Reducing Training Time for Linear SVMs

Nick Arnosti, Williams College

August 6, 2010

***Abstract*—Support Vector Machines (SVMs) have been shown to achieve high performance on classification-related tasks, and are generally portable across domains. Because training time for SVMs typically scales poorly as the size of the training set increases, a great deal of effort has been dedicated to reducing the time required to learn a classification model. Recent work presented by Joachims in [1], and improved by Franc and Sonnenburg in [2] and [3], makes use of a reformulation of the SVM primal problem and cutting-plane approximation algorithms to train linear SVMs in time linear with respect to the size of the training set. In this paper, we observe an inefficiency in the algorithms presented in the above works, and introduce a modification which reduces training time for linear SVMs by up to 40% when compared to the methods presented in [2] and [3]. Additionally, we analyze the effect that runtime parameters and data set attributes have on the performance of our algorithms.**

I. Introduction

When using an SVM to classify objects, it must be provided with training data. This consists of a set of examples of the form $(\mathbf{x}_i, y_i)$, where each $\mathbf{x}_i$ is a vector of feature values (independent variables) taken from the space $\mathcal{X}$, and y_i is a class label (the dependent variable) taken from the set $\mathcal{Y}$. The idealized goal, then, is to come up with a function $g : \mathcal{X} \to \mathcal{Y}$ such that for each i, $g(\mathbf{x}_i) = y_i$. Throughout this paper, n is the number of training examples, s the number of features for each example, $\mathcal{X} = \mathbb{R}^s$, and k is the number of distinct classes.

For binary classification, $\mathcal{Y} = \{-1, +1\}$, and the SVM is biased to look for solutions of the form $g(\mathbf{x}_i) = sgn(\mathbf{w}^T\mathbf{x}_i + b)$ for fixed $\mathbf{w} \in \mathbb{R}^f, b \in \mathbb{R}$ (from [4]). For linearly separable data sets, the goal is to find the choice of $\mathbf{w}$ which maximizes the margin between the two classes. This is equivalent to minimizing $||\mathbf{w}||$ subject to the constraints $y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 \ \forall i$.

In general, the data is not linearly separable, so it is necessary to introduce error terms $\xi_i \geq 0$ for each training example. The goal becomes to maximize the margin between classes while also minimizing error on the training set. Using an L1-loss function, this boils down to the following optimization problem (from [5]):

$$\begin{aligned} \text{minimize:} \quad & \tfrac{1}{2}\mathbf{w}^T\mathbf{w} + \tfrac{C}{n}\textstyle\sum_{i=1}^{n} \xi_i \\ \text{subject to:} \quad & y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 - \xi_i \ \forall i, \end{aligned} \tag{1}$$

where $C \in \mathbb{R}$ is known as the regularization constant and determines the relative importance of the two objectives.

One unfortunate result of this formulation is that because we have with n error terms ξ_i, optimizing them takes $\mathcal{O}(n^2)$ time, making SVM training a very slow process on large data sets. As a result, most well-known SVM solvers, such as LIBSVM (http://www.csie.ntu.edu.tw/ cjlin/libsvm/), scale poorly as the size of the training set increases.

Many multi-class solutions simply break the problem down into a series of binary classification problems, and then create a binary classifier to solve each of these. The simplest and most common approach is referred to as *one-against-all*, which trains one binary SVM, $\mathbf{w}_y$, for each class $y \in \mathcal{Y}$, using as input labels $+1$ if $y_i = y$ and -1 otherwise. The decision function in this case is $y_{predicted} = \text{argmax}_{y\in\mathcal{Y}}(\mathbf{w}_y^T\mathbf{x}_i + b_y)$. Other common techniques are known as *one-against-one*, *half-against-half* and *error-correcting output coding*, described in [6], [7] and [8], respectively. It is noted in [9] that for many data sets, these approaches do not produce results that are as good as coming up with a true multi-class solution.

In the multi-class approach presented in [10], instead of one slack variable for each point in the training set, there are k of them. Making the simplifying (and easily reversible) assumption that each $b_y = 0$, let ξ_{ij} denote the error of point i with respect to class j, and $\mathbf{w}_j$ denote the linear classifier corresponding to class j. Then the optimization problem to be solved is of the form:

$$\begin{aligned} \text{minimize:} \quad & \tfrac{1}{2}\textstyle\sum_{j=1}^{k} \mathbf{w}_j^T\mathbf{w}_j + C\sum_{i=1}^{n}\sum_{j=1}^{k} \xi_{ij} \\ \text{subject to:} \quad & \mathbf{w}_j^T\mathbf{x}_i \geq 1 - \xi_{ij} \ \forall i, j. \end{aligned} \tag{2}$$

Unfortunately, due to the large number of slack variables, solving this optimization problem exactly is very computationally intensive. In [9], the number of slack variables is reduced to n by defining $\xi_i = \max_j\{\xi_{ij}\}$. This simplifies the form of the optimization problem above, but optimizing the $n \times k$ matrix $\mathbf{w}$ remains a time-intensive task as the number of classes and training examples grows. Recent research ([1],[2],[11]) has focused on finding approximate solutions, and has resulted in a significant decrease in SVM training time. Section II discusses the algorithms presented in some of this recent work. Section III makes note of one way in which the algorithms presented in [1] and [2] are sub-optimal and introduces two new algorithms which make use of this observation. We tested our algorithms on several data sets, and the results of these tests are presented and analyzed in Section IV, with additional data provided in the Appendix. In the process of conducting tests, we stumbled across an unexpected benefit to our algorithms, which is presented in Section V. We close with a brief conclusion and discussion of possibilities for future work.

