Online boosting algorithm based on two-phase SVM training and its application to image processing

Author:
Yugov Vsevolod

Supervisor:
Itsuo Kumazawa

Department of Information Processing

Tokyo Institute of Technology

A thesis submitted for the degree of

Doctor of Philosophy

September 2012

Abstract

We describe and analyze a simple and effective two-step online boosting algorithm that allows us to utilize highly effective stochastic gradient descent based methods developed for online SVM training without the need to fine-tune kernel parameters, and show its efficiency by several experiments. Our method is similar to the 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 online setting, allowing for varying number of classifiers. We show the effectiveness of our method by developing applying it to the task of object tracking on the mobile device (iPhone). In order to achieve the real-time processing speed we furthermore describe a set of compact features in order to fully utilize the parallel processing capabilities of the device GPU. We then show that utilizing our algorithm with such features allows for a high discrimination rate even with a small number of features being utilized.

To ...

Acknowledgements

I would like to acknowledge all the people that have helped me with this research problem. First of all, I thank my academic advisor, prof. Kumazawa, for giving me a chance to live and study in Japan, and for his support during my research. I would also like to thank other member of my laboratory, especially Fukushi-san and Matsumura san, as well as recently graduated Quivy-san, for their help in discussing my work and assisting me in data collection, and just for being good friends. I recognize that this research would not have been possible without the financial assistance of Japans Ministry of Education, Culture, Sports, Science and Technology (MEXT), to which I express my gratitude.

Also, I want to thank my family, especially my grandmother who has raised and always supported me, my mother, whose initiative has made my study here possible, and my brother, who has always helped me in tight spots.

Last, but not least, I would like to express my gratitude to Hirasawa Makiko and Saeko, who has provided me home away from home in Japan, and without whose moral support I'd have never made it so far.

Contents

List of Figures

List of Tables

GLOSSARY

1

Introduction

1.1 Aims of research

Recently, there has been several breakthroughs in the area of online learning and classification. This was partly driven by the massive amount and ever increasing speed of data acquisition in our information-driven society. The amount of data, especially in applications related to image processing quickly outstrips the capacity of many common learning algorithms that require all of the data to fit in memory or to be available at the same time. Futhermore, many of them have computational requirements that are polynomial of degree two or above on the amount of data. In particular, algorithms related to Support Vector Machines (SVM), a common and effective classification tool introduced by Vapnik =ref=, are usually quite computationally expensive.

For these reasons, several low-complexity, linear- or near-linear time algorithms has been developed specifically for the tasks that either have online limitations, i.e. only a limited number of samples available at the time and only sequential access to data, or need to process the amount of data that cannot fit in memory.

Two related types of classifier methods enjoy increased attention lately. For one, Support Vector Machines, primarily binary classifiers that have been successfully used for various tasks in image and signal processing, speech recognition and DNA analysis, have been successfully adapted to the online setting by using Stochastic Gradient Descent methods on their primal formulation. However, while incredibly efficient in the case of linear classification, introducing kernels to achieve nonlinearity in the online setting results in the rapid increase of computational complexity, as kernel expansion coefficients are accumulated.

On the othe hand, boosting algorithm for aggregating several simple classifiers into one stronger has also been adapted for various online and adaptation tasks, especially in the areas like object tracking in the video sequence, where the adaptability of a model to changing conditions is paramount. Most of such methods, however, limit the complexity, and, as a result, possible accuracy, by fixing the amount of added classifiers and fixing the feature pool.

Our Goal In this thesis we aim to bridge the gap between boosting and online SVM learning by exploiting the simolarities between both that allow us to introduce a new boosting algorithm based on two-step SVM training. The proposed algorithm allows for greater flexibility, and has smaller computational costs than traditional kernel-based methods while achieving similar or greater accuracy. To our knowledge this is the first such algorithm proposed.

Contrbitions Our contributions in this paper, are therefore as follows. First we propose a new learning method and compare it against existing methods in terms of accuracy and computational complexity, as well as proposing several variations of the method useable for various applications. Then, we introduce a specific application of the descibed method to the task of video tracking on the mobile device to demontrate feasibiity of the method in a practical application. We show that reduced computational costs of our method, as well as highly parallel processing on the device's GPU allows such complex applications to run in the real time. Also, for this application a new set of simple features is introduced and evaluated.

1.2 Overview of common methods

As mentioned above, the algorithm introduced in this paper is based partly on new SGD-based training methods, as well as well known algorithms, such as Adaboost. In particular, our work has been inspired by the following algorithms:

NORMA Naive Online Risk Minimization Algorithm =ref= is one of the first and most generics algorithms based on stochastic gradient descent. It can be applied to various online task that require nonlinear separation of data, including kernel-based SVM, regression and novelty detecion. This paper has served as a basis to many other related algorithms, and provides a solid theoretical background by defining bounds on error and convergence rates of SGD-based methods.

Pegasos Primal Estimated sub-GrAdient SOlver for SVM (PEGASOS) (==ref==) is a more recent algorithm that bridges the span between online and batch learning by allowing several samples to be processed at once. It gives significantly iproves convergence speed compared to NORMA at the price of a slight increase of a computational complexity of a single iteration.

Online AdaBoost In a series of papers, =ref=, and online boosting algorithm for feature selection is introduced, and several applications of it are discussed. This algorithm, and its limitation, are what has originall inspired us to work on a boosting-related methods.

1.3 Algorithm outline

Our algorithm takes at its basis the offline Adaboost algorithm and transfers it to online setting by utilizing its similarity to the SVM formulation. The addition of the weak classifier that requires minimization of the weighted error rate ove the field of available classifiers in AdaBoost is replaced by iterative updates in the form of Pegasos algorithm applied over several data inputs. To reflect the changing parameters of the distribution, the boosting weights themselves are updated in the similar fashion. The algorithm is described in detail in Chapter 3.

1.4 Possible applications

The method introduced in this thesis is a generic online learning method applicable to a wide range of problems, which it shared with other methods in the same area. Two of the example applications include:

Tracking model changes Our proposed algorithm can be used to create and contniuosly update a model changing in time, such as an object model in the tracking applcation, as mentioned in =ref=.

Dealing with data with nonlinear distributions Many data acquisition tasks produce data that is not linearly separable into two classes. While kernel-based methods can be applied to the task of classifying such data, they often fail when the kernel function and its parameters are not chosen well. Our method, however can be applied to arbitrary distribution, as the parameters have minimal imact on the resulting accuracy.

2

Related works

Both the SVM and boosting-based learning algorithms, and their applications to various tasks in image processing are extremely widespread. In this chapter we outline the works most closely related to the presented methods, since the size constraints of the paper do not allow for a detailed review of this area.

2.1 Overview

In general, classification is a problem of identifying to which set, or cathegory belongs the next observation. The cathegories may be given beforehand, or derived from the data itself by application of a chosen clustering method. Usually, a certain set of data samples is given beforehand (training dataset), which serves as a basis by which the membership of the new sample is determined. The data samples are transformed into a set of explanatory variables, or features, and from them the cathegory-defining model, or classifier, is built.

While the distinction between offline and online algorithms for training classifiers is not clearly defined, it is usually accepted that the offline systems have random access to all training data at the same time, and that the model resulting from this data should asymptotically converge to the equilibrium. In this setting, the training time is less important. On the other hand, online systems have only limited access to the data, usuallly to a single sample at a time, or a few consecutive samples, and it is preferrable for the learning iteration to run in the real time, i.e. that the update iteration should take less time then the acquisition of the next data sample.

There are several types of classifiers, such as bunary or multiclass, linear, nonlinear and cathegorical, etc. In this paper, we focus on the linear binary classifiers, which can be used for nonlinear classification by transforming feature space.

