Online (and Offline) on an Even Tighter Budget

Jason Weston
NEC Laboratories
America,
Princeton,
NJ, USA
jasonw@nec-labs.com

Antoine Bordes
NEC Laboratories
America,
Princeton,
NJ, USA
antoine@nec-labs.com

Leon Bottou
NEC Laboratories
America,
Princeton,
NJ, USA
leonb@nec-labs.com

Abstract

We develop a fast online kernel algorithm for classification which can be viewed as an improvement over the one suggested by (Crammer, Kandola and Singer, 2004), titled "Online Classificaton on a Budget". In that previous work, the authors introduced an *on-the-fly* compression of the number of examples used in the prediction function using the size of the margin as a quality measure. Although displaying impressive results on relatively noise-free data we show how their algorithm is susceptible in noisy problems. Utilizing a new quality measure for an included example, namely the error induced on a selected subset of the training data, we gain improved compression rates and insensitivity to noise over the existing approach. Our method is also extendable to the batch training mode case.

1 Introduction

Rosenblatt's Perceptron (Rosenblatt, 1957) efficiently constructs a hyperplane separating labeled examples $(x_i, y_i) \in \mathbb{R}^n \times \{-1, +1\}$. Memory requirements are minimal because the Perceptron is an *online* algorithm: each iteration considers a single example and updates a candidate hyperplane accordingly. Yet it globally converges to a separating hyperplane if such a hyperplane exists.

The Perceptron returns an arbitrary separating hyperplane regardless of the minimal distance, or *margin*, between the hyperplane and the examples. In contrast, the Generalized Portrait algorithm (Vapnik and Lerner, 1963) explictly seeks an hyperplane with maximal margins.

All the above methods produce a hyperplane whose normal vector is expressed as a linear combination of examples. Both training and recognition can be carried out with the only knowledge of the dot products $x_i'x_j$ between examples. Support Vector Machines (Boser, Guyon and Vapnik, 1992) produce maximum margin non-linear separating hypersurfaces by simply replacing the dot products by a Mercer kernel $K(x_i, x_j)$.

Neither the Generalized Portrait nor the Support Vector Machines (SVM) are online algorithms. A set of training examples must be gathered (and stored in memory) prior to running the algorithm. Several authors have proposed online Perceptron variants that feature both the margin and kernel properties. Example of such algorithms include the Relaxed Online Maximum Margin Algorithm (ROMMA) (Li and Long, 2002), the Approximate Maximal Margin Classification Algorithms (ALMA) (Gentile, 2001), and the Margin Infused Relaxed Algorihm (MIRA) (Crammer and Singer, 2003).

The computational requirements[1] of kernel algorithms are closely related to the *sparsity* of the linear combination defining the separating hyper-surface. Each iteration of most Perceptron variants considers a single example and decides whether to insert it into the linear combination. The Budget Perceptron (Crammer, Kandola and Singer, 2004) achieves greater sparsity by also trying to remove some of the examples already present in the linear combination.

This discussion only applies to the case where all examples can be separated by a hyperplane or a hypersurface, that is to say in the absence of noise. Support Vector Machines use Soft Margins (Cortes and Vapnik, 1995) to handle noisy examples at the expense of sparsity. Even in the case where the training examples can be separated, using Soft Margins often improves the test error. Noisy data sharply degrades the performance of all the Perceptron variants discussed above.

We propose a variant of the Perceptron algorithm that addresses this problem by removing examples from the linear combination on the basis of a direct measurement of the training error in the spirit of Kernel Matching Pursuit (KMP) (Vincent and Bengio, 2000). We show that this algorithm has good performance on both noisy and non-noisy data.

[1]and sometimes the generalization properties

2 Learning on a Budget

Figure 1 shows the Budget Perceptron algorithm (Crammer, Kandola and Singer, 2004). Like Rosenblatt's Perceptron, this algorithm ensures that the hyperplane normal $\mathbf{w}_t$ can always be expressed as a linear combination of the examples in set C_t:

$$\mathbf{w}_t = \sum_{i \in C_t} \alpha_i \mathbf{x}_i. \quad (1)$$

Whereas Rosenblatt's Perceptron updates the hyperplane normal w_t whenever the current example $(\mathbf{x}_t, y_t)$ is misclassified, the Budget Perceptron updates the normal whenever the margin is smaller than a predefined parameter $\beta > 0$, that is to say whenever $y_t(\mathbf{x}_t \cdot \mathbf{w}_t) < \beta$.

