Improving Efficiency of SVM k-Fold Cross-Validation by Alpha Seeding

Zeyi Wen[1], Bin Li[2], Kotagiri Ramamohanarao[1], Jian Chen[2*], Yawen Chen[2], Rui Zhang[1]
[1]{zeyi.wen, kotagiri, rui.zhang}@unimelb.edu.au
The University of Melbourne, Australia
[2]{gitlinux@gmail.com, ellachen@scut.edu.cn, elfairyhyuk@gmail.com}
South China University of Technology, China

Abstract

The k-fold cross-validation is commonly used to evaluate the effectiveness of SVMs with the selected hyper-parameters. It is known that the SVM k-fold cross-validation is expensive, since it requires training k SVMs. However, little work has explored reusing the h^{th} SVM for training the $(h+1)^{\text{th}}$ SVM for improving the efficiency of k-fold cross-validation. In this paper, we propose three algorithms that reuse the h^{th} SVM for improving the efficiency of training the $(h+1)^{\text{th}}$ SVM. Our key idea is to efficiently identify the support vectors and to accurately estimate their associated weights (also called alpha values) of the next SVM by using the previous SVM. Our experimental results show that our algorithms are several times faster than the k-fold cross-validation which does not make use of the previously trained SVM. Moreover, our algorithms produce the same results (hence same accuracy) as the k-fold cross-validation which does not make use of the previously trained SVM.

1 Introduction

In order to train an effective SVM classifier[1], the hyper-parameters (e.g. the penalty C) need to be selected carefully. The k-fold cross-validation is a commonly used process to evaluate the effectiveness of SVMs with the selected hyper-parameters. It is known that the SVM k-fold cross-validation is expensive, since it requires training k SVMs with different subsets of the whole dataset. To improve the efficiency of k-fold cross-validation[2], some recent studies (Wen et al. 2014; Athanasopoulos et al. 2011) exploit modern hardware (e.g. Graphic Processing Units). Chu et al. (Chu et al. 2015) proposed to reuse the k linear SVM classifiers trained in the k-fold cross-validation with parameter C for training the k linear SVM classifiers with parameter $(C+\Delta)$. However, little work has explored the possibility of reusing the h^{th} (where $h \in \{1, 2, ..., (k-1)\}$) SVM for improving the efficiency of training the $(h+1)^{\text{th}}$ SVM in the k-fold cross-validation with parameter C.

In this paper, we propose three algorithms that reuse the h^{th} SVM for training the $(h+1)^{\text{th}}$ SVM in k-fold cross-validation. The intuition behind our algorithms is that the hyperplanes of the two SVMs are similar, since many training instances (e.g. more than 80% of the training instances when k is 10) are the same in training the two SVMs. Note that in this paper we are interested in $k > 2$, since when $k = 2$ the two SVMs share no training instance.

We present our ideas in the context of training SVMs using Sequential Minimal Optimisation (SMO) (Platt and others 1998), although our ideas are applicable to other solvers (Osuna, Freund, and Girosi 1997; Joachims 1999). In SMO, the hyperplane of the SVM is represented by a subset of training instances together with their weights, namely alpha values. The training instances with alpha values larger than 0 are called support vectors. Finding the optimal hyperplane is effectively finding the alpha values for all the training instances. Without reusing the previous SVM, the alpha values of all the training instances are initialised to 0. Our key idea is to use the alpha values of the h^{th} SVM to initialise the alpha values for the $(h+1)^{\text{th}}$ SVM. Initialising alpha values using the previous SVM is called *alpha seeding* in the literature of studying leave-one-out cross-validation (DeCoste and Wagstaff 2000). At some risk of confusion to the reader, we will use "alpha seeding" and "initialising alpha values" interchangeably, depending on which interpretation is more natural.

Reusing the h^{th} SVM for training the $(h+1)^{\text{th}}$ SVM in k-fold cross-validation has two key challenges. (i) The training dataset for the h^{th} SVM is different from that for the $(h+1)^{\text{th}}$ SVM, but the initial alpha values for the $(h+1)^{\text{th}}$ SVM should be close to their optimal values; improper initialisation of alpha values leads to slower convergence than that without reusing the h^{th} SVM. (ii) The alpha value initialisation process should be very efficient; otherwise, the time spent in the initialisation may be larger than the time saved in the SVM training. This is perhaps the reason that existing work either (i) reuses the h^{th} SVM trained with parameter C for training the h^{th} SVM with parameter $(C+\Delta)$ where both SVMs have the identical training dataset (Chu et al. 2015), or (ii) only studies alpha seeding in the leave-one-out cross-validation (DeCoste and Wagstaff 2000; Lee et al. 2004) which is a special case of k-fold cross-validation.

Our key contributions in this paper are the proposal of

*Jian Chen is the corresponding author.

[1]For ease of presentation, we discuss binary classification, although our approaches are applicable to multi-class classification and regression.

[2]Without confusion, we omit "SVM" in SVM k-fold cross-validation.

three algorithms (where we progressively refine one algorithm after the other) for reusing the alpha values of the h^{th} SVM for the $(h+1)^{\text{th}}$ SVM. (i) Our first algorithm aims to initialise the alpha values to their optimal values for the $(h+1)^{\text{th}}$ SVM by exploiting the optimality condition of the SVM training. (ii) To efficiently compute the initial alpha values, our second algorithm only estimates the alpha values for the newly added instances, based on the assumption that all the shared instances between the h^{th} and the $(h+1)^{\text{th}}$ SVMs tend to have the same alpha values. (iii) To further improve the efficiency of initialising alpha values, our third algorithm exploits the fact that a training instance in the h^{th} SVM can be potentially replaced by a training instance in the $(h+1)^{\text{th}}$ SVM. Our experimental results show that when $k = 10$, our algorithms are several times faster than the k-fold cross-validation in LibSVM; when $k = 100$, our algorithm dramatically outperforms LibSVM (32 times faster in the Madelon dataset). Moreover, our algorithms produce the same results (hence same accuracy) as LibSVM.

The remainder of this paper is organised as follows. We describe preliminaries in Section 2. Then, we elaborate our three algorithms in Section 3, and report our experimental study in Section 4. In Section 5 and 6, we review the related literature, and conclude this paper.

2 Preliminaries

Here, we give some details of SVMs, and discuss the relationship of two rounds of k-fold cross-validation.

