Pomona College
Department of Computer Science

Updatable Learning of Support Vector Machines using Sequential Minimal Optimization

Joel Detweiler

April 29, 2012

Submitted as part of the senior exercise for the degree of
Bachelor of Arts in Computer Science

Sara Owsley Sood

Abstract

This paper presents a new algorithm for updating trained Support Vector Machines (SVMs) to reflect the addition of new training data. Training of Support Vector Machines is computationally expensive. This cost is undesirable for situations where there is a trained SVM and new data is introduced, as retraining to account for the new data can be prohibitively expensive. Thus, there is a need for an SVM that can be updated for new data. This update should come at minimal time cost in order to be more efficient than retraining with all of the data. Additionally, an updatable SVM should achieve good classification accuracy as compared to a traditional SVM. The updatable SVM presented here utilizes the Sequential Minimal Optimization algorithm in order to update an SVM for a single new example. Evaluation shows significant time savings from using an updatable SVM without a significant decline in classification performance. These results show that updatable SVMs meet the stated goals and show significant potential for utilization in real-world systems.

Contents

List of Figures

Chapter 1

Introduction

Supervised Machine learning methods have become a common tool in our world today, frequently being used in situations with large amounts of data. Typically, learners are trained on large amounts of data and then put into practice. However, it is not uncommon for new data to continually become available; a situation that we would like our learner to account for.

Consider, for example, the case of a medical software provider that seeks to provide an easy way of diagnosing patients with cancer. They may utilize a supervised machine learning approach that trains on data containing basic health and personal information labeled with a corresponding diagnosis. The learner could then be used to easily diagnose future patients without the presence of highly-trained specialized medical staff. In order to be ensure accuracy, such a system may be trained using several million diagnoses. Training on that much data would likely be costly in terms of time and computational resources.

Now the software provider may frequently obtain new diagnosis data as doctors submit real diagnosis information. This data may be important, as there may be new medical trends (e.g. environmental changes) that the learner should reflect in new diagnoses. Since the provider likely wants the learner to adapt and account for the new data in future diagnoses, it will need to retrain the learner on all the data. However, because of the costs associated with training, completely retraining may be infeasible. Furthermore, by the time the learner has retrained, there may already be a backlog of new data to again add.

Clearly, there is a need for learners that can be updated as new labeled data is obtained. Support Vector Machines (SVMs) are a common machine learning approach, yet they lack this means of being updated with new data.

This paper describes an approach for updating Support Vector Machines with new labeled data. In particular, we are interested in the design of SVMs that can be updated efficiently and without loss of accuracy. We will start by describing the relevant previous work in Chapter 2. We will then present our approach for building an updatable SVM in Chapter 3. Our results showing significant time savings and equivalent accuracy are presented in Chapter 4. Finally, our conclusions and possible directions for future work in this field are described in Chapter 5

Chapter 2

Related Work

2.1 Anytime Learning

Training Support Vector Machines is a hard problem. A commonly used approach for hard problems is anytime algorithms [BD94]. The driving concept behind anytime algorithms is the desire to explicitly trade computation speed and the quality of the computation result.

There have been two main proposed classes of anytime algorithms, contract and interruptible [ZR96]. Contract algorithms are given an allocation of total run-time (or other resources) when activated. If the algorithm is interrupted within the allocated time frame, it is not guaranteed to return useful results. An interruptible algorithm, on the other hand, is not given a resource allocation in advance. However, an interruptible algorithm can be interrupted at any time and must have results prepared. Contract algorithms are acceptable in many cases, but "in many real-life applications it is not possible to allocate the resources in advance" [EM07].

Anytime algorithms can be applied to machine learning tasks. Esmeir and Markovitch [EM07] showed how Decision Tree learning can be done as both a contract and interruptible algorithm. The authors presented their approaches for inducing decision trees of minimal size, as most decision tree induction algorithms have an inductive bias for minimal tree size. They showed positive results from their anytime learners, as accuracy increased and tree size decreased with an increased time allotment. From this, we conclude that anytime learning can be useful and is worth exploring for other machine learning techniques.

Our proposal of an updatable SVM shares many similarities to an anytime algorithm. An updatable SVM is not inherently an anytime learner and allows for different capabilities. However, we present methods for anytime learning using an updatable SVM in section 3.3.

2.2 Support Vector Machines

Support Vector Machines (SVMs) are a common machine learning binary classification model. Cortes and Vapnik described SVMs in their current form in their 1995 work [CV95]. SVMs are considered a large-margin classifier, in that they find the hyperplane that linearly separates the training data with the largest margin between vectors of the two classes [CV95]. It does so through the use of support vectors, which are a small subset of the training data that determine the margin.

