Large-Scale Support Vector Machines: Algorithms and Theory

Aditya Krishna Menon

ABSTRACT

Support vector machines (SVMs) are a very popular method for binary classification. Traditional training algorithms for SVMs, such as chunking and SMO, scale superlinearly with the number of examples, which quickly becomes infeasible for large training sets. Since it has been commonly observed that dataset sizes have been growing steadily larger over the past few years, this necessitates the development of training algorithms that scale at worst linearly with the number of examples. We survey work on SVM training methods that target this large-scale learning regime. Most of these algorithms use either (1) variants of primal stochastic gradient descent (SGD), or (2) quadratic programming in the dual. For (1), we discuss why SGD generalizes well even though it is poor at optimization, and describe algorithms such as Pegasos and FOLOS that extend basic SGD to quickly solve the SVM problem. For (2), we survey recent methods such as dual coordinate-descent and BMRM, which have proven competitive with the SGD-based solvers. We also discuss the recent work of [Shalev-Shwartz and Srebro, 2008] that concludes that training time for SVMs should actually *decrease* as the training set size increases, and explain why SGD-based algorithms are able to satisfy this desideratum.

1. WHY LARGE-SCALE LEARNING?

Supervised learning involves analyzing a given set of labelled observations (the *training set*) so as to predict the labels of unlabelled future data (the *test set*). Specifically, the goal is to learn some function that describes the relationship between observations and their labels. Archetypal examples of supervised learning include recognizing handwritten digits and spam classification.

One parameter of interest for a supervised learning problem is the size of the training set. We call a learning problem *large-scale* if its training set cannot be stored in a modern computer's memory [Langford, 2008]. A deeper definition of large-scale learning is that it consists of problems where the main computational constraint is the amount of time available, rather than the number of examples [Bottou and Bousquet, 2007]. A large training set poses a challenge for the computational complexity of a learning algorithm: in order for algorithms to be feasible on such datasets, they must scale at worst linearly with the number of examples.

Most learning problems that have been studied thus far are *medium-scale*, in that they assume that the training set can be stored in memory and repeatedly scanned. However, with the growing volume of data in the last few years, we have started to see problems that are large-scale. An example of this is ad-click data for search engines. When most modern search engines produce results for a query, they also display a number of (hopefully) relevant ads. When the user clicks on an ad, the search engine receives some commission from the ad sponsor. This means that to price the ad reasonably, the search company needs to have a good estimate of whether, for a given query, an ad is likely to be clicked or not. One way to formulate this as a learning problem is to have training examples consisting of an ad and its corresponding search query, and a label denoting whether or not the ad was clicked. We wish to learn a classifier that tells us whether a given ad is likely to be clicked if it were generated for a given query. Given the volume of queries search engines process (Google processes around 7.5 billion queries a month [Searchenginewatch.com, 2008]), the potential size of such a training set can far exceed the memory capacity of a modern system. Conventional learning algorithms cannot handle such problems, because we can no longer store and have ready access to the data in memory. This necessitates the development of new algorithms, and a careful study of the challenges posed by this scale of problem. An extra motivation for studying such algorithms is that they can also be applied to medium-scale problems, which are still of immediate practical interest currently.

Our focus in this document is how a support vector machine (SVM), a popular method for binary classification that is based on strong theory and enjoys good practical performance, can be scaled to work with large training sets. There have been two strands of work in the literature on this topic. The first is a theoretical analysis of the problem, in an attempt to understand how learning algorithms need to be changed to adapt to a large-scale setting. The other is the design of training algorithms for SVMs that work well for these large datasets, including the recent Pegasos solver [Shalev-Shwartz et al., 2007], which leverages the theoretical results on large-scale learning to actually *decrease* its runtime when given more examples. We discuss both strands, and attempt to identify the limitations of current solvers. First, let us define more precisely the large-scale setting that we are considering, and describe some general approaches to solving such problems.

1.1 Batch and online algorithms

When we discuss supervised learning problems with a large training set, we are implicitly assuming that the learning is done in the batch framework. We do *not* focus on the online learning scenario, which consists of a potentially infinite stream of training examples presented one at a time, although such a setting can certainly be thought of as large-scale learning. However, it is possible for an online *algorithm* to solve a batch problem, and in fact this might be desirable in the large-scale setting, as we discuss below. More generally, an intermediate between batch and online algorithms is what we call an *online-style* algorithm. This is an algorithm that assumes a batch setting, but only uses a sublinear amount of memory, and whose computational complexity scales only sublinearly with the number of examples. This precludes batch algorithms that repeatedly process the training set at each iteration. A standard online algorithm can be converted into an online-style algorithm easily: we simply stream in the training set one at a time to the algorithm, and use some combination of the predictors for each example as our predictor for the entire training set. It can be shown that if an online algorithm achieves a regret of $R(T)$ on T rounds of adversarial input, then it can achieve a generalization error of $O(R(T)/T)$ when used as an online-style algorithm [Kakade and Tewari, 2009].

1.2 Approaches to large-scale learning

As we discussed in the previous section, it is possible to use an online algorithm to solve a large-scale task. In fact, there are three obvious ways in which we can solve a large-scale batch learning problem (c.f. [Langford et al., 2008]):

1. We can treat the training data as a stream, and apply an online or online-style learning algorithm. Such algorithms have space requirements that are only mildly (possibly not at all!) dependent on the number of examples, and we do not need to store the entire training set in memory.

2. We can parallelize a batch algorithm, which lets us split the large learning problem into a number of smaller problems. Given enough machines or processor cores, each sub-problem becomes medium-scale, and hence tractable.

3. We can preprocess the training data and sample a small subset of data to train on. This subset may be chosen randomly, or using a more sophisticated approach that tries to pick only informative examples. If we are particularly aggressive in choosing this subset, we might be able to reduce the training set enough to make our training algorithm tractable.

Although parallelization is a natural way to deal with large-scale problems, most literature on large-scale SVM training focusses on the online and online-style solutions. We are aware of only one large-scale SVM solver, OCAS [Franc and Sonnenburg, 2008] (discussed in §4.5.1), that explicitly considers parallelization. Other parallel SVMs, such as [Graf et al., 2005], have not been tested on truly large datasets, and lack strong theoretical guarantees. The idea of preprocessing the training set is plausible in the context of SVMs because as we discuss in §3.3, the final SVM classifier is only a function of a few training examples. While there have been solutions based on this idea (e.g. [Yu, 2003, Schohn and Cohn, 2000]), they are not the main focus of the document, as most of these approaches are heuristics with no theoretical guarantees.

1.3 Outline of paper

The outline of this document is as follows. Section 2 provides some background on regularization, a foundational concept in SVMs, and gradient methods, which are used in several of the training methods we describe. Section 3 gives the necessary background on SVMs. Section 4 is the main section of the document, describing several large-scale SVM training algorithms, and Section 5 analyzes in more detail why algorithms based on stochastic gradient descent perform well on the SVM task. The conclusion, Section 6, identifies directions for further research.

2. OPTIMIZATION IN LEARNING

In this section, we give some background on the connection between optimization and learning, which is important in analyzing training algorithms for SVMs. We define foundational concepts such as generalization error and regularization, and then look at gradient based optimization methods. The latter serve as the foundation for several large-scale SVM training methods we study in subsequent sections.

2.1 Estimating generalization error

A learning problem must have a measure that says how good an algorithm performs on the task. A universal goal for any (batch) supervised learning algorithm is *generalization*, which estimates how well the algorithm will perform on future data. Suppose the data points $x \in \mathcal{X}$ with true labels $y \in \mathcal{Y}$ have a joint distribution $\mathcal{D}$ over $\mathcal{X} \times \mathcal{Y}$. Usually, we want the number of misclassifications to be low: a misclassification is where our prediction function $f : \mathcal{X} \to \mathcal{Y}$ has $f(x) \neq y$. We can define the *generalization error* of a classifier that produces a parameter vector θ to be $g(\theta) = \mathbb{E}_{(x,y)\sim\mathcal{D}}[1[f(x) \neq y)]]$. Here, $1[\cdot]$ is the 0-1 indicator function, also called the *0-1 loss function*.

If we can write an analytic expression for $g(\theta)$, then the learning problem amounts to finding the θ that minimizes this expression. However, in practice the distribution $\mathcal{D}$ of points is unknown. So, we try to minimize some function $\hat{g}(\theta)$ on the training set, $\mathcal{T} = \{(x_i, y_i)\}_{i=1}^n$, that acts as a surrogate for the generalization error. A natural choice is to minimize the 0-1 loss on the training set, but this problem is quite difficult as it is not *convex*. Therefore, we instead introduce another convex (non-negative) loss function $\ell(x, y; \theta)$ that approximates the 0-1 loss, and minimize this over the training set. The training error is therefore

$$\hat{g}(\theta) = \frac{1}{n}\sum_{i=1}^{n} \ell(x_i, y_i; \theta).$$

This expression is also known as the *empirical risk* ("risk" here is used synonymously with "loss").

It should be stressed that $\hat{g}$ is only a surrogate for the true generalization error, and minimizing $\hat{g}$ does not necessarily mean that we minimize the generalization error. In fact, when we try to minimize the loss over the training set, there is the danger of *overfitting*, meaning that we find spurious patterns in the training set that are not manifested in the distribution $\mathcal{D}$. A common way to prevent overfitting during training is *regularization*, which we describe below.

2.2 Regularization

Regularization refers to augmenting the objective function $g(\theta)$ with an extra term that penalizes "complex" θ vectors. Intuitively, this can be seen as an application of Occam's razor. Common measures of complexity include the ℓ_1 and ℓ_2 norms of θ. We incorporate regularization by representing our objective as the sum of the empirical loss and regularization term. The general form becomes

$$\begin{aligned}\hat{g}(\theta) &= \frac{1}{n}\sum_{i=1}^{n} \ell((x_i, y_i); \theta) + r(\theta) \\ &= \ell_{\text{emp}}(\theta) + r(\theta), \qquad (1)\end{aligned}$$

where r is a regularization term, and ℓ_{emp} denotes the empirical loss on the training set. This problem is known as *regularized risk minimization*.

While the choice of a specific regularization term, such as the ℓ_2 norm, can seem arbitrary, in problems such as ridge-regression, one can show that this minimizes the influence of spurious correlations in the data. ℓ_2 regularization is also intimately connected to SVMs, as we show in §3.1.

2.3 Gradient methods for learning

In the previous section, we showed how we can define a learning problem as computing $\theta = \operatorname{argmin} \hat{g}(\theta)$, where $\hat{g}(\theta)$ is the sum of empirical loss and regularization terms. We now look at methods for finding this minimum based on computing the gradient of $\hat{g}$.

2.3.1 Gradient descent

A classical mathematical technique for finding the minimum of a function $f(\theta)$ is *gradient descent* (GD). This method uses the fact that the gradient ∇f of a function points in the direction of greatest increase. This means that $-\nabla f$ points in the direction of greatest decrease, and so a natural iterative algorithm to find the minima of f is to update an estimate θ_t using

$$\theta_{t+1} = \theta_t - \eta_t \nabla f(\theta_t).$$

Here, η_t is the *learning rate*, which determines how far we move in the direction of the gradient. It is important to choose this value carefully, because if it is too small then convergence is slow, and if it is too large then we can overshoot and miss the minimum.

For a learning problem, we are interested in $\nabla \hat{g}$, which is

$$\nabla \hat{g}(\theta_t) = \frac{1}{n} \sum_{i=1}^{n} \nabla \ell(x_i, y_i; \theta_t) + \nabla r(\theta_t).$$

2.3.2 Stochastic gradient descent

In gradient descent, we compute the gradient using the entire training set. A superficially simple (but in fact far-reaching) alteration of this is to find the gradient with respect to a single randomly chosen example. This technique is called *stochastic gradient descent* (SGD). The update rule is then

$$\theta_{t+1} = \theta_t - \eta_t \nabla \ell(x_{i(t)}, y_{i(t)}; \theta_t) - \eta_t \cdot \nabla r(\theta_t),$$

where $i(t)$ is an index drawn randomly from $\{1, 2, \ldots, n\}$. In expectation, this update is the same as gradient descent, because $\mathbb{E}_{i(t)}[\ell(x_{i(t)}, y_{i(t)}; \theta_t)] = \frac{1}{n} \sum_i \ell(x_i, y_i; \theta_t)$.

By using only a single example we are only getting an approximation to the true gradient; therefore, we are no longer guaranteed to move in the direction of greatest descent. Nonetheless, there are at least two important reasons why stochastic gradient descent is useful for learning problems: (1) it is significantly quicker than gradient descent when n is large, and (2) it can be shown that stochastic gradient descent minimizes the generalization error quicker than gradient descent. We discuss the latter in more detail in §5.2.

2.3.3 Second-order gradient methods

Gradient methods only use information about the gradient ∇. However, the Hessian H of second order partial derivatives can provide valuable information about the curvature of a function. Methods that use the Hessian are known as *second-order* or *Newton* methods. We define *second-order gradient descent* (2GD) with the update rule

$$\theta_{n+1} = \theta_n - \gamma [H_f(\theta_n)]^{-1} \nabla f(\theta_n),$$

where H_f is the Hessian of f, and $\gamma > 0$ is the *step size*. In the classic formulation of Newton's method, we have $\gamma = 1$. When γ is chosen differently (e.g. using a line search), the method is sometimes called the *damped Newton method* [Boyd and Vandenberghe, 2004].