Support Vector Machines

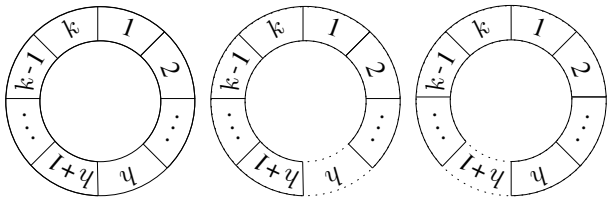
An instance $\boldsymbol{x}_i$ is attached with an integer $y_i \in \{+1, -1\}$ as its label. A positive (negative) instance is an instance with the label of $+1$ (-1). Given a set $\mathcal{X}$ of n training instances, the goal of the SVM training is to find a hyperplane that separates the positive and the negative training instances in $\mathcal{X}$ with the maximum margin and meanwhile, with the minimum misclassification error on the training instances.

To enable handily mapping training instances to other data spaces by kernel functions, finding the hyperplane can be expressed in a *dual form* (Bennett and Bredensteiner 2000) as the following *quadratic programming* problem (Nocedal and Wright 2006).

$$\begin{aligned} \underset{\boldsymbol{\alpha}}{\text{argmax}} \quad & \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\boldsymbol{\alpha}^T \boldsymbol{Q}\boldsymbol{\alpha} \\ \text{subject to} \quad & 0 \le \alpha_i \le C, \forall i \in \{1, ..., n\}; \sum_{i=1}^{n} y_i \alpha_i = 0 \end{aligned} \tag{1}$$

where $\boldsymbol{\alpha} \in \mathbb{R}^n$ is also called a weight vector, and α_i denotes the *weight* of $\boldsymbol{x}_i$; $\boldsymbol{Q}$ denotes an $n \times n$ matrix $[Q_{i,j}]$ and $Q_{i,j} = y_i y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j)$, and $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ is a kernel value computed from a kernel function (e.g. Gaussian kernel, $K(\boldsymbol{x}_i, \boldsymbol{x}_j) = exp\{-\gamma||\boldsymbol{x}_i - \boldsymbol{x}_j||^2\}$). Then, the goal of the SVM training is to find the optimal $\boldsymbol{\alpha}$. If α_i is greater than 0, $\boldsymbol{x}_i$ is called a *support vector*.

In this paper, we present our ideas in the context of using SMO to solve Problem (1), although our key ideas are applicable to other solvers (Osuna, Freund, and Girosi 1997; Joachims 1999). The training process and the derivation of the optimality condition are unimportant for understanding our algorithms, and hence are not discussed here. Next, we present the optimality condition for the SVM training which will be exploited in our proposed algorithms in Section 3.



(a) k subsets (b) h^{th} as test (c) $(h+1)^{\text{th}}$ as test

Figure 1: k-fold cross-validation

The optimality condition for the SVM training In SMO, a training instance $\boldsymbol{x}_i$ is associated with an optimality indicator f_i which is defined as follows.

$$f_i = y_i \sum_{j=1}^{n} \alpha_j Q_{i,j} - y_i \tag{2}$$

The optimality condition of the SVM training is the Karush-Kuhn-Tucker (KKT) (Kuhn 2014) condition. When the optimality condition is met, we have the optimality indicators satisfying the following constraint.

$$\min\{f_i | i \in I_u \cup I_m\} \ge \max\{f_i | i \in I_l \cup I_m\} \tag{3}$$

where

$$\begin{aligned} I_m &= \{i | \boldsymbol{x}_i \in \mathcal{X}, 0 < \alpha_i < C\}, \\ I_u &= \{i | \boldsymbol{x}_i \in \mathcal{X}, y_i = +1, \alpha_i = 0\} \cup \\ &\quad \{i | \boldsymbol{x}_i \in \mathcal{X}, y_i = -1, \alpha_i = C\}, \\ I_l &= \{i | \boldsymbol{x}_i \in \mathcal{X}, y_i = +1, \alpha_i = C\} \cup \\ &\quad \{i | \boldsymbol{x}_i \in \mathcal{X}, y_i = -1, \alpha_i = 0\}. \end{aligned} \tag{4}$$

As observed by Keerthi et al. (Keerthi et al. 2001), Constraint (3) is equivalent to the following constraints.

$$f_i > b \text{ for } i \in I_u; \quad f_i = b \text{ for } i \in I_m; \quad f_i < b \text{ for } i \in I_l \tag{5}$$

where b is the bias of the hyperplane. Our algorithms proposed in Section 3 exploit Constraint (5).

Relationship between the h^{th} round and the $(h+1)^{\text{th}}$ round in k-fold cross-validation

The k-fold cross-validation evenly divides the dataset into k subsets. One subset is used as the test set $\mathcal{T}$, while the rest $(k-1)$ subsets together form the training set $\mathcal{X}$. Suppose we have trained the h^{th} SVM (in the h^{th} round) using the 1^{st} to $(h-1)^{\text{th}}$ and $(h+1)^{\text{th}}$ to k^{th} subsets as the training set, and the h^{th} subset serves as the testing set (cf. Figure 1b). Now we want to train the $(h+1)^{\text{th}}$ SVM. Then, the 1^{st} to $(h-1)^{\text{th}}$ subsets and the $(h+2)^{\text{th}}$ to k^{th} subsets are shared between the two rounds of the training. To convert the training set used in the h^{th} round to the training set for the $(h+1)^{\text{th}}$ round, we just need to remove the $(h+1)^{\text{th}}$ subset from and add the h^{th} subset to the training set used in the h^{th} round. Hereafter, we call the h^{th} and $(h+1)^{\text{th}}$ SVMs *the previous SVM* and *the next SVM*, respectively.