2.2.1 SVM Derivation

According to Cortes and Vapnik, a set of labeled training data with input x and binary classification y

$$(x_1, y_1), \ldots, (x_l, y_l) \tag{2.1}$$

is linearly separable if there exists a vector w and scalar b such that the following inequality holds:

$$y_i(w \cdot x_i + b) \geq 1, \qquad i = 1, \ldots, l \tag{2.2}$$

The optimal hyperplane (w, b) is constructed as

$$w \cdot x + b = 0 \tag{2.3}$$

From the hyperplane (w, b) they derive a linear decision function which is used to classify new data points:

$$l(z) = sign(wz + b) \tag{2.4}$$

In order to create the optimal decision function, however, the optimal hyperplane with the largest margin must be found. For the hyperplane (w, b) the size of the margin is

$$\rho(w, b) = \frac{2}{\sqrt{w \cdot w}} \tag{2.5}$$

Thus, maximizing ρ finds the hyperplane with the largest margin to use in the decision function. Since the goal is to maximize the size of the margin ρ, it is equivalent to minimize

$$\frac{2}{\rho^2} = \frac{1}{2} w \cdot w \tag{2.6}$$

Equations 2.6 and 2.2 can be seen as, respectively, an objective function and constraint in a Quadratic Program (QP). This QP is written in Figure 2.1.

$$\begin{gathered} \min \tfrac{1}{2} w \cdot w \\ y_i(w \cdot x_i + b) \geq 1, \qquad i = 1, \ldots, l \end{gathered}$$

Figure 2.1: The basic Quadratic Program to optimize the hard-margin hyperplane margin.

Note that this QP gives a hard-margin result. That is, it assumes that there are no errors and that the data is completely linearly separable. This is not an easily to solvable Quadratic Program, however. Thus, Cortes and Vapnik present an equivalent QP.

The authors use the Langrangian $L(w, b\alpha)$ and solve for the saddle point to derive the equation

$$W(\alpha) = \sum_{i=1}^{l} a_i - \frac{1}{2} \sum_{i=1}^{l} \sum_{j=1}^{l} \alpha_i \alpha_j y_i y_j x_i x_j \tag{2.7}$$

By finding the saddle point for this Langrangian they also derived the constraint for $W(\alpha)$

$$\sum_{i=1}^{l} y_i \alpha_i = 0 \tag{2.8}$$

Furthermore, it must be true that each Lagrange multiplier is non-negative, so $W(\alpha)$ is also constrained by

$$\alpha_i \geq 0 \qquad i = 1, \ldots, n \tag{2.9}$$

Additionally, solving for the saddle point shows that we can write w as:

$$w = \sum_{i=1}^{l} \alpha_i y_i x_i \tag{2.10}$$

The authors then use the Kuhn-Tucker Theorem to show that the Lagrange multipliers α_i are only non-zero where

$$y_i(x_i \cdot w + b) - 1 = 0$$

Thus, we say a vector x_i is a support vector and $\alpha_i > 0$ if $y_i(w \cdot x_i + b) = 1$.

From the derivation of w in equation 2.10 and our decision function in 2.4, we can see that the only thing that we need to solve for is these support vectors, which define w. We can formulate a Quadratic Program (Figure 2.2) from equations 2.7, 2.8, and 2.9 that solves for the support vectors at our saddle point.

$$\begin{gathered}\max W(\alpha) = \sum_{i=1}^{l} a_i - \frac{1}{2}\sum_{i=1}^{l}\sum_{j=1}^{l} \alpha_i \alpha_j y_i y_j x_i x_j \\ \sum_{i=1}^{l} y_i \alpha_i = 0 \\ \alpha_i \geq 0 \qquad i = 1, \ldots, n\end{gathered}$$

Figure 2.2: The hard-margin Quadratic Program that solves for the support vectors α_i

The authors also show that at the maximum optimal solution $W(\alpha_0)$ to figure 2.2 and minimum optimal solution $\frac{2}{\rho_0^2}$ to the Quadratic Program in figure 2.1, we can use the Kuhn-Tucker theorem to show that:

$$W(\alpha_0) = \frac{2}{\rho_0^2} \tag{2.11}$$

This shows that solving the QP in figure 2.2 to find the support vectors is equivalent to solving the QP in figure 2.1 to find the maximum hyperplane margin. Thus, solving for the support vectors using the QP in figure 2.2 also optimizes the margin, giving us the best possible hyperplane for our linear decision function.

Figure 2.2 gives us a hard-margin classifier. [Vap95] gives us a slight change to this QP that allows for a soft-margin classifier and the use of a kernel function to map to higher dimensions. This new QP is given in Figure 2.3 and is the Quadratic Program that will be used throughout this paper.

Solving the QP in figure 2.3 gives us the support vectors, which the SVM can then use to quickly classify new examples using equation 2.4.

$$\max W(\alpha) = \sum_{i=1}^{l} a_i - \frac{1}{2}\sum_{i=1}^{l}\sum_{j=1}^{l} \alpha_i \alpha_j y_i y_j k(x_i, x_j)$$
$$\sum_{i=1}^{l} y_i \alpha_i = 0$$
$$0 \leq \alpha_i \leq C \qquad i = 1, \ldots, n$$

Figure 2.3: The soft-margin Quadratic Program that solves for the support vectors α_i using a kernel function. Here, C is the maximum penalty for soft classification.