II. Previous Work

In [1], Joachims presents what he calls the "structural formulation" of the primal (1), which reduces the number of slack variables from n to 1. For mathematical simplification, Joachims makes the assumption that $b = 0$, which can easily

be reversed by adding a feature of weight 1 to each vector $\mathbf{x}_i$. This formulation can be written as:

$$\begin{aligned} &\text{minimize} && \tfrac{1}{2}\mathbf{w}^T\mathbf{w} + C\xi \\ &\text{subject to} && \tfrac{1}{n}\sum_{i=1}^{n} \max(0, 1 - y_i\mathbf{w}^T\mathbf{x}_i) \leq \xi \end{aligned} \tag{3}$$

Joachims shows that (1) and (3) are mathematically equivalent after setting $\xi = \frac{1}{n}\sum_{i=1}^{n}\xi_i$, meaning that a vector $\mathbf{w}^*$ is a solution to one if and only if it is a solution to the other. This structural formulation made it possible to come up with very good approximations of the optimal objective value in time linear with respect to the size of the training set - a significant improvement upon past performance.

In the years since Joachims introduced this structural formulation, several others have looked to extend and improve upon his work. One notably successful implementation is the Optimized Cutting Plane Algorithm for Support Vector Machines, or OCA. This paper will adopt the notation used in [2], where the objective function from (3) is given the name $F(\mathbf{w})$, and the expression on the left hand side of the constraint, which represents the risk using an L1-loss function, is dubbed $R(\mathbf{w})$. Rather than solving the problem directly, Joachims' cutting-plane algorithm (henceforth referred to as CPA) creates ever-better approximations $R_t(\mathbf{w})$ of $R(\mathbf{w})$, where t indicates the number of iterations.

In [1], Joachims conceives of the constraint from (3) as 2^n distinct constraints. Each iteration modifies $R_t(\mathbf{w})$ by taking into account the "most violated constraint." In the language of [2], this corresponds to taking a subgradient of $R(\mathbf{w})$. Given a choice of $\mathbf{w}'$, the subgradient of interest $\mathbf{a}'$ can be computed as follows:

$$\mathbf{a}' = -\frac{1}{n}\sum_{i=1}^{n}\pi_i y_i \mathbf{x}_i \quad \text{where } \pi_i = \begin{cases} 1 & \text{if } y_i\langle \mathbf{w}', \mathbf{x}_i\rangle \leq 1 \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

Because R is a convex function, for any point $\mathbf{w}'$ and the corresponding subgradient $\mathbf{a}'$, the linear approximation $R(\mathbf{w}') + \langle \mathbf{a}', \mathbf{w} - \mathbf{w}'\rangle$ provides a lower bound for $R(\mathbf{w})$. For notational simplicity, the authors of [2] define $b' = R(\mathbf{w}') - \langle \mathbf{a}', \mathbf{w}'\rangle$, so that $R(\mathbf{w}) \geq \langle \mathbf{a}', \mathbf{w}\rangle + b'$. Given a collection of cutting planes defined by pairs $(\mathbf{a}_i, b_i)$, $R_t(\mathbf{w})$ is their pointwise maximum (subjected to a non-negativity constraint): $R_t(\mathbf{w}) = \max(0, \max_{1\leq i\leq t}\{\langle \mathbf{a}_i, \mathbf{w}\rangle + b_i\})$.

One iteration of Joachims' CPA computes $\mathbf{w}_{t+1}$ by solving the simplified optimization problem generated by substituting $R_t(\mathbf{w})$ for $R(\mathbf{w})$. Then $\mathbf{a}_{t+1}$ and b_{t+1} are computed by the rules above. The algorithm terminates when $1 - \frac{F_t(\mathbf{w}_t)}{F(\mathbf{w}_t)} \leq \epsilon$, a constant provided as a parameter to the solver. This guarantees that $F(\mathbf{w}_t)(1-\epsilon) \leq F(\mathbf{w}^*)$, where $F(\mathbf{w}^*)$ is the solution to (1).

There are several advantages to this technique. Computing $\mathbf{a}_i$ and b_i takes $\mathcal{O}(ns)$ time. In [1], Joachims proves that the number of iterations required to reach ϵ-precision depends only on ϵ (he bounds the growth at a rate of $\mathcal{O}(C/\epsilon)$) and not on the number of training examples n. The time required to solve the reduced optimization problem grows superlinearly with the number of iterations, but as the number of constraints is bounded by a constant independent of n and s, CPA runs in $\mathcal{O}(ns)$ time. Further details and analysis of CPA can be found in [12].

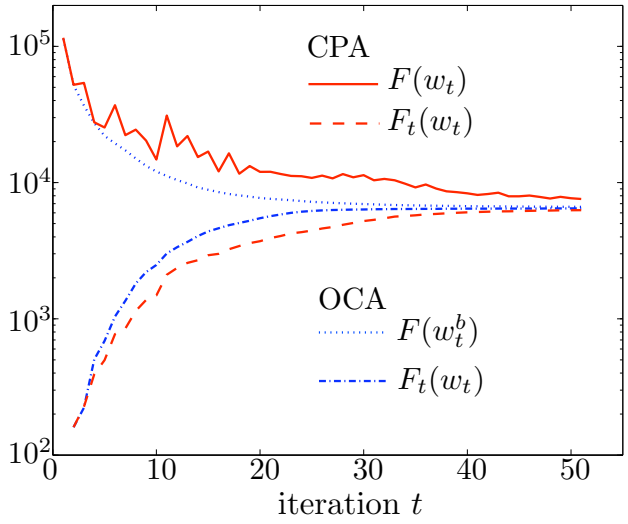

Fig. 1. The effect of the OCA changes, with objective value on the y-axis. Note that the CPA overestimate $F(\mathbf{w}_t)$ fluctuates significantly, while the OCA overestimate $F(\mathbf{w}_t^b)$ decreases monotonically, causing faster convergence.

In [2], the authors note several ways in which CPA is sub-optimal. Most notably, while the under-estimate $F_t(\mathbf{w}_t)$ increases monotonically with respect to t, the over-estimate $F(\mathbf{w}_t)$ fluctuates significantly, which can lead to slow rates of convergence. To solve this problem, a new value $\mathbf{w}_t^b$ is defined by $\mathbf{w}_t^b = \text{argmin}_{\mu\geq 0} F(\mu\mathbf{w}_t + (1-\mu)\mathbf{w}_{t-1}^b)$. Note that for $\mu = 0$, the argument to F is $\mathbf{w}_{t-1}^b$, so we have that $F(\mathbf{w}_t^b) \leq F(\mathbf{w}_{t-1}^b)$, and the over-estimate $F(\mathbf{w}_t^b)$ decreases monotonically. A visualization of the difference is taken from [2] and presented in Figure 1. While this modification does not change the growth rate, $\mathcal{O}(C/\epsilon)$, for the number of iterations, in practice OCA requires many fewer iterations to converge than CPA does. As a result, OCA training time is significantly less than that of CPA, often by an order of magnitude or more [2].