As with standard GD, we can use second-order stochastic gradient descent (2SGD) by using a stochastic approximation $\tilde{\nabla} f$ to the true gradient. The drawback of a second-order method is that the Hessian for a training set with d features is a $d \times d$ matrix. When d is large, computing this matrix can be infeasible. Even if H_f can be computed efficiently, storing it can be infeasible (as it is typically a dense matrix). This motivates quasi-Newton methods, such as LBFGS [Liu et al., 1989].

2.3.4 Subgradient descent and projection

Gradient and stochastic gradient descent assume that $f(\theta)$ is a differentiable function of θ, so that ∇f is always well defined. But if we wish to minimize $f(\theta) = \hat{g}(\theta)$, this assumes that $\ell((x, y); \theta)$ and the regularization term $r(\theta)$ are differentiable in θ, which does not always hold. For example, ℓ_1 regularization, where $r(\theta) = |\theta|_1$, is not differentiable at $\theta = 0$. Fortunately, if ℓ_{emp} and r are both *convex* functions, then even if they are not differentiable, they can be lower bounded at any point by a *subgradient*. The subgradient $\partial f(\theta)$ consists of all vectors that are lower bounds to the function at θ:

$$\partial f(\theta) = \{v \in \mathbb{R}^d : (\forall \theta') f(\theta') \geq f(\theta) + v \cdot (\theta' - \theta)\}.$$

The update for subgradient descent is almost identical to gradient descent:

$$\theta_{t+1} = \theta_t - \eta_t \nabla^{(s)} f(\theta_t).$$

Here, we use $\nabla^{(s)} f(\theta_t)$ to denote any vector $v \in \partial f(\theta_t)$.

Now, suppose that we wish to do the optimization over a constrained set Ω of θ values. An extension to subgradient descent that allows us to solve this problem is *subgradient projection*, where after performing a subgradient descent step, we project the resulting vector into Ω, giving the following update:

$$\theta_{t+1} = \Pi_\Omega(\theta_t - \eta_t \nabla^{(s)} f(\theta_t)),$$

where Π_Ω denotes Euclidean projection[1] onto the set Ω.

3. SUPPORT VECTOR MACHINES

Support vector machines (SVMs) are a popular method for binary classification. SVMs can be seen as an extension of the perceptron, which tries to find a hyperplane that separates the data. The perceptron simply tries to find *any* separating hyperplane, without considering how clearly the hyperplane separates the data. But intuitively, a hyperplane that is as far away as possible from either class is preferable, because we expect this to generalize better to unseen data (an example is shown in Figure 1). A technical measure of how clearly a hyperplane separates data is its *margin*. This is the distance of the hyperplane to the closest point in the dataset; a large margin means that the hyperplane very clearly separates the data.

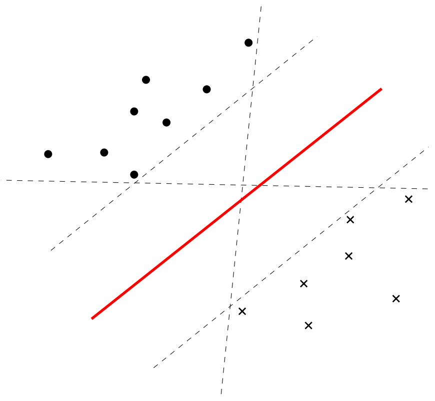

Figure 1: There can be several hyperplanes (shown as dashed lines) that separate a dataset, but intuitively the one with the largest margin (the red line) has the best generalization.

[1]If Ω is the space spanned by basis vectors $u_1, \ldots, u_k$, and if $A = [u_1 \quad u_2 \quad \ldots \quad u_k]$, then the Euclidean projection of x onto Ω is $A(A^T A)^{-1} A^T x$.

Given the definition of margin, we can define the goal of an SVM: for a training set $\{(x_i, y_i)\}_{i=1}^n$, where x_i are the observed data and y_i the labels, an SVM finds a *maximum margin* separating hyperplane. In the standard setting, we have real valued data $x_i \in \mathbb{R}^d$, and binary labels $y_i \in \{\pm 1\}$. We now formally define SVMs as an optimization problem.

3.1 The primal SVM formulation

In formal terms, the parameter θ that SVMs learn consists solely of a vector w, representing the normal to the separating hyperplane with maximum margin. In the rest of this document, we will use w in place of θ whenever we refer to a problem for which $\theta = \langle w \rangle$. There are two ways of treating the SVM problem. The classical method is the hard margin SVM [Vapnik and Lerner, 1963], which assumes that the dataset is linearly separable: hence, every point must be correctly classified by the maximum margin hyperplane. The soft margin SVM [Bennett and Mangasarian, 1992, Cortes and Vapnik, 1995] allows for some points to be misclassified, but penalizes these points appropriately. The latter is more useful in practical settings where data is unlikely to be perfectly separable (e.g. due to noise), and so we focus on this version. It can be represented with the following optimization problem.

THEOREM 1. *Given a training set $\{(x_i, y_i)\}_{i=1}^n$ of training examples where $y_i \in \{\pm 1\}$, the hyperplane parameterized by normal vector w that balances the goal of separating the data and maximizing margin can be found by solving the following optimization problem:*

$$\textit{minimize}_{w} \ \frac{\lambda}{2}||w||^2 + \frac{1}{n}\sum_{i=1}^{n} \max(0, 1 - y(w \cdot x)), \tag{2}$$

where $\lambda \geq 0$ is called the regularization parameter.

As a note on terminology: instead of using a regularization parameter λ that scales $||w||^2$, we can use a misclassification parameter C that scales the empirical loss term. The two are related by $\lambda = \frac{1}{nC}$.

We can think of the problem as requiring the minimization of empirical loss, plus a regularization term that limits the complexity of our solution [Shalev-Shwartz et al., 2007], like the form of §2.2. In fact, in the inseparable case, this is a more convincing motivation for the SVM problem than maximizing the margin (whose definition is not as clear when we allow for misclassifications) [Poggio and Smale, 2005]. We discuss the choice of loss function in §3.4.

3.2 The dual formulation and kernels

The above analysis assumed that we were seeking a hyperplane in the same space as the dataset, that is, a linear classifier. SVMs can be used as nonlinear classifiers using the classic *kernel trick*. The idea is to embed the data points x_i into some higher dimensional space $\mathcal{F}$ using some mapping $\Phi : \mathcal{X} \to \mathcal{F}$, and to seek a linear classifier in $\mathcal{F}$. This will of course be nonlinear in the original space $\mathcal{X}$. The kernel trick allows us to make the mapping Φ implicit, by defining the *kernel matrix* K, where $K_{ij} = \Phi(x_i) \cdot \Phi(x_j)$. If in the original space we only employ dot-products $x_i \cdot x_j$ to solve the learning problem, we can seamlessly transfer to a high dimensional space by simply replacing these dot-products by K_{ij}. This suggests that we can use any kernel matrix K that defines a valid dot-product in a high dimensional space, without knowing precisely what the high dimensional mapping Φ is! Mercer's theorem [Cristianini and Shawe-Taylor, 2000, pp. 35–41] tells us that any positive semi-definite matrix is a valid kernel matrix.

For SVMs, the primal problem we described in the previous section is its most natural form, because it captures exactly the quantity we wish to minimize. When moving to kernels with SVMs, however, it is standard to work with the *dual* optimization problem. The reason is that the dual version uses explicit dot-products, as we show below.

THEOREM 2. *The dual formulation of the SVM optimization problem in Equation 2 is*

$$\textit{maximize}_{\alpha} \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y_i y_j (x_i \cdot x_j) \ \textit{subject to} \ 0 \leq \alpha_i \leq \frac{1}{\lambda n}. \tag{3}$$

Since we have an explicit dot-product, using a kernel is simple: we replace $x_i \cdot x_j$ by K_{ij}, giving the dual problem

$$\text{maximize}_{\alpha} \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y_i y_j K(x_i, x_j) \text{ subject to } 0 \leq \alpha_i \leq \frac{1}{\lambda n}.$$

In §4.1, we show how we can incorporate kernels into the primal problem.

Several practical applications of SVMs use nonlinear kernels, such as the polynomial and RBF kernel. However, in applications like text classification, linear SVMs are still used, because it has been observed that many text classification problems are linearly separable [Joachims, 1998]. Most literature on large-scale SVM training, starting with SVM$^{\text{perf}}$, have targetted the linear SVM problem, citing this fact. Perhaps a more pragmatic reason for this is that linear SVMs make a lot of analysis simpler. Commonly used datasets to test these methods include the Reuters `CCAT` data, and the `astro-ph` data consisting of abstracts from arXiV's Astrophysics section[2].

3.3 What are support vectors?

The dual SVM problem lets us define the important concept of *support vectors* that give SVMs their name. These vectors are the training points which are not classified with confidence; that is, they are either misclassified, or are correctly classified but fall inside the margin region (see §3.1). Equivalently, they are the examples whose corresponding α_i values are non-zero. Recalling the representer theorem from the previous section, this tells us that the optimal weight vector w^* is a linear combination of the support vectors. Therefore, the support vectors are the "essential" training points, and the goal of training is to discover them.

The number n_{SV} of support vectors also characterizes the complexity of the learning task: if n_{SV} is small, then that suggests that only a few examples are important, and that we can disregard many examples without any loss in accuracy. However, if n_{SV} is large, then nearly every example is important for accuracy. [Steinwart, 2004] showed that under general assumptions about the loss function and underlying distribution $\mathcal{D}$ of the training data, $n_{SV} = \Omega(n)$. This suggests that asymptotically, *all* points are critical for training! While this seemingly gives an $\Omega(n)$ bound on training time, we note that this is only to solve the SVM problem *exactly*. As we discuss in §5.2, it is of greater practical interest to look approximate solutions to the SVM problem, where this bound no longer applies. Further, datasets that arise in practice need not necessarily have $\Theta(n)$ support vectors: it has been observed that the `usps` dataset has $n_{SV} = O(\sqrt{n})$, for example [Chapelle, 2007].

[2] `http://arxiv.org/archive/astro-ph`

3.4 Loss functions

We make a brief comment about the choice of loss function in SVMs. The standard definition of SVMs uses $\ell(x, y; w) = \max(0, 1 - y(w \cdot x))$, as we have done; this is known as the *hinge-loss*. This penalizes errors linearly, but has the disadvantage that it is not differentiable everywhere. A common variant of the classical SVM definition is to instead use the *square-loss*, $\ell(x, y; w) = \max(0, 1 - y(w \cdot x))^2$, which is differentiable everywhere. An intermediate solution is the Huber loss, which is linear for $y(w \cdot x) < 0$, but quadratic for $0 \leq y(w \cdot x) \leq 1$, and hence also differentiable everywhere. All these loss functions are *convex*, which means that it is easy to optimize with them, and they also upper-bound the 0-1 loss function (see §2.1). This means that minimizing these losses also makes bounds the misclassification error of our classifier.

In practice, the differentiability of the square and Huber loss is appealing, and indeed these losses are sometimes used for this reason. Theoretically however, it has been shown that using the hinge-loss, SVMs approach the optimal classifier (also called the *Bayes classifier*), whereas the square and higher order losses approach only a distribution whose *sign* gives the optimal classifier [Lin, 2002]. Having said this, it is not clear whether there is a discernible practical difference in accuracy when one does not use the hinge-loss.

It is straightforward to incorporate different losses into the primal version of the SVM problem (Equation 2): treating $\max(0, 1 - y_i(w \cdot x_i))$ as a special case of a loss function $\ell(x_i, y_i; w_i)$, we can plug in the square or Huber loss and get a similar optimization problem. It is not as obvious how the change in loss function affects the dual problem. The dual problem for the hinge- and square-loss can be expressed as as

$$\text{maximize} \sum_i \alpha_i - \frac{1}{2}\alpha^T Q \alpha \text{ subject to } 0 \leq \alpha_i \leq U, \qquad (4)$$

where $Q_{ij} = y_i y_j (x_i \cdot x_j) + D_{ij}$ for some diagonal matrix D, and U is some constant. The values of D, U depend on the loss function: for hinge-loss, $D_{ii} = 0$ and $U = \frac{1}{\lambda n}$. For square-loss, $D_{ii} = \frac{\lambda n}{2}, U = \infty$.

4. LARGE-SCALE SVM SOLVERS

We look at a range of large-scale SVM solvers, starting with a brief discussion of traditional optimization methods whose ideas have been considerably extended. A summary of these solvers and some of their properties is given in Table 1. We can see that most solvers fall into two categories: (1) primal gradient based methods, and (2) dual quadratic programming based methods. The two exceptions, Core Vector Machines and OCAS, are very interesting directions, and there is evidence that both are competitive with most other methods. We begin with a discussion of the traditional dichotomy between primal and dual methods, and why primal methods have received renewed interest of late.

4.1 Primal vs dual form

In §3.2, we showed how the dual SVM problem can naturally be extended to handle kernels. As pointed out in [Chapelle, 2007], the same can be done for the primal formulation when we have hinge- or square-loss. In the case of hinge-loss for example, we write the primal optimization problem as

$$\underset{f \in H}{\text{minimize}} \ \frac{\lambda}{2}||f||_H^2 + \frac{1}{n}\sum_{i=1}^n \max(0, 1 - y_i f(x_i)),$$

where H is a reproducing kernel Hilbert space (RKHS)[3] with associated kernel K. This can now be solved using standard techniques such as gradient descent; see §4.3.1 for details. As with the linear SVM case, we can view the problem as regularized risk minimization. The ℓ_2 regularizer $||f||_H^2$ is essential here, as the Hilbert space implicitly defined by the kernel K may be infinite dimensional! Without some form of regularization, we run the risk of easily overfitting on the training data.