2.2 Offline classification algorithms

2.2.1 Support vector machines

The support vector machines are a class of linear binary classifiers that attempt to maximize the minimal distance (margin) between classes, i.e. to construct a hyperplane in the feature space that separates the two classes and is located, intuitively, exactly in the "middle" between them (see =fig=). The mathematical formulation for this problem is: given a set of training samples $\vec{x}_i$ and associated labels y_i, $i \in [1..n]$, minimize in$\vec{w}, b$ $\frac{1}{2}||\vec{w}||^2$ subject to $y_i(\vec{w}\vec{x}_i - b) \geq 1$. Introducing Lagrangian multiplers α_i, the primal formulation then becomes as follows:

$$\min_{\vec{w},b} \max_{\alpha_i} \left\{ \frac{1}{2}||\vec{w}||^2 - \sum_{i=1}^{n} \alpha_i (y_i(\vec{w}\vec{x}_i - b) - 1) \right\} \tag{2.1}$$

The classifier output is then the sign of the confidence function $f(\vec{x}, \vec{w}) = \vec{w} * \vec{x} - b$,

$$H(\vec{x}) = sign(f(\vec{x}, \vec{w}))$$

The dual formulation of the above problem

$$\max_{\alpha_i} \left(\sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum j = 1^n \alpha_i \alpha_j y_i y_j k(\vec{x_i}, \vec{x_j}) \right) \tag{2.2}$$

subject to $\alpha_i \geq 0$,where, in original linear case, $k(\vec{x_i}, \vec{x_j}) = \vec{x_i} \cdot \vec{x_j}$, $\cdot$ denoting an inner product, and $\vec{w} = \sum_{i=1}^{n} \alpha_i y_i \vec{x}_i$. In this formulation, data points $\vec{x}_i$ for which $\alpha_i > 0$ are called the support vectors, giving rise to the name of the method. Intuitively, these are the points that lie on the margin hyperplanes.

Original =ref= formulation assumed that the data was linearly separable, and couldn't be solved for the noisy data. One of the most important results related to this version of classifier was that, when solvable, it was shown =ref= to minimize the theoretical upper bound on the testing error rate. This bound is related to the VC dimension of the classifier, and governs the relation between the capacity of a learning machine and its performance. The bound is as follows: if the classifier with parameters

$\vec{\alpha}$ achieves empirical error rate, i.e. error rate on a set used for training, $R_{emp}(\vec{\alpha})$, then with probability $1 - \eta$, the following bound holds:

$$R(\vec{\alpha}) \leq R_{emp}(\vec{\alpha}) + \sqrt{\left(\frac{h(log(2l/h) + 1) - log(\eta/4)}{l}\right)} \tag{2.3}$$

where l is the number of training samples and h is the VC dimension of a classifier function, defined as the number maximum cardinality of a data point set that can be shattered (separated for any assignment of labels $y \in -1; 1$ to data points). For example, a linear classifier of dimension n has VC dimension of $n + 1$.

SVM were then originally derived as a family of classifiers minimizing right-hand part of =equation=, thus minimizing the expected risk and generalization error. The original classification, however, was too rigid and not very usable on sets of data from the real world, so in =ref=, Eq. (2.1) was rewritten, replacing hard constraints with a loss function:

$$\min_{\vec{w}} \left\{ \frac{\lambda}{2} ||\vec{w}||^2 + \frac{1}{n} \sum_{i=1}^{n} l(\vec{w}, (\vec{x}, y) \right\} \tag{2.4}$$

where $l(\vec{w}, (\vec{x}, y)$ is a loss function determining penalty for the outlier and λ is a parameter that determines the "softness" of the margin, with large λ favoring larger number of outliers with smaller margin.

The most commonly used loss function l is a hinge-loss function, that linearly punishes the outliers and margin errors, i.e. data points that are classified correctly but lie between margin hyperplanes

$$l(\vec{w}, (\vec{x}, y) = \max(0, \rho - y(\vec{w} \cdot \vec{x}))$$

where ρ is a margin parameter, usually assumed to be 1 for maximum margin algorithms.

When using Eq. (2.2.1) , dual form of Eq. (2.4) is the same as Eq. (2.2) , with the constraints changed to $0 \leq \alpha_i \leq C$, $C \propto \frac{1}{\lambda}$. This constrained quadratic problem is the one that is usually solved by most common offline methods.

2.2.1.1 Using kernel trick to create nonlinear classifier

It is easy to see that all of the above formulas can be expressed in terms of linear combinations of kernel function $k(\cdot, \cdot)$ on the input data samples and coefficients α_i.

For instance, confidence function of the classifier $f(\vec{x}, \vec{w}) = \vec{w} * \vec{x}$ can be replaced with $f(\vec{x}; \{\vec{x_i}, \alpha_i\, y_i\}) = \sum_{i=1}^{n} \alpha_i y_i k(\vec{x}, \vec{x_i})$. The linear function $k(\vec{x_i}, \vec{x_j}) = \vec{x_i} \cdot \vec{x_j}$ of the original formulation can then be replaced by any function satisfying Mercers condition, i.e. any positive definite kernel. There also exists a body of research (such as =ref=) that deals with practical applications of non-positive definite kernels (such as well known sigmoid function), but that lies beyond the scope of a current work.

The Mercers condition, in essence, guarantees that there exists a feature space V, that has an operation of inner product defined in it, and a map from the input data space S, $\phi : S \leftarrow V$so that kernel function between two vectors $\vec{x}, \vec{y} \in S$ is equivalent to inner product: $k(\vec{(}x, \vec{y}) = \phi(\vec{x}) \cdot \phi(\vec{y})$. Essentially, the kernel trick maps input vectors $\vec{x_i}$ into larger-dimensional feature vector space V , where, hopefully, the data becomes linearly separable (see =fig= for illustration).

This techniques allows using SVM as nonlinear classifier, and greatly expands the variety of possible applications since as long as the kernel function is defined, the input vectors $\vec{x_i}$ do not even have to be numbers,a nd can instead be words or area descriptors in an image (=ref=]). However, its application also increases the costs associated with learning and using the classifier, since instead of a single vector $\vec{w}$ we have to store all nonzero α_i and associated $\vec{x}_i$, and the increased dimensionality of the feature space gurantees the larger possible amount of support vectors as compared to the linear formulation. Also, the same increased dimensionality increases the VC dimension of the classifier, raising bounds on the generalization error, i.e. increasing the risk of overfitting. In addition, the kernel formulation makes the primal problem much harder to solve, biasing the existing research towards quadratic dual formulation, although the methods of dealing with it exist and are introduced in section 2.3.1.1 below.

The most commonly used kernels are listed in =table=.

2.2.1.2 Common methods for offline SVM training

Most common methods for SVM training deal with the solution to the Eq. (2.2) , and as such are usually applicable to the Quadratic Programming problems in general. Several classes of such silution techniques should be mentioned in this overview.

Interior point methods IP methods (for example, =ref=) replace linear constraints of the primal with a barrier function, similar to the IP methods for linear programming. The result is a sequence of unconstrained problems which can be optimized very efficiently using Newton or Quasi-Newton methods. The advantage of IP methods is that they achive rapid convergence to a given accuracy bound in terms of a number of iterations. Unfortunately, they typically require run time which is cubic in the number of data samples n. Moreover, the memory requirements of IP very large, so such methods are not suitable for the training sets with large number of samples.

Segmentation-based methods To overcome the quadratic memory requirement of IP methods, decomposition methods such as as well-known SMO =ref= and SVM-Light =ref= work with dual variables α_i, constrained by a set of conditions that are derived from the current state of the solution, and thus change every iteration. In the extreme case, the active set consists of a single constraint. Therefore, the larger quadratic problem is segmented into a number of smaller ones, SMO emplying the smalles subset of just two variabes being optimized at a time. While algorithms of this family are simple to implement and have general asymptotic convergence properties, the time complexity of is still typically super linear in the training set size n.

Some of the decomposition methods can be adapted to the online learning setting, but the results are typically inferior in terms of convergence rate and accuracy to the methods specifically developed for the online setting.

Gradient-based methods Unconstrained gradient methods used to be common before the emergence of the more modern methods. While gradient based methods are usually known to exhibit slow convergence rates, the computational demands imposed by large scale classification problems of high dimension feature space, such as the ones common in image processing, has revived the theoretical and applied interest in gradient methods. Many of the online methods described below, as well as our proposed algorithm, were based on the modifications of gradient methods.

2.2.2 Boosting

Boosting is a name of a family of meta-learning algorithms that boost the performance of several low- accuracy claasifiers by combining them into a single classifier with increased accuracy. Usually, a linear combination of the classifiers' outputs is used, with the goal of the corresponding algorithm being the assigment of the weights to each classifier. In other words, given a set, or a pool of M classifiers $h_i(\vec{x})$, each with error ratio ϵ_i being arbitrarily close to a result of a random classification, 0.5, over a training dataset (the error ratio may be unknown), the boosting algorithm attempts to find a set of boosting coeffficients $\{\beta_i\}$ to form a confidence function $F(\vec{x}, \{\beta_i\}) = \sum_{i=1}^{M} \beta_i h_i(\vec{x})$, so that a classifier

$$H(\vec{x}, \{\beta_i\}) = sign(F(\vec{x}, \{\beta_i\})) \tag{2.5}$$

would achieve error rate below arbitrary threshold.