For notational convenience, define $f(\mu) = F(\mu\mathbf{w}_t + (1-\mu)\mathbf{w}_{t-1}^b)$. The process of finding the value of μ which minimizes f shall be referred as the "line search." It is shown in [2] that $\partial f(\mu)$ is a piecewise linear function with discontinuities at n values of μ. Solving $\partial f(\mu) = 0$ requires finding the points of discontinuity (each one can be computed in $\mathcal{O}(f)$ time), sorting them (which requires $\mathcal{O}(n\log n)$ time), and then evaluating the function at some of these points to determine where $\partial f(\mu)$ crosses the x-axis.

In [3], the authors present their multi-class implementation of OCA. In this formulation $\mathbf{w} \in \mathbb{R}^d$ (rather than $\mathbb{R}^f$) for some choice of d, and given a function $\Psi : \mathcal{X}\times\mathcal{Y} \to \mathbb{R}^d$, define the decision function by $h_{\mathbf{w}}(\mathbf{x}) = \text{argmax}_{y\in\mathcal{Y}}\langle \mathbf{w}, \Psi(\mathbf{x}, y)\rangle$. Again, the objective function $F(\mathbf{w})$ is $\frac{1}{2}\mathbf{w}^T\mathbf{w} + CR(\mathbf{w})$, where risk $R(\mathbf{w})$ is defined by:

$$\frac{1}{n}\sum_{i=1}^{n}\max_{y\in\mathcal{Y}}(\delta(y, y_i) + \langle \Psi(\mathbf{x}_i, y) - \Psi(\mathbf{x}_i, y_i), \mathbf{w}\rangle) \tag{5}$$

and $\delta(\cdot,\cdot)$ is the zero-one loss function. This is a sensible choice of risk, as a point can only be classified if its contribution to the summand exceeds 1. As a result, this risk function provides an upper-bound on the average training error (or empirical risk) $R_{emp}(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}\delta(h_{\mathbf{w}}(\mathbf{x}), y_i)$.

Vectors $\mathbf{w}_t$ and $\mathbf{w}_t^b$ are computed as in the binary case. The latter computation again boils down to minimizing $f(\mu) = F(\mu\mathbf{w}_t + (1-\mu)\mathbf{w}_{t-1}^b)$. In this case, $\partial f(\mu)$ has discontinuities at $n(k-1)$values of μ (recall $k = |\mathcal{Y}|$). Finding each of these values can be done in $\mathcal{O}(k^2)$ time, and sorting them can be done in $\mathcal{O}(nk\log nk)$ time, so the overall time complexity of the line search is $\mathcal{O}(nk^2 + nk\log nk)$.

III. Improving Upon OCA

Though the math behind completing the line search for an optimal value of μ is elegant, the superlinear asymptotic runtime is an immediate warning flag, as Joachims' methods scale linearly with n. It seems that this portion of the OCA leaves something to be desired. When looking to improve the OCA algorithm, our primary focus was on developing faster techniques for choosing a good value of μ.

Using the terminology presented in [2], the CPA presented in [1] uses $\mu = 1$ for each iteration. While this hastens each iteration by avoiding a line search, the effect is a notable increase in the number of iterations required. The thought which inspired our research is that perhaps finding good approximations for μ could be done substantially faster than the complete line search (thus reducing time spent on each iteration), while keeping the number of iterations from increasing as dramatically as results from simply setting $\mu = 1$.

A. The Grid Search Cutting Plane Algorithm (GCA)

Our initial grid-search cutting-plane algorithm, GCA, rather than solving analytically for the maximum of $f(\mu)$, computes its value at several points and uses these to make an intelligent choice. The method by which this choice is made is inspired by the grid search for optimal parameter values presented in [13]. Given a parameter $\beta \in (0, 0.5]$, GCA computes $f(m\beta)$ for $m = 0, 1, \ldots \lfloor \frac{2}{\beta} \rfloor$. Then, a more refined search is conducted in the neighborhood of the best choice of $m\beta$, incrementing by β^2 at each step. Finally, a third search occurs in the most promising region identified by the second pass, incrementing by β^3 at each step. The final output is the value of μ among those checked which minimized $f(\mu)$. This process is detailed in Algorithm 1, and makes use of a function **makePass** and a global variable $bestval$.

It should be noted that evaluating $f(\mu)$ requires $\mathcal{O}(nk)$ time, and that for any choice of β, the number of times that $f(\mu)$ is evaluated is bounded by a constant independent of n, k, C, or ϵ. As a result, GCA reduces the asymptotic runtime of the line search from $\mathcal{O}(nk^2 + nk\log nk)$ to $\mathcal{O}(nk)$.

One might be concerned that GCA will get stuck in a local minimum which is far from the true optimal value of μ. Fortunately, this cannot occur. The shape of the function $f(\mu)$ is described extensively in [2] and [3]. For the purposes of this paper, it is enough to note that ∂f, though not continuous, increases monotonically. In other words, the concavity of f never changes: it is always concave-up. As a result, f has no extraneous local minima for the grid search to find.

double **makePass**($init$, max, inc)
  $\mu' \leftarrow \frac{init+max}{2}$
  for ($\mu \leftarrow init; \mu < max; \mu \leftarrow \mu + inc$) **do**
    if ($f(\mu) < bestval$) **then**
      $bestval \leftarrow f(\mu)$
      $\mu' \leftarrow \mu$
    end if
  end for
  return μ'

Algorithm 1 Grid Search Algorithm (GCA)

$bestval \leftarrow \infty$
$\mu_1 \leftarrow$**makePass**(0,2,β)
$\mu_2 \leftarrow$**makePass**($\mu_1 - \frac{\beta}{2}, \mu_1 + \frac{\beta}{2}, \beta^2$)
$\mu_3 \leftarrow$**makePass**($\mu_2 - \frac{\beta^2}{2}, \mu_2 + \frac{\beta^2}{2}, \beta^3$)
return μ_3

The computationally expensive part of the grid search is repeatedly computing $f(\mu)$. If this is done too frequently, the GCA grid search will run more slowly than the OCA line search. This was the motivation behind the three-tiered approach described above. Conducting a single pass at high granularity would require computing $f(\mu)$ too many times. Conducting a single pass at low granularity would alleviate this problem, but at the expense of having a very rough estimate for the optimal value of μ. Instead, GCA essentially "zooms in" on the most promising sections by conducting high-granularity searches only in regions where $f(\mu)$ is relatively low.

