case of SVM, many boosting methods were designed for offline setting, where all of the training examples are given a priori, and share the same set of problems when dealing with larger datasets. Recently, there were several successful approaches ([7], [5]) designed to shift boosting to online setting, however, they assume that the number of weak classifiers in the boosting process stays the same, which limits the flexibility of their methods.

1.3 Outline of the paper

The remainder of this article is organized as follows: In section 2 we give the description of our algorithm. In section 3 we compare it to several existing online training methods in terms of computational cost and flexibility. Finally, we present our conclusion and further directions of research.

2 Algorithm description

2.1 Overview of SVM and Pegasos algorithm

Support Vector Machines are a useful classification tool, developed by Vapnik et. al. ([3]), that attempt to construct a hyperplane separating the data that has the largest distance to any point in any class (see fig. 1). The original formulation assumed data to be linearly separable, with no noise. Later, a loss term was added to account for noisy data. The most widely used loss function for SVM training is hinge-loss:

$$l(f, y) = \max(0, \rho - yf) \tag{1}$$

where ρ is called a margin parameter, y is a label of a sample and f is classification function of the svm, sometimes also used as a confidence measure. For linear SVMs,f is a simple inner product of input and weight vectors, while for SVMs using kernel trick for nonlinear classification

$$f(\vec{x}) = \sum_{i=1}^{N} \alpha_i k(\vec{x_i}, \vec{x}) + b$$

where $k(\cdot, \cdot)$ is a kernel function satisfying Mercer's condition and b is a bias term.

Several methods exist for solving both primal and dual formulations of the SVM optimization problem. In this article, we employ the Primal Estimated Subgradient Solver (Pegasos) method, as described in [11], with some modifications.

The primal form of SVM problem is a minimization problem with added regularization term. Formally, for a reproducing kernel Hilbert space $\mathscr{H}$ with kernel $k : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$, and a set of vectors $\mathscr{X} \in \mathbb{R}^n$ with corresponding labels $y : \mathscr{X} \to \{-1; 1\}$, find function f such that:

$$f = \arg\min_{f \in \mathscr{H}} \left(\frac{\lambda}{2}||f||_{\mathscr{H}} + \frac{1}{m} \sum_{\vec{x} \in \mathscr{X}} l(f(\vec{x}), y(\vec{x}))\right) \tag{2}$$

In the linear case, function f can be represented as $f(\vec{x}) =< \vec{\alpha}, \vec{x} >$, where $< \cdot, \cdot >$ denote an inner product, and the equation becomes

$$\vec{\alpha} = \arg\min_{\vec{\alpha} \in \mathbb{R}^n} \left(\frac{\lambda}{2}||\vec{\alpha}||_2 + \frac{1}{m} \sum_{\vec{x} \in \mathscr{X}} l(< \vec{\alpha}, \vec{x} >, y(\vec{x}))\right) \tag{3}$$

In our work, we also incorporate a bias term b by substituting $\vec{x}$ with $\vec{x}^e = \{\vec{x}, 1\}$. In this case, $\vec{\alpha} \in \mathbb{R}^{n+1}$. To simplify notation, we assume that all input vectors has been extended in such a way, and simply use $\vec{x}$.

In online setting, only one of the sample-label pairs is available at a time. There exist several algorithms to deal with such a problem that have well-established convergence bound. Amongst them, methods using Stochastic Gradient Descent(or, in case of hinge-loss function, subgradient descent) are most prominent. In this paper, we choose a Pegasos algorithm for its rapid convergence and simplicity of implementation. Here we only present algorithm with our modifications for weighting sample, for detailed analysis please refer to the original article ([11]).

On each iteration, the Pegasos algorithm is given iteration number t, regularization coefficient λ (regulating how "soft" the resulting margin is, i.e. whether the priority is given to low error rate over training set or larger margin between classes), sample vector $\vec{x}$ with weight w and label y, and updates vector $\vec{\alpha}$ as shown on fig. 2.