For ease of presentation, we denote the shared subsets—$(k-2)$ subsets in total—by $\mathcal{S}$, denote the unshared subset in the training of the previous round by $\mathcal{R}$, and denote the subset for testing in the previous round by $\mathcal{T}$. Let us continue to use the example shown in Figure 1, $\mathcal{S}$ consists of the 1^{st} to $(h-1)^{th}$ subsets and the $(h+2)^{\text{th}}$ to k^{th} subsets; $\mathcal{R}$ is the $(h+1)^{\text{th}}$ subset; $\mathcal{T}$ is the h^{th} subset. To convert the training set $\mathcal{X}$ used in the h^{th} round to the training set $\mathcal{X}'$ for the $(h+1)^{\text{th}}$ round, we just need to remove $\mathcal{R}$ from $\mathcal{X}$ and add $\mathcal{T}$ to $\mathcal{X}$, i.e. $\mathcal{X}' = \mathcal{T} \cup \mathcal{X} \setminus \mathcal{R} = \mathcal{T} \cup \mathcal{S}$. We denote three sets of indices as follows corresponding to $\mathcal{R}$, $\mathcal{T}$ and $\mathcal{S}$ by $I_{\mathcal{R}}$, $I_{\mathcal{T}}$ and $I_{\mathcal{S}}$, respectively.

$$I_{\mathcal{R}} = \{i | \boldsymbol{x}_i \in \mathcal{R}\}, I_{\mathcal{T}} = \{i | \boldsymbol{x}_i \in \mathcal{T}\}, I_{\mathcal{S}} = \{i | \boldsymbol{x}_i \in \mathcal{S}\} \quad (6)$$

Two rounds of the k-fold cross-validation often have many training instances in common, i.e. large $\mathcal{S}$. E.g. when k is 10, $\frac{8}{9}$ (or $\sim 90\%$) of instances in $\mathcal{X}$ and $\mathcal{X}'$ are the instances of $\mathcal{S}$. Next, we study three algorithms for reusing the previous SVM to train the next SVM.

3 Reusing the previous SVM in k-fold cross-validation

We present three algorithms that reuse the previous SVM for training the next SVM, where we progressively refine one algorithm after the other. (i) Our first algorithm aims to initialise the alpha values $\boldsymbol{\alpha}'$ to their optimal values for the next SVM, based on the alpha values $\boldsymbol{\alpha}$ of the previous SVM. We call the first algorithm Adjusting Alpha Towards Optimum (ATO). (ii) To efficiently initialise $\boldsymbol{\alpha}'$, our second algorithm keeps the alpha values of the instances in $\mathcal{S}$ unchanged (i.e. $\alpha'_s = \alpha_s$ for $s \in I_{\mathcal{S}}$), and estimates α'_t for $t \in I_{\mathcal{T}}$. This algorithm effectively performs alpha value initialisation via replacing $\mathcal{R}$ by $\mathcal{T}$ under constraints of Problem (1), and hence we call the algorithm Multiple Instance Replacement (MIR). (iii) Similar to MIR, our third algorithm also keeps the alpha values of the instances in $\mathcal{S}$ unchanged; different from MIR, the algorithm replaces the instances in $\mathcal{R}$ by the instances in $\mathcal{T}$ one at a time, which dramatically reduces the time for initialising $\boldsymbol{\alpha}'$. We call the third algorithm Single Instance Replacement (SIR). Next, we elaborate these three algorithms.

Adjusting Alpha Towards Optimum (ATO)

ATO aims to initialise the alpha values to their optimal values. It employs the technique for online SVM training, designed by Karasuyama and Takeuchi (Karasuyama and Takeuchi 2009), for the k-fold cross-validation. In the online SVM training, a subset $\mathcal{R}$ of outdated training instances is removed from the training set $\mathcal{X}$, i.e. $\mathcal{X}' = \mathcal{X} \setminus \mathcal{R}$; a subset $\mathcal{T}$ of newly arrived training instances is added to the training set, i.e. $\mathcal{X}' = \mathcal{X}' \cup \mathcal{T}$. The previous SVM trained using $\mathcal{X}$ is adjusted by removing and adding subsets of instances to obtain the next SVM.

In the ATO algorithm, we first construct a new training dataset $\mathcal{X}'$ where $\mathcal{X}' = \mathcal{S} = \mathcal{X} \setminus \mathcal{R}$. Then, we gradually increase alpha values of the instances in $\mathcal{T}$ (i.e. increase α'_t for $t \in I_{\mathcal{T}}$), denoted by $\boldsymbol{\alpha}'_{\mathcal{T}}$, to (near) their optimal values; meanwhile, we gradually decrease the alpha values of the instances in $\mathcal{R}$ (i.e. decrease α'_r for $r \in I_{\mathcal{R}}$), denoted by $\boldsymbol{\alpha}'_{\mathcal{R}}$, to 0. Once the alpha value of an instance in $\mathcal{T}$ satisfies the optimal condition (i.e. Constraint (5)), we move the instance from $\mathcal{T}$ to the training set $\mathcal{X}'$; similarly once the alpha value of an instance in $\mathcal{R}$ equals to 0 (becoming a non-support vector), we remove the instance from $\mathcal{R}$. ATO terminates the alpha value initialisation when $\mathcal{R}$ is empty.

Updating the alpha values Next, we present details of increasing $\boldsymbol{\alpha}'_{\mathcal{T}}$ and decreasing $\boldsymbol{\alpha}'_{\mathcal{R}}$. We denote the step size for an increment on $\boldsymbol{\alpha}'_{\mathcal{T}}$ and decrement on $\boldsymbol{\alpha}'_{\mathcal{R}}$ by η. From constraints of Problem (1), all the alpha values must be in $[0, C]$. Hence, for $t \in I_{\mathcal{T}}$ the increment of α'_t, denoted by $\Delta\alpha'_t$, cannot exceed $(C - \alpha'_t)$; for $r \in I_{\mathcal{R}}$ the decrement of α'_r, denoted by $\Delta\alpha'_r$, cannot exceed α'_r. We denote the change of all the alpha values of the instances in $\mathcal{T}$ by $\Delta\boldsymbol{\alpha}'_{\mathcal{T}}$ and the change of all the alpha values of the instances in $\mathcal{R}$ by $\Delta\boldsymbol{\alpha}'_{\mathcal{R}}$. Then, we can compute $\Delta\boldsymbol{\alpha}'_{\mathcal{T}}$ and $\Delta\boldsymbol{\alpha}'_{\mathcal{R}}$ as follows.