Choosing a large β ensures that the hyperplane will eventually become close to the maximal margin hyperplane. This also increases the likelihood that an arbitrary example will become part of the expansion (1) and make the final solution less sparse.

The Budget Perceptron addresses this problem with a *removal* process. Whenever the number of expansion examples exceeds a predefined threshold p, the removal process excludes one example from the expansion. More specifically, the removal process (steps $1a$–$1c$, figure 1) simulates the removal of each example and eventually selects the example i that, when removed, remains recognized with the largest margin:

$$i = \arg\max_{j \in C_t} \{ y_j(\mathbf{w}_{t-1} - \alpha_j y_j \mathbf{x}_j) \cdot \mathbf{x}_j \}$$

The justification for such a strategy is that the Perceptron algorithm only adds examples to the cache when they are errors. Early on in the training, examples may be added because the decision rule learnt thus far is relatively inaccurate, however later on these examples may be well classified as the direction of the hyperplane has changed considerably. The standard Perceptron algorithm does not have any removal procedure.

Several variants of this algorithms can be derived by changing the update formula (figure 2) or by replacing the dot products by suitable kernel functions. The maximum size of the expansion can be fixed or variable (Crammer, Kandola and Singer, 2004). Essentially, to adapt to the variable case one removes all examples that violate $y_j(w_{t-1} - \alpha_j y j x_j) \cdot x_j < \beta$ on each iteration. For simplicity however, in the remainder of the paper we will present algorithms in the simplest linear setup with Perceptron update and fixed sized cache, and leave such variants to the reader.

Experimental results (Crammer, Kandola and Singer, 2004) demonstrate that the Budget Perceptron performs extremely well on relatively noiseless problems. However, it degrades quickly on noisy problems. Suppose for instance that we randomly flip the labels of a small proportion η of both the training and test examples. The misclassification rate of the best hyperplane is at least η. Such misclassified examples accumulate into the Budget Perceptron expansion because only examples which are *classified well* are removed. Mislabeled examples reverse the direction of the normal w_t, and poor performance follows.

```
**Input**: Margin $\beta > 0$, Cache Limit $p$.

**Initialize**: Set $\forall t\ \alpha_t = 0$, $\mathbf{w}_0 = 0$, $C_0 = \emptyset$

**Loop**: For $t = 1, \ldots, T$
  – Get a new instance $\mathbf{x}_t \in \mathbb{R}^n$, $y_t = \pm 1$.
  – Predict $\hat{y}_t = \text{sign}(y_t(\mathbf{x}_t \cdot \mathbf{w}_{t-1}))$
  – If $y_t(\mathbf{x}_t \cdot \mathbf{w}_{t-1}) \leq \beta$ update:
    1. If $|C_t| = p$ remove one example:
       a Find $i = \arg\max_{j \in C_t} \{ y_j(\mathbf{w}_{t-1} - \alpha_j y_j \mathbf{x}_j) \cdot \mathbf{x}_j \}$
       b Update $\mathbf{w}_{t-1} \leftarrow \mathbf{w}_{t-1} - \alpha_i y_i \mathbf{x}_i$.
       c Remove $C_{t-1} \leftarrow C_{t-1} \setminus \{i\}$.
    2. Insert $C_t \leftarrow C_{t-1} \cup \{t\}$.
    3. Set $\alpha_t = 1$.
    4. Compute $\mathbf{w}_t \leftarrow \mathbf{w}_{t-1} + \alpha_t y_t \mathbf{x}_t$.

**Output**: $H(x) = \text{sign}(\mathbf{w}_T \cdot \mathbf{x})$.
```

Figure 1: The Budget Perceptron algorithm (Crammer, Kandola and Singer, 2004).