The only difference from [11] is addition of weighting term w, the significance of which will be explained below.

function PEGASOS($\vec{\alpha}, \lambda, \vec{x}, y, t, w$)
$l = \max(0, 1 - < \vec{\alpha}, \vec{x} > y)$
$\sigma = I(l > 0)$
$\eta = \frac{1}{\lambda t}$
$\vec{\alpha} = (1 - \eta\lambda)\vec{\alpha} + w\sigma\eta y\vec{x}$
$\vec{\alpha} = min\left(1, \frac{1}{||w||_2\sqrt{\lambda}}\right)\vec{\alpha}$
end function

Figure 2: The Pegasos algorithm with weighted samples.

The number of simultaneous input samples k, used in the original Pegasos algorithm, is taken to be 1, reducing Pegasos to an SGD algorithm with an additional projection step.

2.2 Overview of AdaBoost

AdaBoost [4] is an algorithm that iteratively adds weighted weak classifiers h_i to obtain a strong classifier H. Each additional classifier is selected from the pool of available classifiers to minimize the weighted error rate over training samples. This error rate was also used to calculate the weight of h_i in H and to update training sample weights in such a way so that the next classifier would favor samples misclassified by the current strong classifier, and ignore the ones classified correctly.

As most learning algorithms of that time, AdaBoost was designed for offline (batch) training, with the error rate estimated over all available samples. There are, therefore, several difficulties involved in employing boosting as an online training algorithm, several of which were mentioned in [5]. Their solution was to use a limited number of meta-classifiers, called selectors, each selecting a single weak classifier with least estimated error from a pool, combined into H. Both the weak classifiers and selectors were updated each iteration. The limited number of selectors and features allowed for simplified online algorithm. Their algorithm, however had two potential problems, one being that a limited number of selectors limited flexibility of the classifier, the second being that the effect constant updates have on the error rate of the weak classifiers was never addressed.

In our work, we note that the expression for the confidence function of strong classifier in AdaBoost

$$F(\vec{x}) = \sum_{i=1}^{T} \beta_i h_i(\vec{x}) \tag{4}$$

$$H = sign(f_H) \tag{5}$$

can be expressed in the same form as objective function f of SVM:

$$F(\vec{h}) =< \vec{\beta}, \vec{h} > \tag{6}$$

where $\vec{h} = \{h_1(\vec{x})...h_T(\vec{x})\}$ is a vector that consists of outputs provided by a set of T weak classifiers, $h_i(\vec{x}) \in \{-1; 1\}$. That means that an SVM training algorithm can be applied for training weights $\vec{\beta}$, with the same guarantees for convergence rate and generalization error bound shown in ([12]).

2.3 Resulting Algorithm

Using the above similarity, we separate the boosting process into two phases: the training of each successful weak classifier, and adjusting the boosting weights that combine weak classifiers into a strong one. Both phases can then be cast as a primal SVM problem, same as Eq. 3, with the difference that in phase 1 (weak classifier training) each sample is weighted according to the result provided by the current strong classifier. To reduce the amount of calculations, we have chosen the loss function for the strong classifier H, L, as a weight, although other weighting solutions are possible. Also unlike [5], we only train the last weak classifier of the set rather than all of them. The reason for this is that fixing parameters of all classifiers previously added increases the accuracy of the error rate estimation for the training of additional classifier, as well as significantly decreasing the amount of calculations needed for a single update. We choose the cutoff criteria, based either on the drop of estimated error rate of the weak classifier or the number of iterations the weak classifier was trained, after which the weights of the weak classifier are fixed, and the next classifier is added and begins training. We use the following notation for our algorithm:

Notation summary

$\vec{x}$ - an input sample vector. A single vector is provided for each iteration of the algorithm. Sample vectors are assumed to be extended with an additional constant element representing bias term.

y - the label provided for $\vec{x}$, $y \in \{-1; 1\}$

i - the number of current iteration.

λ - regularization parameter from Pegasos algorithm used for training weak classifiers.

λ_H - regularization parameter used for training strong classifier.

T - the number of weak classifiers in the current pool. This number is incremented each time a new classifier is added.