$$\Delta\boldsymbol{\alpha}'_{\mathcal{T}} = \eta(C\mathbf{1} - \boldsymbol{\alpha}'_{\mathcal{T}}), \quad \Delta\boldsymbol{\alpha}'_{\mathcal{R}} = -\eta\boldsymbol{\alpha}'_{\mathcal{R}} \quad (7)$$

where $\mathbf{1}$ is a vector with all the dimensions of 1. When we add $\Delta\boldsymbol{\alpha}'_{\mathcal{T}}$ to $\boldsymbol{\alpha}'_{\mathcal{T}}$ and $\Delta\boldsymbol{\alpha}'_{\mathcal{R}}$ to $\boldsymbol{\alpha}'_{\mathcal{R}}$, constraints of Problem (1) must be satisfied. However, after adjusting $\boldsymbol{\alpha}'_{\mathcal{T}}$ and $\boldsymbol{\alpha}'_{\mathcal{R}}$, the constraint $\sum_{i \in I_{\mathcal{T}} \cup I_{\mathcal{S}} \cup I_{\mathcal{R}}} y_i \alpha'_i = 0$ is often violated, so we need to adjust the alpha values of the training instances in $\mathcal{X}'$ (recall that at this stage $\mathcal{X}' = \mathcal{S}$). We propose to adjust the alpha values of the training instances in $\mathcal{X}'$ which are also in $\mathcal{M}$ where $\boldsymbol{x}_i \in \mathcal{M}$ given $i \in I_m$. In summary, after increasing $\boldsymbol{\alpha}'_{\mathcal{T}}$ and decreasing $\boldsymbol{\alpha}'_{\mathcal{R}}$, we adjust $\boldsymbol{\alpha}'_{\mathcal{M}}$. So when adjusting $\boldsymbol{\alpha}'_{\mathcal{T}}$, $\boldsymbol{\alpha}'_{\mathcal{R}}$ and $\boldsymbol{\alpha}'_{\mathcal{M}}$, we have the following equation according to constraints of Problem (1).

$$\sum_{t \in I_{\mathcal{T}}} y_t \Delta\alpha'_t + \sum_{r \in I_{\mathcal{R}}} y_r \Delta\alpha'_r + \sum_{i \in I_m} y_i \Delta\alpha'_i = 0 \quad (8)$$

$\mathcal{M}$ often has a large number of instances, and there are many possible ways to adjust $\boldsymbol{\alpha}'_{\mathcal{M}}$. Here, we propose to use the adjustment on $\boldsymbol{\alpha}'_{\mathcal{M}}$ that ensures all the training instances in $\mathcal{M}$ satisfy the optimality condition (i.e. Constraint (5)). According to Constraint (5), we have $\forall i \in I_m$ and $f_i = b$. Combining $f_i = b$ and the definition of f_i (cf. Equation (2)), we have the following equation for each $i \in I_m$.

$$y_i\Big(\sum_{t \in I_{\mathcal{T}}} Q_{i,t}\Delta\alpha'_t + \sum_{r \in I_{\mathcal{R}}} Q_{i,r}\Delta\alpha'_r + \sum_{j \in I_m} Q_{i,j}\Delta\alpha'_j\Big) = 0 \quad (9)$$

Note that y_i can be omitted in the above equation. We can rewrite Equation (8) and Equation (9) using the matrix notation for all the training instances in $\mathcal{M}$.

$$\begin{bmatrix} \boldsymbol{y}_{\mathcal{T}}^T & \boldsymbol{y}_{\mathcal{R}}^T \\ \boldsymbol{Q}_{\mathcal{M},\mathcal{T}} & \boldsymbol{Q}_{\mathcal{M},\mathcal{R}} \end{bmatrix} \begin{bmatrix} \Delta\boldsymbol{\alpha}'_{\mathcal{T}} \\ \Delta\boldsymbol{\alpha}'_{\mathcal{R}} \end{bmatrix} + \begin{bmatrix} \boldsymbol{y}_{\mathcal{M}}^T \\ \boldsymbol{Q}_{\mathcal{M},\mathcal{M}} \end{bmatrix} \Delta\boldsymbol{\alpha}'_M = 0$$

We substitute $\Delta\boldsymbol{\alpha}'_{\mathcal{T}}$ and $\Delta\boldsymbol{\alpha}'_{\mathcal{R}}$ using Equation (7); the above equation can be rewritten as follows.

$$\Delta\boldsymbol{\alpha}'_{\mathcal{M}} = -\eta\Phi \quad (10)$$

where $\Phi = \begin{bmatrix} \boldsymbol{y}_{\mathcal{M}}^T \\ \boldsymbol{Q}_{\mathcal{M},\mathcal{M}} \end{bmatrix}^{-1} \begin{bmatrix} \boldsymbol{y}_{\mathcal{T}}^T & \boldsymbol{y}_{\mathcal{R}}^T \\ \boldsymbol{Q}_{\mathcal{M},\mathcal{T}} & \boldsymbol{Q}_{\mathcal{M},\mathcal{R}} \end{bmatrix} \begin{bmatrix} C\mathbf{1} - \boldsymbol{\alpha}'_{\mathcal{T}} \\ -\boldsymbol{\alpha}'_{\mathcal{R}} \end{bmatrix}$. If the inverse of the matrix in Equation (10) does not exist, we find the pseudo inverse (Greville 1960)

Computing step size η: Given an η, we can use Equations (7) and (10) to adjust $\boldsymbol{\alpha}'_{\mathcal{M}}$, $\boldsymbol{\alpha}'_{\mathcal{T}}$ and $\boldsymbol{\alpha}'_{\mathcal{R}}$. The changes

of the alpha values lead to the change of all the optimality indicators $\boldsymbol{f}$. We denote the change to $\boldsymbol{f}$ by $\Delta\boldsymbol{f}$ which can be computed by the following equation derived from Equation (2).

$$\boldsymbol{y} \odot \Delta\boldsymbol{f} = \eta[-\boldsymbol{Q}_{\mathcal{X},\mathcal{M}}\Phi + \boldsymbol{Q}_{\mathcal{X},\mathcal{T}}(C\mathbf{1} - \boldsymbol{\alpha}'_{\mathcal{T}}) - \boldsymbol{Q}_{\mathcal{X},\mathcal{R}}\boldsymbol{\alpha}'_{\mathcal{R}}] \quad (11)$$

where $\odot$ is the hadamard product (i.e. element-wise product (Schott 2005)).

If the step size η is too large, more optimality indicators tend to violate Constraint (5). Here, we use Equation (11) to compute the step size η by letting the updated f_i (where $i \in I_u \cup I_l$) just violate Constraint (5), i.e. $f_i + \Delta f_i = b$ for $i \in I_u \cup I_l$.

Updating $\boldsymbol{f}$ After updating $\boldsymbol{\alpha}'$, we update $\boldsymbol{f}$ using Equations (2) and (11). Then, we update the sets I_m, I_u and I_l according to Constraint (5).

The process of computing η and updating $\boldsymbol{\alpha}'$ and $\boldsymbol{f}$ are repeated until $\mathcal{R}$ is empty.

Termination When $\mathcal{R}$ is empty, the SVM may not be optimal, because the set $\mathcal{T}$ may not be empty. The alpha values obtained from the above process serve as the initial alpha values for the next SVM. To obtain the optimal SVM, we use SMO to adjust the initial alpha values until optimal condition is met. The pseudo-code of the full algorithm is shown in Algorithm 1 in Wen et al. (2016).