Complexity Assuming we use an RBF kernel, the insertion step requires $O(pn)$ operations where p is the cache size and n is the input dimensionality. The deletion step requires $O(p)$ operations, assuming all kernel calculations are cached. Note that the latter cost is only incurred for margin errors when the cache is full.

Perceptron (Rosenblatt, 1957)

$$\alpha_t = 1$$

MIRA (Crammer and Singer, 2003)

$$\alpha_t = \min\left(1, \max\left(0, \frac{-y_i(\mathbf{w} \cdot \mathbf{x}_i)}{\mathbf{x}_i \cdot \mathbf{x}_i}\right)\right)$$

No-Bias-SVM (Kecman, Vogt and Huang, 2003) $\beta = 1$

$$\alpha_t = \min\left(C, \max\left(0, \frac{1 - y_i(\mathbf{w} \cdot \mathbf{x}_i)}{\mathbf{x}_i \cdot \mathbf{x}_i}\right)\right)$$

Figure 2: **Update Rules for Various Algorithms.** These can be used to replace step 3 in figure 1 or 3.

3 Learning on a Tighter Budget

The Budget Perceptron removal process simulates the removal of each example and eventually selects the example that remains recognized with the largest margin. This margin can be viewed as an indirect estimate of the impact of the removal on the overall performance of the hyperplane. Thus, to improve the Budget algorithm we propose to replace this indirect estimate by a direct evaluation of the misclassification rate. We term this algorithm the Tighter Budget Perceptron. The idea is simply to replace the measure of margin

$$i = \arg\max_{j \in C_t} \{ y_j (\mathbf{w}_{t-1} - \alpha_j y_j \mathbf{x}_j) \cdot \mathbf{x}_j \} \qquad (2)$$

with the overall error rate on all currently seen examples:

$$i = \arg\min_{j \in C_t} \{ \sum_{k=1}^{t} L(y_k, \text{sign}((\mathbf{w}_{t-1} - \alpha_j y_j \mathbf{x}_j) \cdot \mathbf{x}_k) \}.$$

Intuitively, if an example is well classified (has a large margin) then not only will it be correctly classified when it is removed as in equation (2) but also all other examples will still be well classified as well. On the other hand, if an example is an outlier then its contribution to the expansion of w is likely to classify points incorrectly. Therefore when removing an example from the cache one is likely to either remove noise points or well-classified points first. Apart from when the kernel matrix has very low rank we do indeed observe this behaviour, e.g. in figure 8.

Compared to the original Budget Perceptron, this removal rule is more expensive to compute, now it requires $O(t(p + n))$ operations (see section 2). Therefore in section 3.2 we discuss ways of approximating this computation whilst still retaining its desirable properties. First, however we discuss the relationship between this algorithm and existing approaches.

```
Input: Margin $\beta > 0$, Cache Limit $p$.

Initialize: Set $\forall t\ \alpha_t = 0$, $\mathbf{w}_0 = 0$, $C_0 = \emptyset$.

Loop: For $t = 1, \ldots, T$
  – Get a new instance $\mathbf{x}_t \in \mathbb{R}^n$, $y_t = \pm 1$.
  – Predict $\hat{y}_t = \text{sign}(y_t(\mathbf{x}_t \cdot \mathbf{w}_{t-1}))$.
  – Get a new label $y_t$.
  – If $y_t(\mathbf{x}_t \cdot \mathbf{w}_{t-1}) \le \beta$ update:
    1. If $|C_t| = p$ remove one example:
       a Find $i = \arg\min_{j \in C_t} \{\sum_{k=1}^{t} L(y_k, \text{sign}((\mathbf{w}_{t-1} - \alpha_j y_j \mathbf{x}_j) \cdot \mathbf{x}_k)\}$.
       b Update $\mathbf{w}_{t-1} \leftarrow \mathbf{w}_{t-1} - \alpha_i y_i \mathbf{x}_i$
       c Remove $C_{t-1} \leftarrow C_{t-1} \setminus \{i\}$.
    2. Insert $C_t \leftarrow C_{t-1} \cup \{t\}$.
    3. Set $\alpha_t = 1$.
    4. Compute $\mathbf{w}_t \leftarrow \mathbf{w}_{t-1} + \alpha_t y_t \mathbf{x}_t$.

Output: $H(x) = \text{sign}(\mathbf{w}_T \cdot \mathbf{x})$.
```

Figure 3: The Tighter Budget Perceptron algorithm.