$\vec{\alpha}_k$ - weights of the k's weak classifier, $k = 1, 2..T$. When an additional classifier is created, a new $\vec{\alpha}_{T+1}$ is created and initialized with zeroes.

i_w - the number of iterations current weak classifier have been trained. Using established convergence bounds of Pegasos algorithm, can be used as a simplest form of cutoff parameter, i.e. each additional weak classifier is trained on a fixed number of samples.

r - a simplified cutoff parameter. A new classifier is added if $i_w > r$.

$f_t = f_t(\vec{x})$ - objective function of a t's weak classifier.

$F = F(\vec{h})$ - objective function of a strong classifier.

L - a loss function for the combined strong classifier. In this work we use hinge-loss function (see eq. 1).

h_t - output value of a weak classifier, $h_t = sign(f_t)$.

$\vec{\beta}$ - boosting coefficients, used for combining several weak classifiers into a stronger one. With the bias term, the number of elements in $\vec{\beta}$ is $T + 1$.

H - a combined strong classifier.

Using this notation, the algorithm is initialized with the

following data:

$$\begin{aligned} T &= 1 \\ \vec{\alpha}_1 &= \vec{0} \\ \vec{\beta} &= (1,0) \\ i &= 0 \\ i_w &= 0 \end{aligned}$$

Assuming a simplified cutoff criteria for weak classifier training, single iteration of the algorithm takes the form illustrated on fig. 3.

Input: Iteration numbers i,i_w, $\vec{x}$, y, λ and λ_H, vectors $\vec{\alpha}_k$, $k = 1..T$, $\vec{\beta}$
  for $t = 1 \rightarrow T$ **do**
    $f_t(\vec{x}) =< \vec{\alpha}_t, \vec{x} >$
    $h_t = sign(f_t)$,
  end for
    $\vec{h} = h_1, ...h_t, 1$
    $F =< \vec{\beta}, \vec{h} >$,
    $L = \max(0, 1 - yF)$,
    $i_w = i_w + 1$
    Pegasos$(\vec{\alpha}_T, \lambda, \vec{x}, y, i_w, L)$
    Pegasos$(\vec{\beta}, \lambda_H, \vec{h}, y, i, 1)$
  if $i_w > r$ **then**
    $i_w = 0$
    $T = T + 1$
    $\vec{\alpha}_T = \vec{0}$
    $\beta_T = 0$
  end if

Figure 3: Proposed algorithm.

First, the input vector is classified by each of the weak classifiers already added to the pool, forming vector $\vec{h}$ of classifier outputs. This pool has $T \ll i$ classifiers in it, starting from a single classifier on the first iteration, and an additional bias expansion term, as described in section 2.1.

Next, objective function and F and the loss function L for the strong classifier are calculated, given boosting weights $\vec{\beta}$. The loss function is then used as a weight for training the weights of the latest weak classifier $\vec{\alpha}_T$ with the Pegasos algorithm. It can be seen that, similar to AdaBoost family of boosting algorithms, our algorithm increases the weights of samples misclassified by the current strong classifier, and decreases the weights of samples classified correctly. In fact, due to the use of the hinge-loss function for weight calculation, samples classified with high confidence by the already existing classifiers have no effect on the weights $\vec{\alpha}_T$. This allows creation of the classifiers that compensate for the errors introduced by the previous ones, and eventual convergence to the true distribution of the samples.

After the weak classifier, boosting weights are adjusted, using vector $\vec{h}$ as an input. Usage of Pegasos algorithm guarantees that eventually, the weights would converge arbitrarily close to the optimum described by the function (3).

As the last step of an iteration, we calculate the cutoff parameter for adding new classifier. If the preset threshold is reached (in this case, the number of training iterations i_w reaches r), the value of T is increased, and a new classifier is initialized with zero weights.

Each iteration of the algorithm produces a strong classifier that can be used to get the class of vector $\vec{x}$:

$$H(\vec{x}) = sign(\sum_{i=1}^{T} \beta_i sign(< \vec{\alpha}_i, \vec{x} >) \quad (7)$$

There are several key differences between our algorithms and SVM using Mercer kernels as described in [11], [6]. The first and possibly most important one is that the number of weak classifiers T is much less than the number of input samples, i. While the algorithm described in [6] allows truncation of the kernel expansion coefficients, this is only applicable to the SGD algorithms with constant learning rate, and results in an accuracy penalty. The second difference is the update of vector $\vec{\beta}$, which allows change in the weights different from the exponential decay of [6]. As our experiments in section 3 show, these differences allow our algorithm to achieve better accuracy while being less resource intensive.