Multiple Instance Replacement (MIR)

A limitation of ATO is that it requires adjusting *all* the alpha values for an *unbounded* number of times (i.e. until $\mathcal{R}$ is empty). Hence, the cost of initialising the alpha values may be very high. In what follows, we propose the Multiple Instance Replacement (MIR) algorithm that only needs to adjust $\boldsymbol{\alpha}'_{\mathcal{T}}$ once. The alpha values of the shared instances between the two rounds stay unchanged (i.e. $\boldsymbol{\alpha}'_{\mathcal{S}} = \boldsymbol{\alpha}_{\mathcal{S}}$), the intuition is that many support vectors tend to stay unchanged. The key idea of MIR is to replace $\mathcal{R}$ by $\mathcal{T}$ at once.

We obtain the alpha values of the instances in $\mathcal{S}$ and $\mathcal{R}$ from the previous SVM, and those alpha values satisfy the following constraint.

$$\sum_{s\in I_{\mathcal{S}}} y_s\alpha_s + \sum_{r\in I_{\mathcal{R}}} y_r\alpha_r = 0 \quad (12)$$

In the next round of SVM k-fold cross-validation, $\mathcal{R}$ is removed and $\mathcal{T}$ is added. When reusing alpha values, we should guarantee that the above constraint holds. To improve the efficiency of initialising alpha values, we do not change alpha values in first term of Constraint (12), i.e. $\sum_{s\in I_{\mathcal{S}}} y_s\alpha_s$.

To satisfy the above constraint after replacing $\mathcal{R}$ by $\mathcal{T}$, we only need to ensure $\sum_{r\in I_{\mathcal{R}}} y_r\alpha_r = \sum_{t\in I_{\mathcal{T}}} y_t\alpha'_t$. Next, we present an approach to compute $\boldsymbol{\alpha}'_{\mathcal{T}}$.

According to Equation (2), we can rewrite f_i before replacing $\mathcal{R}$ by $\mathcal{T}$ as follows.

$$f_i = y_i(\sum_{r\in I_{\mathcal{R}}} \alpha_r Q_{i,r} + \sum_{s\in I_{\mathcal{S}}} \alpha_s Q_{i,s} - 1) \quad (13)$$

After replacing $\mathcal{R}$ by $\mathcal{T}$, f_i can be computed as follows.

$$f_i = y_i(\sum_{t\in I_{\mathcal{T}}} \alpha'_t Q_{i,t} + \sum_{s\in I_{\mathcal{S}}} \alpha'_s Q_{i,s} - 1) \quad (14)$$

where $\alpha'_s = \alpha_s$, i.e. the alpha values in $\mathcal{S}$ stay unchanged. We can compute the change of f_i, denoted by Δf_i, by subtracting Equation (13) from Equation (14). Then, we have the following equation.

$$\Delta f_i = y_i[\sum_{t\in I_{\mathcal{T}}} \alpha'_t Q_{i,t} - \sum_{r\in I_{\mathcal{R}}} \alpha_r Q_{i,r}] \quad (15)$$

To meet the constraint $\sum y_i\alpha_i = 0$ after replacing $\mathcal{R}$ by $\mathcal{T}$, we have the following equation.

$$\sum_{s\in I_{\mathcal{S}}} y_s\alpha_s + \sum_{r\in I_{\mathcal{R}}} y_r\alpha_r = \sum_{s\in I_{\mathcal{S}}} y_s\alpha'_s + \sum_{t\in I_{\mathcal{T}}} y_t\alpha'_t$$

As $\alpha'_s = \alpha_s$, we rewrite the above equation as follows.

$$\sum_{r\in I_{\mathcal{R}}} y_r\alpha_r = \sum_{t\in I_{\mathcal{T}}} y_t\alpha'_t \quad (16)$$

We write Equations (15) and (16) together as follows.

$$\begin{bmatrix} \boldsymbol{y} \odot \Delta\boldsymbol{f} + \boldsymbol{Q}_{\mathcal{X},\mathcal{R}}\boldsymbol{\alpha}_{\mathcal{R}} \\ \boldsymbol{y}_{\mathcal{R}}^T \cdot \boldsymbol{\alpha}_{\mathcal{R}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{Q}_{\mathcal{X},\mathcal{T}} \\ \boldsymbol{y}_{\mathcal{T}}^T \end{bmatrix} \boldsymbol{\alpha}'_{\mathcal{T}} \quad (17)$$

Similar to the way we compute Δf_i in the ATO algorithm, given i in $I_u \cup I_l$ we compute Δf_i by letting $f_i + \Delta f_i = b$ (cf. Constraint (5)). Given i in I_m, we set $\Delta f_i = 0$ since we try to avoid f_i violating Constraint (5). Once we have $\Delta\boldsymbol{f}$, the only unknown in Equation (17) is $\boldsymbol{\alpha}'_{\mathcal{T}}$.

Finding an approximate solution for $\boldsymbol{\alpha}'_{\mathcal{T}}$ The linear system shown in Equation (17) may have no solution. This is because $\boldsymbol{\alpha}'_{\mathcal{S}}$ may also need to be adjusted, but is not considered in Equation (17). Here, we propose to find the approximate solution $\boldsymbol{\alpha}'_{\mathcal{T}}$ for Equation (17) by using linear least squares (Lawson and Hanson 1974) and we have the following equation.

$$\begin{bmatrix} \boldsymbol{Q}_{\mathcal{X},\mathcal{T}} \\ \boldsymbol{y}_{\mathcal{T}}^T \end{bmatrix}^T \begin{bmatrix} \boldsymbol{y} \odot \Delta\boldsymbol{f} + \boldsymbol{Q}_{\mathcal{X},\mathcal{R}}\boldsymbol{\alpha}_{\mathcal{R}} \\ \boldsymbol{y}_{\mathcal{R}}^T \cdot \boldsymbol{\alpha}_{\mathcal{R}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{Q}_{\mathcal{X},\mathcal{T}} \\ \boldsymbol{y}_{\mathcal{T}}^T \end{bmatrix}^T \begin{bmatrix} \boldsymbol{Q}_{\mathcal{X},\mathcal{T}} \\ \boldsymbol{y}_{\mathcal{T}}^T \end{bmatrix} \boldsymbol{\alpha}'_{\mathcal{T}}$$

Then we can compute $\boldsymbol{\alpha}'_{\mathcal{T}}$ using the following equation.