The reason this is a natural extension of the linear primal problem is that the classic *representer theorem* [Kimeldorf and Wahba, 1971] tells us that the optimal solution w^* of the primal problem can be represented as a linear combination of the training examples in the high-dimensional space: $w^* = \sum_i \alpha_i^* y_i \Phi(x_i)$, where Φ is the high-dimensional mapping implicitly defined by K. In fact, these α_i^*'s are precisely the optimal values of the dual problem. Now comparing the above formulation to the original primal problem, we see that we have replaced $||w||$ with a Hilbert norm, and $w \cdot x$ with an evaluation in the Hilbert space (which is equivalent to a dot-product). If we choose $K(x_i, x_j) = x_i \cdot x_j$, then we recover the linear primal problem.

Given the fact that the primal SVM problem can handle kernels, it is instructive to ask why the dual version was traditionally preferred as the de facto space to work in. One historic reason for working in the dual is that it more *obviously* lends itself to non-linear SVMs, as it makes explicit use of dot-products. The idea of invoking the representer theorem to incorporate kernels into the primal problem, while obvious in hindsight, does not seem to have been considered until [Chapelle, 2007]. Another potential historic reason for the use of the dual, suggested by [Chapelle, 2007], is that the original hard margin formulation of SVMs has a series of constraints of the form $1 - y_i(w \cdot x_i) \leq 0$. By contrast, the dual constraints are of the form $0 \leq \alpha_i \leq 1/\lambda n$, which are significantly simpler as they merely requiring the variables being optimized to lie in a bounded interval (these are known as *box constraints* in optimization theory). However, as we showed in Equation 2, the soft margin primal SVM can be represented as an unconstrained minimization, hence avoiding this issue. This unconstrained formulation has only been recently considered with the advent of gradient-based SVM solvers (which we discuss in §4.3).

One convenience of the primal formulation is that it is simpler to deal with approximate solutions [Chapelle, 2007]. A ρ-approximate[4] solution $\tilde{w}$ to the SVM problem satisfies $\hat{g}(w) \leq \hat{g}(w^*) + \rho$, where w^* is the true minimizer of the training error. If we work with the primal problem, then it is simple to stop our algorithm at an appropriate point to get an approximate solution. But with a dual solver, we have the problem that a ρ-approximate dual solution does not necessarily correspond to a ρ-approximate primal solution. Hence, we need to devise more sophisticated stopping conditions, such as ones based on the KKT criteria. This is a big conceptual simplification of the SVM training process. Naturally, this is only interesting if approximate solutions to the SVM problem are viable in practice. It turns out that such approximations are theoretically well-motivated, and allow us to view large-scale learning in a very different light. We look at this issue more in §5. Interestingly, classical solvers were interested in *exact* solutions to the SVM problem; we discuss a couple of them below.

4.2 Traditional methods

Early SVM solvers focussed on quadratic-programming approaches

[3]Recall that a Hilbert space is a complete inner product space, and a RKHS with kernel K is the space $\{f : f(x) = \sum_i \alpha_i K(x, x_i)\}$.

[4]We will use ρ henceforth to refer to the optimization tolerance parameter.

Algorithm	Citation	SVM type	Optimization type	Style	Runtime
SMO	[Platt, 1999]	Kernel	Dual QP	Batch	$\Omega(n^2 d)$
SVM$^{\text{light}}$	[Joachims, 1999]	Kernel	Dual QP	Batch	$\Omega(n^2 d)$
Core Vector Machine	[Tsang et al., 2005, 2007]	SL Kernel	Dual geometry	Batch	$O(s/\rho^4)$
SVM$^{\text{perf}}$	[Joachims, 2006]	Linear	Dual QP	Batch	$O(ns/\lambda\rho^2)$
NORMA	[Kivinen et al., 2004]	Kernel	Primal SGD	Online(-style)	$\tilde{O}(s/\rho^2)$
SVM-SGD	[Bottou, 2007]	Linear	Primal SGD	Online-style	Unknown
Pegasos	[Shalev-Shwartz et al., 2007]	Kernel	Primal SGD/SGP	Online-style	$\tilde{O}(s/\lambda\rho)$
LibLinear	[Hsieh et al., 2008]	Linear	Dual coordinate descent	Batch	$O(nd \cdot \log(1/\rho))$
SGD-QN	[Bordes and Bottou, 2008]	Linear	Primal 2SGD	Online-style	Unknown
FOLOS	[Duchi and Singer, 2008]	Linear	Primal SGP	Online-style	$\tilde{O}(s/\lambda\rho)$
BMRM	[Smola et al., 2007]	Linear	Dual QP	Batch	$O(d/\lambda\rho)$
OCAS	[Franc and Sonnenburg, 2008]	Linear	Primal QP	Batch	$O(nd)$

Table 1: A comparison of various SVM solvers discussed in this document. "QP" refers to a quadratic programming technique, "SGD" to stochastic (sub)gradient descent, and "SGP" to stochastic (sub)gradient projection. "SL" means the method only works with square-loss. The runtime is for a problem with n training examples and d features, with an average of s non-zero features per example. λ is the SVM regularization parameter, and ρ the optimization tolerance. "Unknown" means there is no known formal bound on the runtime.

to directly solve the dual optimization problem (Equation 3), for reasons discussed above. These solvers have the disadvantage of being computationally expensive, with superlinear dependence on the number of examples n (for some solvers, the worst case runtime is at least quadratic in n). All the algorithms we discuss subsequently dramatically improve this runtime. Here, we briefly discuss two traditionally important training methods: Sequential Minimal Optimization [Platt, 1999] and SVM$^{\text{light}}$ [Joachims, 1999]. Both algorithms have served as the inspiration for modern, faster training algorithms (LaRank and SVM$^{\text{perf}}$ / BMRM).

The Sequential Minimal Optimization (SMO) technique [Platt, 1999] is a historically important SVM training method. It attempts to break up the SVM optimization problem into a number of smaller subproblems, which can be efficiently solved and pieced together. To be precise, at each iteration SMO tries to optimize a pair of Lagrange multipliers α_i, α_j. This is contrast to the original QP solver for SVMs (the "chunking" approach [Vapnik, 1982]), which tries to solve a problem involving all non-zero Lagrange multipliers. This made SMO quite attractive at the time, since it was faster than several competing methods. Its runtime is superlinear in the number of training examples, in the worst case being $\Omega(n^2 d)$: this is a big improvement over older methods that scale like n^3, but is obviously infeasible on large datasets.

SVM$^{\text{light}}$ is a *dual-decomposition* method, meaning that it decomposes the standard dual problem into a series of subproblems. Then, by working with only a small subset of these problems at a time, it has a space requirement that is linear in the number of examples n. This was one of the first linear-space SVM training algorithms, and its software package of the same name has been regularly updated. Like most dual methods, it handles nonlinear SVMs. Based on empirical analysis, its runtime is known to be $\Omega(n^2 d)$, which like SMO is infeasible for large data sets. It is still a versatile approach, however, and can be applied to transductive SVMs, regression and ranking problems, and so on. Hence, it is still a popular package in some application domains.

4.3 Primal gradient-based methods

Most large-scale primal methods use some variant of SGD to quickly solve the SVM problem. As mentioned in §2.3.2, SGD has the advantage of being significantly quicker than methods that use the true gradient at each iteration. In practice, this is the reason why SGD is sometimes favoured even though it is known to be slow at convergence. It seems then that these methods cannot hope to compete with approaches that solve the SVM optimization problem more strenuously. In §5, we show why this intuition is not necessarily true. We start by looking at some simple SGD based methods.

4.3.1 Stochastic gradient descent solvers

NORMA [Kivinen et al., 2004] is an online learning algorithm for solving nonlinear SVMs. It is essentially an application of SGD to the primal SVM objective function (to be precise, for the case of hinge-loss, we apply stochastic *subgradient* descent). Recall from §3.2 that in the nonlinear SVM problem, we seek to find some f in a RKHS H such that

$$\frac{\lambda}{2}||f||_H^2 + \frac{1}{n}\sum_{i=1}^{n} \max(0, 1 - y_i f(x_i))$$

is minimized. It can be shown that the corresponding SGD update is

$$f_{t+1} = (1 - \eta\lambda) f_t - \eta \ell'(f_t(x_t), y_t) k(x_t, \cdot),$$

where $\ell'(x, y) = \frac{\partial}{\partial x}\ell(x, y)$. Since f resides in a Hilbert space, this is an update of a *function*, not a vector. To move to the simpler case of a vector update, we invoke the representer theorem (see §3.2), which tells us that $f_t(x) = \sum_{i=1}^{t-1} \alpha_i k(x_i, x)$. Therefore, the update can alternately be expressed in terms of the α_i variables, yielding

$$\alpha_t = -\eta_t \ell'(f_t(x_t), y_t)$$

$$\alpha_{t'} = (1 - \eta_t\lambda)\alpha_{t'} \text{ for } t' < t.$$

For the case of SVMs with hinge-loss, the update for α_t is

$$\alpha_t = \eta_t y_t 1[y_t f_t(x_t) - 1 \leq 0].$$

Importantly, it is shown that one can truncate the expansion of $f_t(x)$ with only exponentially small loss in accuracy: this lets one keep a constant number τ of terms at any given time, which means that the space requirements of the algorithm are not prohibitive. Further, both a regret and training error bound are provided for the algorithm: it can be shown that in either case, the algorithm requires $O(1/\rho^2)$ iterations to converge to an ρ-optimal solution.

NORMA is the first application of SGD to the SVM problem that we are aware of, and the idea seems to have been rediscovered at least twice for the linear SVM case. Bottou proposed essentially the same algorithm with SVM-SGD[5] [Bottou, 2007] for linear SVMs, and showed that SVM-SGD achieves comparable test set performance to other more complicated methods, such as SVM$^{\text{light}}$ and SVM$^{\text{perf}}$, in dramatically less time (it is 20 times faster than SVM$^{\text{light}}$). [Zhang, 2004] also proposed essentially the same method, albeit in the more general framework of linear prediction, and was more interested in the theoretical performance of SGD.

It is initially surprising that SGD based methods can achieve such good performance in much less time. Since the gradient estimates are based on one example at a time, they do a factor of n less work than standard batch methods per example. Of course, the bound on the number of iterations, which is $O(1/\rho^2)$ for NORMA, is much higher than logarithmic convergence rates usually enjoyed by batch methods. There is also the issue of tuning the learning rate, a common complaint about gradient methods. However, we will see that the iteration bound can be sharpened to $O(1/\rho)$ in the process of solving the learning rate issue. Further, a detailed comparison of runtimes in §5 will show why the independence on n has very strong implications. We first look at the Pegasos solver, which is an important extension of plain SGD.

4.3.2 Pegasos

Pegasos [Shalev-Shwartz et al., 2007] is a SVM optimization algorithm whose runtime is independent of the number of training examples. (In §5.3, we will study Pegasos' runtime more closely.) The algorithm operates on mini-batches of the training data, with the batch size specified by a user-supplied parameter k. This allows the solver to vary between stochastic gradient descent and subgradient projection: when $k = 1$, Pegasos is nothing but SGD augmented with a projection step, and when $k = n$, the number of training examples, it is simply subgradient projection. However, despite the superficial similarity to SGD, an important property about Pegasos is that it converges to the ρ-approximate solution in $\tilde{O}\left(\frac{d}{\lambda\rho}\right)$ iterations, where d is the maximum number of nonzero features in each example. This is in contrast to the $O(1/\rho^2)$ convergence of methods based on plain SGD, such as NORMA.

The algorithm proceeds as follows: at each iteration, a random subset of k training examples is chosen. The weight vector is updated by the subgradient of the objective function evaluated with these k examples. Then, the vector is projected onto a ball of radius $1/\sqrt{\lambda}$, because it can be shown that the optimal solution lies inside this ball (see Lemma 1). The formal description of the algorithm is given in Algorithm 2. In the case $k = 1$, there are a couple of important distinctions from plain SGD that allow Pegasos to converge to the optimal solution quickly. First, the learning rate is decayed by $\eta_t = \frac{1}{\lambda t}$, where λ is the regularization parameter. Second, we always project w so that $||w|| \leq 1/\sqrt{\lambda}$. The projection step ensures that the sharp decrease in learning rate is feasible, in that it is still possible to bound the number of iterations by $O(1/\rho)$.

Since Pegasos' runtime is parameterized by k, like the SGD methods of the previous section, its runtime is independent of the number of examples. Naturally, this would not be useful if it caused the algorithm to converge very slowly. In fact, Pegasos converges to the ρ-approximate solution in only $\tilde{O}\left(\frac{d}{\lambda\rho}\right)$ iterations. This guarantee relies on the following simple (but non-obvious) fact about the optimal SVM solution, which says the projection step always brings us closer to the SVM solution.

```
for $t = 1 \dots T$ do
    Pick random $A_t \subseteq \mathcal{T}$ such that $|A_t| = k$
    $\mathcal{M} := \{(x, y) \in A_t : 1 - y(w \cdot x) > 0\}$
    $\nabla_t := \lambda w_t - \frac{1}{|\mathcal{M}|} \sum_{(x,y)\in\mathcal{M}} yx$
    Update $w_{t+\frac{1}{2}} \leftarrow w_t - \frac{1}{\lambda t} \cdot \nabla_t$
    Let $w_{t+1} \leftarrow \min\left(1, \frac{1}{\sqrt{\lambda}||w_{t+\frac{1}{2}}||}\right) w_{t+\frac{1}{2}}$
end for
return $w_{T+1}$
```

Algorithm 2: The Pegasos algorithm.