2.2.2 Previous SVM Work

Support Vector Machines have been shown to produce excellent results on classifying tasks. Cortez and Vapnik evaluated the use of SVMs for digit recognition, showing that SVMs can perform comparably with other classical algorithms [CV95]. Additionally, Joachims explored using SVMs for text classification [Joa98b]. He showed significantly better classification success as compared to the Naive Bayes, Rocchio, C4.5 Decision Trees, and K Nearest Neighbor algorithms. These results show that SVMs are a legitimate machine learning classifier and worthy of studying and developing further.

2.3 Sequential Minimal Optimization

The time to train SVMs is dominated by the time it takes to solve the Quadratic program in figure 2.2.1. However, solving Quadratic Programs is computationally expensive. For large amounts of data with many attributes, training of an SVM can be extremely slow. In 1998 John Platt introduced a method for fast training of SVMs called Sequential Minimal Optimization (SMO) [Pla99]. SMO works by breaking the Quadratic Program into a series of the smallest possible subproblems, which it solves analytically rather than through a slower numerical optimization process. For the SVM Quadratic Program, "the smallest possible optimization problem involves two Lagrange multipliers because the Lagrange multipliers must obey a linear equality constraint" [Pla99]. SMO is an iterative algorithm, at each step choosing two Lagrange multipliers and optimizing their values. The details of this optimization process and a derivation can be found in [Pla99]. In choosing the first of the two multipliers to optimize at each iteration, it checks the current value of the multiplier using the Karush-Kuhn-Tucker

(KKT) conditions.

$$\begin{aligned} a_i = 0 &\Rightarrow y_i f(\vec{x}_i) \geq 1, \\ 0 < a_i =< C &\Rightarrow y_i f(\vec{x}_i) = 1, \\ a_i = C &\Rightarrow y_i f(\vec{x}_i) \leq 1, \end{aligned}$$

Figure 2.4: Karush-Kuhn-Tucker (KKT) Conditions. A QP is solved when these conditions are met for all i.

The SMO algorithm iteratively traverses the Lagrange multipliers. If a multiplier is found to violate the KKT conditions, it uses a heuristic designed to speed convergence to choose a second multiplier. It then analytically optimizes the pair. The algorithm continues until the Lagrange multiplier values converge. Throughout this process the constraints of the QP are never violated, ensuring that feasibility is always maintained.

Pseudo-code for the SMO algorithm is given in figure 2.5 [Pla99]. While much of the original pseudo-code is omitted, the main routine is included. Note that the `examineExample(I)` procedure first checks the Lagrange multiplier at index I using the KKT conditions. If it does not violate these conditions, it does nothing and returns a value of 0. If it does violate the KKT conditions, it uses a second procedure that uses a heuristic to choose a second multiplier and then optimizes the pair.

The goal of the SMO algorithm was to achieve fast training times through the solving of many computationally quick subproblems. Platt presented results showing this to be true [Pla99]. They found that the memory footprint of SMO grows linearly and scales well. Thus, it performs well even on large problems. Evaluation showed that SMO, as compared to the common “chunking” approach to SVM training, trained up to 1200 times faster for linear SVMs and up to 15 times faster for non-linear SVMs. The author concludes that these results suggest that SMO should be a common tool for training SVMs. Indeed, this is the case as at least one commonly used SVM tool utilizes a SMO approach [CL11]. These results, as well as the iterative structure of the algorithm, suggest that SMO is a good candidate for utilization in an updatable SVM algorithm.

```
main routine:
     initialize alpha array to all zero
     initialize threshold to zero
     numChanged = 0
     examineAll = 1
     while (numChanged > 0 | examineAll)
         numChanged = 0
         if (examineAll)
             loop I over all training examples
                 numChanged += examineExample(I)
         else
             loop I over examples where alpha is not 0 & not C
                 numChanged += examineExample(I)
         if (examineAll)
             examineAll = false
         else if (numChanged == 0)
             examineAll = true
```

Figure 2.5: Pseudo-code for the main routine of the SMO algorithm. The procedure examineExample(I) is used to check if I satisfies the KKT conditions. If not, it chooses a second Lagrange multiplier and optimizes the pair.

Chapter 3

Methods

3.1 Tools

This work builds on the Cognitive Foundry framework developed at the Sandia National Laboratories [BBD08]. The Cognitive Foundry is a "a unified collection of software tools for Cognitive Science and Technology." This framework proved to be easily usable and adaptable to our research. They had previously implemented the Sequential Minimal Optimization algorithm described in Chapter 2. Furthermore, the Cognitive Foundry provided useful tools for data management as well as testing.

3.2 Updatable SVM

We seek to design an updatable Support Vector Machine, which is an SVM that can, once trained, be updated to reflect the addition of a single training example without having to completely re-train. We contrast this with a "batch" SVM, which requires training on the entire set of examples at one time.

The intuition for creating an updatable SVM lies within the understanding of the Quadratic Programming problem in figure 2.2.1. An SVM must solve this QP problem, therefore finding the optimal solution. Now if we want to modify the QP by adding a new training example, we would not expect this process to be difficult. That is, if we are at an optimal solution for n examples, we would expect that solution to be near-optimal for $n + 1$ examples. Thus, the work to update the SVM for that $n + 1$th example should be small. The goal in designing an updatable SVM is to capitalize on that idea and quickly update an SVM for a new example.