$$\boldsymbol{\alpha}'_{\mathcal{T}} = \left(\begin{bmatrix} \boldsymbol{Q}_{\mathcal{X},\mathcal{T}} \\ \boldsymbol{y}_{\mathcal{T}}^T \end{bmatrix}^T \begin{bmatrix} \boldsymbol{Q}_{\mathcal{X},\mathcal{T}} \\ \boldsymbol{y}_{\mathcal{T}}^T \end{bmatrix} \right)^{-1} \begin{bmatrix} \boldsymbol{Q}_{\mathcal{X},\mathcal{T}} \\ \boldsymbol{y}_{\mathcal{T}}^T \end{bmatrix}^T \begin{bmatrix} \boldsymbol{y} \odot \Delta\boldsymbol{f} + \boldsymbol{Q}_{\mathcal{X},\mathcal{R}}\boldsymbol{\alpha}_{\mathcal{R}} \\ \boldsymbol{y}_{\mathcal{R}}^T \cdot \boldsymbol{\alpha}_{\mathcal{R}} \end{bmatrix} \quad (18)$$

If the inverse of the matrix in above equation does not exist, we find the pseudo inverse similar to ATO.

Adjusting $\boldsymbol{\alpha}'_{\mathcal{T}}$ Due to the approximation, the constraints $0 \le \alpha'_t \le C$ and $\sum_{r\in I_{\mathcal{R}}} y_r\alpha_r = \sum_{t\in I_{\mathcal{T}}} y_t\alpha'_t$ may not hold. Therefore, we need to adjust $\boldsymbol{\alpha}'_{\mathcal{T}}$ to satisfy the constraints, and we perform the following steps.

- If $\alpha'_t < 0$, we set $\alpha'_t = 0$; if $\alpha'_t > C$, we set $\alpha'_t = C$.
- If $\sum_{t\in I_{\mathcal{T}}} y_t\alpha'_t > \sum_{r\in I_{\mathcal{R}}} y_r\alpha_r$ (if $\sum_{t\in I_{\mathcal{T}}} y_t\alpha'_t < \sum_{r\in I_{\mathcal{R}}} y_r\alpha_r$), we uniformly decrease (increase) all the $y_t\alpha'_t$ until $\sum_{t\in I_{\mathcal{T}}} y_t\alpha'_t = \sum_{r\in I_{\mathcal{R}}} y_r\alpha_r$, subjected to the constraint $0 \le \alpha'_t \le C$.

After the above adjusting, α'_t satisfies the constraints $0 \le \alpha'_t \le C$ and $\sum_{r\in I_{\mathcal{R}}} y_r\alpha_r = \sum_{t\in I_{\mathcal{T}}} y_t\alpha'_t$. Then, we use SMO with $\boldsymbol{\alpha}'$ (where $\boldsymbol{\alpha}' = \boldsymbol{\alpha}'_{\mathcal{S}} \cup \boldsymbol{\alpha}'_{\mathcal{T}}$) as the initial alpha values for training an optimal SVM. The pseudo-code of whole algorithm is shown in Algorithm 2 in Wen et al. (2016).

Single Instance Replacement (SIR)

Both ATO and MIR have the following major limitation: the computation for $\boldsymbol{\alpha}'_{\mathcal{T}}$ is expensive (e.g. require computing the inverse of a matrix). The goal of the ATO and MIR is to minimise the number of instances that violate the optimality condition. In the algorithm we propose here, we try to *minimise* Δf_i with a hope that the small change to f_i will not violate the optimality condition. This slight change of the goal leads to a much cheaper computation cost on computing $\boldsymbol{\alpha}'_{\mathcal{T}}$. Our key idea is to replace the instance in $\mathcal{R}$ one after another with a similar instance in $\mathcal{T}$. Since we replace one instance in $\mathcal{R}$ by an instance in $\mathcal{T}$ each time, we call this algorithm Single Instance Replacement (SIR). Next, we present the details of the SIR algorithm.

According to Equation (2), we can rewrite f_i of the previous SVM as follows.

$$f_i = y_i\Big(\sum_{j\in I_{\mathcal{S}}\cup I_{\mathcal{R}}\setminus\{p\}} \alpha_j Q_{i,j} + \alpha_p Q_{i,p} - 1\Big) \tag{19}$$

where $p \in I_{\mathcal{R}}$. We replace the training instance $\boldsymbol{x}_p$ by $\boldsymbol{x}_q$ where $q \in I_{\mathcal{T}}$, and then the value of f_i after replacing $\boldsymbol{x}_p$ by $\boldsymbol{x}_q$ is as follows.

$$f_i = y_i\Big(\sum_{j\in I_{\mathcal{S}}\cup I_{\mathcal{R}}\setminus\{p\}} \alpha_j Q_{i,j} + \alpha'_q Q_{i,q} - 1\Big) \tag{20}$$

where $\alpha'_q = \alpha_p$. By subtracting Equation (19) from Equation (20), the change of f_i, denoted by Δf_i, can be computed by $\Delta f_i = y_i\alpha_p(Q_{i,q} - Q_{i,p})$. Recall that $Q_{i,j} = y_i y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j)$. We can write Δf_i as follows.

$$\Delta f_i = \alpha_p(y_q K(\boldsymbol{x}_i, \boldsymbol{x}_q) - y_p K(\boldsymbol{x}_i, \boldsymbol{x}_p)) \tag{21}$$

Recall also that in SIR we want to replace $\boldsymbol{x}_p$ by an instance, denoted by $\boldsymbol{x}_q$, that minimises Δf_i. When $\alpha_p = 0$, Δf_i has no change after replacing $\boldsymbol{x}_p$ by $\boldsymbol{x}_q$. In what follows, we focus on the case that $\alpha_p > 0$.

We propose to replace $\boldsymbol{x}_p$ by $\boldsymbol{x}_q$ if $\boldsymbol{x}_q$ is the "most similar" instance to $\boldsymbol{x}_p$ among all the instances in $\mathcal{T}$. The instance $\boldsymbol{x}_q$ is called the most similar to the instance $\boldsymbol{x}_p$ among all the instances in $\mathcal{T}$, when the following two conditions are satisfied.

- $\boldsymbol{x}_p$ and $\boldsymbol{x}_q$ have the same label, i.e. $y_p = y_q$.
- $K(\boldsymbol{x}_p, \boldsymbol{x}_q) \geq K(\boldsymbol{x}_p, \boldsymbol{x}_t)$ for all $\boldsymbol{x}_t \in \mathcal{T}$.