Unsurprisingly, the value of the parameter β has a significant effect on the efficiency of the grid search. For small values of β, $f(\mu)$ is computed too frequently, leading to poor performance. For large values of β, $f(\mu)$ is evaluated fewer times, but the precision of the search is worse, so the number of iterations required to converge is greater. We experimentally determined that setting $\beta = 0.2$ consistently proved to be a good compromise between these influences. Unfortunately, even for large values of β (such as 0.5), a naïve grid search evaluates $f(\mu)$ too many times with each iteration, causing the grid search to be slower than the OCA line search. At a glance, this seems to spell doom for using a grid-search to determine μ.

B. An Improved Grid Search (GCA2)

Fortunately, there are ways to get around this problem while maintaining the core idea of using a grid-search. While GCA neatly encapsulates the central idea behind the grid-search, it requires several modifications in order to outperform OCA. Accordingly, we developed a revised grid-search algorithm, GCA2, which makes two crucial changes to GCA. These modifications are displayed in Algorithm 2 (which makes use of a modified routine **makePass2**), and described in more detail below.

The first of these optimizations takes advantage of the concavity of f by truncating the grid search if $f(m\beta) <$

double **makepass2**($init$, inc)
  $\mu \leftarrow init$
  while $(f(\mu + inc) < f(\mu))$ **do**
    $\mu \leftarrow \mu + inc$
  end while
  return μ

Algorithm 2 Modified Grid Search Algorithm (GCA2)

if $(uncertainty \geq t_2)$ **then**
    $\mu_1 \leftarrow \mathbf{makepass2}(\mu_{prev} - 1, \beta)$
else $\mu_1 \leftarrow \mu_{prev}$
end if
if $(uncertainty \geq t_1)$ **then**
    $\mu_2 \leftarrow \mathbf{makepass2}(\mu_1 - \frac{\beta}{2}, \beta^2)$
else $\mu_2 \leftarrow \mu_{prev}$
end if
$\mu_3 \leftarrow \mathbf{makepass2}(\mu_2 - \frac{\beta^2}{2}, \beta^3)$
if $(|\mu_3 - \mu_{prev}| \leq h(uncertainty))$ **then**
    decrease $uncertainty$
else increase $uncertainty$
end if
$\mu_{prev} \leftarrow \mu_3$
return μ_3

$f((m+1)\beta)$, as this indicates that the search has passed the optimal value of μ. The technique ensures that time is not wasted calculating values of f at points which are increasingly far from optimal, and is also applied during the second and third passes of the algorithm. Another advantage to this is that there is no longer the arbitrary restriction $\mu < 2$.

The second change takes advantage of the observation that after the first few iterations, the optimal value of μ, as calculated by the OCA line search, was generally quite low (certainly below 0.1). In other words, as the algorithm progressed, $\mathbf{w}_t^b$ was modified only slightly with each iteration. As a result, a full grid search (even truncated as described above), wasted too much time checking large values of μ. To avoid this, we introduced a new variable, `uncertainty`, which essentially determines the width of the search window for each iteration.

Initially, `uncertainty` is set at 1, and all three passes of the line search are conducted. From then on, each iteration searches values of μ in an interval close the value of μ selected by the previous iteration. The width of this interval is determined by the value of `uncertainty`. There are two threshold values $t_1 < t_2$. If `uncertainty` $> t_2$, three passes are conducted. If $t_1 <$ `uncertainty` $< t_2$, the first (coarsest) search is bypassed, narrowing the search window. If `uncertainty` $< t_1$, only the final (most refined) search is conducted. After each iteration, the value of `uncertainty` is adjusted according to how much the optimal value of μ has changed from the previous iteration to the current one. If the difference between these is less than a threshold value h, `uncertainty` is decreased. Otherwise, it seems possible that a broader search was needed, so `uncertainty` is increased. It is best to make h depend on the width of the search region (which in turn is a function of `uncertainty`).

This algorithm takes advantage of the fact that most late iterations only modify $\mathbf{w}_t^b$ slightly by saving time when `uncertainty` is low, while still allowing the flexibility of a full grid search when necessary. A summary of the performance of GCA2 is presented in the following section, with more complete data provided in the Appendix.

C. The Three-point Cutting Plane Algorithm (TCA)

Inspired by previous work and hoping to take advantage of the shape of $f(\mu)$, we developed a second way to approximate optimal values of μ, which we refer to as the Three-point Cutting Plance Algorithm (TCA). This algorithm makes use of three possibilities for μ, designated `low`, `mid`, and `high`. These values define a search window which is initially centered at the value of μ selected by the previous iteration. The width of the initial window is determined by the variable `uncertainty`. If $f($`mid`$)$ is less than both $f($`low`$)$ and $f($`high`$)$, this indicates that the optimal value of μ lies within the current search region, so the search region is narrowed by shifting the values `low` and `high` towards `mid`. If $f($`low`$)$ is smaller than the other values, then our current search region is to the "right" of the optimal value of μ, so the entire window is shifted left. Similarly, if $f($`high`$)$ is the smallest value, the search region is shifted right. This continues until the width of the window, `high`$-$`low`, is less than a prescribed parameter γ, at which point we set $\mu =$ `mid`.