The SMO algorithm lends itself to being restructured. SMO solves the QP by iteratively moving through the Lagrange multipliers (which correspond to training examples), choosing two to optimize at a time. The algorithm traverses the examples in the given order and for each "uses heuristics to choose which two Lagrange multipliers to jointly optimize" [Pla99]. The choice of multipliers is made to speed convergence rather than a necessary part of finding the optimal solution. Thus, the order of traversal and the choosing of multiplier pairs can be manipulated without hindering convergence (although we can expect an increased time to convergence).

A new example can be viewed, therefore, as a Lagrange multiplier with a value of zero that has yet to be optimized with any other examples. This means that the new example is not being considered a support vector. However, this new multiplier can be optimized in order to further improve the solution to the QP problem. Thus, we can begin optimizing for these $n+1$ examples from the optimal solution for n examples, a significantly smaller problem than optimizing for all $n+1$ examples at one time. In doing so, the cost for adding the new example is reduced to the time it takes to make this small optimization.

The method for updating our SVM is described as follows. First, the Lagrange multiplier for the new example is initialized to zero. By initializing it to zero, we ensure that our QP constraints will not be violated by the addition of this example. The Karush-Kuhn-Tucker conditions are then checked for this example in order to see if this new example's Lagrange multiplier needs optimizing. If not, the SVM does not need any updating and the new example is not considered a support vector. If the KKT conditions are not met, however, another Lagrange multiplier is chosen and the pair is optimized. Then the process of iteratively optimizing pairs of Lagrange multipliers is begun again. This process stops once the KKT conditions have been satisfied for each training example, meaning that an optimal solution has been reached.

This update algorithm is described in greater detail using pseudo-code in figure 3.1. This update function relies on the design of the SMO algorithm and is essentially a re-write of the main procedure for SMO. The difference is that only a single example is being added. Thus, if there are no changes after calling `examineExample(i)`, then the SVM is already optimal and no changes are needed. This is significant, because it means that adding new training data comes at essentially no cost for these cases. However, if it does make a change to the new multiplier, then it is uncertain if the rest of the multipliers are optimized correctly. Thus, the process of iteratively optimizing Lagrange multipliers is started up again and continues until convergence.

By implementing this new update method, we have made our batch SVM into an updatable SVM.

3.3 Anytime Learning

By creating an updatable SVM, there is now the means to make an SVM that is an anytime learner. The anytime SVM can be parameterized by n, the number of training examples to use. That is, we can make an anytime SVM learner where the constrained resource would be the number of training examples.

It is trivial to design a contract SVM algorithm that is constrained by the number of training examples n. This is because it can simply randomly pick n of the total possible training examples and train on them using our normal batch method. This problem is therefore rather trivial.

The update algorithm allows for the design of an interruptible anytime SVM. The interruptible algorithm can utilize the update function to iteratively update the SVM with new examples until either it runs out of training data or is interrupted, upon which it returns the resulting learner. Some pseudo-code showing how one might utilize an updatable SVM to design an interruptible anytime SVM algorithm is shown in Figure 3.2.

It is important to note that the interruptible anytime SVM algorithm cannot run indefinitely and continue to improve. This is because of the nature of the SMO algorithm in that it optimizes the Quadratic Program given in Figure 2.3. Once the algorithm runs out of examples to train on, its solution can no longer be optimized. Thus, we cannot expect continued improvement of the interruptible anytime SVM once all the examples have been used in training.

```
@precondition: The SVM has already been trained initially
fun updateSVM(Example e)
     Add e to Data List
     i = index of e
     alpha_i = 0
     add alpha_i to alpha array
     changed = examineExample(i)
     if (changed == 1)
         numChanged = 0
         examineAll = true
         while (numChanged > 0 | examineAll)
             numChanged = 0
             if (examineAll)
                 loop I over all training examples
                     numChanged += examineExample(I)
             else
                 loop I over examples where alpha is not 0 & not C
                     numChanged += examineExample(I)
             if (examineAll)
                 examineAll = false
             else if (numChanged == 0)
                 examineAll = true
```

Figure 3.1: Pseudo-code for updating the SVM using a modified SMO algorithm. The method examineExample(I) is a method of the SMO algorithm used to check if I satisfies the KKT conditions. If not, it chooses a second Lagrange multiplier and optimizes the pair.

```
fun interruptibleSVM(data)
     learner = new Updatable SVM Learner
     while (not interrupted and data is not empty)
         nextData = data.pop()
         learner.update(nextData)
return learner
```

Figure 3.2: Pseudo-code for an interruptible anytime SVM

Chapter 4

Evaluation

4.1 Goals and Methods

Our goals in making an updatable SVM algorithm are twofold. The first goal is to achieve significant time savings by updating rather than re-training. That is, we want to show that updating an SVM with a new example is significantly faster than completely re-training using both the old and new data. The belief here is that if the original SVM is already optimal, then it should be possible to quickly optimize our old solution to account for the new data. Thus, updating an SVM should come at a low time cost.