3.1 Relation to Other Algorithms

Kernel Matching Pursuit The idea of kernel matching pursuit (KMP) (Vincent and Bengio, 2000) is to build a predictor $w = \sum_i \alpha_i x_i$ greedily by adding one example at a time, until a pre-chosen cache size p is found. The example to add is chosen by searching for the example which gives the largest decrease in error rate, usually in terms of squared loss, but other choices of loss function are possible. While this procedure is for *batch* learning, and not *online* learning, clearly this criteria for *addition* is the same as our criteria for *deletion*.

There are various variants of KMP, two of them called basic- and backfitting- are described in figure 4. Basic adapts only a single α_i in the insertion step, whereas backfitting adjusts all α_i of previously chosen points. The latter can be computed efficiently if the kernel matrix can be fit into memory (the algorithm is given in (Vincent and Bengio, 2000)), but is expensive for large datasets. The basic algorithm, on the other hand does not perform well for classification, as shown in figure 8.

Note that we could adapt our algorithm's addition step to also be based on training error. However, using the Perceptron rule, an example is only added to the cache if it is an error, making it more efficient to compute. Note that variants of KMP have also been introduced that incorporate a deletion as well as an insertion step (Nair, Choudhury and Keane, 2002).

Condense and Multi-edit Condense and multi-edit (Devijver and Kittler, 1982) are editing algorithms to "sparsify" k-NN. Condense removes examples that are far from the decision boundary. The Perceptron and the SVM already have their own "condense" step as such points typically have $\alpha_i = 0$. The Budget Perceptron is an attempt to make the condense step of the Perceptron more aggressive. Multi-edit attempts to remove all the examples that are on the wrong side of the Bayes decision boundary. One is then left with learning a decision rule with non-overlapping classes with the same Bayes decision boundary as before, but with Bayes risk equal to zero. Note that neither the Perceptron nor the SVM (with soft margin) perform this step[2], and all incorrectly classified examples become support vec-

[2]An algorithm designed to combine the multi-edit step into SVMs is developed in (Bakır, Bottou and Weston, 2004).

```
**Input**: Cache Limit $p$.

**Initialize**: Set $\forall t\ \alpha_t = 0$, $\mathbf{w}_0 = 0$, $C_0 = \emptyset$.

**Loop**: For $t = 1, \ldots, p$
  – Choose $(k, \alpha) = \arg\min_{\alpha,\ j=1\ldots m} \sum_{i=1}^{m} (y_j - (\mathbf{w}_t + \alpha \mathbf{x}_j)\cdot x_i)^2$.
  – Insert $C_t \leftarrow C_{t-1} \cup \{t\}$.
  – *Basic-KMP:*
    Set $\mathbf{w}_t \leftarrow \mathbf{w}_{t-1} + \alpha \mathbf{x}_k$.
    *Backfitting-KMP:*
    Set $\mathbf{w}_t \leftarrow \sum_{i \in C_t} \alpha_i \mathbf{x}_i$ where $\{\alpha_i\} = \arg\min_{\{\alpha_i\}} \sum_{j=1}^{m} \left( y_j - \sum_{i \in C_t} \alpha_i \mathbf{x}_i \cdot \mathbf{x}_j \right)^2$

**Output**: $H(x) = \text{sign}(\mathbf{w}_T \cdot \mathbf{x})$.
```

Figure 4: The Basic and Backfitting Kernel Matching Pursuit (KMP) Algorithms (Vincent and Bengio, 2000).

tors with $\alpha_i > 0$. Combining condense and multi-edit together one only tries to keep the correctly classified examples close to the decision boundary. The Tighter Budget Perceptron is also an approximate way of trying to achieve these two goals, as previously discussed.