Boosting algorithms were originally an answer to the question posed by Kearns (=ref=, of whether a set of weak classifiers can be combined to form an arbitrary strong classifier. The proof of the possiblity delivered by Shapite =ref= has significant impact on the field of classification, and has lead to many related algorithms being developed. Amongst the most effective and popular is the AdaBoost, first introduced in =ref=, which shall be reviewed in more detail below, since it forms part of the basis of our proposed method.

2.2.3 AdaBoost

In this section, we shall briefly describe the AdaBoost algorithm for later reference. For the detailed derivation, please refer to te =ref=.

AdaBoost, short for Adaptive Boosting, is a greedy algorithm formulated by Yoav Freund and Robert Schapire, that can be used in conjunction with many other lesrning algorithms serving as a source for the set of the weak classifiers. AdaBoost is adaptive in the sense that subsequent weak classifiers added to the solution are tweaked in favor of those instances misclassified by previous classifiers. For this reason, AdaBoost can be sensitive to noisy data, however, it performs well on most datasets, and have been successfully used for the variety of tasks. Of the particuar interest to our work is its application to image processing and feature selection, deescribed in =ref=.

AdaBoost adds a new weak classifier in each of a series of iterations $t = 1, \ldots, T$. On each iteration, a distribution of weights D_t is updated that indicates the importance of examples in the data set for the classification. On each round, the weights of each incorrectly classified example are increased, and the weights of each correctly classified example are decreased, so the new classifier focuses on the examples which have been misclassified by the previous classifiers.

For binary classifications, the algorithm is given input data samples, $x_i \in X$, corresponding labels $y_i \in \{-1; 1\}$, $i = 1, \ldots, n$, and a family of weak classifiers $\mathfrak{H}$. It initiazises a weight $D_i = 1$ for each sample. Then, for each iteration, the algorithm proceeds as described below.

1. A weak classifier is selected from a provided family $\mathfrak{H}$ that minimizes the weighted error rate over the training dataset:

$$h_t = \underset{h_t \in \mathfrak{H}}{\operatorname{argmax}} \; |0.5 - \epsilon_t|$$

$$\epsilon_t = \frac{\sum_{i=1}^{n} D_i I(h_t(\vec{x}_i) y_i < 0)}{\sum_{i=1}^{n} D_i}$$

2. Set $\beta_i = \frac{1}{2} ln \left(\frac{1-\epsilon_t}{\epsilon_t} \right)$
3. Update for all i: $D_i = D_i e^{-\beta_i y_i h_t(\vec{x_i})}$. This step decreases the weigths of the successfully classified samples, and decreases the weight of misclassified ones.

After T iterations, the resulting strong classifier can be calculated by Eq. (2.5) Adaboost can be seen as a minimization of the

2.2.4 Other boosting algorithms

Boosting algorithms mainly differ in te way they esmimate weights β_i. Some of them prioritize misclassified examples, same as the AdaBoost, while others, like BrownBoost =ref=, attempt to increase robustness to noise and outliers by "giving up", and decreasing weights of the samples that has been repeatedly misclassified.

While our proposed learning technique uses AdaBoost as the basis for sample weighting, it is flexible enough to be easily adjusted to other methods.

2.3 Online classification algorithms

As mentioned above, many of the offline classification algorithms suffer from superlinear computational costs in the number of training samples. As the amount of training data increases, the training quickly becomes unfeasible. Also, with the increasing amount of common devices capable of data acquisition, such as mobile phones wit video cameras, etc, real-time classification tasks that allow for the adaptation to the incoming data stream and operate on a limited amount of data available dureing a single frame become more and more relevant. This is one of the reasons for the development of our proposed algorithm and sample application.

In this section therefore, we introduce the algorithms specifically developed or modified for the online setting, that can be used as the replacement of the algorithms described above.

2.3.1 Online SVM

In this section, we review two algorithms for online SVM training that have a direct bearing on out work. Both of these algorithms use a variant of the Stochastic Gradient Descent in order to upadate the solution on each iteration.

2.3.1.1 NORMA

NORMA =ref= is a generic method for onilne risk minimization using stochastic gradient descent, with a particular focus on its application to kernel-based Support Vector Machines and regression. In the founding paper, Kivinen et.al. both describe the method and give theoretical bounds on its accuracy and convergence rate. They show that the convergence rate of NORMA is independent of the size of the dataset, if a limited dataset is used, and is instead of the order $O(\frac{X}{\epsilon^2})$, where X is the bound on the absolute value of the kernel function, and ϵ is an error rate. They also show that the truncation error resulting from removing older kernel expansion vectors decreases exponentially with the number of preserved vectors-coefficient pairs, as long as the kernel fuction values over the space of input data samples are bounded.

The agorithm itself attempts to minimize the primal formulation of an SVM problem (Eq. (2.4)) rewritten to allow the usage of the kernel trick. To do that, the concept of Reproducible Kernel Hilbert Spaces (RKHS) with defined inner product is used to replace $\vec{w}$ of the linear SVM with the function f int the RKHS $\mathscr{H}$ defined by kernel $k(\cdot,\cdot)$

$$\min_{f\in\mathscr{H}} \left\{ \frac{\lambda}{2}||f||^2_{\mathscr{H}} + \frac{1}{n}\sum_{i=1}^{n} l(f(\vec{x}_i, y_i)) \right\} \tag{2.6}$$

where $||f||^2_{\mathscr{H}} = f \cdot f$ However, in the online setting, assuming that a single data sample $\vec{x}_t$ and label y_t are provided on the iteration t, and other data samples unavailable Eq.

(2.6) instead takes the form of instantanious risk:

$$\min_{f \in \mathscr{H}} \left\{ \frac{\lambda}{2} ||f||^2_{\mathscr{H}} + l(f(\vec{x}_t, y_t)) \right\} \tag{2.7}$$

The above expression is then minimalized by utilising a simple subgradient descent (gradient if the function l is differentiable) with a learning rate η_t, i.e., on each iteration the function f is updated:

$$f_{t+1} = f_t - \eta_t \partial_f \left(\frac{\lambda}{2} ||f||^2_{\mathscr{H}} + l(f(\vec{x}_t, y_t)) \right)$$

which can be simplified to

$$f_{t+1} = (1 - \eta_t) f_t - l'_f(f(\vec{x}_t, y_t) k(\cdot, \vec{x}_t) \tag{2.8}$$

Since as long as the kernel used satisfies Mercer's condition, function f can be represented in the form already mentioned above, i. e., on the iteration t

$$f(\vec{x}) = \sum_{i=1}^{t} \alpha_i k(\vec{x}, \vec{x}_i) \tag{2.9}$$

Substituting Eq. (**??**) Fexpansion) into Eq. (2.8) , the final formula dealing with the updates of α_i becomes:

$$\alpha_t = -\eta_t l'(f(\vec{x}_t, y_t) \quad i = t$$

$$\alpha_i = (1 - \eta_t) \alpha_i \quad i < t$$

Here one of the drawbacks of the kernel bases-methods becomes obvious, since it can be seen that on each iteration an additional kernel expansion vector is added, quickly increasing the storage requirements.

2.3.1.2 Pegasos

Pegasos algorithm, introduced in =ref=, is conceptually similar to the NORMA, with two key differences, one being that it incorporates an additional parameter k, which denotes the number of input samples accepted at a single iteration, and averages the loss function in Eq. (2.7) over k. The other key difference is the additional scaling step after each iteration, which drastically increases the convergence rate.

$$f_{t+1} = min(1, \frac{1}{\sqrt{\lambda ||f_{t+0.5}||^2_{\mathscr{H}}}}) f_{t+0.5}$$

Using these steps allow Pegasos to effectively use agressive update schedule $\eta_t = \frac{1}{t}$, and achieve conergence rates proportional to $O(\frac{1}{\epsilon})$. Unlike NORMA, pegasos is primarily oriented toward linear SVM optimization. This fact, combined with the increased convergence rate and simplicity of implementation lead to us adopting it over NORMA as a basis of our method.

2.3.2 Online boosting

Online boosting algorithms, such as the ones presented in =ref=, =ref=, are a modification of AdaBoost approach with the exact error rate of the classifier ϵ_t being replaced with the online estimate $\tilde{\epsilon_t}$ that is updated on each iteration. In essense, they propose rerunning AdaBoost algorithm on limited subsets of weak classifiers called selectors on each iteration. Since this method is not essential to the understanding of our proposed algorithm, we shall refrain from desribing it in further detail here.

2.4 Overview of applications

Below are several examples of the practical applications of the described algorithms. While the applications for large scale and online SVM learning and boosting are spanning most of the areas of modern research that deal with large amounts of data, the examples presented below are most relevant to our study topic.