Note that in the second condition, we use the fact that the kernel function approximates the similarity between two instances (Balcan, Blum, and Srebro 2008). If we can find the most similar instance to each instance in $\mathcal{R}$, the constraint $\sum_{s\in I_{\mathcal{S}}} y_s\alpha'_s + \sum_{t\in I_{\mathcal{T}}} y_t\alpha'_t = 0$ will be satisfied after the replacing $\mathcal{R}$ by $\mathcal{T}$. Whereas, if we cannot find any instance in $\mathcal{T}$ that has the same label as $\boldsymbol{x}_p$, we randomly pick an instance from $\mathcal{T}$ to replace $\boldsymbol{x}_p$. When the above situation happens, the constraint $\sum_{s\in I_{\mathcal{S}}} y_s\alpha'_s + \sum_{t\in I_{\mathcal{T}}} y_t\alpha'_t = 0$ is violated. Hence, we need to adjust $\boldsymbol{\alpha}'_{\mathcal{T}}$ to make the constraint hold. We use the same approach as MIR to adjusting $\boldsymbol{\alpha}'_{\mathcal{T}}$. The pseudo code for SIR is given in Algorithm 3 in Wen et al. (2016).

4 Experimental studies

We empirically evaluate our proposed algorithms using five datasets from the LibSVM website (Chang and Lin 2011). All our proposed algorithms were implemented in C++. The experiments were conducted on a desktop computer running Linux with a 6-core E5-2620 CPU and 128GB main memory. Following the common settings, we used the Gaussian kernel function and by default k is set to 10. The hyper-parameters for each dataset are identical to the existing studies (Catanzaro, Sundaram, and Keutzer 2008; Smirnov, Sprinkhuizen-Kuyper, and Nalbantov 2004; Wu and Li 2006). Table 2 gives more details about the datasets. We study the k-fold cross-validation under the setting of binary classification.

Next, we first show the overall efficiency of our algorithms compared with LibSVM. Then, we study the effect of varying k from 3 to 100 in the k-fold cross-validation.

Overall efficiency on different datasets

We measured the total elapsed time of each algorithm to test their efficiency. The total elapsed time consists of the alpha initialisation time and the time for the rest of the 10-fold cross-validation. The result is shown in Table 1. To make the table to fit in the page, we do not provide the total elapsed time of ATO, MIR and SIR for each dataset. But the total elapsed time can be easily computed by adding the time for alpha initialisation and the time for the rest. Note that the time for "the rest" (e.g. the fourth column of Table 1) includes the time for partitioning dataset into 10 subsets, training (the most significant part) and classification.

As we can see from the table, the total elapsed time of MIR and SIR is much smaller than LibSVM. In the Madelon dataset, MIR and SIR are about 2 times and 4 times faster than LibSVM, respectively. In comparison, ATO does not show obvious advantages over MIR and SIR, and is even slower than LibSVM on the Adult dataset due to spending too much time on alpha value initialisation. Another observation from the table is SIR spent the smallest amount of time on the alpha initialisation among our three algorithms, while SIR has the similar "effectiveness" as MIR on reusing the alpha values. The effectiveness on reusing the alpha values is reflected by the total number of training iterations during the 10-fold cross-validation. More specifically, according to the ninth to twelfth columns of Table 1, LibSVM often requires more training iterations than MIR and SIR; SIR and MIR have similar number of iterations, and in some datasets (e.g. Adult and MNIST) SIR needs fewer iterations, although SIR saves much time in the initialisation. More importantly, the improvement on the efficiency does

Table 2: Datasets and kernel parameters

Dataset	Cardinality	Dimension	C	γ
Adult	32,561	123	100	0.5
Heart	270	13	2182	0.2
Madelon	2,000	500	1	0.7071
MNIST	60,000	780	10	0.125
Webdata	49,749	300	64	7.8125

Table 1: Efficiency comparison ($k = 10$)

dataset	elapsed time (sec)							number of iterations			
	libsvm	ATO		MIR		SIR		libsvm	ATO	MIR	SIR
		init.	the rest	init.	the rest	init.	the rest				
adult	6,783	3,824	5,738	2,034	3,717	57	3,705	397,565	361,914	318,169	$3.2\text{x}10^5$
heart	0.36	0.016	0.19	0.058	0.083	0.003	0.24	6,988	4,882	1,443	3,968
madel.	54.5	2.0	24.6	1.7	12.8	1.2	13.5	9,000	5,408	1,800	1,800
mnist	$1.7\text{x}10^5$	35,410	69,435	30,897	38,696	1,416	36,406	$1.3\text{x}10^6$	575,250	280,820	$2.6\text{x}10^5$
webda.	24,689	11,166	9,394	6,172	7,574	133	11,901	783,208	245,385	230,357	$3.6\text{x}10^5$

Table 3: Effect of k on total elapsed time (sec)

Dataset	$k = 3$		$k = 100$	
	libsvm	SIR	libsvm	SIR
Adult	733	683	41,288	33,877
Heart	0.09	0.08	3.39	1.17
Madelon	8.8	7.8	620	19.5
MNIST	29,692	22,296	2,508,684	61,016
Webdata	3,941	2,342	190,817	31,918

not sacrifice the accuracy. Due to the space limitation, we omit providing the accuracy comparison here. More details about accuracy can be found in Wen et al. (2016).

Effect of varying k

We varied k from 3 to 100 to study the effect of the value of k. Note that the elapsed time for $k = 10$ can be calculated from Table 1. Moreover, because conducting this set of experiments is very time consuming especially when $k = 100$, we only compare SIR (the best among the our three algorithms according to results in Table 1) with LibSVM.

Table 3 shows the results. As LibSVM was very slow when $k = 100$ on the MNIST dataset, we only ran the first 30 rounds to estimate the total time. As we can see from the table, SIR consistently outperforms LibSVM. When $k = 100$, SIR is about 32 times faster than LibSVM in the Madelon dataset. The experimental result for the leave-one-out (i.e. k equals to the dataset size) cross-validation is similar to $k = 100$, and is available in Figure 2 in Wen et al. (2016).

5 Related work

We categorise the related studies into two groups: on alpha seeding, and on online SVM training.