LEMMA 1. *The optimal weight vector w^* that minimizes the primal SVM problem satisfies $||w^*|| \leq \frac{1}{\sqrt{\lambda}}$.*

PROOF. The *strong duality* theorem says that when we try to minimize a convex function subject to a set of linear constraints, the values of the optimal primal and dual solutions are equal; in other words, the so called *duality gap* is zero. Since the optimization problems for SVMs satisfy these conditions, if we consider the optimal solutions w^* and α^* to the primal and dual problems respectively, we must have[6]

$$\frac{\lambda}{2}||w^*||^2 + \frac{1}{n}\sum_{i=1}^{n}\ell(x_i, y_i; w^*) = \frac{1}{n}\sum_{i=1}^{n}\alpha_i^* - \frac{1}{2\lambda n^2} \cdot \sum_{i,j}\alpha_i^*\alpha_j^* y_i y_j (x_i \cdot x_j).$$

But using the fact that $w^* = \sum \alpha_i^* y_i x_i$ and $\alpha_i^* \geq 0$, the right hand side can be rewritten:

$$\frac{\lambda}{2}||w^*||^2 + \frac{1}{n}\sum_{i=1}^{n}\ell(x_i, y_i; w^*) = \frac{||\alpha^*||_1}{n} - \frac{\lambda}{2}||w^*||^2.$$

Rearranging,

$$\lambda||w^*||^2 = \frac{||\alpha^*||_1 - \sum_{i=1}^{n}\ell(x_i, y_i; w^*)}{n}.$$

The claim follows from the fact that $||\alpha^*||_1 \leq n$ and that loss functions are positive. □

An interesting observation about Pegasos is that the parameter k does not appear as an assumption or runtime parameter for the proof of convergence. This suggests that for *any* value of k, Pegasos will converge to an approximately optimal solution. Since $k = 1$ is the least expensive choice of k, we can take this to be the default value. So, Pegasos can effectively be thought of as SGP, and demonstrates that SGD can be used to design very efficient and competitive learning algorithms.

We make a brief note on the importance of Pegasos' learning rate. Aside from the favourable convergence guarantees, by fixing the learning rate to be $\eta_t = 1/\lambda t$, we eliminate the need to discover a sensible value for this rate. This parameter tuning is commonly cited as a drawback of SGD, and so by setting it to be a fixed analytic expression, we avoid the issue altogether. Of course, we are still required to find a sensible value for λ, but this is true for *any*

[5]Note that there is no formal publication of the work, although all code is freely available.

[6]The α variables in the dual formulation has been scaled by $n\lambda$: this accounts for the fact that we scaled the primal problem by λ.

SVM training method. Choosing a good regularization parameter λ (or equivalently, the loss factor C) is entirely problem dependent, not algorithm dependent. Small values of λ (or large values of C) mean that we severely penalize misclassified points; if a small λ is required for a good solution, this indicates that the problem is difficult. The fact that Pegasos explicitly depends on $1/\lambda$ means that it accounts for the difficulty of the problem!

Pegasos can also be extended to use nonlinear kernels using the representer theorem. Recall from §3.2 that this theorem let us express w as a linear combination of the training examples, $w = \sum_i \alpha_i y_i \Phi(x_i)$. It was observed in §3.3 that only some $\alpha_i \neq 0$, and that these training examples are called the support vectors. If we let I be the indices of examples for which $\alpha_i \neq 0$, then we can represent w implicitly using I and the corresponding α_i values. This is because we only perform two operations on w: scaling by a constant, and finding the dot-product. Both can be done only knowing I and α_i; the latter is because the dot-product can be written using the kernel:

$$w \cdot \Phi(x_j) = \sum_{i \in I} \alpha_i y_i K(x_i, x_j).$$

So, by keeping track of I and the α_i values, we can easily handle kernels in Pegasos, with the same convergence guarantee. The only caveat is that the runtime per iteration now becomes $O(|I|)$, which is $\min(n, \tilde{O}(d/\lambda\rho))$ in the worst case.

4.3.3 SGD-QN

SGD-QN [Bordes and Bottou, 2008] is a recent method that topped the wild track of the ICML '08 Pascal Challenge [Pascal, 2008]. This algorithm combines SVM-SGD with a quasi-Newton (QN) method. Recall that a quasi-Newton method approximates the inverse Hessian, and uses this to scale the gradient at each iteration. Quasi-Newton methods are by nature batch methods, and so have been typically eschewed for large-scale problems. However, a recent paper [Schraudolph et al., 2007] developed an online version of LBFGS called oLBFGS, eliminating all steps that assume that there is batch access to the training data at every step. This nearly lets LBFGS be used as-is for training a large-scale SVM, but for the fact that LBFGS assumes that the function being minimized is smooth, which is not the case for the hinge-loss formulation of SVMs. Recent work has lifted this assumption as well [Yu et al., 2008], which would make LBFGS a viable approach for the SVM problem if combined with the oLBFGS framework. We discuss this further in §6.

SGD-QN is essentially a simplification of oLBFGS where, instead of keeping an accurate estimate of the inverse Hessian, a diagonal scaling matrix D is used instead. With this diagonal matrix, the update for the weight vector becomes

$$w_{t+1} \leftarrow w_t - \frac{1}{t} D \cdot \nabla^{(s)} \hat{g}(w_t).$$

Here, $1/t$ represents the learning rate, inspired by the success of Pegasos. At iteration $t+1$, the estimate D is updated using

$$D \leftarrow \left(1 - \frac{2}{t+1}\right) D + \frac{2}{t+1} \cdot \frac{w_{t+1} - w_t}{\nabla_t^{(s)} \hat{g}(w_{t+1}) - \nabla_t^{(s)} \hat{g}(w_t)},$$

where the $2/(t+1)$ term is a dampening factor, $\nabla_t^{(s)}$ is the subgradient evaluated at the training example (x_t, y_t), and $(w_{t+1} - w_t)/(\nabla_t^{(s)} \hat{g}(w_{t+1}) - \nabla_t^{(s)} \hat{g}(w_t))$ is an estimate of the inverse second derivative[7].

[7]This is because $f'(x) \approx \frac{f(x+h)-f(x)}{h}$ for h small

The use of a diagonal scaling matrix to approximate second-order gradient methods has been studied previously in the context of neural networks [Becker and LeCun, 1989]. An important point, which is also addressed in oLBFGS, is that this requires an extra gradient computation compared to SGD: in SGD, we only find $\nabla_t \hat{g}(w_t)$, and the next iteration finds $\nabla_{t+1} \hat{g}(w_{t+1})$, which is distinct from the $\nabla_t \hat{g}(w_{t+1})$ term required here.

The only results on SGD-QN are those from the Pascal Challenge, though these are promising: the other entries in the wild track included algorithms such as LibLinear and SVM-SGD. A formal publication is required to study it more carefully, though. In particular, it is not clear if the diagonal scaling matrix improves the convergence rate of SVM-SGD. While it uses the $1/t$ learning rate of Pegasos, it does not use a projection step: is it the case that the diagonal scaling allows such a learning rate to be feasible? And would adding a projection step improve performance? Finally, updating D at every iteration would require $O(n)$ work, and so the authors prescribe doing this only once in a while. It is not clear how often is enough to ensure good performance. These questions are important, because a heuristic, while potentially useful, is not theoretically convincing.

4.3.4 FOLOS

While we have studied solvers specifically for SVMs thus far, as we discussed in §3.1, SVMs fall under the general class of methods having an objective function of the form $\hat{g}(w) = \ell_{\text{emp}}(w) + r(w)$, where $\ell_{\text{emp}}(w)$ is a convex measure of empirical loss, and $r(w)$ is a convex regularization term. Therefore, a convex optimizer that solves a general problem of this form is trivially also an SVM solver.

FOLOS [Duchi and Singer, 2008] is such a solver for the loss-plus-regularization class of convex optimization problems. The algorithm is a simple extension to the subgradient projection method for solving convex problems, where instead of doing a projection on the result of a gradient step, FOLOS does an analytic minimization. The intuition is that to minimize a function $\ell_{\text{emp}}(w) + r(w)$, we want to be close to whatever w_t minimizes $\ell_{\text{emp}}(w)$, but we also want $r(w)$ to be small as well. So, FOLOS uses the following pair of updates:

$$w_{t+\frac{1}{2}} = w_t - \eta_t \nabla^{(s)} \ell_{\text{emp}}(w_t)$$

$$w_{t+1} = \underset{w}{\text{argmin}} \left(\frac{1}{2} ||w - w_{t+\frac{1}{2}}||^2 + \eta_{t+\frac{1}{2}} r(w) \right).$$

The w_{t+1} update tries to find a weight vector that is close $w_{t+\frac{1}{2}}$ (which in turn is close to the minimum of $\ell_{\text{emp}}(w)$), but is also close to the minimum of $r(w)$.

Now, since $0 \in \partial f(w) \iff (\forall v) f(v) \geq f(w)$, w is the minimum of the function ℓ_{emp} iff 0 belongs in the subgradient of ℓ_{emp} evaluated at w. So, since w_{t+1} is defined to be a minimizer, it can be shown that this property implies

$$w_{t+1} = w_t - \eta_t \nabla^{(s)} \ell_{\text{emp}}(w_t) - \eta_{t+\frac{1}{2}} \nabla^{(s)} r(w_{t+1}).$$

This equation gives an alternate expression for the w_{t+1} vector that we end up with after performing the projection step given earlier. The critical point of this equation is that w_{t+1} involves not only the subgradient $\nabla^{(s)} \ell_{\text{emp}}(w_t)$, but also $\nabla^{(s)} r(w_{t+1})$, which is the gradient of $r(w)$ at w_{t+1}; that is, w_{t+1} implicitly influences our update even before we have evaluated it! This is the reason for the name FOLOS, as the gradient is *forward looking*. The algorithmic implication of this is that our computation of w_{t+1} implicitly includes a regularization term on w_{t+1}, which can lead to sparse solutions if we choose ℓ_1 regularization. Note that we could not have done this update directly, since we can't directly compute $r(w_{t+1})$ from w_t. If we instead used $r(w_t)$ in the update, we would not get a truly sparse solution; this is because $r(w_t)$ would only penalize the *previous* weight vector, rather than the current one.

While intuitively sensible, an important question is whether these updates converge to the right solution. The authors show that this is the case both in the batch and online cases. For the batch case, the FOLOS algorithm has $O(1/\sqrt{T})$ convergence to the optimal solution, where T is the number of iterations of the algorithm. More precisely, with a learning rate $\eta_t \propto \frac{1}{\sqrt{t}}$,

$$\min_{t \in \{1 \dots T\}} \hat{g}(w_t) - \hat{g}(w^*) = O\left(GD\frac{\log T}{\sqrt{T}}\right),$$

where w^* is the optimal solution, $||w^*|| < D$ and G is a bound on the norm of any subgradient in ∂f and ∂r.

For the online case, assuming that one of ℓ_{emp}, r is *H-strongly convex*[8] for some constant H, one can show that with a learning rate of $\eta_t = 1/t$, the regret after T iterations is

$$R(T) = O\left(\frac{G^2 \log T}{H}\right).$$

Combined with the online-to-batch generalization result of [Kakade and Tewari, 2009], this tells us that FOLOS converges in $\tilde{O}(\frac{1}{H\epsilon})$ iterations. This is comparable to the Pegasos iteration bound when FOLOS is applied to the SVM problem.

As the above follows from the classical subgradient projection technique, the gradients used are the true gradients over the entire training set, *not* the stochastic gradients evaluated at each example. If this assumption were required in FOLOS, it would restrict its applicability in the batch learning framework. Fortunately, the authors state if we replace $\nabla^{(s)}\ell_{\text{emp}}$ with a stochastic approximation $\tilde{\nabla}^{(s)}\ell_{\text{emp}}$ (e.g. the subgradient evaluated only at a single example, as with SGD), the above guarantees still hold in expectation. Therefore, FOLOS can be operate as an online-style algorithm.

The FOLOS framework is quite general, encompassing a recent truncated gradient approach of [Langford et al., 2008] when we use $r(w) = ||w||_1$. When we use $r(w) = \frac{\lambda}{2}||w||_2^2$, as with SVMs, then the update rule is

$$w_{t+1} = \frac{w_t - \eta_t \nabla^{(s)}\ell_{\text{emp}}}{1 + \eta_t \lambda}.$$

This means that the update is a stochastic subgradient step that then shrinks the result towards 0, with the aim of reducing the norm. This is similar in intent to the projection step of Pegasos.

An important direction for further study of FOLOS is comparing it to other methods on realistic data. The experiments in the paper are, by the authors' own admission, preliminary, and reflect a medium- scale learning task (the training set consists of 1000 normally distributed examples (plus noise) in $\mathbb{R}^{400}$). The results on this synthetic dataset are promising: FOLOS is competitive with Pegasos, and learns a sparse weight vector when ℓ_1 regularization is used.

4.4 Dual quadratic programming methods

As we discussed in §3.2, arguably the main historic reason for favouring the dual SVM problem is that it can handle kernels easily. With the focus on linear SVMs in the large-scale setting, we have seen several methods that shifted back to the primal. But even in the linear case, the dual is amenable to certain optimization techniques (such as coordinate descent) that have been well-studied in optimization literature, and have proven to converge quickly. The dual solvers that we study use one or more of these special techniques to quickly achieve a good solution.