When designing the algorithm, two aspects of it were tweaked to determine what provided the best performance. The first was, unsurprisingly, the parameter γ. The second was the amount by which `low` and `high` are shifted inwards when $f($`mid`$)$ is the lowest value. In general, the values were computed to be a weighted average of their previous values and `mid`. As is displayed in Algorithm 3, the relative weights of these points, α_l and α_h, are computed using a function g which takes as input $f($`mid`$)$ and $f($`low`$)$ or $f($`high`$)$, respectively. Intuitively, the closer $f($`low`$)$ is to $f($`mid`$)$, the less the point `low` should move, and so the larger the weight for α_l. After some experimentation, it was determined that $\gamma = 0.02$, $g(x, y) = (\frac{x}{y})^2$ provided good performance. Our implementation uses the equation $h(x) = x/2$ to determine whether to increase or decrease `uncertainty`.

IV. Experimental Results

Tests were primarily run on two data sets, both available from the LIBSVM data sets website (http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/).
The first, Covtype, provided cartographic data on sections of forest, and classified each location based on the dominant form of tree cover. The training set consists of 581012 examples, each with 54 features, divided into 7 classes. All features were linearly scaled to take values in the range [0,1]. The second data set, MNIST, presents a hand-written digit recognition problem. The training set consists of 60000 examples, each with 780 features, divided into 10 classes. An additional data set, RCV1, which uses data from the Reuters corpus and was also obtained from the LIBSVM website,

Algorithm 3 Three-point Cutting Plane Algorithm (TCA)

$low \leftarrow \mu_{prev} - uncertainty$
$mid \leftarrow \mu_{prev}$
$high \leftarrow \mu_{prev} + uncertainty$
while $(high - low > \gamma)$ **do**
  if $(f(low) < f(mid))$ **then** /* shift search left */
    $high \leftarrow mid$
    $mid \leftarrow low$
    $low \leftarrow 2 \cdot mid - high$
  else if $(f(high) < f(mid))$ **then** /* shift search right */
    $low \leftarrow mid$
    $mid \leftarrow high$
    $high \leftarrow 2 \cdot low - mid$
  else /* contract search window */
    $\alpha_l \leftarrow g(f(mid), f(low))$
    $\alpha_h \leftarrow g(f(mid), f(high))$
    $low \leftarrow \frac{mid + \alpha_l \cdot low}{1+\alpha_l}$
    $high \leftarrow \frac{mid + \alpha_h \cdot high}{1+\alpha_h}$
  end if
end while
if $(|mid - \mu_{prev}| \leq h(uncertainty))$ **then**
  decrease $uncertainty$
else increase $uncertainty$
end if
$\mu_{prev} \leftarrow mid$
return mid

was used for some experiments. Its test set (which we used for training) contains 518571 examples, each consisting of 47236 features. These examples are divided into 53 classes. Because the focus of this project is on data sets which require significant training time, these data sets were selected from the LIBSVM site primarily because they were among the largest. All tests were run on a Linux VirtualBox machine installed on a 2.8 GHz Intel Core i7 processor with 8 GB main memory.

A. Results Using Default Parameter Values

The graphs in Figures 2 and 3 show results when using the default parameter values of $C = 1, \epsilon = 0.01$. Search Time represents the total time spent determining which value of μ to use for each iteration. All tests reached the termination criteria $1 - \frac{F_t(\mathbf{w}_t^b)}{F(\mathbf{w}_t)} < \epsilon$, and classification performance for the three models was equivalent (OCA, GCA2 and TCA error rates were 35.92%, 35.90%, and 35.84%, respectively, on Covtype, and 5.57%, 5.56%, and 5.56% on MNIST). Data from more experiments, along with discussion of the effect of changing these parameters, follows in sections IV-B, IV-C, and the Appendix.

The algorithms GCA2 and TCA notably outperform OCA on both of the primary data sets. On MNIST, the total time spent finding a good value for μ decreased by over 70% for both algorithms, causing the time spent per iteration to decrease by just over 16%. As the number of iterations increased by only 2-3%, the net effect was approximately a 14% improvement in training time. On Covtype, results were even better: both algorithms take over 38% less time to train than OCA. This is achieved by reducing the time spent selecting μ by over 65-70% while increasing the number of iterations by less than 10%.

It is not surprising that GCA2 and TCA perform relatively better on Covtype than on MNIST, as the OCA line search consumes over 56% of the total time on the former, and only 21% of the total time on the latter. When tested on RCV1 the three algorithms performed equivalently (slight fluctuations from one trial to the next caused variation in which algorithm terminated first). It is unsurprising that GCA2 and TCA do not improve upon OCA in this case, as under 8% of OCA training time was spent on the line search. It is natural to ask what causes this discrepancy, since understanding it would make it possible to determine (without explicitly running tests) whether or not the algorithms presented in this paper were likely to provide a significant reduction in training time on a particular data set.

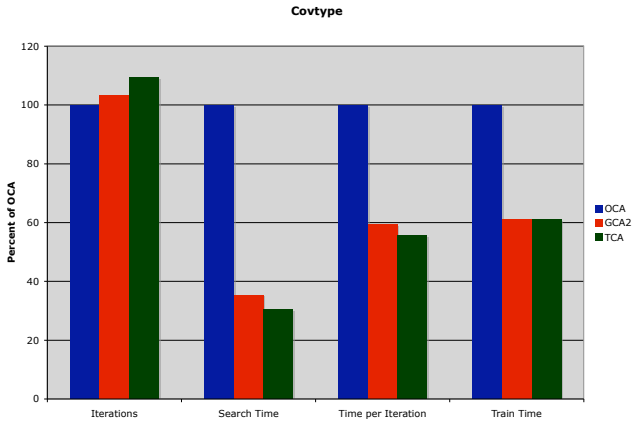

Fig. 2. Performance on Covtype, normalized to OCA performance. Note that search time, the focus of GCA2 and TCA, is reduced by 60-70%, causing an improvement in time of about 40%.