- **Object recognition**

  As mentioned in =ref= and briefly described in =ref=, the above mentioned methods or theor derivatives can be used to quickly and adaptively organize various image features (HOG in the case of =ref=, or an ensemble of simple features in =ref=) to reliably detect certain objects. The adaptive nature of such algorithms allows the detection to remain stable even for an object with changing form.

- **Object tracking** Similar to above, only in this case the original position of an object is given beforehand and has to be updated each frame as it moves.As shown in =ref=, online boosting for simple features, in particular, seems to be well suited for such a task.

- **Text classification** Text classification, i.e. the task of defining whether the text belongs to a certain cathegory or not, often has to be updates online, as the user inserts new texts or updates old labels. The online SVM-based methods are well suited for such tasks both due to adaptability and their ability to process large amounts of data easily. Due to near-linearity of many text classification based tasks, Pegasos is better suited to such applications than Norma.

- **Advertisement selection** Once again, a modification of the above mentioned text classification, this application deals with determining whether a certain link is relevant or not to a particular user based on his history of previous searches and network surfing.

3

Description of the proposed two-step algorithm

3.1 Derivation of the algorithm

In this section, we describe in detail the way our proposed algorithm was conceived and derived.

3.1.1 Drawbacks of the existing online algorithms

The method described in chap. 2 represent, in a certain way, two extremes of a spectrum. On one hand, the SVM training algorithms like Pegasos and NORMA may be used for either linear classification, which is fast and has low memory requirements, but has the drawback of being of limited utility, since the majority of data distributions in the natural datasets is nonlinear, or nonlinear kernel-based classification, which is imprecise in the case the optimal kernel for the data distribution is unknown, and, even in the optimal case, often exhibits growth of computational complexity by accumulating kernel expansion coefficients, that is roughly proportional to half the the processed data (see =fig==). The reason for this growth is partially explained by the fact that the provided data points cannot perfectly describe distribution of data in the transformed feature space, and thus the algorithms has to compensate with the larger number of samples to provide approximation (see =fig=). In this way, the kernel versions of the SVM training algorithms represent algorithms with unlimited complexity growth over time.

On the other hands, online boosting algorithms presented in several sources (for example, see =ref=, =ref==, etc) usually adopt the approach of a fixed complexity opposite to that of the offline boosting by fixing the number of classifiers being added to the model at the beginning of the training, which limits the possibly accuracy. They are also limitied by the need to estimate and update a large number of pre-selected classifiers on each iteration, although this problem is somewhat mitigated by the simplicity of the classifiers.

Our goal in this paper is to develop a middle-ground classifier that can provide adjustable levels of complexity growth or decay depending on the needs of the application. For that reason, we took a look at a structure of a single AdaBoost update and have adapted it to the online setting by using SVM training as a method of choosing the next classifier.

3.1.2 Similarity of AdaBoost and linear SVM

We shall start the explanation of our algorithm by noting the similarity between a classifier resulting from a linear SVM training and strong classifier of AdaBoost (this similaity being analyses in more detail in =ref=). It is easy to see, that Eq. (2.2.1) and Eq. (2.5) are identical, both being the sign value of a confidence function, which in turn is a linear combination of an input. The main difference is in the fact that in case of boosting the original inputs $\vec{x}_i$ were transformed by a set of classifiers to a kind of bimary-valued feature space. However, one cannot assume that this makes boosting completely equivalent to the kernel-based SVM, since the *sign* function, as well ass sigmoid function, the extreme case of which it is does, in fact, not satisfy Mercer condition, this fact proven in =ref=. This makes direct application of kernel SVM-related methods unpredictable. However, if we were to treat a vector of weak classifier outputs $h_t(\vec{x})$ as a kind of input vectors to the linear SVM classifier, we could use online SVM adaptation to iteratively change the weights β_i in the Eq. (2.5) .

3.1.3 Adapting AdaBoost to online setting by using modified Pegasos algorithm

AdaBoost iteration consists, essentially, from the two parts - the selection of the weak classifier $h_t(\vec{x})$ minimizing error in respect to the weights provided by the iteration's strong classifier, and selection of the weight for that classifier.

In the online setting, however, we do not have the option of evaluating each datapoint for each classifier to determine the optimum, although error values on the unweighted dataset can be estimated simply by adding the error of each consecutive iteration. It is also not possible to reach back and reevaluate data points when the weights of the classifiers placed earlier in the AdaBoost iteration chain change due to the changed accuracy estimate. The solution to this problem employed in =ref= is to both fix the number of classifiers and to separate them into unrelated sets called selectors (although the latter requirement is relaxed in the later articles =ref=), resulting in the totality of classifiers in each selector to slowly adapt their accuracy values according to their position in the booster chain.

Our solution, however, is to add a single classifier (linear SVM) at a time and then train it according the the weights provided by the current strong classifier, which is also being trained to better fit the global data distribution. This training can be achieved in the variety of ways, but the similarity between the Adaboost and the SVM described in the previous section has lead us to consider one of the online SVM training algorithm for adjusting the boosting weights. Given the simplicity and the higher convergence rate we have chosen Pegasos as our basis training method.

The online setting also naturaly raises the question of calculating the weight of each incoming data point for training the additional classifier. To answer this, we consider the weights provided by the AdaBoost algorithm at a given iteration t

$$D_{i,t} = e^{-\beta_1 h_1(x_i) y_i} e^{-\beta_2 h_2(x_i) y_i} \dots e^{-\beta_t h_t(x_i) y_i} = e^(- y_i F(\vec{x_i}))$$

That is, the weight for each data point is a constant e to the power of the confidence function of the currently available multiplied by the opposite of a given label, which results in a positive value in case the current version of the strong classifier misclassifies the sample and negative if the available classification if correct. Since in our algorithm the we have constant access to the confidence function of the updated strong classifier, the calculation of the weight is relatively straightforward. Several loss functions can be used, including exponential function as in AdaBoost, or the hinge loss function (Eq. (2.2.1)) common to SVM, with varying results, which will be explored later in more detail.

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

Figure 3.1: The Pegasos algorithm with weighted samples. Parameters: $\vec{w}$ - weight vector for the linear SVM, λ - regularization parameter, $\vec{x}$ -input training sample, y - training label, t_w - number of current iteration for weak classifier, c - weight parameter

3.1.4 Linear SVM as weak classifiers

The last step in defining our algorithm is the definition of the weak classifier. In our work, we use a linear SVM for that purpose. Even a random linear classifier would provide an error rate differing from random classification value of 0.5, in all but extremely rare degenerate cases, which is the only requirement for the weak classifier in AdaBoost (see =fig= for illustration).

The accuracy of the SVM is then improved by it being trained online by the same Pegasos algorithm as the global classifier with a single difference being an adaptation to the weights provided by the global classifier. Pseudocode and required parameters for the weighted for Pegasos iteration are shown on fig. 3.1. It should be noted that this is not the only way to incorporate weights into the online update algorithm. For example, =ref=, as well as =ref= propose using several updates in a row with the number being drawn according to the poisson distribution. However, we find our adaptation much simpler, and sufficient in terms of accuracy.

3.2 Description of the resulting algorithm

3.2.1 Algorithm description

The proposed algorithm can the be summarized as follows:

- It takes 3 parameters, λ, λ_H, and parameter r regulating the complexity growth.
- On each iteration, it takes the following inputs: number of iteration t, the number of iterations the weak classifier has been trained t_w, $\vec{x}$, y, and the outputs of the

previous iterations: vectors $\vec{w}_k$, $k = 1 \cdots K$, K being the number of SVM serving as weak classifiers already incorporated into a strong classifier, and $\vec{\beta} = \beta_1 \cdots \beta_K$ as boosting coefficients for weak classifiers. On the first iteration, $\vec{w}_1 = \vec{0}$, $K = 1$, $\beta_1 = 1$ that is, the classifier being trained is considered incorporated into a strong classfier.

- Outputs of weak classifiers are calculated and accumulated into a vector $\vec{h} = h_1 \cdots h_K$, $h_k = sign(\vec{w}_k \cdot \vec{x})$
- Confidence function for the resulting global classified is calculated and used to estimate the weight for the training of weak classifier: $F(\vec{h}) = \vec{\beta} \cdot \vec{h}$, $c = l(F(\vec{h})$, l defined as a hinge-loss function (Eq. (2.2.1))
- Weak classifier currently in training (defined by vector $\vec{w}_K$) is updated according to the algorithm illustrated on fig. 3.1.
- Vector β of the strong classifier is updated using Pegasos algorithm without weight modification, using the calculated confidence value F and regularization parameter λ_H.
- Parameters t and t_w are increased: $t = t + 1$, $t_w = t_w + 1$.
- If $t_w > r$, the values of $\vec{w}_K$ are fixed and a new weak classifier is initialized: $K = K + 1$, $\vec{w}_K = \vec{0}$, $\beta_K = 0$. Since the classifier is initialized with 0, it does not start to affect strong classifier H until the next update.