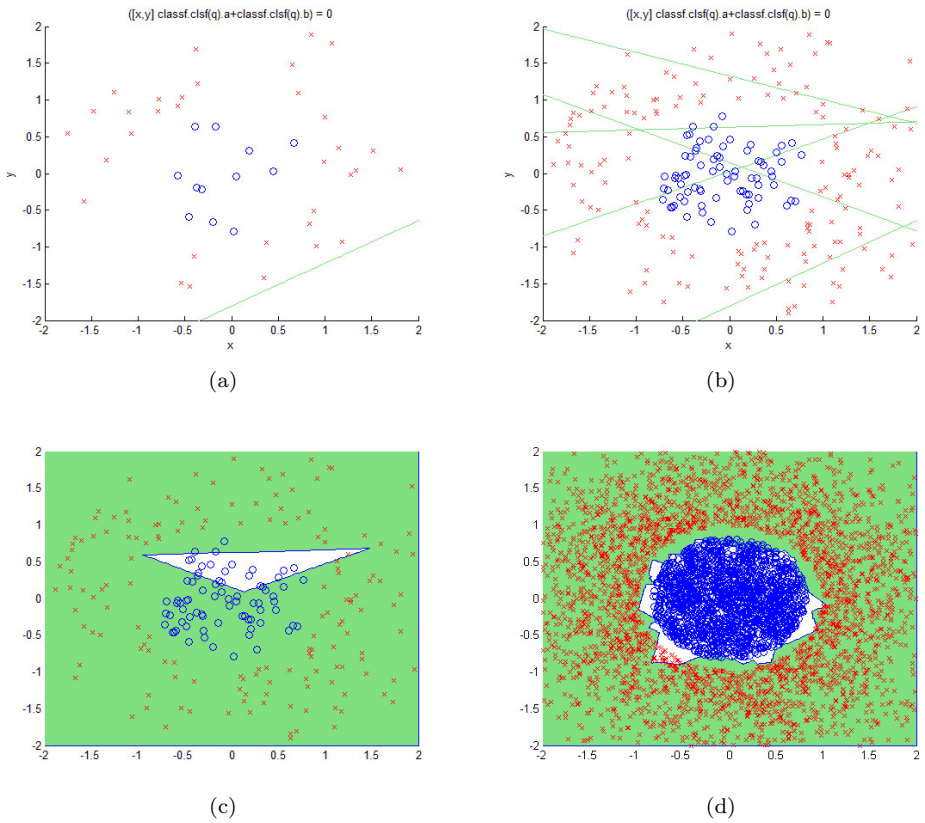
It is also important to note, that while parameters λ, λ_H and r somewhat affect the convergence rate, they are independent from the form of the class-separating surface, i.e., unlike the kernel methods, the convergence rate and resulting accuracy both depend heavily on the type and parameters of the kernel selected, our method converges similarly to the AdaBoost, with the resulting accuracy depending mainly on regularization parameters and resulting amount of weak classifiers T. This is shown in our Experiments section, where our algorithm is running on the same set of parameters for datasets with different variable distributions, and often outperforming even the kernel-using algorithms with ideal kernel settings.

2.4 Discussion on the proposed method

The process of convergence is illustrated in fig. 4 for the case of two-dimensional dataset, where the data is classified according to whether it is inside unit circle about origin. The parameters used for this illustration, $\lambda = 0.02$, $\lambda_H = 0.02$, $r = 50$. As can be seen, the original separation is quite bad since a linear classifier cannot converge with such data distribution, however, as new weak classifiers are added, the separation achieved by strong classifier becomes closer and closer to the true data distribution.