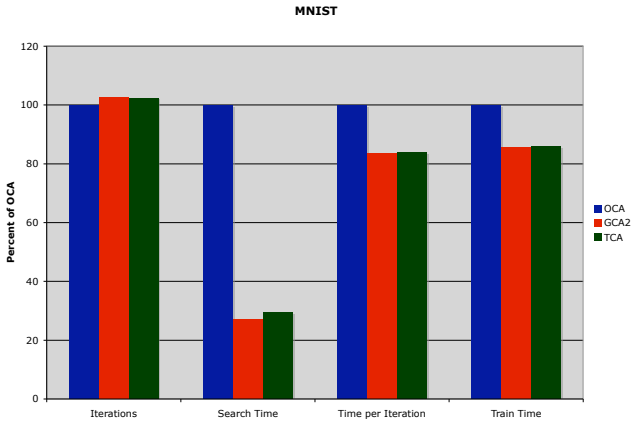

Fig. 3. Performance on MNIST, normalized to OCA performance. Note that despite improving search time by over 70%, GCA2 and TCA training times do not decrease as substantially as for Covtype.

B. Examining the Number of Features

It should be noted that RCV1, for which the line search consumed only 7.8% of the total time, has over 47000 features. On MNIST, which makes use of 780 features, 21% of the total time was devoted to the line search. Covtype, meanwhile, uses only 54 features, and the line search consumes 56% of the total time. This pattern seemed worth examining more closely.

From a theoretical perspective, the OCA line search consists of computing $\mathcal{O}(nk)$ values of μ where the gradient of $f(\mu)$ might have a discontinuity, and then sorting these values. While time spent on the initial computation depends on the number of features used, the line search time is dominated by the sorting of the values [2], which takes time dependent only on n. Accordingly, while time spent on the line search is not technically independent of the number of features s, for data sets with a large number of training examples (which are the only ones of relevance for this paper), the time spent on the line search should not be greatly affected by the number of features. Other portions of the algorithm, however, take time scaling linearly with s. As a result, the fraction of training time spent on the line search should increase as the number of features decreases.

In order to test this hypothesis, we created new training sets from the RCV1 test set and MNIST training set. These training sets contained all of the original data points, but left out many of the original features. We then ran the OCAS solver on each of these new data sets. The results, using $C = 10, \epsilon = 0.01$ for MNIST and $C = 1, \epsilon = 0.01$ for RCV1, are shown in Figure 4.

As the number of features decreases, the total time spent per iteration decreases, while the time spent during each iteration on the line search remains fairly constant. As a result, the percentage of the total time spent on the line search increases notably. Because GCA2 and TCA focus on improving the OCA line search, this indicates that these methods will outperform OCA by the most on data sets with relatively few features.

Of course, one cannot arbitrarily reduce the dimensionality of a data set without potentially reducing classification performance. Because decreasing numbers of features are accompanied by decreasing training time, there is already a notable incentive to conduct research into how best to identify and remove less informative features early in the classification process. This paper does not seek to develop techniques to accomplish this. Instead, we stop at noting which data sets are most impacted by GCA2 and TCA modifications, and observing that if techniques for the elimination of uninformative features improve, the algorithms presented here will become relatively more advantageous.

It seems likely that a more thorough examination of this phenomena, rather than looking at the raw number of features s for a data set, should consider how s compares with the number of training examples n, as this value crucially determines run-times for various parts of the algorithm. We leave this for future research.

Features	% Search	Features	% Search
780	18.8	4373	30.0
473	25.1	2368	31.9
337	36.7	676	42.5
139	57.2	186	56.4

Fig. 4. For both MNIST (left) and RCV1 (right), as the number of features increases, the percentage of training time spent on the OCA line search increases.

C. Effects of Parameters C and ϵ

It should be noted that the percentage of the training time spent on the line search depends not only on the data, but also on the parameters used when solving.

If the tolerance ϵ is very small, then the program will have to go through more iterations to obtain the desired accuracy. Because the set of constraints grows with the number of iterations, the amount of time required to solve these constraints (referred to as Quadratic Programming, or QP, Time) grows super-linearly with respect to the number of iterations. The other steps of the algorithm, including the line search, require a constant amount of time per iteration. This is shown in Figure 5. As a result, as ϵ decreases, QP time comes to dominate the training time, and so the percentage of the process spent on the line search decreases. Accordingly, methods which aim to decrease training time by focusing on the line search, such as GCA2 and TCA, are less likely to demonstrate a notable reduction in training time.

At some point, any additional iterations required as a result of sub-optimal determination of μ should outweigh the reduced time spent on the line search, causing OCA to outperform GCA2 and TCA. It should be noted, however, that classification accuracy universally converges to its final value for values of ϵ much greater than this threshold value. Correspondingly, there is no need for the user of the program to set ϵ to values low enough to cause this behavior.

Unsurprisingly, given the bound of $\mathcal{O}(C/\epsilon)$ on the number of iterations, increasing C has a similar effect to decreasing ϵ. Thus, for sufficiently large values of C, time spent solving the

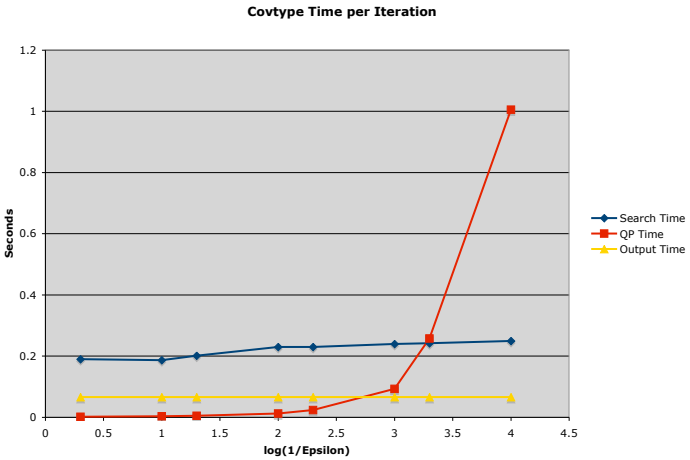

Fig. 5. Covtype Time Breakdown. As ϵ decreases, the number of iterations required to converge increases. While most steps in the algorithm take a constant amount of time per iteration, this causes time spent solving the reduced optimization problem, QP Time, to increase dramatically.