[8]A function f is H-strongly convex if for any x, y, $f(y) \geq f(x) + \nabla f(x) \cdot (y - x) + \frac{H}{2}||y - x||^2$. It can be shown that f is H-strongly convex iff $f - ||\cdot||^2/2$ is convex. The SVM problem is strongly-convex, because the ℓ_2 regularizer $r(w) = \frac{\lambda}{2}||w||^2$ is strongly-convex.

4.4.1 SVM$^{\text{perf}}$

SVM$^{\text{perf}}$ [Joachims, 2006] is a popular solver linear SVMs, and was one of the first SVM training algorithms to have *linear* dependence on both the number of examples and average number of non-zero feature values. This fact, combined with a free implementation, has made it a standard solver that is used as a baseline for all modern training algorithms.

SVM$^{\text{perf}}$ solves the standard SVM problem by viewing it as a special case of *structural* SVMs, which are used to predict structured output. The primal problem for a structural SVM is

$$\text{minimize } \frac{\lambda}{2}||w||^2 + \zeta \text{ subject to } \forall c \in \{0,1\}^n,$$
$$\sum_{i=1}^n c_i y_i (w \cdot x_i) \geq \sum_{i=1}^n c_i - \zeta n.$$

Notice that there are exponentially many constraints to the problem, which is to be expected for structural prediction. What is desirable however is that there is only a single slack variable, ζ, shared across each of these constraints. This affords some flexibility in solving the problem. The paper shows how to solve the structural problem with a *cutting plane* method, where instead of solving a difficult convex problem, we solve an approximation of it consisting of a number of hyperplanes (these are the cutting planes); this is similar to the idea behind bundle methods, which we analyze in §4.4.3, where it turns out that SVM$^{\text{perf}}$ can be viewed as a special case of a bundle method. We can solve the structural SVM problem iteratively by keeping a working set $\mathcal{W}$ of constraint indices, and solving the problem restricted to the constraints in $\mathcal{W}$. Each element of $\mathcal{W}$ is a vector $w \in \{0,1\}^n$, and so can be thought of as some combination of training point indices; when $\mathcal{W} = \{0,1\}^n$, then we solve the exact structural problem. This working set is updated at each iteration to include the indices for the points that are currently misclassified: we add a new binary vector w to $\mathcal{W}$, where w_i denotes if the ith training example is misclassified or not.

The algorithm terminates when it is within ρ of the optimal primal solution, and this is shown to happen in at most $O(1/\lambda\rho^2)$ iterations, and so SVM$^{\text{perf}}$, like Pegasos, depends on the difficulty of the task ($\lambda \to 0$ the training time increases). Unlike Pegasos, the dependence on $1/\rho$ is quadratic, meaning that it takes longer to reach an approximate solution.

The benefits of studying the structural SVM problem are also seen in the performance of the LaRank algorithm [Bordes et al., 2007]. LaRank was originally designed for dealing with very large multi-class prediction problems, such as those encountered when doing structured prediction, but has been shown to be competitive for binary classification problems as well.

4.4.2 LibLinear

LibLinear is a popular linear classification package that is related to LibSVM, a library for general (not necessarily linear) SVMs. The algorithm behind LibLinear is described in [Hsieh et al., 2008], which uses *coordinate descent* on the dual SVM formulation. The idea of coordinate descent is to perform a series of univariate optimizations, one for each coordinate of the variable of interest. Roughly, to optimize a function $f(x_1, \dots, x_n)$, for each coordinate i, we fix all other coordinates and try to find the optimal value of x_i. In other words, we assume that the values $\{x_1, \ldots, x_{i-1}, x_{i+1}, x_n\}$ are all correct, and that only x_i is suboptimal.

The algorithm applies this idea to the dual SVM problem; recall that this is (Equation 4)

$$\underset{\alpha}{\text{minimize}}\ f(\alpha) = \sum \alpha_i - \frac{1}{2}\alpha^T Q \alpha \text{ subject to } 0 \le \alpha_i \le U,$$

where the values of Q, U depend on the choice of loss function (hinge- or square-loss). Suppose we have some current (suboptimal) solution vector α, and want to find the optimal value for the ith coordinate given all other coordinates. This can obviously be solved using

$$\underset{d}{\text{minimize}}\ f(\alpha + d e_i) \text{ subject to } 0 \le \alpha_i + d \le U, \tag{5}$$

where e_i is the ith standard basis vector. When $d = 0$, this means that the value of α_i is optimal. It is easy to check that this function is a quadratic, and its Taylor expansion is:

$$f(\alpha + d e_i) = K + d \cdot \nabla_i f(\alpha) + \frac{d^2}{2} Q_{ii},$$

where K is some constant. Clearly $d = 0$ is the minimizer of Equation 5 iff $\max(0, \nabla_i f(\alpha)) = 0$. Otherwise, the unconstrained optimal d value is

$$d = -\frac{\nabla_i f(\alpha)}{Q_{ii}}.$$

To compute $\nabla_i f(\alpha)$ efficiently, it is useful to also store the primal weight vector w in addition to α. Using the representer theorem for w, $\nabla_i f(\alpha)$ can be shown to be

$$\nabla_i f(\alpha) = y_i(w \cdot x) - 1 + D_{ii}\alpha_i,$$

from which we deduce the update for α is

$$\alpha_i \leftarrow \min(\max(\alpha_i - \nabla / Q_{ii}, 0), U).$$

Of course, one also needs to update w in order to use it to compute $\nabla_i f$. The update for w has the simple form

$$w \leftarrow w + (\alpha_i - \alpha_i^{\text{old}}) y_i x_i.$$

The algorithm is presented in Algorithm 3. The fact that we store both α and w means that, while the algorithm is derived using the dual SVM formulation, we can analyze it in terms of how it updates w. To compare the update of w to that of SGD, note that the step of picking a coordinate can be done in several ways: one can simply cycle through all points in presented order, one can shuffle the points and cycle through them, or one can pick a point at random. The latter is similar to an SGD approach. In fact, the update of w is reminiscent of the SGD update, but for two differences: (1) there is no learning rate, but instead the scaling factor $(\alpha_i - \alpha_i^{\text{old}})$ is used, (2) instead of using the (sub)gradient $\nabla f_{\text{primal}} = \lambda w - y_i x_i \cdot 1[y_i(w \cdot x_i) < 1]$, we use the direction $y_i x_i$. Loosely, point (1) can be viewed in a similar light to the quasi-Newton update, where scaling is taken care of by the Hessian matrix rather than a manually tuned learning rate. Notice also that we need to project the α_i's to ensure that the box constraints are met. This means that the algorithm can be thought of as a generalized SGP method, with the learning rate found using analytic minimization.

It is shown that this algorithm converges in only $O(\log 1/\rho)$ iterations, which makes it very attractive in practice. Experiments in the paper show that the algorithm achieves a low generalization error solution quicker than Pegasos, likely because the latter needs $\tilde{O}(d/\lambda\rho)$ iterations. However, in §5.3 we point out that Pegasos' runtime *decreases* on larger training sets, and it is not clear that the same is true of this method.

```
Let $\alpha \leftarrow$ zero vector in $\mathbb{R}^n$
Let $w \leftarrow$ zero vector in $\mathbb{R}^d$

while $\alpha$ is not optimal do
  Pick an index $i \in \{1, \ldots, n\}$
  $\alpha_i^{\text{old}} \leftarrow \alpha_i$
  $\nabla \leftarrow y_i(w \cdot x_i) - 1 + D_{ii}\alpha_i$
  $\nabla_P \leftarrow \min(\max(\nabla, 0), U)$

  if $\nabla_P \neq 0$ then
    $\alpha_i \leftarrow \min(\max(\alpha_i - \nabla/Q_{ii}, 0), U)$
    $w \leftarrow w + (\alpha_i - \alpha_i^{\text{old}}) y_i x_i$
  end if
end while
```

Algorithm 3: The dual coordinate descent algorithm of Hsieh et al. [2008].

4.4.3 Bundle method for risk minimization (BMRM)

Recall our characterization of the SVM problem as regularized risk minimization, where risk is measured using hinge-loss. The BMRM approach [Smola et al., 2007] is a solver of general risk minimization problems that uses bundle methods. A bundle method refers to any technique that minimizes a non-smooth (but convex) objective function by approximating it using an envelope of subgradients (Figure 2). One can then try to minimize this envelope, which is a simpler problem as it constitutes a set of linear functions. Further, it has been observed that the number of subgradients required for a good fit of the true objective is independent of the number of training examples that constitute the risk. This is similar to SVM$^{\text{perf}}$'s idea of keeping a small set of linear constraints, and indeed it turns out that BMRM is an extension of SVM$^{\text{perf}}$ to a more general framework. The advantage of this is that it allows us to prove much sharper convergence bounds.

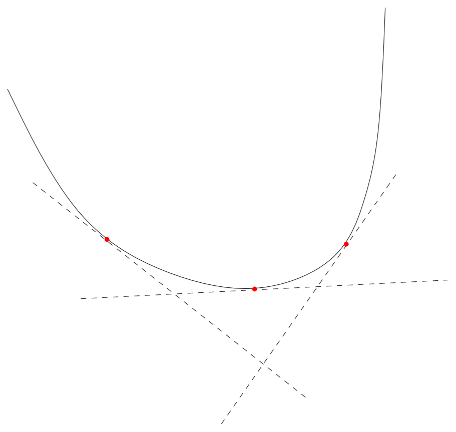

Figure 2: An example of approximating a convex function using a lower envelope of tangential hyperplanes.

In the BMRM approach, we minimize regularized risk $\hat{g}(w) = \ell_{\text{emp}}(w) + r(w)$, as in Equation 1, by keeping an envelope of subgradients for $\hat{g}(w)$ at iterates $\{w_1, w_2, \ldots\}$. At each iteration, this envelope is updated to include the subgradient at the current point w_t. We do this by keeping an offset b_t, which is computed using

$$b_{t+1} = \ell_{\text{emp}}(w_t) - \nabla^{(s)}\ell_{\text{emp}}(w_t) \cdot w_t.$$

With this offset, the function $f(w) = b_t + \nabla^{(s)}\ell_{\text{emp}}(w_t) \cdot w$ defines a hyperplane tangential to $\hat{g}(w)$ at the point $w = w_t$. By keeping a lower envelope of these hyperplanes, we get an increasingly refined approximation to $\hat{g}(w)$. The iterates of w are simply taken to be the minimizers of the current approximation of the objective function:

$$w_{t+1} = \operatorname*{argmin}_{w} \left\{ r(w) + \max_{t' \leq t+1} \left[b_{t'} + (\nabla^{(s)} \ell_{\text{emp}}(w_{t'})) \cdot w \right] \right\}.$$

Of course, this minimization is not trivial as-is; to solve it, the authors move into the dual-space, and show that the minimization problem is equivalent to

$$\max_{\alpha} -\frac{1}{2\lambda} \alpha^T Q \alpha + \alpha \cdot b \text{ such that } ||\alpha||_1 = 1, \alpha_i \geq 0,$$

where $Q_{ij} = \nabla^{(s)} \ell_{\text{emp}}(w_i) \cdot \nabla^{(s)} \ell_{\text{emp}}(w_j)$. The size of this program is $t \times t$ at the tth iteration, which means it can be quickly solved when t is small. The authors show that the bundle method converges in $O(1/\rho)$ iterations, which means that in the worst case, we solve a problem of size $O(1/\rho) \times O(1/\rho)$. In fact, for continuously differentiable objective functions, this bound is drastically improved to $O(\log 1/\rho)$ iterations.

The paper's experimental comparison with Pegasos show that BMRM is a better optimizer, as it approaches the optimal solution faster. However, as we discuss in §5.2, this does not necessarily translate into significantly better generalization. Another question is the relation of SVM$^{\text{perf}}$ and BMRM. According to the paper, SVM$^{\text{perf}}$ is essentially equivalent to BMRM when it is applied to the SVM problem. But the results for SVM$^{\text{perf}}$ reported in [Shalev-Shwartz et al., 2007] are markedly different than the ones in this paper; in [Shalev-Shwartz et al., 2007], SVM$^{\text{perf}}$ is sometimes a factor of 10 slower than Pegasos. It is not clear why there is a substantial difference between the two sets of results (a similar issue of conflicting results was cited as a motivation for the ICML'08 Pascal Challenge [Pascal, 2008]). Interestingly, Pegasos was similarly reported to be worse than SVM$^{\text{perf}}$ in the results published in [Franc and Sonnenburg, 2008], which we discuss in §4.5.1.

4.5 Other solvers

While most large-scale SVM solvers have been based on primal gradient descent or dual quadratic-programming, this does not represent the complete spectrum of solvers. Here, we discuss two promising approaches, one working with the primal and the other with the dual.

4.5.1 OCAS

OCAS [Franc and Sonnenburg, 2008] is an SVM solver that uses an improved cutting plane technique, building on the work of SVM$^{\text{perf}}$ and BMRM. It is based on the simple observation that when we minimize the lower envelope in a bundle method, we do not necessarily minimize the true objective function. That is, if we let $f(w)$ denote the approximation to the regularized risk term $\hat{g}(w)$, then the iterates $\{w_t\}$ of a bundle method are guaranteed to satisfy $f(w_{t+1}) < f(w_t)$, but $g(w_{t+1}) > g(w_t)$ is possible. While the bundle method eventually converges to the optimal solution, such iterates are not helpful and slow convergence. The idea of OCAS is to force the true objective function to also decrease monotonically across iterates. To do this, we do a line-search to minimize the true objective function. This initially seems like it might be as complicated as the original problem; minimizing the true objective via a line search is not obviously easier than minimizing it globally. However, the authors show how this line search can be done in $O(n \log n)$ time, using a simple sort-based algorithm.