Each phase of our training algorithm is trying to solve the primal formulation of SVM (eq. 2), using Pegasos algorithm, which has a convergence rate of $O\left(\frac{R^2}{\lambda\epsilon}\right)$, where R is a bound on the norm of input vector, and ϵ is desired accuracy. It is easy to see, that for the second phase $R = T$, T being the number of added classifiers, and each classifier producing output $h_i \in \{-1; 1\}$, so the convergence rate slowly decreases as additional classifiers are added. To combat this, certain classifiers with lows weights may be removed from the pool. This has a small additional effect of increasing the ability of algorithm to adapt to a changing classification target.

The changing weights of sample vectors in the first phase, as well as increasing the size of classifier pool, can



(a) (b)

(c) (d)

Figure 4: Illustration of the convergence process: (a) The first linear classifier trained ($i = 50$, $T = 1$), (b) Several additional weak classifiers added ($i = 250$, $T = 5$) (c) The data separation corresponding to the set of classifiers in (b). Background color shows class generated by classifier, form and color of data points show actual label y (d) Data separation achieved after convergence ($i = 5000$, $T = 100$).

be easily recast as a moving classification goal, described in [6]. According to them, movement of the target introduces an accuracy penalty that is approximately linear to the total distance traveled by the target, which suggests that in our case the convergence should be slower than the convergence of an SVM training algorithm using kernel parameters appropriate for the sample distribution. This is an additional reason why we only update a single weak classifier, rather than all of them, since otherwise the drift would be proportional to the number of classifiers added, significantly penalizing convergence rate.

However, the bounds mentioned in [6] are not tight, and it is not clear how their bounds are altered by the additional projection step in Pegasos. Adding to this the fact that in most real-time applications the exact form of the data distribution is not known, we have decided to compare algorithm performance using experimental data.

2.5 Possible extensions

In this section we discuss several possible extensions of our algorithm. For example, as can be easily noticed, while parameter T is much less than the kernel expansion terms, it still grows with additional samples, which may lead to loss of effectiveness and overfitting. There are several possible extensions that may allow to avoid this. One way is to remove classifiers $\vec{\alpha}_i$ for which the condition $|\beta_i| < \epsilon$ held for several iterations in a row. This will also allow the algorithm to adapt better to the case of changing distribution. The other way is to increase the parameter r depending on T, or to choose a different cutoff algorithm altogether.

Also, to increase adaptability to changing input conditions, it is possible to change the calculation of the learning rate of the Pegasos algorithm, for example, stopping its decay on a certain threshold. However, such experiments are outside the scope of this paper.

3 Experiments

3.1 Description

We compare our algorithm to both Pegasos [11] and Norma [6], implemented on MATLAB for both the linear and the kernel-based case. The experiments are being run on AMD Phenom X4 965, with only one core being used for calculations. We perform several experiments, aiming to compare generalization error and convergence rates over different datasets, as well as the ability of the algorithm to adapt to the distribution with the changing parameters (flexibility) .

We use several artificial datasets with known distributions and separation properties, and a Forest Covertype dataset (separating class 5 from other classes), originally used in [1], and also used for comparison of convergence

speed in [11]. The artificial datasets are generated according to the following distributions:

1. *High-dimensional linearly separable data (Linear)* A random hyperplane is created in 50-dimensional space. Data points are generated randomly to both positive and negative sides of the hyperplane. Data points too close to the hyperplane are filtered out.
2. *High-dimensional linearly separable data with noise (Linear + noise)* Same as 1, but 10% of the labels are switched, simulating salt-and-pepper noise.
3. *Bayes-separable data (Bayesian)* This dataset is generated as described in [6], i.e. in such a way so that data is clearly separable using ideal Bayesian classifier for known class distributions.
4. *Bayes-separable data with moving distribution (Drifting)* As in 3, but the parameters of a distribution are changed slightly each iteration, simulating target movement.
5. *Bayes-separable data with switching distribution (Switching)* Once again, a dataset generated according to the description in [6], with the distribution changed drastically every 1000 iterations.

For each distribution we measure the decrease of estimated error rate over training dataset (estimated error being simply the number of misclassified training samples divided by number of iterations), and the resulting error rate over the testing dataset (generated without noise in case of noisy distribution).