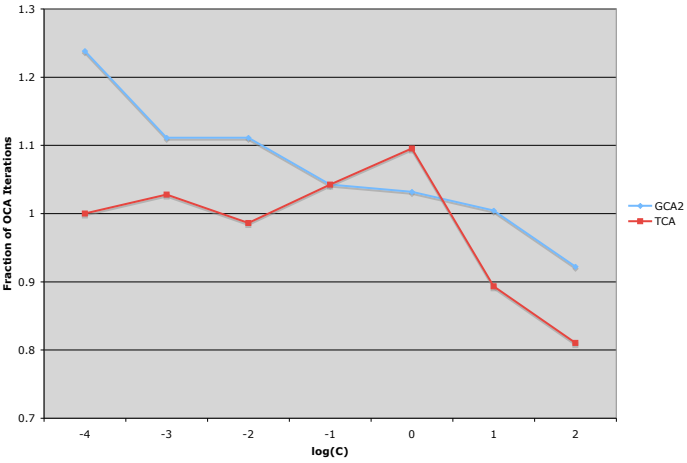

Fig. 6. Effect of C on Iterations Ratio. As the regularization constant C increased, the ratio of iterations required by GCA2 and TCA to the number required by OCA decreased. Thus, even as search time becomes a negligible fraction of training time, these new algorithms continue to outperform OCA.

quadratic programming problem dwarfs all other components. Interestingly, however, increasing C sometimes has another effect on the training times for the three algorithms being discussed. On the Covtype data set, as C grows very large, OCA becomes increasingly likely to use $\mu = 0$ for many consecutive iterations. For reasons described in the following section, this frequently results in GCA2 and TCA terminating in fewer iterations than OCA. Thus, even for values of C where time spent on the line search comprises a negligible fraction of the total time, GCA2 and TCA may improve upon the the performance of OCA. Figure 6 displays the relative number of iterations for GCA2 and TCA (when compared to OCA) for different values of C (logarithm for x-axis is base 10). Tables showing the full results of tests run with different values of C and ϵ are provided in the Appendix.

V. Another Benefit to GCA2 and TCA

When seeking an approximate solution to the primal optimization problem (1), both CPA and OCA use a greedy approach: each iteration attempts to reduce the objective value by as much as possible. Since the ultimate goal is to find an objective value within a tolerance ϵ of the optimal value, this is a reasonable approach. It should be noted, however, that choices for $\mathbf{w}_t^b$ have a notable impact on values of $\mathbf{w}_i^b$ for $i > t$. This means that a slightly "worse" choice of $\mathbf{w}_t^b$ during the current iteration may result in better options in the future, and lead to faster convergence.

Recall that in OCA, $\mathbf{w}_t^b$ is found by searching along a line between $\mathbf{w}_{t-1}^b$ and $\mathbf{w}_t$. Thus, the following section will discuss finding optimal values of μ (rather than $\mathbf{w}_t^b$), as the two are equivalent.

The inspiration for GCA2 and TCA was that by approximating the optimal value of μ, rather than finding it exactly, the time spent on each iteration of the algorithm would decrease. It was assumed that there was a tradeoff: in return for reducing the time per iteration, the values of μ that were found would not be as good as those found with OCA, and as a result the above two algorithms would require more iterations to converge. While this is often the case, for a notable number of experiments, GCA2 and TCA terminated after *fewer* iterations than OCA (in addition to requiring less time per iteration). This is only possible because of the observation that all of the algorithms presented in this paper use a greedy solution to a problem which is not inherently greedy.

The above observation is only interesting if it is possible to identify values of μ which will be particularly good or bad for future convergence. In general, this is difficult. Nevertheless, there is one value of μ which has a clear drawback. Note that if $\mu = 0$, $\mathbf{w}_t^b = \mathbf{w}_{t-1}^b$. While it is true that $\mathbf{w}_t$ and $\mathbf{w}_{t+1}$ will be different (so the algorithm will never truly get "stuck"), if $\mu = 0$ for many iterations in a row, the algorithm has spent all of those iterations without improving its best so-far solution.

Perhaps the data set which best illustrates this danger is "Poker," which can be obtained from the UCI machine learning repository of data bases (http://archive.ics.uci.edu/ml/datasets.html). It contains eight million example 5-card poker hands, with ten features used to describe each (the suit and rank of each card). There are 10 class labels, which describe the quality of the hand (high card, pair, two pairs, etc). When run on this data set with the default values of $C = 1, \epsilon = 0.01$ TCA and GCA2 converged in 234 and 235 iterations, respectively, while OCA required 1454 iterations and took over twelve times as long to find an equivalent solution. Upon further investigation, it became apparent that the concern noted above (namely, repeatedly determining $\mu = 0$ to be optimal) was precisely what caused OCA to perform so poorly. For 1451 consecutive iterations of OCA, $\mu = 0$ was computed to be optimal. Thus, $\mathbf{w}_t^b$ remained at its default value, $\mathbf{0}$, and classification error was 100%. Because of the way that GCA2 and TCA are implemented, these algorithms tended to find small (but nonzero) values for μ, and thus avoided the trap into which OCA fell.

Interestingly, this problem can be avoided by a very simple fix. After modifying OCA to arbitrarily set $\mu = 0.02$ for the first five iterations (after which it returned to its usual line search), the algorithm terminated in 231 iterations, with comparable objective value and classification performance to GCA2 and TCA.

The fact that for $\epsilon > .011$, OCA terminated without ever modifying $\mathbf{w}$ serves as a potent reminder that while solving the primal problem and achieving accurate classification score are often linked (and most SVMs are compared on the basis of their respective objective values), it is possible to succeed at the first without making notable progress on the second.

It should be noted that for many reasons, the Poker data set is not well-suited to multi-class SVM classification. With so few features relative to the number of examples, the SVM is unable to learn a good model - the best it can do is to get about 50% correct. Even this is unimpressive, as over 92% of the examples belong to two of the classes, and so the final predictions are no better than simply assigning the most common label to every hand.

We argue that the limitations of the Poker data set should not undermine the validity of the observations above. The first

reason for this is that even if it is not possible to achieve high classification accuracy, one would hope to achieve the best result possible in relatively few iterations. The second is that while Poker was selected because it provides a particularly glaring example of this concern, the same issue arises when working with other data sets.