Theoretically, OCAS shares the same convergence bound as SVM$^{\text{perf}}$; this is not surprising, as each step of OCAS can be no worse than that of SVM$^{\text{perf}}$. As a result, its runtime is simply stated as $O(nd)$, which is the same as SVM$^{\text{perf}}$. In the reported experimental results, however, OCAS is significantly faster than SVM$^{\text{perf}}$, usually by a factor of 5, indicating that OCAS has a smaller constant in its big-O bound. Surprisingly, OCAS is also reported to be faster than Pegasos, by virtue of Pegasos being *slower* than SVM$^{\text{perf}}$ on some datasets. This is completely at odds with the results from the Pegasos paper, which are on the same datasets! Interestingly, as we noted earlier, the results of the BMRM paper also show SVM$^{\text{perf}}$ outperforming Pegasos. Consequently, Pegasos' true behaviour is by no means firmly established, and requires close study.

4.5.2 Core vector machines

The margin-based motivation for SVMs suggests there is some geometry involved in the method. A geometric formulation of SVMs has been established [Bennett and Bredensteiner, 2000], which allows for SVMs to be solved using results from computational geometry. One of the most interesting geometrically inspired SVM solvers is the Core Vector Machine (CVM) algorithm [Tsang et al., 2005], which has theoretically appealing scalability results. The CVM is the first work we are aware of where the training time is linear in the number of examples (it precedes SVM$^{\text{perf}}$ by a year): to achieve an accuracy of ρ, the CVM runs in time $O(d \cdot (n/\rho^2 + 1/\rho^4))$. (In fact, the paper proposes a speedup through randomization that makes its runtime *independent* of the number of examples, preceding Pegasos by two years.) It is also one of the first solvers to look explicitly for an approximate solution to the SVM problem, motivated by the success of approximation algorithms in theoretical computer science. As we will see in §5, [Bottou and Bousquet, 2007] shows that the SVM problem does not need to be solved exactly to achieve good generalization. Therefore, CVMs can be seen as an application of this major principle of [Bottou and Bousquet, 2007], although they were designed before that paper (and are not cited).

The idea of the CVM algorithm is to connect SVMs to a problem from computational geometry known as the *minimum enclosing ball* (MEB). This simply asks for the ball of smallest radius that encloses a given set of points. Similar to the optimization problem for SVMs, an exact solution to this problem is expensive in terms of time and space. However, if one looks at the ρ-approximate minimum enclosing ball, then one can find a solution in time linear in the number of points. The authors show how certain types of nonlinear SVMs can be solved by repeated computation of MEBs, *provided* we use the square-loss.

The authors show how the runtime can be made *independent* of the number of examples, but in this case the runtime becomes $O(d/\rho^8)$, which is likely infeasible for small ρ. Subsequent optimizations by the authors [Tsang et al., 2007] have reduced this to $O(d/\rho^4)$, which is still expensive. However, in light of [Bottou and Bousquet, 2007], we only need ρ to be "small enough", as strenuous optimization does not necessarily correspond to greatly reduced generalization error. In this case, an $O(d/\rho^4)$ algorithm could be faster than one that has superlinear dependence on the number of examples n, even if the ρ dependence of the latter is better.

One limit on the scope of the CVM algorithm is that it assumes the use of *normalized kernels*. This means that $K(x, x)$ is assumed to be a constant for any x: the examples are taken to be mapped onto a hypersphere by the implicit kernel mapping Φ. The authors claim that many real-world kernels satisfy this property, but this is in fact a fairly strong assumption about how the data is scaled, even in the case of a linear kernel; [Joachims, 2006] for example disagrees with the authors' assertion.

Interest in CVMs may have been dampened by claims that their experiments were not comprehensive enough [Loosli and Canu, 2007], and that its behaviour with respect to the SVM parameter C [9] was erratic. However, in [Tsang and Kwok, 2007], it is pointed out that one reason for these apparent anomalies is a buggy implementation of `random` in Linux. ([Sonnenburg et al., 2007] cites this as a good example of why code for machine learning algorithms needs to be published as open source, which we agree with.) But it is worth noting that even in the authors' updated results, CVMs do not achieve good test set error on a separable dataset (`checkers`), where SVMs should be able to achieve zero test error. As with all SVM training methods, a detailed comparison to subsequent solvers is required to assess the current viability of CVMs.

5. WHY SGD WORKS ON LARGE TRAINING SETS

In §4.3, we presented a number of SVM solvers based on some version of SGD, with their gradient computations based on a single example. We noted that while such methods can complete iterations much quicker than those that use the true gradient, it is well known that SGD is a poor optimization algorithm; hence, one might expect that these training methods sacrifice accuracy for speed. In this section, we show why this is not the case, presenting results that show that SGD is poor at optimization but good at generalization. The latter is the true goal of learning; optimization is only a means toward this end. We then present a very surprising result (§5.3) due to [Shalev-Shwartz and Srebro, 2008] that shows why SVM training time should *decrease* as the number of examples increases! The paper shows that Pegasos satisfies this desideratum, proving the important place that SGD has for SVM training. To understand this analysis, we first need to make explicit the distinction between an empirical and learning optimum.

5.1 Empirical optimum vs. learning optimum

[Bottou and Le Cun, 2004, 2005] show how an online-style, generalized second order [10] stochastic gradient algorithm can be a better choice than a batch second order gradient algorithm for large-scale problems. Here, "better" means that a stochastic algorithm can achieve lower generalization error than a batch algorithm in a fixed amount of time. This is initially surprising: while it has been observed that (2)SGD achieves low test-set error in specific practical applications despite being a poor optimization algorithm, this paper shows why 2SGD will *always* outperform 2GD on very large datasets. The key fact, one which the subsequent papers we discuss also exploit, is that the optimization problem in learning is only a surrogate for our true goal, which is achieving good generalization. While 2GD can solve the training optimization much quicker than 2SGD, it minimizes the generalization error at a slower rate.

The paper specifically compares 2SGD and 2GD in minimizing *unregularized* generalization error. This means that the result does not directly apply to SVMs, but later sections show how this extension can be done. We first notice that when we have infinitely many examples, 2SGD directly minimizes this error, since it randomly chooses a training example at each iteration; when $n \to \infty$, these examples follow the asymptotic distribution of the data. To compare its performance to 2GD, one needs to study how fast 2GD converges to the generalization optimum. To do this, we need to find how large n needs to be for the 2GD solution to be as close to the generalization optimum as 2SGD. Clearly, if we could process infinitely many examples, minimizing the training error would be equivalent to minimizing the generalization error. In practice, we can only process a finite subset of these examples, and so we need to see how large this subset has to be. We solve this problem as follows. Let θ_n^* denote the optimal batch solution obtainable n examples, which is the result of 2GD with infinitely many epochs. We want to know how fast this empirical optimum θ_n^* converges to $\theta^* = \theta_\infty^*$, the optimal parameter vector when we have access to infinitely many examples. (As argued earlier, $\theta^* = \text{argmin } g(\theta)$.)

Once we find this rate of convergence, we can find how fast $\theta_n(t)$ converges to θ^*, and thus how large n needs to be. The authors show that, surprisingly,

$$\theta_n^* = \theta_{n-1}^* - \frac{\Psi_n}{n}\frac{\partial \ell}{\partial \theta}(x_n, y_n, \theta_{n-1}^*) + O(1/n^2),$$

where Ψ_n is a matrix that converges to H^{-1}. This is surprising because it is almost identical to a 2SGD update, indicating that the behaviour of the empirical optimums and the solutions for 2SGD have the same general form. The difference between the two recurrences is the nature of the scaling matrices, and the precise nature of the lower order terms. It is plausible that these might affect the two rates of convergence; however, the authors show that assuming only the convergence of the scaling matrices to H^{-1}, neither of these differences has an influence on convergence, and so the empirical optimums of 2GD converge as slowly to the generalization optimum as 2SGD, despite doing a factor of $O(n)$ extra work!

Now suppose that we want 2GD to achieve the same generalization error as 2SGD. Given the above result, it can be shown that 2SGD will be a factor of $O(\log \log n)$ faster in reaching the same solution as 2GD. Of course, $\log \log n$ is a small constant in practice, the point of the paper is that 2SGD is *at least* as good at 2GD for learning. This was observed empirically at the time, but the paper was the first work to theoretically prove why this is the case.

Note that the paper's result only works when we are using a second-order SGD method, or a method that is asymptotically second-order (i.e. it uses a scaling matrix that converges to the Hessian). We **cannot** claim that plain SGD is similarly better than a batch algorithm. The authors point out that an important theoretical direction is to analyze what happens when we use a low-rank approximation to the Hessian. Also, it is unclear whether a *different* batch algorithm might be able to converge to the global optimum faster than 2GD, and hence faster than 2SGD.

This work raises the distinction between the *empirical* and *learning* optimum. 2SGD is able to give better generalization performance than 2GD when both are run for the same amount of time, despite 2SGD being worse at finding the empirical optimum to the training optimization problem. It reminds us that the optimization formulation of a learning problem is only a surrogate for finding the solution with best generalization. To further investigate the performance of SGD based algorithms, the next paper explicitly incorporates approximation into the sources of error for learning.

5.2 Stochastic gradient and estimation error

[Bottou and Bousquet, 2007] build on the previous paper to show concretely why a learning algorithm can be poor at optimization, but excellent at generalization for a large-scale learning task. They develop a framework that explicitly incorporates the effect of optimization error in generalization, and analyze the asymptotic behaviour of this error when the number of examples is large. This is done by looking at the behaviour of the *estimation error*, which is intuitively a measure of how representative the training set is of the underlying distribution. By analyzing the interplay of these errors,

[9] Recall that we set $\lambda = 1/nC$.

[10] A generalized second order method is simply one that uses a scaling matrix that *converges* to the inverse Hessian.

their conclusion is that stochastic gradient algorithms (SGD and 2SGD) are able to leverage a decrease in estimation error in large datasets to achieve a low generalization error quicker than batch gradient algorithms (GD and 2GD). This shows that in addition to 2SGD being superior to 2GD, as evinced in the previous section, plain SGD is quantifiably better than plain GD.

One of the motivations of the paper is the question of how useful approximate solutions to learning problems are: how much do we pay if we only approximately solve the optimization problem that models our learning goal? The authors answer this question by showing that one need not focus on very accurate solutions for large scale problems, as it is still possible to achieve a good generalization error with only approximate solutions. This is in itself interesting, because usually (e.g. in theoretical CS) the optimization problem is precisely the mathematical encoding of the problem of interest. However, in learning the true quantity of interest is the generalization error; the optimization formulation of the problem is simply a compensation for the fact that we don't know the underlying distribution of test data. Explicitly noting this fact is one of the paper's most important contributions.

Formally, suppose we have an unregularized learning problem, and wish to minimize empirical loss ℓ_{emp} on the training set. Suppose that $\hat{\theta} = \text{argmin } \ell_{\text{emp}}(\theta)$ is the empirical minimizer. A learning algorithm tries to find this vector using an optimization algorithm, which in practice can only produce a solution $\tilde{\theta}$ such that $\ell_{\text{emp}}(\tilde{\theta}) \leq \ell_{\text{emp}}(\hat{\theta}) + \rho$, for some positive constant ρ. For this $\tilde{\theta}$, the authors decompose the generalization error $\epsilon = g(\tilde{\theta})$ as $\epsilon = \epsilon_{\text{app}} + \epsilon_{\text{est}} + \epsilon_{\text{opt}}$, where:

1. ϵ_{app} is the *approximation error*, which measures how good the hypothesis class is. This is $\min g(\theta)$, the minimum error that *any* model in this hypothesis class must incur. When our objective function is regularized, ϵ_{app} can be controlled by changing the regularization parameter λ.

2. ϵ_{est} is the *estimation error*, which measures how good the training set is. It is defined as $\epsilon_{\text{est}} = g(\hat{\theta}) - \min g(\theta)$, where $\hat{\theta} = \text{argmin } \ell_{\text{emp}}(\theta)$, the expected difference between the minima of the training and true (i.e. expected) loss. When the training set is large and representative of the underlying distribution of the data, then this difference will be small. So, ϵ_{est} can be controlled by training on a larger number of examples.

3. ϵ_{opt} is the *optimization error*, which measures how accurate the optimization solver is. This is controlled by the optimization tolerance ϵ_{tol}, which we denote by ρ to avoid confusion from the generalization error components. ρ is controlled by stopping our optimization algorithm at the appropriate point. With ρ, the optimization error is defined as $g(\tilde{\theta}) - g(\hat{\theta})$. We will use ρ to denote the tolerance in the rest of this document; however, it is important to note that most literature on SVM training uses ϵ instead, because they do not focus explicitly on generalization.