The second goal is to achieve rates of success comparable to the batch approach. Specifically, we hope that training on data by continually using the update method would not be sacrificing any success that could be achieved through the batch method.

Based on these two goals, two different ways of measuring our success were used. In order to perform evaluation, two different SVMs were trained and tested by moving through the training data example by example. For the first SVM, an updatable SVM, the SVM is updated with the new example and the time it takes to do so is recorded. The second SVM, a batch SVM, trains on all the data seen so far. Then, the testing data is used to evaluate the accuracy of each of the SVMs at the current training example. For example, when it comes to the 50th training example, the first SVM is updated with it and the second SVM is trained using all 50 example. The time it takes to update and re-train are measured and recorded. The accuracy of each SVM is calculated using all of the testing data results are stored. It then moves on to the 51st training example and repeats the process.

We can expect certain results from our evaluation. In terms of time, we

would expect the SVM to take longer to re-train for each new data point. That is, training for the first $n+1$ training examples takes longer than training on the first n examples. We would expect an increase in time for both the batch and updatable SVMs, and we would hope to see a
significantly smaller time increase for the updatable SVM. Similarly, we would expect both the batch and updatable SVM to have increasing accuracy as more training examples are added and our SVMs become better fit to the data. Here, we would like to see the accuracy at each training example to be comparable for the batch and updatable SVM.

4.2 Measuring Success

The chosen measure of success or accuracy in terms of classifying examples correctly is the F1 Measure. F1 Measure has been shown to be a good measure of success for machine learning classifiers, as simple *accuracy* is not a good performance measure in cases of unbalanced class sizes [Joa98b]. The F1 Measure is on a scale of 0 to 1, with 1 being the most successful. F1 measure is defined as:

$$F1 = \frac{2(P * R)}{P + R} \tag{4.1}$$

Figure 4.1: The mathematical definition for F1 Measure. Here, P is the Precision of the learner and R is the recall.

4.3 Data

All evaluation was performed using data sets from the UCI Machine Learning Repository [FA10], a common source of data for Machine Learning tasks. Several data sets that differed in number of examples as well as number of attributes and area of application were used. Specifically, the *Iris*, *Wine*, and *Ionosphere* data sets were used for evaluation. Details for these data sets can be found in Table 4.1.

Name	# Instances	# Attributes	Attribute Type	Area
Iris	150	4	Real	Life
Wine	178	13	Real, Integer	Physical
Ionosphere	351	34	Real, Integer	Physical

Table 4.1: Basic Information on the Data Sets Used in Testing

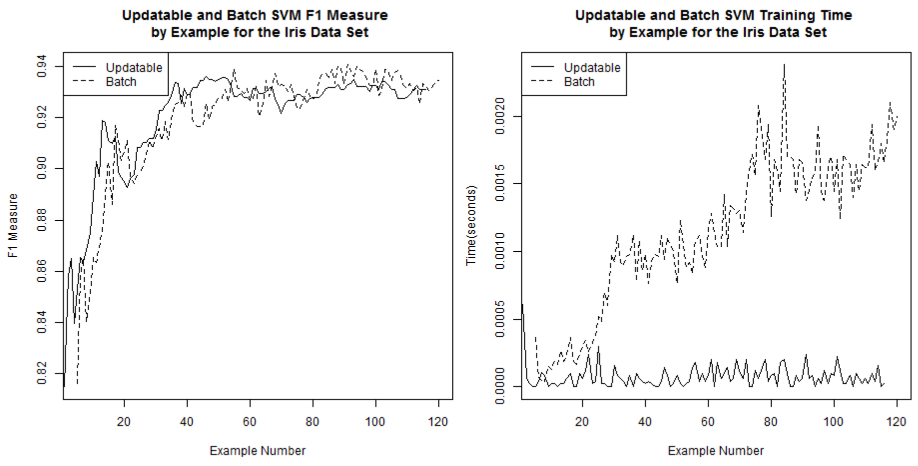
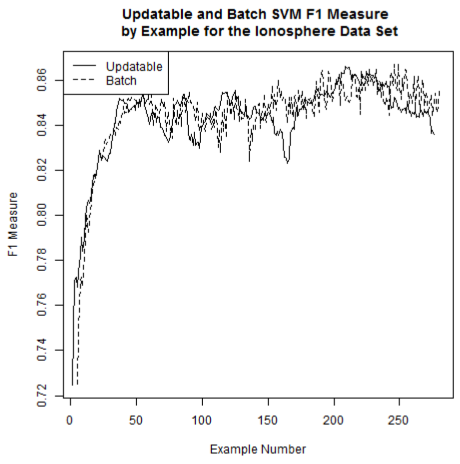


Figure 4.2: Evaluation Results showing training time and F1 Measure for each new training example for batch and updatable SVMs on the Iris data set.

4.4 Results

Evaluation was performed using 10 trials of 5-fold cross-validation. As described in section 4.1, training time and F1 measure were determined for both the updatable and batch SVMs after each additional training example was added. Thus, for each example in the training set, we present an average of 50 training times and F1 measures where have trained on a random 80% of the training data and tested on the remainder.