Furthermore, to increase the flexibility of the resulting algorithm, both the weak and the strong classifiers are assumed to incorporate a bias term into an input vector, according to the method described in =ref=, that is, both $\vec{x}$ and $\vec{\beta}$ are assumed to have an additional element equal to 1.

From the above definition, it is easy to see that at each iteration our algorithm has a string classifier ready, same as =ref=. The additional parameter r is used to control the complexity and non-linearity of the output, although the experiments have shown that for most non-linear application, the constant value of r about 100 is sufficient. Higher values of r result in a fewer number of better-trained classifiers combined into a strong classifiers, limiting possible task complexity, while lower result in the weak classifiers being closer to random and increase complexity without appreaciable incease in accuracy for most datasets.

The result is an algorithm with controllable storage and computational requirements that can be used for online training of strong classifiers on nonlinear data distribution without the use of kernels, and therefore without the need to adjust or fine-tune kernel parameters. Experiments in section =secref= show tha a constant value of parameters works well enough for most complex datasets, proving the simplicity and large applicability range of proposed method.

The convergence of our meethod on a sample 2-dimentional dataset is illustrated on =fig=

3.2.2 Possible modifications

The above algorithm can be modified in several ways to adapt for a specific application requirements or for increased flexibility

Different loss functions for weak classifier training. The function l used to calculate weight for weak classifier training can be changed to, essentially, any monotonically increasing function. Several loss functions popular in optimization are shown on =fig=.

Removal of weak classifiers While our algorithm allows for accuracy levels comparable or higher to that of kernel-based SVM, with much lower computational complexity, computational and storage costs of our algorithm still grow with input samples. One way to stop this growth is to remove those weak classifiers absolute values of coefficients for which fall below a certain threshold, resulting in eventual replacement of ill-fitting classifiers. This technique also increases the flexibility of the method

Limiting the global iteration number t Since the learning rate of the algorithm decreases proportionally to teration number t, for applications that deal with constantly changing data distribution it is beneficial to limit the growth of the number t, stopping the increase at certain threshold corresponding to the level of flexibility desired.

Varying number and kind of weak classifiers trained simultaneously Since the trainer for strong classifier is unaware of a kind and number of weak classifiers being trained, different features may be added when each new weak classifier is initialized, and several distinct classifiers may be trained at once, especially if the setting favors parallel processing. This particular modification is discussed in more detail in the next section.

3.2.3 Comparison to other algorithms

Online SVM trainning methods (NORMA and Pegasos) Our methods compares favorably to both NORMA and Pegasos in case of nonlinear data distribution, since they allow for much lower computational and storage requirements. If, however, the parameter r is set to values near 1, our method is reduced to, essentially, fitting a set of random classifiers to data distribution, with the storage and computational requirement approaching those of kernel methods. However, the extimated computational requrements are still somewhat lower that for most commonly used kernels due to the simplicity of the sign function.

For the case of linarly separable data, our algorithm is essentially indistinguishable from Pegasos, on which it is based, with additional overhead due to added classifiers.

Online boosting methods It is difficult to compare our method to the online boosting methods due to the difference in methodology. In many ways, our methods are quite opposite. Methods proposed in =ref= apply learning updates to the preselected set of classifiers based on random features, while our method updates usually only one classifier per iteration. Also, their algorithm assigns weights to classifiers in a manner similar to Adaboost, all at the same iteration, whle ours adjusts the weights iteratively over time.

4

Application of proposed method to object tracking on a mobile device

Recently, more and more powerful mobile devices become publically available. Every next iteration increases the CPU speed and memory available. However, there is still a significant dearth of resources on the mobile devices as compared to the desktop machines, especially when it comes with the CPU power. Yet, many consumer-level devices start to include a graphical processing unit that is capable of rapidly performing parallel programming. In this section, we show how our learning method canbe implemented on the mobile device for the challenging purpose of object tracking. The simplicity and speed of our method, combined with original simple features proposed in the section, allow us to develop a tracking algorithm that performs in near real time. This shows the feasibility of using out method for real-world apllications in image processing.

4.1 Resources available on the mobile device: iPhone 4S

4.1.1 Overview

As our target mobile device, we have chosen the latest iteration of the popular iPhone brand: iPhone 4S. Our choice was partially guided by the fact that the device itself, and the means to program it were readily available and the proggramming lay well within our area of expertise. Much more important, however, was the possibility of harnessing the power of GPU, thereby increasing the speed of the parallel computations.

iPhone 4S is a touch-sceen based smartphone released by Apple in the fall of 2011. It's design is similar to that of it's predescessor, iPhone 4 (=fig==). Its processing capabilities consist of the Aplle A5 system-on-a-chip, that contains a dual-core ARM Cortex-A9 MPCore CPU, that includes NEON SIMD coprocessor for vector operations on the floating-point values, running at 1Ghz, image signal processing unit, performing such operations as face detection, white balance and image stabilization, and most importanly to us, an Imagination Technologies PowerVR SGX543 dual-core graphics processing unit. Unfortunately, the direct access to the ISP unit is limited to the system software, so it cannot be used or programmed to increase image processing speed, leaving us to rely on GPU only.

This phone demonstrates the increasing speed with which consumer-level mobile devices approach the capabilities of desktop computers. In fact, this a starling example of the Moore's law, since the processing capabilities of iPhone 4S are nearly double of its predescessor of only a year prior.

Latest versions of the iPhone system software (iOS 4 and 5) support the OpenGL 2.0 graphics library, which removes the fixed graphics pipeline, and introduces programmable vertex and fragment shaders, which allow not only a variety of imaging and video effects, but also, to some extent general data processing in parallel.

4.2 General-purpose computing on graphics processing units

General-purpose computing on graphics processing units (GPGPU) is the means of using a graphics processing unit (GPU), which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the central processing unit (CPU). GPGPU requires a specific combination between hardware components and software that allows the use of a traditional GPU to perform computing tasks that are extremely demanding in terms of processing power. Different GPU have different capabilities in that regard, with the GPUs for desktop video card allowing more flexibility and even the usage of a programming language developed specifically for GPGPU, while GPU on the mobile devices still offer only a limited programming capacity.

General purpose computing on the GPU has greatly benefited from the new architectural approach GPU manufacturers have taken in their latest GPUs. In specific, these new GPU architectures come with a high grade of programmability, something not previously found in older generations of GPU architectures. Thanks to its broader range of programmability, the graphics processing unit has now been opened up to many kinds of applications and code.

Although GPGPU attempts were made in the past, they weren't very successful primarily because previous GPU architectures were very difficult to program for parallel processing.

The effectiveness of the GPU useage largely depends on application. Tasks that are similar to graphics processing, that is, massively paraller, with low to none interdependency between the values being calculated. Usually, a cerain amount of interaction between CPU ang GPU is beneficial to the program's performance, however, one must always keep in mind that on most systems transfer of data between CPU and GPU memory is very slow, creating a programming bottleneck.

As we have already established, GPUs are particularly well suited for parallel processing, a situation typically found in image processing, generic patterns analysis, search for oilfields and natural resources, and analysis of financial risk calculation patterns. On the other hand, using GPUs for databases, data compression, recursive algorithms, and processes that require a high logical control of calculation is not ideal. A traditional CPU architecture would be far more efficient in this situation.

Unfortunaltely, the low-power devices like iPhone do not benefit from the advantage of specially developed GPGPU languages, like CUDA or Cg. This means that we have to explore the programmable capabilities of the shaders in the graphics pipeline, while dealing with severe limitations described in detail below.

4.2.1 Open GL ES programmable graphic pipeline

OPenGL ES is a verstion of OpenGL graphics library for use on the mobile devices. Starting from OpenGL ES 2.0, it has received a complete overhaul, and for the first time has enabled usage of programmable graphics pipeline on the mobile devices. While before, only some parameters of the graphic processing were available to change , OPenGL

ES 2.0 added the ability of adding small programs called shaders (due to their original usage for achieving various shading effects), that are executed on GPU at key points during the graphic rendering process. The graphic processing pipeline is illustrated on =fig=

More specifically, ES 2.0 standard allow the usage of the so called vertex and freagment shaders. Vertex shaders are programs that are called once for each reddered vertex, and allow certain parameters to be interpolated between vertices on the same primitive. When applied for general calculations, they are generally used to set up environment for the fragment shader.