For all experiments, the parameters of our algorithm were fixed, with $\lambda = 0.02$, $\lambda_H = 0.03$ and the cutoff parameter $r = 150$. Pegasos and Norma used parameter $\lambda = 0.02$, and either a linear kernel or a Gaussian RBF kernel with $\gamma = 0.01$, which is the same value of γ used for generating Bayes - separable datasets.

3.2 Experimental results

The graphs for the estimated error rate are shown on figure 5, while the resulting error rate on the test datasets is shown in Table 1. It can be seen, that for linearly separable problems our algorithms performs on par with the Pegagos algorithm, with slight increase of the error rate possibly due to the overfitting. For kernel-based methods, however our algorithm usually outperforms both Norma and Pegasos, unless the exact kernel parameters are used, and even then (see fig. 5b) our algorithm performs slightly better in the long run. It is interesting to note, that for switching dataset, NORMA actually outperforms Pegasos by a considerable margin, indicating that Pegasos algorithm is more sensitive to rapid changes in the classification target, most likely due to the rapid decay of the learning rate with time, while our algorithm was largely able to compensate.

For the Covertype dataset, linear classifiers work best, and approach the error rates indicated in the paper([1]), with Pegasos and our algorithm giving virtually the same results.

It is also important to note that, when compared to kernel-based SVM algorithms, our algorithm is much more efficient both in terms of computing and storage requirements, since the amount of weak classifiers, each requiring only a single inner product calculation, is much lower than the amount of kernel expansion terms produced by both NORMA and Pegasos for the same accuracy levels. For example, in the test shown on fig. 5c, the resulting amount of kernel expansion terms was over 5000 after 10000 iterations (for both Norma and Pegasos), while the amount of weak classifiers generated by our method was only 67.

Dataset	O	PL	NL	PG	NG
Linear	0.04	0.03	0.09	0.07	0.1
Linear+ noise	0.02	0.002	0.005	0.02	0.08
Bayesian	0.03	0.17	0.22	0.08	0.15
Drifting	0.04	0.28	0.35	0.15	0.18
Switching	0.11	0.20	0.21	0.32	0.13
Covertype	0.19	0.2	0.35	0.43	0.48

Table 1: Error rate over the testing dataset, O - our algorithm, PL -linear Pegasos, NL - linear NORMA, PG - Pegasos using Gaussian RBF kernel with $\gamma = 0.01$, NG - NORMA with the same kernel

4 Conclusion and future work

We have shown how combination of boosting and online SVM training creates an algorithm that outperforms standard training algorithms in the case when the kernel parameters are not known, and in general allows for the creating more efficient classifiers that simple kernel expansion. The drawbacks of our algorithm include the fact that it is not as efficient in case of linearly separable problems, and that it inherits some of the sensibility to the rapid changes in target function from Pegasos.

The possible future expansions include incorporating non-linear kernels into algorithm, and investigation various possible applications of our algorithm to online learning and image processing problems.

References

[1] J. Blackard. Comparative accuracies of artificial neural networks and discriminant analysis in predicting forest cover types from cartographic variables. *Computers and Electronics in Agriculture*, 24(3):131–151, December 1999.

[2] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, New York, NY, USA, 2004.

[3] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Mach. Learn.*, 20(3):273–297, September 1995.

[4] Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *J. Comput. Syst. Sci.*, 55(1):119–139, August 1997.