Related work on alpha seeding

DeCoste and Wagstaff (2000) first introduced the reuse of alpha values in the SVM leave-one-out cross-validation. Their method (i.e. AVG discussed in Supplementary Material) has two main steps: (i) train an SVM with the whole dataset; (ii) remove an instance from the SVM and distribute the associated alpha value uniformly among all the support vectors. Lee et al. (Lee et al. 2004) proposed a technique (i.e. TOP discussed in Supplementary Material) to improve the above method. Instead of uniformly distributing alpha value among all the support vectors, the method distributes the alpha value to the instance with the largest kernel value.

Existing studies called "Warm Start" (Kao et al. 2004; Chu et al. 2015) apply alpha seeding in selecting the parameter C for linear SVMs. Concretely, $\boldsymbol{\alpha}$ obtained from training the h^{th} linear SVM with C is used for training the h^{th} linear SVM with $(C + \Delta)$ in the **two** k-fold cross-validation processes by simply setting $\boldsymbol{\alpha}' = r\boldsymbol{\alpha}$ where r is a ratio computed from C and Δ. In those studies, no alpha seeding technique is used when training the k SVMs with parameter C. Our work aims to reuse the h^{th} SVM for training the $(h+1)^{\text{th}}$ SVM for k-fold cross-validation with parameter C.

Related work on online SVM training

Gauwenberghs and Poggio (2001) introduced an algorithm for training SVM online where the algorithm handles adding or removing one training instance. Karasuyama and Takeuchi (2009) extended the above algorithm to the cases where multiple instances need to be added or removed. Their key idea is to gradually reduce the alpha values of the outdated instances to 0, and meanwhile, to gradually increase the alpha values of the new instances. Due to the efficiency concern, the algorithm produces *approximate* SVMs. We aim to train SVMs which meet the optimality condition.

6 Conclusion

To improve the efficiency of the k-fold cross-validation, we have proposed three algorithms that reuse the previously trained SVM to initialise the next SVM, such that the training process for the next SVM reaches the optimal condition faster. We have conducted extensive experiments to validate the effectiveness and efficiency of our proposed algorithms. Our experimental results have shown that the best algorithm among the three is SIR. When $k = 10$, SIR is several times faster than the k-fold cross-validation in LibSVM which does not make use of the previously trained SVM; when $k = 100$, SIR dramatically outperforms LibSVM (32 times faster than LibSVM in the Madelon dataset). Moreover, our algorithms produce same results (hence same accuracy) as the k-fold cross-validation in LibSVM does.

Acknowledgments This work is supported by Australian Research Council (ARC) Discovery Project DP130104587 and ARC Future Fellowships Project FT120100832. Prof. Jian Chen is supported by the Fundamental Research Funds for the Central Universities (Grant No. 2015ZZ029) and the Opening Project of Guangdong Province Key Laboratory of Big Data Analysis and Processing.

References

Athanasopoulos, A.; Dimou, A.; Mezaris, V.; and Kompatsiaris, I. 2011. GPU acceleration for support vector machines. In *International Workshop on Image Analysis for Multimedia Interactive Services*.

Balcan, M.-F.; Blum, A.; and Srebro, N. 2008. A theory of learning with similarity functions. *Machine Learning* 72(1-2):89–112.

Bennett, K. P., and Bredensteiner, E. J. 2000. Duality and geometry in svm classifiers. In *ICML*, 57–64.

Catanzaro, B.; Sundaram, N.; and Keutzer, K. 2008. Fast support vector machine training and classification on graphics processors. In *ICML*, 104–111. ACM.

Cauwenberghs, G., and Poggio, T. 2001. Incremental and decremental support vector machine learning. *Advances in neural information processing sys.* 409–415.

Chang, C.-C., and Lin, C.-J. 2011. LIBSVM: a library for support vector machines. *TIST* 2(3):27.

Chu, B.-Y.; Ho, C.-H.; Tsai, C.-H.; Lin, C.-Y.; and Lin, C.-J. 2015. Warm start for parameter selection of linear classifiers. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 149–158. ACM.

DeCoste, D., and Wagstaff, K. 2000. Alpha seeding for support vector machines. In *SIGKDD*, 345–349. ACM.

Greville, T. 1960. Some applications of the pseudoinverse of a matrix. *SIAM review* 2(1):15–22.

Joachims, T. 1999. Making large scale svm learning practical. Technical report, Universität Dortmund.

Kao, W.-C.; Chung, K.-M.; Sun, C.-L.; and Lin, C.-J. 2004. Decomposition methods for linear support vector machines. *Neural Computation* 16(8):1689–1704.

Karasuyama, M., and Takeuchi, I. 2009. Multiple incremental decremental learning of support vector machines. In *Advances in neural information processing systems*, 907–915.

Keerthi, S. S.; Shevade, S. K.; Bhattacharyya, C.; and Murthy, K. R. K. 2001. Improvements to platt's smo algorithm for svm classifier design. *Neural Computation* 13(3):637–649.

Kuhn, H. W. 2014. Nonlinear programming: a historical view. In *Traces and Emergence of Nonlinear Programming*. Springer. 393–414.

Lawson, C. L., and Hanson, R. J. 1974. *Solving least squares problems*, volume 161. SIAM.

Lee, M. M.; Keerthi, S. S.; Ong, C. J.; and DeCoste, D. 2004. An efficient method for computing leave-one-out error in support vector machines with gaussian kernels. *Neural Networks, IEEE Transactions on* 15(3):750–757.

Nocedal, J., and Wright, S. 2006. *Numerical optimization*. Springer Science & Business Media.

Osuna, E.; Freund, R.; and Girosi, F. 1997. An improved training algorithm for support vector machines. In *IEEE Workshop on Neural Networks for Signal Processing*, 276–285.

Platt, J., et al. 1998. Sequential minimal optimization: A fast algorithm for training support vector machines.

Schott, J. R. 2005. Matrix analysis for statistics.

Smirnov, E.; Sprinkhuizen-Kuyper, I.; and Nalbantov, G. 2004. Unanimous voting using support vector machines. In *Belgium-Netherlands Conference on Artificial Intelligence*, 43–50.

Wen, Z.; Zhang, R.; Ramamohanarao, K.; Qi, J.; and Taylor, K. 2014. Mascot: fast and highly scalable SVM cross-validation using GPUs and SSDs. In *International Conference in Data Mining*, 580–589. IEEE.

Wen, Z.; Li, B.; Kotagiri, R.; Chen, J.; Chen, Y.; and Zhang, R. 2016. Improving efficiency of SVM k-fold cross-validation by alpha seeding. *Technical Report arXiv:1611.07659 [cs.LG], arXiv*.

Wu, Z., and Li, C. 2006. Feature selection for classification using transductive support vector machines.