A fragment shader is called once per fragment, which usually corresponds to a single pixel on the screen or texture. Therefore, this is the shader where the majority ofthe parallel processing takes place. However, there are several limitations to the usage of the shader program, not the least of which being that a single shader can ony produce a single 4-byte output as of OpenGL ES 2.1.

However, as can be seen from the illustration of the GPU architecture =fig=, the advantage in using shaders lies in the fact that a large amount of them are executed at the same time, significantly reducing computational times for similar operations.

4.2.2 GPUImage programming library

In our work, we use GPUImage programming library for iOS, written in Objective C. While in general, attaching and switching shaders is a relatively complex programming task due to an amount of auxillary OpenGL code, the use of this library allows us to simplify and streamline what would otherwise be nearly impossible to program by exposing a simple and user-friendly API.

The GPUImage framework is a BSD-licensed iOS library that lets you apply GPU-accelerated filters and other effects to images, live camera video, and movies. This library is available as an open source project on Github, and allows us to express image processing tasks as a series of completely customizable filters that can be written using the OpenGL Shading Language. In our work, we develop several filters specific for our application, and also make use of several simple filters that are already available.

In particular, we make use of the following built-in filters:

- GPUImageBlendFilter

- GPUImageCropFilter
- GPUImageFastBlurFilter
- GPUImageRotationFilter

Also, we make use of the video input exposed by the provided API.

4.2.3 Limitations

In this section, we outline several limitations of the OpenGL ES shader system, as opposed to the GPGPU system available on desktop PC, and our adjustments to them.

- **Lack of support for the floating point textures and limited precision**

  The most important limitation is the fact that all highly parallel inputs and outputs in OpenGL ES are supposed to be textures, i.e. 2D arrays of data coded in, 4-byte vectors representing color. While desktop OpenGL allows the use of floating point textures, where the 4 bytes represent a single floating-point number of high precision or 2 of lower precision, OpenGL ES lack this feature, necessitaing either recoding of incoming and outgoing data, which results in severe slowndowns rendering application useless, or working with low-precision data.

  This precludes the GPU computation and usage of such tools as the integral image, which lead to the fact that such image features as Haar-like features and histogram-based features cannot be computed effectively.

- **No support for preloaded render calls**

  When it is necessary to perform a sequence of operations on the GPU, especially such series in which every consecutive step utilizes data output form the previous one, it would often be beneficial to store the sequence on the GPU before execution. In OpenGL ES, we, hovewer, have to wait for each step to finish and inform CPU before starting the next one, significantly impacting performance. The only solution to this is to minimize the number of such operation series.

- **No procedures available for linear (non-2D) arrays**

  Once again, all operations on the mobile GPU are performed with textures sserving as a data sotrage. In addition to the precision problems, there is little to no support for large one-dimantional arrays, necessitating packing such arrays into 2D form, thus adding unnecesary operations in the shader code.

- **Opaque memory layout** On the desktop, the layout of the memory available to each shader is well known, allowing to perform certain optimization techniques improving performance by efficient memory caching. In OpenGL ES, memory layout is more or less completely opaque.

- **Insufficient tools for synchronization**

  As was partially mentioned in the above point, possibly the only tool available for the synchronization between data operation in OpenGL ES is waiting for all GL render calls to complete. This decreases efficiency drastically.

4.3 Modification of Pegasos algorithm for parallel processing

In this section we describe the modification to the Pegasos algorithm described in secref for more efficient parallel processing, and compare it to the original.

4.3.1 Modification description

The original Pegasos algorithm was designed for sequential or batch-sequential updates, i.e. only allowing for a single or limited number of input samples $\vec{x}, y$ per iteration. In the image processing task, however, we usually have the situation where a large number of input samples, possibly one for each pixel, is available at the same time when the frame is processed. One solution would be to just use all these inputs as a single batch, but that would decrease the efficiency of the algorithm.

Either way, for estimating the Pegasos update it is necessary to add the weighted value of misclassified sample together, which, while not a simple task for parallel processing, is relatively well developed =ref=. Therefore, based on the common algorithms for the parallel summations, we have developed a modification of the Pegasos algorithm which can be called Pyramidal Pegasos (=fig=). It uses the simple hierarchial scheme for summation, but in each step replaces summation with the following steps, assuming that each shader program has access to a sample vector and label pair $\vec{x}_i, y_i$, and a pair of weight vectors $\vec{w}_i$,$\vec{w}_{i+1}$ from the previous level(initialized at $\vec{0}$ on the first level):

1. Average the weight vectors $\vec{w}_o = \frac{1}{2}\left(\vec{w}_i + \vec{w}_{i+1}\right)$
2. Perform Pegasos update iteration (normal or weighted) on the vector $\vec{w}_o$ with $\vec{x}_i, y_i$
3. Output updated $\vec{w}_o$ to use in the next level

The result of the algorithm is a single vector of weights $\vec{w}$. It is easy enough to see that the amount of Pegasos updates performed is two times larger than in the case of sequential updates, but this expense is easily enough offset by the parallel processing gains. It is also possible to eliminate the first layer and distribute the data samples over layers , reducing the total number of updates to the number of input data samples.

4.3.2 Evaluation

In this section, we give a brief evaluation of the Pyramidal Pegasos modification, comparing its performance to the original Pegasos in order to determine whether we can use it in our application. For simplicity, we only evaluate the convergence on the linearly separable dataset. The results are shown on =fig=. It can be seen that the convergence rate of this modifications is similar to the original algorithm. Unfortunately, the actual implementation of the algorithm on GPU with the use of OpenGL ES programming framework has shown us, that due to a large overhead associated with additional rendering calls for each pyramid level, the speed of the algorithm suffes in comparison to the CPU-only implementation. This drawback, however, is not present on the desktop GPU. Therefore , for the current version of our testing application, we have decided to use the standard sequential algorithm. However, we still consider this algorithm to be useful learning tool for the case of better optimized parallel implementation.

4.4 Simple local image features

Due to the limitations of the OpenGL ES archtecture and high computational cost, extraction of the global image features, even the simple features like Haar-like features =ref= of local binary patterns(LBP) over large image region is unfeasible on the iPhone for any usable size of the region of interest, we had to develop our own set of fetures to estmiate the feasibility of using outlearning algorithm for image processing tasks.

The features that we use in our experiments a partially based on the same idea as Haar-like features and LBP, but extremely simplified to minimize the number of texture fetches necessary. Essentially, where their features deal with image regions, ours mainly deal with singular pixels, possibly afer passing the image through gaussian filter for increased stability. We show, that even when using such features, our algorithm can provide a remarkable level of distinctivenes between image regions.

The feature vector for a single randomized feature is collected as follows (=fig=):

1. For each feature, two random offsets $\vec{\Delta}_1$, $\vec{\Delta}_2$ are chosen
2. For each pixel with coordinate $\vec{p}$ in the ROI, luminance values of the pixels with coordinates $\vec{p}+\vec{\Delta}_1$ and $\vec{p}+\vec{\Delta}_2$ are compared, and the result of comparison added as a feature
3. r, g, normalized color values for pixels at coordinates $\vec{p}$, $\vec{p}+\vec{\Delta}_1$ and $\vec{p}+\vec{\Delta}_2$ are added to the feature vector.

As a result a single feature simply gives information of the color values of three pixels arranged in a certain configuration, and indicates whether one of the pixels in brigther than another, possibly indication an edge.

On its own, as single feature vector like that does not have much discriminative power, however, as we will show later, when several such features are combined using our proposed learning methods, they become capable of distinguishing a wide range of objects, although the performance is still inferior to the more robust global features.

4.5 Tracking

In this work, we evaluate the applicabity of the features desribed above combined with our learning method to image processing tasks. As an example task, inspired by =ref=, we have chosen simple object tracking.

As shown in =ref=, object tracking on a n image can be cast as a binary classification problem over a region of interest, separating an area containing object (usually a bounding box rectangle) from all other bositions in the neghborhood. Some kind of movement model may be used to constrain the search area in order to reduce the amount of necessary computations. Also, since object changes from frame to frame, online adaptation can be used to update object model, as described in =ref=. Since our model is geared towars online learning, such updates are also possible, and theeir effect is investigated below.

4.5.1 Training regime

First, in order to separate object position from the surrounding background, an object model in the feature space have to be trained. Usuallym this training is assumed to be done offline, afyer the original object position is . In our case, since we don't have a specific object, an area in the middle of the screen is trained as an object to be tracked. A ROI around the object position is selected, and a large amount of features are extracted and passed through a trainer with high lerning rate set. Several pixels in the center of ROI are assumed to have label value of 1, i.e. are assumed to be the true position of an object being trained, while all other pixels are assumed to have label value of -1.