Figure 4.2 shows evaluation results on the Iris data set. The graph on the left shows improvement in the F1 Measure as new training examples are added for both the batch and updatable SVM, as we expected. We can also see that the results for each are comparable. That is, at any given training example addition, the F1 measures are close. For some examples



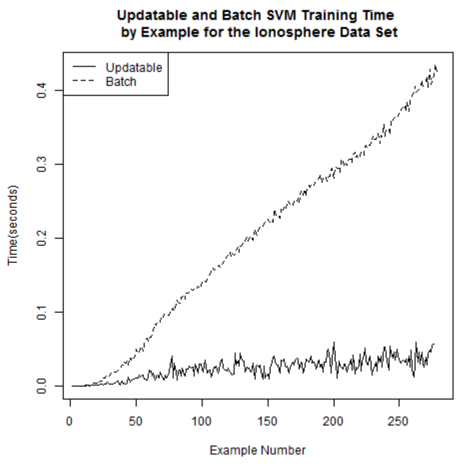


Figure 4.3: Evaluation Results showing training time and F1 Measure for each new training example for batch and updatable SVMs on the Ionosphere data set.

the updatable SVM outperforms the batch SVM, and for others the batch performs better.

The graph on the right shows the training time as each new example is added using the updatable and batch SVMs. The time savings for using the updatable SVM at any given point can be calculated by subtracting the training time for the updatable SVM from the batch SVM training time. For example, the time savings for using an updatable SVM when adding the 120th training example is $0.002 - 0.00002 = .00198$, which is a savings of 99%. This is an extremely significant time savings, and our first graph shows that it comes at no cost in terms of F1 measure.

Figure 4.3 shows evaluation results on the Ionosphere data set. Again, the graph on the left shows comparable testing results after the addition of each new training example. Additionally, the graph on the right shows significant time savings by using the updatable SVM. For example, the time savings for using the updatable SVM for the 280th training example is $0.426 - 0.057 = 0.369$, which is a savings of 86.6%. Once again, we see that the updatable SVM brings significant time savings without an expected cost in terms of F1 Measure.

Figure 4.4 shows evaluation results on the Wine data set. Here, our graph of F1 Measures actually shows that the updatable SVM significantly

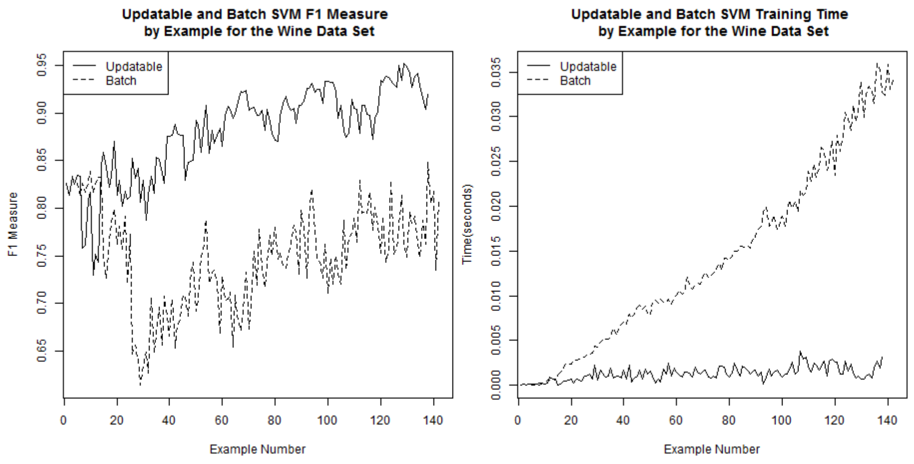


Figure 4.4: Evaluation Results showing training time and F1 Measure for each new training example for batch and updatable SVMs on the Wine data set.

outperforms the batch SVM. This is an interesting result, and is discussed more in section 4.5. Our graph of training times again shows significant savings for using the updatable SVM. For our 143rd training example, we have a time savings of, $0.0341 - 0.0031 = 0.0314$, which is a savings of 91%.

Additionally, our evaluation shows that the rate of increase in training time for the batch SVM is significantly larger than for the updatable SVM for each of our data sets. Thus, as shown in figure 4.5, the time savings for using an updatable SVM in terms of the percentage of batch training time will grow larger as the size of the training data increases. From this we suggest that the time savings for using an updatable SVM is largest when there are large amounts of training data. A possible explanation for this phenomena lies in the structure of the update method. If the new example does not violate the KKT conditions, then the current model is already optimal and no changes are needed. As the SVM becomes better fit to the data, it is increasingly likely that new data points would not change the model. Thus, the probability that updating comes at no cost increases as the size of the training data increases.