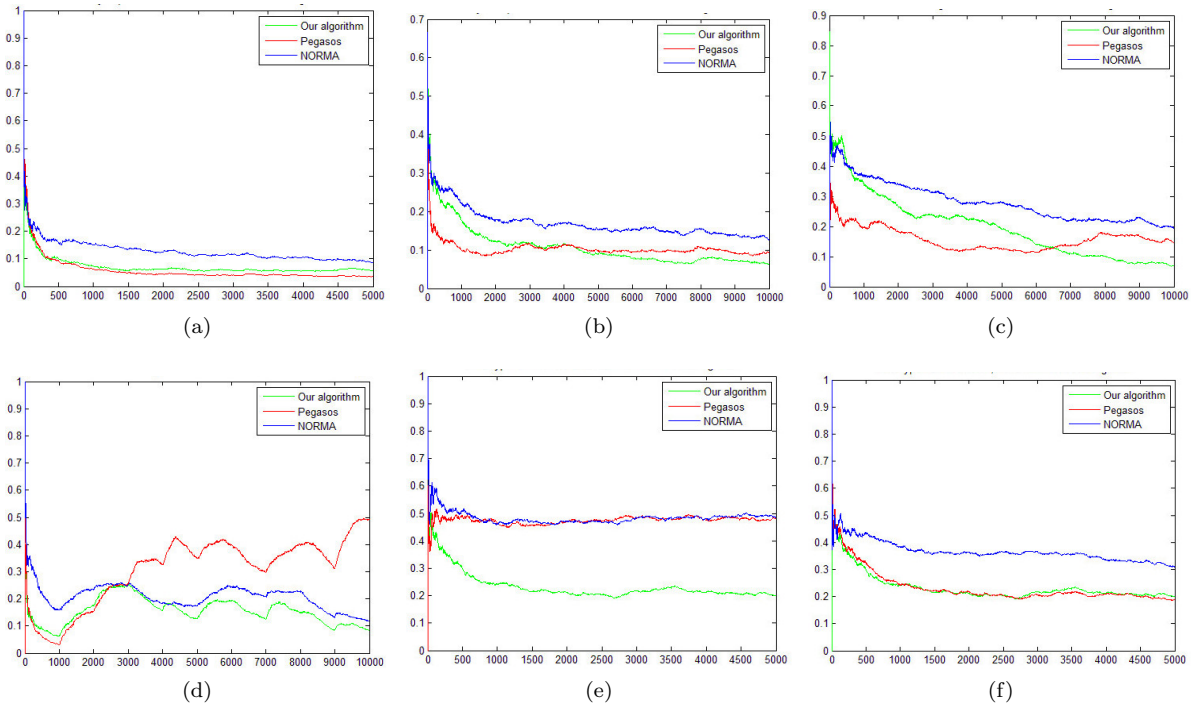


Figure 5: Experimental results.(a) Linearly separable dataset, Linear Norma and Pegasos, (b) Bayes-separable dataset, Pegasos and Norma using RBF kernel (c) Bayes-separable dataset with drifting distribution parameters, Pegasos and Norma using RBF kernel, (d) Bayes-separable dataset with distribution parameters being switched every 1000 iterations, Pegasos and Norma using RBF kernel, (e) Covertype dataset, Pegasos and Norma using RBF kernel (do not converge), (f) Covertype dataset, linear Pegasos and Norma

[5] M. Grabner H. Grabner and H. Bischof. Real-time tracking via on-line boosting. In *Proceedings British Machine Vision Conference (BMVC), volume 1*, pages 47–56, 2006.

[6] Jyrki Kivinen, Alexander J. Smola, and Robert C. Williamson. Online learning with kernels. *IEEE Transactions on Signal Processing*, 52(8):2165–2176, 2004.

[7] N. Oza and S. Russell. Online bagging and boosting. In *Artificial Intelligence and Statistics 2001*, pages 105–112. Morgan Kaufmann, 2001.

[8] John C. Platt. Advances in kernel methods. chapter Fast training of support vector machines using sequential minimal optimization, pages 185–208. MIT Press, Cambridge, MA, USA, 1999.

[9] Gunnar Ratsch, Bernhard Scholkopf, Sebastian Mika, and Klaus-Robert Muller. Svm and boosting: One class. Technical report, 2000.

[10] Robert E. Schapire, Yoav Freund, Peter Bartlett, and Wee S. Lee. Boosting the Margin: A New Explanation for the Effectiveness of Voting Methods. *The Annals of Statistics*, 26(5):1651–1686, 1998.

[11] Shai Shalev-Shwartz, Yoram Singer, and Nathan Srebro. Pegasos: Primal estimated sub-gradient solver for svm. In *Proceedings of the 24th international conference on Machine learning*, ICML '07, pages 807–814, New York, NY, USA, 2007. ACM.

[12] Vladimir N. Vapnik. *The Nature of Statistical Learning Theory*. Springer New York Inc., New York, NY, USA, 1995.