The goal of a learning algorithm is to minimize ϵ, subject to two constraints: the number of examples n we process, and the time T available are both bounded by some constants. If we let $n \to \infty$, we can make ϵ_{est} small by looking at as many training examples as possible. In this case, we can choose ρ to be some moderate positive constant, sacrificing exact optimization due to the benefit from looking at more training examples. To make the tradeoff more concrete, we need a way to connect ρ and ϵ. The authors use *estimation bounds*, which study the rate of decrease of ϵ_{est} as the training set size n increases. In a large-scale learning setting, these bounds are of immediate interest, because they let us quantify the tradeoff between looking at more examples (i.e. effectively working with a larger training set) versus optimizing the empirical objective function. The authors' bound is

$$0 \leq \epsilon - \epsilon^* \leq c \cdot \left(\epsilon_{\text{app}} + \frac{d}{n} \log \frac{n}{d} + \rho \right), \tag{6}$$

for some positive constant c, where ϵ^* denotes the minimum generalization error over all hypothesis spaces. This tells us that we should optimally choose $\rho = O\left(\frac{d}{n} \log \frac{n}{d}\right)$.[11] Further optimization is unnecessary, as the generalization error ϵ would be asymptotically unaffected. We can use this to compare first- and second-order gradient and stochastic gradient descent by finding the time the algorithms take to reach a generalization error that is at most $c \cdot (\epsilon_{\text{app}} + \mathcal{E})$ as a function of the additive term $\mathcal{E}$. To do this, we simply set $\rho = O(\mathcal{E})$, which implies that $n = O(d \log(1/\mathcal{E})/\mathcal{E})$ examples are sufficient; while the training set is taken to be practically infinite, further training examples are unnecessary for asymptotic improvement in the generalization bound. The results of this analysis are shown in Table 2. We see that SGD, although poor at optimization, is significantly faster at minimizing generalization error than both GD and 2SGD. This explains the common empirical observation that SGD performs very well on practical learning problems, even though it is a relatively poor optimization algorithm. While GD can find the empirical optimum much quicker than 2SGD when ρ is small, it pays the price of scaling linearly with the number of training examples n. SGD on the other hand remains agnostic about the number of examples for its runtime, but still enjoys the decrease in estimation error as n increases.

By extending the analysis of [Bottou and Le Cun, 2004], this paper shows concretely why finding a solution with low optimization error is not the same as finding one with low generalization error. On very large datasets, the estimation error can be shown to decay like roughly $O(\log n/n)$, and so the optimization accuracy ρ can be increased. Since stochastic gradient algorithms do not explicitly depend on the number of examples, they can take use of the increase in ρ without spending more time on the larger dataset. While the result is still for unregularized learning, the next paper builds on this idea, and applies it to the SVM problem.

5.3 Decreasing SVM training time on larger training sets

The previous paper showed why, for unregularized problems, we can expect SGD to generalize better than GD on a large training set. The key fact was that an increase in n decreases the estimation error, and yet does not increase the runtime of SGD. Building on this analysis for the SVM problem, Shalev-Shwartz and Srebro [Shalev-Shwartz and Srebro, 2008] argue that when presented with more training data, the runtime of an SVM solver should actually *decrease*. This is in contrast to most SVM solvers, whose runtime increases with more data, usually superlinearly! The authors point out that traditional runtime analyses of SVM solvers study them from the point of view of how well they solve the optimization problem. But of course, their real goal is to find a solution with good generalization. Using the decomposition of [Bottou and Bousquet, 2007], the authors shows why an SVM solver ought to use the decreased estimation error that more examples afford, and consequently run faster on this larger dataset. Further, they show that Pegasos meets this requirement, being the first solver to do so.

[11]The authors' analysis is a little more specific than this, and includes a so-called estimation rate α; our summary is for the case $\alpha = 1$.

Algorithm	Time to reach optimization accuracy ρ	Time to reach generalization error $\leq c \cdot (\epsilon_{\text{app}} + \mathcal{E})$
GD	$O\left(nd \log \frac{1}{\rho}\right)$	$O\left(\frac{d^2}{\mathcal{E}} \log^2 \frac{1}{\mathcal{E}}\right)$
SGD	$O\left(\frac{d}{\rho}\right)$	$O\left(\frac{d}{\mathcal{E}}\right)$
2GD	$O\left((d^2 + nd) \log \log \frac{1}{\rho}\right)$	$O(\frac{d^2}{\mathcal{E}} \log \frac{1}{\mathcal{E}} \log \log \frac{1}{\mathcal{E}})$
2SGD	$O\left(\frac{d^2}{\rho}\right)$	$O\left(\frac{d^2}{\mathcal{E}}\right)$

Table 2: Bounds on the time for gradient based algorithms to reach a specific optimization and generalization error. The latter is for the case where we have an infinite training set.

The first point the authors make is that the runtime should never *increase* with data set size if the generalization error is fixed: they give the example of a predictor that achieves accuracy of 5% with a thousand examples, and point out that if we are given ten thousand examples, just ignoring nine thousand of them gives us the same accuracy and runtime.[12] The question is whether we can get 5% accuracy with *less* time, given these extra examples. The rationale for why this is possible is the following. When we increase the training set size, the estimation error decreases; therefore, with a very large training set, ϵ_{est} will be small. So, to get some fixed generalization error ϵ, one can increase the optimization tolerance ρ. Optimizing to a higher tolerance will reduce the runtime compared to a small training set, *provided* it takes us less time to look through the new examples than it would to spend optimizing. That is, we now have a battle between the potential increase in runtime when we have more training examples, and the decrease in runtime due to the requirement of a lower optimization tolerance. A crucial fact about SGD is that its runtime does *not* depend on the number of examples; therefore, the authors characterize the scenario of increasing the number of examples with SGD as a "pure win". In contrast, methods such as SVM$^{\text{perf}}$ have an increased runtime in each step when the number of examples increases. This leads to a critical number of samples, before which the runtime decreases, but after which the cost of processing the new examples dominates and the runtime increases.

At first glance, our earlier claim that Pegasos' runtime decreases with more examples seems to contradict the bound on its number of iterations, which was stated as $\tilde{O}(1/\delta\lambda\rho)$. However, we can show that λ, ρ depend on the estimation error, and so can be changed based on n. This lets us derive a bound on the time required to reach a generalization of ϵ. Therefore, to properly study the behaviour of SVM solvers on large-scale problems, we need to describe the effect of estimation error, and connect this to the various SVM parameters. For the case of unlimited data ($n \to \infty$), the authors prove the following theorem that bounds the generalization error for an SVM.

THEOREM 3. *Let w be the learned weight vector when SVM optimization is done up to tolerance ρ, and let w_0 be any other weight vector. Then,*

$$g(w) \leq g(w_0) + 2\rho + \frac{\lambda}{2}||w_0||^2 + \tilde{O}(1/\lambda n),$$

where $g(w)$ is the generalization error with weight vector w.

We can view this similarly to Equation 6. Suppose that w_0 is the optimal weight vector, so that $g(w_0) = \epsilon_{\text{app}}$. Essentially, this bound decomposes the generalization error in terms of ϵ_{app}, the optimization tolerance ρ, and a decaying term $\tilde{O}(1/\lambda n)$ that takes the place of the estimation error. This rate is much stronger than the $\log n/n$ bound in Equation 6. The proof of the theorem is simple (and instructive), and we summarize it below.

PROOF. First, let $f(w) := g(w) + \frac{\lambda}{2}||w||^2$ denote the regularized generalization error, and $\hat{f}(w) := \hat{g}(w) + \frac{\lambda}{2}||w||^2$ the training error (with regularization). Now decompose $g(w)$ as

$$\begin{aligned} g(w) &= f(w) - \frac{\lambda}{2}||w||^2 \\ &= f(w) - \frac{\lambda}{2}||w||^2 - \left(f(w_0) - g(w_0) - \frac{\lambda}{2}||w_0||^2\right) \\ &= g(w_0) + (f(w) - f(w_0)) + \frac{\lambda}{2}||w_0||^2 - \frac{\lambda}{2}||w||^2 \\ &= g(w_0) + (f(w) - f(w^*)) + (f(w^*) - f(w_0)) \\ &+ \frac{\lambda}{2}||w_0||^2 - \frac{\lambda}{2}||w||^2. \end{aligned}$$

where w^* is the minimizer of the generalization error, $\text{argmin}\, g(w)$.

The second term in this expression is a difference of expected losses, and can be bounded using the corresponding empirical losses [Sridharan et al., 2008]:

$$\begin{aligned} f(w) - f(w^*) &\leq 2\max(0, \hat{f}(w) - \hat{f}(w^*)) + O\left(\frac{\log 1/\delta}{\lambda n}\right) \\ &= 2\rho + O\left(\frac{\log 1/\delta}{\lambda n}\right) \text{ by definition of } w \end{aligned}$$

Finally, by the optimality of w^*, we must have $f(w^*) - f(w_0) \leq 0$. Combining these facts,

$$g(w) \leq g(w_0) + 2\rho + \frac{\lambda}{2}||w_0||^2 + \tilde{O}(1/\lambda n).$$

□

This tells us that if we want a generalization error of $g(w) = g(w_0) + \mathcal{E}$, we can precisely bound ρ, λ and n asymptotically in terms of $\mathcal{E}, w_0$:

$$\begin{aligned} \rho &= O(\mathcal{E}) \\ \lambda &= O(\mathcal{E}/||w_0||^2) \\ n &= \Omega(||w_0||^2/\mathcal{E}^2). \end{aligned}$$

The bound on n is of particular interest, as it tells us roughly how many examples need to be looked at to allow for sufficient estimation error. Even if an algorithm's runtime scales linearly with n, in the infinite data case it can work on a subset of roughly $||w_0||^2/\mathcal{E}^2$ examples in order to achieve an excess generalization error of $\mathcal{E}$. This is beneficial because it lets us compare the runtimes of various SVM solvers for the case of practically infinite

[12]Of course, this assumes that the data is presented in random order, because otherwise we have the problem of choosing *which* examples to ignore. If we ignore the informative examples, we cannot hope to achieve the same generalization error as with fewer examples.

data (we cannot just set $n = \infty$!). For Pegasos, the earlier runtime of $\tilde{O}(d/\lambda\rho)$ becomes $\tilde{O}(d||w_0||^2/\mathcal{E}^2)$. In contrast, SVM$^{\text{perf}}$'s runtime of $O(nd\log(1/\rho)/\lambda)$ becomes roughly $O(d||w_0||^4/\mathcal{E}^3)$. Therefore, Pegasos is able to achieve a generalization error of ϵ much faster than SVM$^{\text{perf}}$ on infinitely large datasets.

With this limiting case analyzed, we can now focus on the practical case of finite (but potentially large) data. Does Pegasos only beat SVM$^{\text{perf}}$ in the limit, while the latter is superior on all practical datasets? The authors show that the finite data behaviour of Pegasos matches the desideratum of the runtime decreasing as the training set gets larger. In contrast, SVM$^{\text{perf}}$ does not display this behaviour because its runtime scales linearly with the number of examples. To carry out this analysis, the authors revisit the statement of Theorem 3, replacing the optimization accuracy ρ with the number of iterations T it takes to achieve such an accuracy. For the case of Pegasos, we know that the number of iterations is $T = \tilde{O}(d/\lambda\rho)$; substituting this,

$$g(w) \leq g(w_0) + \tilde{O}(d/\lambda T) + \frac{\lambda}{2}||w_0||^2 + \tilde{O}(1/\lambda n).$$

We can now minimize the right hand side as a function of λ: this happens when $\lambda = \tilde{\Theta}\left(\frac{\sqrt{nd/(T+1)}}{||w_0||}\right)$. If we plug in this value, and make T the subject of the equation, we find that the time $T(n, \mathcal{E})$ to reach a solution with excess generalization error $\mathcal{E}$ using n examples is

$$T(n, \mathcal{E}) = \tilde{O}\left(\frac{d}{(\mathcal{E}/||w_0|| - O(1/\sqrt{n}))^2}\right).$$

Notice that this has a vertical asymptote as $n \to ||w_0||^2/\mathcal{E}$, but, more interestingly, that T is a monotone decreasing function of n! As hoped, Pegasos' runtime decreases as it is presented more examples. In contrast, the same contrast for SVM$^{\text{perf}}$ reveals its runtime behaves like

$$T(n, \mathcal{E}) = O\left(\frac{nd}{(\mathcal{E}/||w_0|| - O(1/\sqrt{n}))^2}\right).$$

Since there is now an extra n on the numerator, we actually have a critical point $n_0 = 4||w_0||^2/\mathcal{E}^2$; before this value, the runtime T decreases a function of n, but after this point, the runtime steadily increases. Intuitively, this is the point where the cost of doing $O(n)$ work per iteration outweighs the benefits of the decreased estimation error. A rough display of the behaviour of the two methods is given in Figure 3.

6. SUMMARY AND FUTURE DIRECTIONS

The current trend of growth of information means that it is inevitable that large-scale learning problems become the norm. There is consequently a need to develop algorithms that deal with them efficiently. In this document, we have looked at large-scale training algorithms for support vector machines (SVMs), one of the most popular binary classifiers. We now look at some open directions for further research in the area.

The work of [Bottou and Bousquet, 2007] introduced formally the idea of approximate solutions to the SVM problem, and why such solutions might be preferable in that they obviously take less time to compute, but also do not affect generalization error significantly. Of course, as pointed out in [Franc and Sonnenburg, 2008], when one looks for approximate solutions to the SVM problem, then one is potentially comparing methods that have very different purposes. So, one has to choose whether to assess SVM solver based on its ability to generalize, or on its ability to solve the SVM problem exactly. The latter raises precisely the issue of [Bottou and Bousquet, 2007], namely that it treats optimization as the goal of learning. The former raises the issue of whether we should compare these SVM solvers against *other* classification methods; for example, there has been work on applying the kernel perceptron efficiently in an online setting [Dekel et al., 2008].

The last point is of independent interest: while the SVM has been very popular as a binary classification method, these do not capture all practically relevant large-scale learning problems. One very natural extension is to SVM for regression, which we have not discussed in this document; SVM$^{\text{perf}}$ has support for this. Another example is multivariate performance measures such as the F_1 score [Joachims, 2005]. While the literature on training large-scale binary SVMs is potentially relevant - especially the distinction between optimization and generalization error - it by no means represents the boundaries of what is to be studied in the field of large-scale learning.