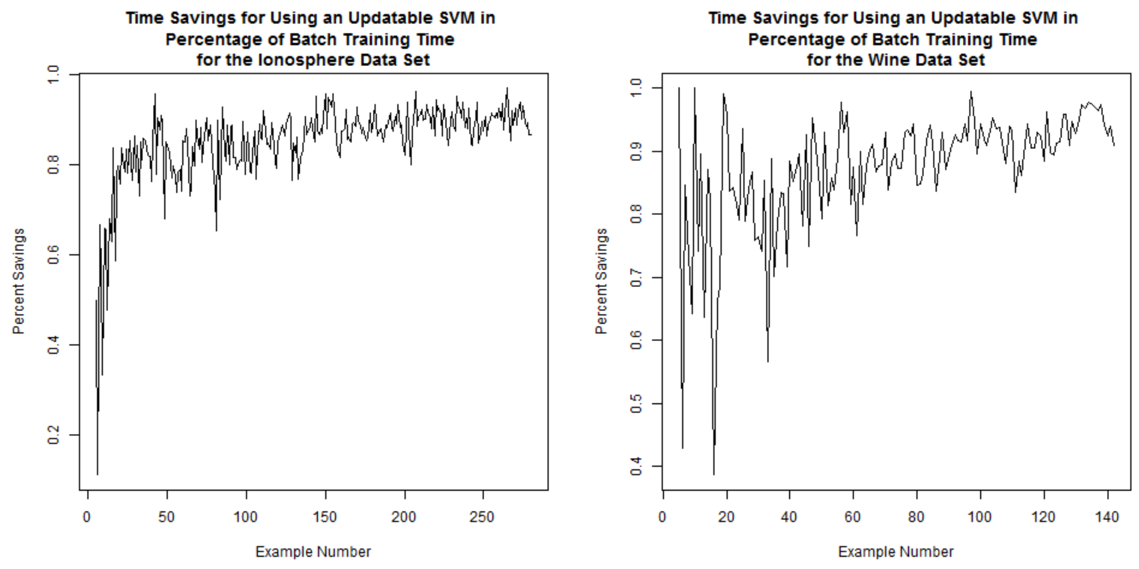


Figure 4.5: Graphs showing the time savings for using an updatable over a batch SVM in terms of the percentage of the batch training time for the Ionosphere and Wine data sets.

4.5 Discussion

It is of importance to note that we do not achieve precisely the same results using a batch and updatable SVM. This is important because while there may be significant time savings when using an updatable SVM, the time savings may be overshadowed by a significant drop in accuracy. In real world situations, the trade-off between time and accuracy may not be acceptable. Thus, we seek to explain these differences and explore their consequences.

Fundamentally, the SMO algorithm is an approximation of the Quadratic Programming problem from figure **??**. Our updatable SMO algorithm is also an approximation of the QP problem. The approximation is different because of how the update method manipulates the traversal of the data. It forces a different order of optimization, which can lead to a different result. Ultimately, what they're trying to find is the maximal value of the objective function while meeting the conditions of the constraints. Both the batch and updatable SMO algorithm ensure that the constraints are satisfied. Therefore, the differences in classification results come from a difference in objective function values. If the two approaches gave almost

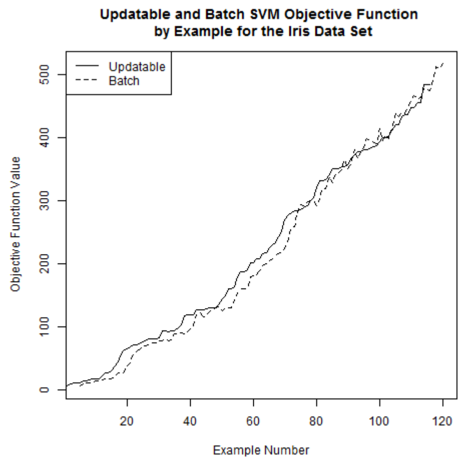
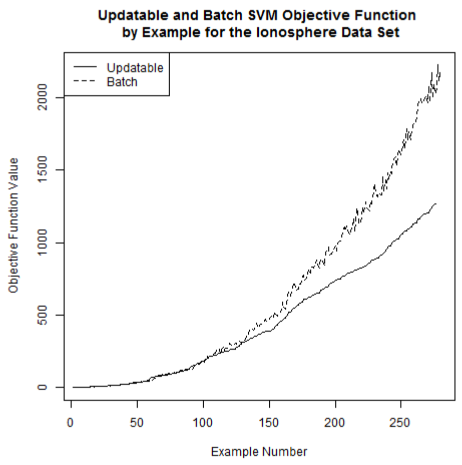


Figure 4.6: Graph showing the change in objective function values for the updatable and batch SVMs as each training example is added for the Iris data sets. Note that they are always close, showing that the two methods produce very similar results.

equivalent results, we would expect a graph of the objective function values to look like that for the Iris data set, shown in figure 4.6. However, this is not always the result. Figure 4.7 shows the changes in objective function values for the Ionosphere and Wine data sets.

Here, there is a divergence in our objective function values, with the batch SMO achieving better results for Ionosphere and the updatable SMO performing better for the wine data set. These results are significant in that they are inconclusive. It is clear that both the updatable and batch approaches perform well and see an increase in accuracy as examples are added, as would be expected. However, it cannot be concluded that the methods produce equivalent objective function values or that one is better than the other. On the other hand, the graphs in figures 4.2, 4.4, and 4.4 show that end results (as classification success) are similar. Thus, we conclude that the updatable SVM algorithm achieves the goals of time savings and comparable accuracy to the batch SVM.