The large amount of negative samples compared to the positive ones creates an extremely imbalanced data set, which negatively affects the results of the learning algorithms based on gradient descent, like ours. In order to balance that, the positive samples are fed into algorithm multiple times.

For this experimental application, we use a multiple feature modification of our algorithm described in =secref=, with each new featue being randomly generated, as well as the removal of weak classifier due to the computational constraints of the mobile device.

4.5.2 Position estimation

After the initial model is trained, the application starts tracking the trained object, as well as continuosly updating the model. In order to calculate the position of the object, we simpy apply thresholding to the confindence function in the ROI, and calculate weighted average position on the remaining values. If all values are below threshold, the object is considered lost. This simple algorithm works resonably well for many cases, although it has several drawbacks.

4.6 Resulting Application layout

The diagram on =fig= shows the resulting layout of the filters used by experimental application. The layers used are, in order

1. **Input** An input texture uploaded from the iPhone video camera, at the resolution 640x480
2. **Crop filter** Input image is cropped to an expaned region of interest.
3. **Blur filter** Cropped image is blurred by a low amount of gaussian blur to increase stability.
4. **Color normalization filter** The RGB values of each pixels are normalized to rgl, $l = \frac{R+G+B}{3}$, $r = \frac{R}{3*l+c}$, $g = \frac{G}{3*l+c}$, where c is a small constant to prevent division by zero.
5. **Feature sampler filter** A set of two filters sampling feature vectors from cropped image for the training of the current weak classifier.
6. **Weak classifier filters** A set of filters equal to the number of weak classifiers in the models that combine feature extraction and classifier evaluation.
7. **Weak classifier extractors** A set of filters extracting calculated classifier values for training of the strong classifier.
8. **A training block** Due to high accuracy demands exhibited by trainer, and GPU limitation, the actual trainings performed on CPU. With feature extraction and weak classifier estimation being performed in parallel, the computetional demands of the algorithm itself are extremely low (more than 100K samples can be processed per second by our measurements).
9. **A multi-texture blending filter** Unlike other filter,this one has only a simple passthrough shader, instead achieving summation of the classifier outputs from the previous stage by enabling built-in OpenGL blending and rendering all the input texture onto a single output.
10. **Position estimate block** This block uses output of the blend filter and claclulates the position update of an object. Uses CPU.

11. **Rectangle filter** This filter simply add the rectangle on the estimated object position.
12. **Output** Output may be in form of the video file (movie encoder) or a screen output, or both. Since the CPU in this case is largely free of calculation, there is more than enough processing power for video compression.

4.7 Results overview

Some of the results of running sample application on the iPhone and attempting to define (segregate) and track various objects are illustrated on figure =fig= The following conclusions can be drawn:

- The proposed features, while sufficient in many cases to separate object from the local background (=fig=), are not very robust due to the fact that they employ data from a low number of pixels which may be noisy. They are also sensitive to scale and rotation. This sensitivity, combined with the fact that it is impossible to hold the phone absolutely steady while acquiring a target object, lead to the lower overall accuracy.
- Even with such limitations imposed by features and precision provided by GPU, the proposed learning algorithm can efficiently organize simple features to achieve object detection for the purposes of tracking, although the tracking is not very accurate (=fig=).
- The resulting algorithm is efficient enough to run in near real-time (12-17 fps) with the number of weak classifiers capped at 15.
- However the simple position estimator suffers from the drawback of losing the tracked object when the movemet exceeds its search area, while expanding search area leads to reduced efficiency of the classifier, since it has to offset the larger number of errors.
- The bottleneck of the algorithm appears to be a number of selectors, growing with the number of features. Repeated data transfers fro GPU to CPU memory, coupled with multiple rendering operations per selector reduce performance drastically. Removal of the need for them should nearly double the frame rate.

- The learning algorithm is simple and efficient enough to bealmost unnoticeable in terms of computational and memory requirements when compared to feature extraction and especially selectors.
- The usage of online adaptive algorithms for tracking can actually decrease the tracker's quality, a fact mentioned in =ref= due to an effect called drifting, where a classifier essentially retrains itself to accept other objects by accumulating errors. Due to high adaptability of our algorithm on limited number of weak classifiers, and high sensitivity of the features used, this effect is quite prominent unless care is taken to avoid it.

To conclude, while the proposed experimental application clearly shows the merit of our learning method as applied to the image processing tasks like object recognition and tracking due to its simplicity and efficiency, the application itself is of limited practical value due to several unresolved problems in implementation. As such, it can be considered a proof-of-concept application rather than a final product.

5

Evaluation of our algorithm in comparison to related algorithms

5.1 Comparison to online SVM training algorithms: Pegasos and NORMA

We compare our algorithm to both Pegasos =ref= and Norma =ref=, 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 and robustness) .

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 =ref=, and also used for comparison of convergence speed in =ref=. 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 to create a clear margin.

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 (**?**), 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. This experiment estimates the ability of the algorithms to adapt to gradual changes in the data distribution.

5. *Bayes-separable data with switching distribution (Switching)* Once again, a dataset generated according to the description in (**?**), with the distribution changed drastically every 1000 iterations. This experiment shows the ability of the algorithms to completely relearn a distribution.

For each distribution we measure the decrease of estimated error rate over training dataset (estimated error being simply the running average of the number of misclassified training samples divided by number of iterations in the averaging window), and the resulting error rate over the testing dataset (generated without noise in case of noisy distribution). In case of the dataset with the changing distribution, the distribution at the last iteration is used for testing.

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, which makes the resulting kernel an ideal case for separation.

5.1.1 Experimental results

The graphs for the estimated error rate are shown on figure 5.1 for linear implementation of NORMA and Pegasos, and on figure 5.2 for the kernel implementation, while the resulting error rate on the test datasets is shown in Table 5.1. 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. **??**) our

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 5.1: 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

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, demonstrating stability of the method to condition changes.

For the Covertype dataset, linear classifiers work best, and approach the error rates indicated in the paper((**?**)), 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. **??**, 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, i.e, both the memory and computational requirements (per classification) were less by a factor of around 75.

5.2 Comparison to offline algorithm: AdaBoost

In the interests of fairness, we also compare our algorithm to the AdaBoost in the offline setting. Since our algorithm is designed for online usage, we simulate the onile environment by sequentially feeding it data samples randomly selected from the training dataset. Since a single iteration of the AdaBoost results in an additional weak classifier being added, while our algorithm takes r iterations to do the same, to obtain equivalent conditions for testing we only evaluate the accuracy of our algorithm every r's iteration. Also for equivalence, we use a pool of M randomly selected linear classifiers for weak classifier selection.

It should be noted that the above condition does not mean that our algorithm is more computationally intensive, since to choose a classifier for addition the AdaBoost algorithm has to calculate weighted error rates of every classifier in the selection pool to select a maximum, or use some optimization algorithm to obtain best classifier in the case of continuous classfier pool, resulting, in the discrete case, in MN classifier evaluations, where N is the number of training samples. Our algorithm, however, only performs rK evaluations, with $r \ll N$ and K being the number of classifiers already added. In fact, in the online case our algorithm is similar to AdaBoost evaluating on the random subset of the classifier pool on the random subset of training data.

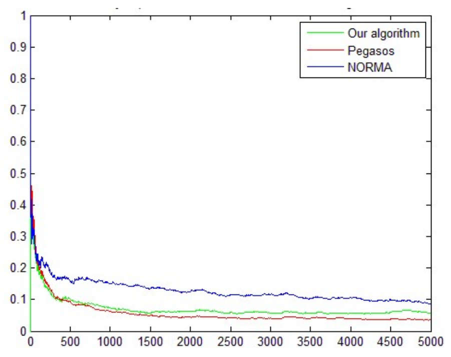
We evalulate the methods on two datasets, one being a synthetic nonlinearly separable dataset, and the other being the Abalone dataset from the UCI database, originated by =ref=, with classes 10-29 merged into a positive class and other classes being negative. The results of generalized error on the number of classifiers added are presented on =fig=. They clearly show, that while on the offline setting AdaBoost outperforms our algorithm, the incease in the convergence speed is not very large. This advantage is due to the fact that our algorithm only selects near-optimal classifier on each addition, while AdaBoost uses the optimally selected one.

5.3 Comparison to the online AdaBoost

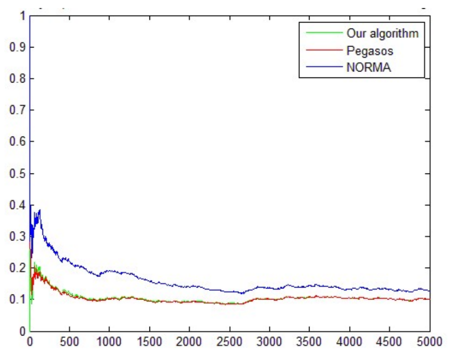
IWe also compare our algorithm to the performance of the online AdaBoost presented in=ref=. We use the same datasets as in above section(5.2), with the difference being that the synthetic dataset is not pregenerated, since both of the algorithms are adapted to the online usage.