An issue that we have not considered is that in many real-world instances of large-scale learning, such as bioinformatics, we cannot assume that the training data is completely labelled. This necessitates the development of large-scale semi-supervised SVMs, or SVMs based on active learning. There has been some nascent work in this field [Sindhwani and Keerthi, 2006, Schohn and Cohn, 2000]. As argued earlier, the refinement of methods for the fully supervised case will likely be beneficial for the semi-supervised setting, but the opposite may also be true: the results of the active learning SVM approach [Schohn and Cohn, 2000] raises some interesting questions about its use in the standard labelled setting. Might it be the case that active learning plays a similar role to SGD, that sees only a fraction of the training set and gets equivalent generalization error?

In terms of SVM solvers, there are a few directions of further research. In §4.3.3, we briefly mentioned recent work that has extended the LBFGS technique to both the online and non-smooth cases. If these strands are connected, then that allows LBFGS to be applied to the SVM problem. Given its prominence in the field of general nonlinear optimization, it is plausible that it will perform very well on the SVM task, and will be competitive at least with the approaches that seek high-accuracy solutions to the SVM problem.

The surprising conclusion of [Shalev-Shwartz and Srebro, 2008] that SVM training time should decrease with larger training sets casts new light on how we assess a training algorithm, and suggests that if done correctly, large-scale learning can in some sense be easier! The result seems to strongly advocate the use of stochastic gradient based techniques, although it is by no means the only method that can satisfy this desideratum. The essential requirement of a solver is simply that the time to process more examples does not exceed the reduced time afforded due to the reduction of estimation error.

7. ACKNOWLEDGMENTS

I would like to thank Serge Belongie and Lawrence Saul for their time serving on my committee, and their helpful comments. I thank Charles Elkan for introducing me to this topic, and for his numerous insightful suggestions as I was preparing for my exam.

References

S. Becker and Y. LeCun. Improving the convergence of back-propagation learning with second-order methods. In D. Touretzky, G. Hinton, and T. Sejnowski, editors, *Proc. of the 1988 Connectionist Models Summer School*, pages 29–37, San Mateo, 1989. Morgan Kaufman.

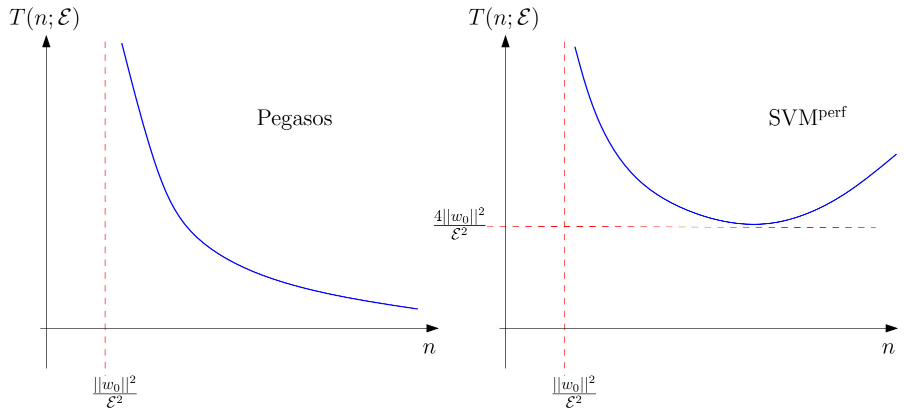

Figure 3: A visualization of the runtime behaviour of Pegasos and SVM$^{\text{perf}}$ as a function of the training set size, n. While Pegasos' runtime smoothly decreases as a function of n, SVM$^{\text{perf}}$ has a critical point n_0 beyond which it runtime starts to increase. This is the effect of the $O(n)$ processing per iteration. A similar figure appeared in the original paper [Shalev-Shwartz and Srebro, 2008].

K.P. Bennett and O.L. Mangasarian. Robust linear programming discrimination of two linearly inseparable sets. *Optimization Methods and Software*, 1:23–24, 1992.

Kristin P. Bennett and Erin J. Bredensteiner. Duality and geometry in svm classifiers. In *ICML '00: Proceedings of the Seventeenth International Conference on Machine Learning*, pages 57–64, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc. ISBN 1-55860-707-2.

Antoine Bordes and Léon Bottou. SGD-QN, LaRank. `http://largescale.first.fraunhofer.de/media/slides/bordes.pdf`, 2008.

Antoine Bordes, Léon Bottou, Patrick Gallinari, and Jason Weston. Solving MultiClass Support Vector Machines with LaRank. In Zoubin Ghahramani, editor, *Proceedings of the 24th International Machine Learning Conference*, pages 89–96, Corvallis, Oregon, 2007. OmniPress. URL `http://leon.bottou.org/papers/bordes-2007`.

Léon Bottou. SVM-SGD. `http://leon.bottou.org/projects/sgd`, 2007.

Léon Bottou and Olivier Bousquet. The tradeoffs of large scale learning. In *Advances in Neural Information Processing Systems*, volume 20, 2007.

Léon Bottou and Yann Le Cun. Large scale online learning. In *Advances in Neural Information Processing Systems 16*, page 2004. MIT Press, 2004.

Léon Bottou and Yann Le Cun. On-line learning for very large data sets: Research articles. *Applied Stochastic Models in Business and Industry*, 21(2):137–151, 2005. ISSN 1524-1904.

Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, March 2004. ISBN 0521833787.

Olivier Chapelle. Training a support vector machine in the primal. *Neural Computation*, 19(5):1155–1178, 2007.

Corinna Cortes and Vladimir Vapnik. Support vector networks. In *Machine Learning*, pages 273–297, 1995.

Nello Cristianini and John Shawe-Taylor. *An Introduction to Support Vector Machines*. Cambridge University Press, 2000.

Ofer Dekel, Shai Shalev-Shwartz, and Yoram Singer. The forgetron: A kernel-based perceptron on a budget. *SIAM Journal on Computing*, 37(5):1342–1372, 2008. ISSN 0097-5397.

John Duchi and Yoram Singer. Online and Batch Learning using Forward Looking Subgradients, 2008. Manuscript.

Vojtěch Franc and Soeren Sonnenburg. Optimized cutting plane algorithm for support vector machines. In *ICML '08: Proceedings of the 25th international conference on Machine learning*, pages 320–327, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-205-4.

Hans Peter Graf, Eric Cosatto, Léon Bottou, Igor Dourdanovic, and Vladimir Vapnik. Parallel support vector machines: The cascade svm. In Lawrence K. Saul, Yair Weiss, and Léon Bottou, editors, *Advances in Neural Information Processing Systems 17*, pages 521–528. MIT Press, Cambridge, MA, 2005.

Cho-Jui Hsieh, Kai-Wei Chang, Chih-Jen Lin, S. Sathiya Keerthi, and Sellamanickam Sundararajan. A dual coordinate descent method for large-scale linear SVM. In *Proceedings of the Twenty Fifth International Conference on Machine Learning (ICML)*, 2008.

Thorsten Joachims. A support vector method for multivariate performance measures. In *ICML '05: Proceedings of the 22nd international conference on Machine learning*, pages 377–384, New York, NY, USA, 2005. ACM. ISBN 1-59593-180-5.

Thorsten Joachims. Text categorization with support vector machines: Learning with many relevant features. In *Proceedings of ECML-98: 10th European Conference on Machine Learning*, pages 137–142. Springer Verlag, 1998.

Thorsten Joachims. Making large-scale support vector machine learning practical. *Advances in kernel methods: support vector learning*, pages 169–184, 1999.

Thorsten Joachims. Training linear SVMs in linear time. In *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 217–226, New York, NY, USA, 2006. ACM. ISBN 1-59593-339-5.

Sham M Kakade and Ambuj Tewari. On the generalization ability of online strongly convex programming algorithms. In D. Koller,

D. Schuurmans, Y. Bengio, and L. Bottou, editors, *Advances in Neural Information Processing Systems 21*, 2009.

George Kimeldorf and Grace Wahba. Some results on Tchebycheffian spline functions. *Journal of Mathematical Analysis and Applications*, 33(1):82–95, 1971.

Jyrki Kivinen, Alexander J. Smola, and Rober C. Williamson. Online learning with kernels. *IEEE Transactions on Signal Processing*, 52(8), August 2004.

John Langford. Concerns about the Large Scale Learning Challenge. `http://hunch.net/?p=330`, 2008.

John Langford, Lihong Li, and Tong Zhang. Sparse online learning via truncated gradient. In *Advances in Neural Information Processing Systems 21*, 2008.

Yi Lin. Support Vector Machines and the Bayes Rule in Classification. *Data Mining and Knowledge Discovery*, 6(3):259–275, 2002. ISSN 1384-5810.

Dong C. Liu, Jorge Nocedal, and Dong C. On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, 45:503–528, 1989.

Gaëlle Loosli and Stéphane Canu. Comments on the "core vector machines: Fast SVM training on very large data sets". *Journal of Machine Learning Research*, 8, 2007. ISSN 1533-7928.

Pascal. Pascal large scale learning challenge. `http://largescale.first.fraunhofer.de/about/`, 2008.

John C. Platt. Fast training of support vector machines using sequential minimal optimization. In B. Schölkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods - Support Vector Learning*, pages 185–208. MIT Press, 1999.

Tomaso Poggio and Steve Smale. The mathematics of learning: Dealing with data. *International Conference on Neural Networks and Brain, 2005*, 1:PL–5–PL–23, October 2005.

Greg Schohn and David Cohn. Less is more: Active learning with support vector machines. In *ICML '00: Proceedings of the Seventeenth International Conference on Machine Learning*, pages 839–846, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc. ISBN 1558607072.

Nicol Schraudolph, Jin Yu, and Simon Günter. A stochastic quasi-Newton method for online convex optimization. In *Proceedings of the 11th International Conference on Artificial Intelligence and Statistics (AIstats)*, pages 433–440, San Juan, Puerto Rico, March 2007. Society for Artificial Intelligence and Statistics.

Searchenginewatch.com. Almost 12 billion u.s. searches conducted in July. `http://searchenginewatch.com/3630718`, September 2008.

Shai Shalev-Shwartz and Nathan Srebro. SVM optimization: inverse dependence on training set size. In *ICML '08: Proceedings of the 25th International Conference on Machine learning*, pages 928–935, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-205-4.

Shai Shalev-Shwartz, Yoram Singer, and Nathan Srebro. Pegasos: Primal Estimated sub-GrAdient SOlver for SVM. In *ICML '07: Proceedings of the 24th International Conference on Machine learning*, pages 807–814, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-793-3.

Vikas Sindhwani and S. Sathiya Keerthi. Large scale semi-supervised linear svms. In *SIGIR '06: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 477–484, New York, NY, USA, 2006. ACM. ISBN 1-59593-369-7.

A. J. Smola, S. V. N. Vishwanathan, and Q. V. Le. Bundle methods for machine learning. In J. C. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems 20*, Cambridge MA, 2007. MIT Press.

Sören Sonnenburg, Mikio L Braun, Cheng Soon Ong, Samy Bengio, Leon Bottou, Geoffrey Holmes, Yann LeCun, Klaus-Robert Müller, Fernando Pereira, Carl Edward Rasmussen, Gunnar Rätsch, Bernhard Schölkopf, Alexander Smola, Pascal Vincent, Jason Weston, and Robert C Williamson. The need for open source software in machine learning. *Journal of Machine Learning Research*, 8:2443–2466, 2007.

Karthik Sridharan, Nathan Srebro, and Shai Shalev-Shwartz. Fast convergence rates for excess regularized risk with application to SVM. `http://ttic.uchicago.edu/~karthik/con.pdf`, 2008.

Ingo Steinwart. Sparseness of support vector machines—some asymptotically sharp bounds. In Sebastian Thrun, Lawrence Saul, and Bernhard Schölkopf, editors, *Advances in Neural Information Processing Systems 16*. MIT Press, Cambridge, MA, 2004.

Ivor W. Tsang and James T. Kwok. Authors' reply to the "comments on the core vector machines: Fast SVM training on very large data sets". *Journal of Machine Learning Research*, 2007. Submitted.

Ivor W. Tsang, James T. Kwok, and Pak-Ming Cheung. Core vector machines: Fast SVM training on very large data sets. *Journal of Machine Learning Research*, 6:363–392, 2005. ISSN 1533-7928.

Ivor W. Tsang, András Kocsor, and James T. Kwok. Simpler core vector machines with enclosing balls. In *ICML*, pages 911–918, 2007.

V Vapnik and A Lerner. Pattern recognition using generalized portrait method. *Automation and Remote Control*, 24, 1963.

Vladimir Vapnik. *Estimation of Dependences Based on Empirical Data*. Springer-Verlag, 1982.

Hwanjo Yu. Classifying large data sets using svms with hierarchical clusters. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 306–315, 2003.

J. Yu, S. Vishwanathan, S. Günter, and N. Schraudolph. A Quasi-Newton approach to nonsmooth convex optimization. In Andrew McCallum and Sam Roweis, editors, *Proceedings of the 25th Annual International Conference on Machine Learning (ICML 2008)*, pages 1216–1223. Omnipress, 2008.

Tong Zhang. Solving large scale linear prediction problems using stochastic gradient descent algorithms. In *ICML '04: Proceedings of the twenty-first international conference on Machine learning*, page 116, New York, NY, USA, 2004. ACM. ISBN 1-58113-828-5.