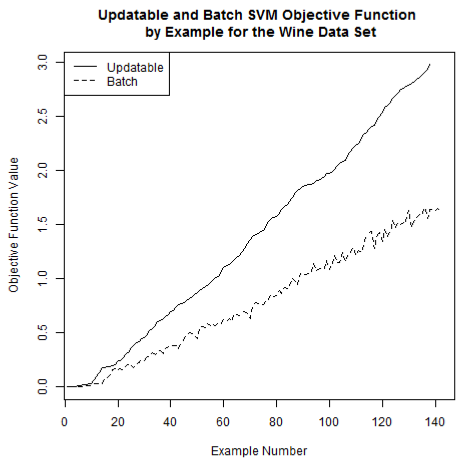


Figure 4.7: Graphs showing the change in objective function values for the updatable and batch SVMs as each training example is added for the Ionosphere and Wine data sets. Note the divergence in objective function values for each.

Chapter 5

Conclusion

Results such as those in figures 4.2, 4.3 and 4.4 show that the updatable SVM method presented here is successful at meeting the goals in section 4.1. The updatable SVM does indeed achieve comparable results to the batch SVM method (although sometimes slightly better or worse). Most importantly, however, it has been shown that the updatable SVM presents a significant time savings as compared to the batch approach. Additionally, the time savings increases for larger, more computationally expensive problems.

5.1 Contributions

An updatable SVM that performs well and comes at little time cost could be useful in real-world situations where the flow of data is continuous. Consider again the example of a medical software system from Chapter 1. An SVM that can be updated in the time frames demonstrated here could initially be trained on some labeled diagnoses from doctors. The software using this SVM could then be deployed. Since the SVM is updatable, the software provider could then continually improve the SVM as new data is added. Doing so would come at low cost and would create a streamlined process where new data is almost immediately being put into use for new diagnoses using the software.

There are many real-world situations where there is a constant influx of data and a desire to account for that, as shown in the example of the medical software. The updatable SVM method presented here has the potential to make a significant impact in areas such as these.

5.2 Future Work

There are ways in which updatable SVMs could be explored further. First, the unpredictability of the success of the updatable SVM as compared to the batch SVM could potentially be of concern in situations where accuracy cannot be greatly sacrificed for the sake of time savings. Thus, it would be worthwhile to demonstrate the range of variance for the difference between the objective function values for an updatable SVM and a batch SVM. This would require a deep exploration of the mathematics behind SVMs and the SMO algorithm and was beyond the scope of this research.

It may also be worthwhile to explore possible alternatives for an updatable SVM algorithm. SMO is just one of several SVM learning algorithms. A reasonable place to begin this work would be with other existing SVM algorithms. It would be interesting to compare the time savings and accuracy for the approach presented here with other updatable SVM algorithms.

Another path for future work would be further exploration of the fields of use for an updatable SVM. Evaluation could be performed using new data types, such as categorical data, as well as larger data sets. Furthermore, it would be useful to explore the use of an updatable SVM in a real-world situation to evaluate the practicality of using an updatable SVM.

Finally, it would be interesting to compare results from updatable learning of SVMs with updatable versions of other machine learning algorithms. SVMs have been shown to perform comparably and sometimes better than other supervised machine learning algorithms. However, it is unknown how the updatable SVM algorithm would compare to other updatable learners in terms of time and accuracy.

Bibliography

[BBD08] Justin Basilco, Zachary Benz, and Kevin R. Dixon. The cognitive foundry: A flexible platform for intelligent agent modeling. In *Proceedings of the 2008 Behavior Representation in Modeling and Simulation (BRIMS) Conference*, 2008.

[BD94] Mark Boddy and Thomas L Dean. Deliberation scheduling for problem solving in time-constrained environments. *Artificial Intelligence*, 67(2):245 – 285, 1994.

[CL11] Chih-Chung Chang and Chih-Jen Lin. Libsvm: A library for support vector machines. *ACM Trans. Intell. Syst. Technol.*, 2(3):27:1–27:27, May 2011.

[CV95] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20:273–297, 1995.

[EM07] Saher Esmeir and Shaul Markovitch. Anytime learning of decision trees. *The Journal of Machine Learning Research*, 8:891–933, 2007.

[FA10] A. Frank and A. Asuncion. UCI machine learning repository, 2010.

[Joa98a] T. Joachims. Making large-scale svm learning practical. LS8-Report 24, Universität Dortmund, LS VIII-Report, 1998.

[Joa98b] T. Joachims. Text categorization with support vector machines: Learning with many relevant features. In *European Conference on Machine Learning (ECML)*, pages 137–142, Berlin, 1998. Springer.

[Pla99] John C. Platt. Fast training of support vector machines using sequential minimal optimization. In *Advances in Kernel Methods - Support Vector Learning*, pages 185–208. MIT Press, 1999.

[Vap95] Vladmir Vapnik. *The Nature of Statistical Learning Theory.* Springer Verlag, 1995.

[ZR96] Shlomo Zilberstein and Stuart J. Russell. Optimal composition of real-time systems. *Artificial Intelligence*, 82(1-2):181–213, 1996.