For weak classifiers, we again use randomly generated linear classifiers, and for update stage of online Boosting algorithm we use the Pegasos iteration.

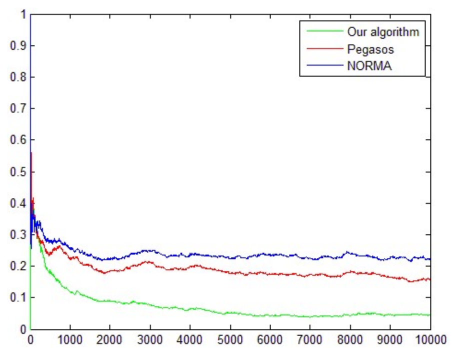
The results of convergence experiments are shown on =fig=. They show that the online boosting algorithm underperforms compared to our algorithm, and is less stable. For the noisy, inseparable Abalone datasets, for example, its accuracy actually decreases with the increasing number of iterations. It is also much more computaionally expensive for the number of selectors equal to the number of classifiers added by our algorithm due to the fact that it erforms updates on the whole pool of classifiers at once to estimate the optimal one. We can conclude that for most cases our algorithm would be superior to the online boosting.



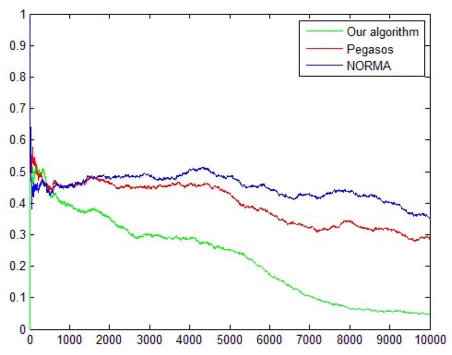
(a) Linearly separable dataset



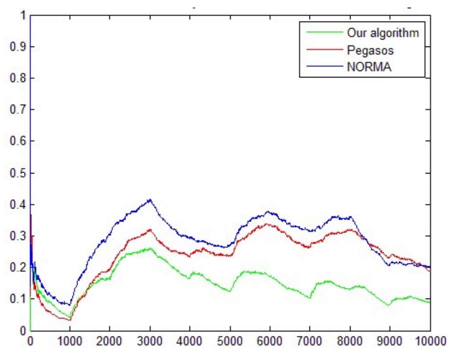
(b) Linear dataset with noise



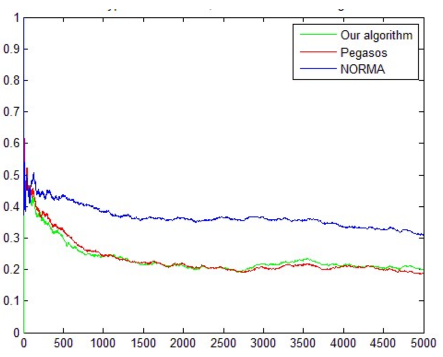
(c) Bayes-separable dataset with fixed distribution parameters



(d) Bayes-separable dataset with drifting distribution parameters

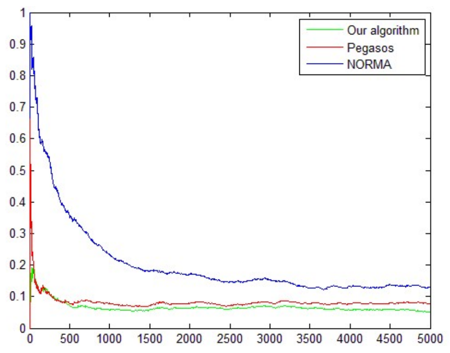


(e) Bayes-separable dataset with parameters being switched every 1000 iterations

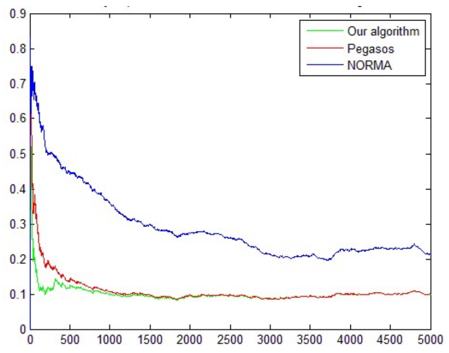


(f) Covertype dataset

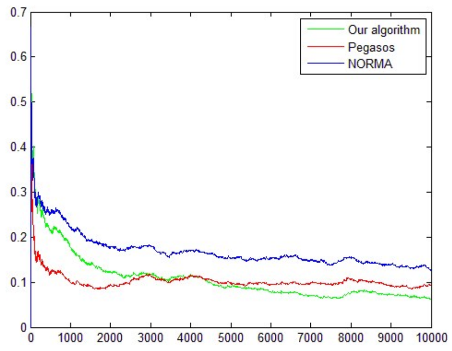
Figure 5.1: Experimental results comparing the performance of our algorithm to linear implementation of Pegasos and NORMA



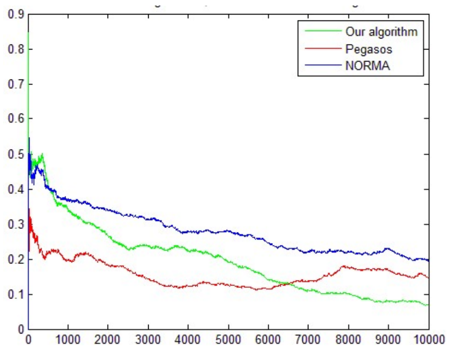
(a) Linearly separable dataset



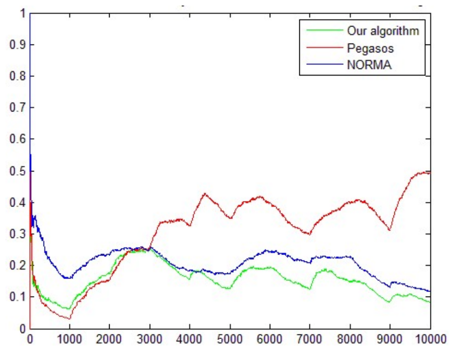
(b) Linear dataset with noise



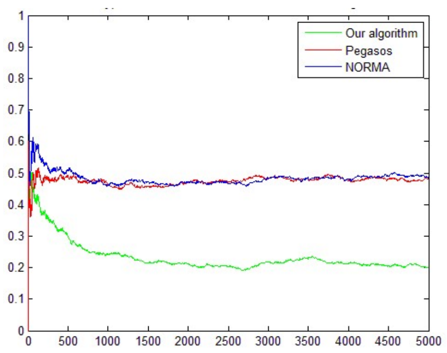
(c) Bayes-separable dataset with fixed distribution parameters



(d) Bayes-separable dataset with drifting distribution parameters



(e) Bayes-separable dataset with parameters being switched every 1000 iterations



(f) Covertype dataset

Figure 5.2: Experimental results comparing the performance of our algorithm implementation of Pegasos and NORMA using Gaussian RBF

6

Conclusion

The main goal of the work presented in this thesis is develeopment of the new online methods, and proving its applicability to the real-world tasks in image processing. After reviewing the state-of-the art methods, includiing online SVM training and boosting, we have developed our own method combining effectiveness of the offline AdaBoost and convergence properties of the online methods like Pegasos. We have proposed a methods iteratively adding and training weak classifiers while at the same time updating the boosting coefficients combining them into a strong classifier with increased accuracy. The resulting algorithm can be used for both linear and non-linear classification, and exhibits high stbility in regard to improperly tuned parameter, allowing to use the same parameter set for varying datadistribution. Furthermore, it has lower computational and memory reqirements than the the state-of-the art methods it was derived from. To show the effectiveness of the proposed algorithms have evaluated our method against Pegasos =ref=, NORMA =ref= and online boosting, all methods quite popular in the scientific community for use with large datasets. We used several synthetic and publically available natural datasets to show the stability and improved convergence rate of our method compared to other methods.

To further show the practiacal apllicability of our algorithm, we have developed a simple tracking application for the mobile device that uses our learning method to learn and maintain object model. We have inroduced a set of simple features sufficient to idetify certain classes of object, and have shown that our method allows their combination into a relatively stable tracker that runs at near real-time speed (12-15 fps) even with unoptimized code. This applicatio, then serves as a proof-of-concept application demostraing the utility of our learning technique.