To prove this point, we examined the trials run on the Covtype data set, with $C = 100, \epsilon = 0.01$. In this case, OCA requires 744 iterations to converge, while GCA2 and TCA require 686 and 603, respectively. Upon investigation, it was determined that on 490 of the 744 OCA iterations (or 66%), the optimal value of μ was determined to be 0. After modifying the code so that any time $\mu = 0$ was determined to be optimal, μ was instead set to 0.02, OCA terminated in 569 iterations.

These results indicate that the shortcomings of the greedy approaches used by all cutting plane algorithms described in this paper may, in some cases, be quite significant. The quick fixes mentioned here may not be the best way to avoid these concerns, and an investigation into better ways to approach this problem would be fascinating and potentially very useful. Such an exploration, however, is beyond the scope of this paper. For our purposes, it suffices to note that while setting $\mu = 0$ may not always be problematic, repeatedly choosing doing so can result in poor performance.

VI. Conclusion and Future Work

In this paper, we have observed that the line search used to compute the best-so-far solution $\mathbf{w}_t^b$ at each iteration of the OCA algorithm is fairly computationally costly. We present two new algorithms, GCA2 and TCA, designed to approximate this solution in a substantially shorter amount of time. These solutions reduce the asymptotic run-time for the search from $\mathcal{O}(nk^2 + nk \log nk)$ to $\mathcal{O}(nk)$, and demonstrate a notable decrease in training time when tested empirically. Both algorithms reduce the search time by 65-75% while seeing an increase in the number of iterations of approximately 3%. On the two primary data sets used for our tests, Covtype and MNIST, the net effect was a decrease training time by roughly 39% and 14%, respectively.

Additionally, we have examined and analyzed the ways in which the regularization constant C, the tolerance ϵ, and the number of features in the training set impact the performance of GCA2 and TCA relative to that of OCA. For reasonable choices of C and ϵ, the number of features present in a data set was shown to significantly impact the percentage of OCA training time spent on the line search: search time was relatively independent of the number of features, while other portions of the algorithm became significantly slower on data sets with large numbers of features. As a consequence of this fact, GCA2 and TCA are most beneficial on large data sets with relatively few features.

Perhaps one of the most interesting observations was that while GCA2 and TCA do not find the optimal choice of $\mathbf{w}_t^b$ at each iteration, for some data sets they converge in *fewer* iterations than OCA, occasionally by quite a significant margin. This suggests that the approach of reducing the objective function by as much as possible at each iteration can yield far-from-optimal rates of convergence. An open question for future investigation is *which* greedy choices are problematic and whether there are better methods for choosing $\mathbf{w}_t^b$ than the approaches that have so far been employed.

Finally, it is worth noting that while the experiments used for this paper were restricted to multi-class classification tasks, the conceptual ideas presented in this paper apply equally well to binary and multi-class problems. It would be informative to conduct tests on binary data sets, to see whether empirical results are are similar in the binary and multi-class cases. In addition to observing whether GCA2 and TCA cause training times to decrease by comparable ratios to those reported in this paper, we would like to investigate whether the cases where GCA2 and TCA terminate in fewer iterations than OCA are more or less prevalent when solving binary classification problems.

Acknowledgement

This research was conducted with support from NSF grant ARRA 0851783. The author would like to thank Dr. Jugal Kalita for pointing the way to relevant recent publications and providing valuable feedback during the editing process.

References

[1] T. Joachims, "Training Linear SVMs in Linear Time," in *Proceedings of the 12th ACM SIGKDD international conference on knowledge discovery and data mining*. ACM New York, NY, USA, 2006, pp. 217-226.

[2] V. Franc and S. Sonnenburg, "OCAS Optimized Cutting Plane Algorithm for Support Vector Machines", in *Proceedings of International Machine Learning Conference*, pages 320327. ACM Press, 2008a.

[3] V. Franc and S. Sonnenburg, "Optimized Cutting Plane Algorithm for Large-Scale Risk Minimization", in *Jounal of Machine Learning Research*, October 2009.

[4] E. Mayoraz and E. Alpaydin, "Support Vector Machines for Multi-Class Classification", in *IWANN, vol. 2*, 1999, pp. 833-842.

[5] C. Burges, "A Tutorial on Support Vector Machines for Pattern Recognition," in *Data Mining and Knowledge Discovery*. Kluwer Academic Publishers, Bowston, MA, USA, 1998, pp. 121-167.

[6] C. Hsu and C. Lin, "Comparison of Methods for Multiclass Support Vector Machines" in *IEEE Transactions on Neural Networks*, vol. 13, no. 2, March 2002.

[7] H. Lei and V. Govindaraju, "Half-Against-Half Multi-class Support Vector Machines," in *Proc. of the 6th International Workshop on Multiple Classifier Systems*, Seaside, CA, USA, 2005.

[8] T. Dietterich and G. Bakiri, "Solving Multiclass Learning Problems via Error-Correcting Output Codes," in *Journal of Artificial Intelligence Research 2*, 1995.

[9] K. Crammer and Y. Singer, "On the Algorithmic Implementation of Multiclass Kernel-Based Vector Machines" in *The Journal of Machine Learning Research*, vol. 2, 2002, pp. 265-292.

[10] J. Weston and C. Watkins, "Support Vector Machines for Multi-Class Pattern Recognition", in *European Symposium on Artificial Neural Networks*. D-Facto public, Belgium, 1999, pp. 219-224.

[11] S. Keerthi, S. Sundararajan, K Change, C. Hsieh, and C. Lin, "A Sequential Dual Method for Large Scale Multi-Class Linear SVMs" in *Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, August 24-27, 2008, Las Vegas, Nevada, USA.

[12] T. Joachims, T. Finley, and C.N. Yu, "Cutting-plane training of structural SVMs" in *Machine Learning*, 76(1), May 2009.

[13] C. Hsu, C. Chang, and C. Lin, "A Practical Guide to Support Vector Classification." Technical report, Department of Computer Science and Information Engineering, National Taiwan University, Taipei, 2003. http://www.csie.ntu.edu.tw/~cjlin/libsvm/.