Regularization One could also view the Tighter Budget Perceptron as an approximation of minimizing a regularized objective of the form

$$\frac{1}{m} \sum L(y_i, f(x_i)) + \gamma ||\alpha||_0.$$

where operator $||\cdot||_0$ is defined as counting the number of nonzero coefficients. That is to say, the fixed sized cache chosen acts a regularizer to reduce the capacity of the set of functions implementable by the Perceptron rule, the goal of which is to minimize the classification loss. This means that for noisy problems, with a reduced cache size one should see improved generalization error compared to a standard Perceptron using the Tighter Budget Perceptron, and we indeed find experimentally that this is the case.

3.2 Making the per-time-step complexity bounded by a constant independent of t

An important requirement of online algorithms is that their per-time-step complexity should be bounded by a constant independent of t (t being the time-step index), for it is assumed that samples arrive at a constant rate. The algorithm in figure 3 grows linearly in the time, t, because of the computation in step 1(a), that is when we choose the example

```
**Input:** $Q_{t-1}, \mathbf{x}_t, \mathbf{s}$
  – $Q_t \leftarrow Q_{t-1} \cup \mathbf{x}_t$.
  – If $|Q_t| > q$
      1. $i = \arg\max_{i \in Q_{t-1}} \mathbf{s}_i$
      2. $Q_t \leftarrow Q_t \setminus \mathbf{x}_i$

**Output**: $Q_t$
```

Figure 5: Algorithm for maintaining a fixed cache size q of relevant examples for estimating the training error. The idea is to maintain a count s_i of the number of times the prediction changes label for example i. One then retains the examples which change labels most often.

in the cache to delete which results in the minimal loss over all t observations:

$$i = \arg\min_{j \in C_t} \{ \sum_{k=1}^{t} L(y_k, \text{sign}((\mathbf{w}_{t-1} - \alpha_j y_j \mathbf{x}_j) \cdot \mathbf{x}_k) \}.$$

(Note that this extra computational expense is only invoked when $\mathbf{x}_t$ is a margin error, which if the problem has a low error rate, is only on a small fraction of the iterations.) Nevertheless, it is possible to consider approximations to this equation to make the algorithm independent of t.

We could simply reduce the measure of loss to only the fixed p examples in the cache:

$$i = \arg\min_{j \in C_t} \{ \sum_{k \in C_t} L(y_k, \text{sign}((\mathbf{w}_{t-1} - \alpha_j y_j \mathbf{x}_j) \cdot \mathbf{x}_k) \}. \tag{3}$$

While this is faster to compute, it may be suboptimal as we wish to have an estimator of the loss that is as unbiased as possible, and the points that are selected in the cache are a biased sample. However, they do have the advantage that many of them may be close to the decision surface.

A more unbiased sample could be chosen simply by picking a fixed number of randomly chosen examples, say q examples, where we choose q in advance. We define this subset as Q_t, where $|Q_t| = \min(q, t)$ which is taken from the t available examples until the cache is filled. Then we compute:

$$i = \arg\min_{j \in C_t} \{ \sum_{k \in Q_t} L(y_k, \text{sign}((\mathbf{w}_{t-1} - \alpha_j y_j \mathbf{x}_j) \cdot \mathbf{x}_k) \}. \tag{4}$$

The problem with this strategy is that many of these examples may be either very easy to classify or always mislabeled (noise) so this could be wastful.

We therefore suggest a secondary caching scheme to choose the q examples with which we estimate the error. We wish to keep the examples that are most likely to change label as these are most likely to give us information about the performance of the classifier. If an example is well classified it will not change label easily when the classifier changes slightly. Likewise, if an example is an outlier it will be consisently incorrectly classified. In fact the number of examples that are relevant in this context should be relatively small. We therefore keep a count $\mathbf{s}_i$ of the number of times example $\mathbf{x}_i$ has changed label, divided by the amount of time it has been in the cache. If this value is small then we can consider removing this point from the secondary cache. When we receive a new observation at $\mathbf{x}_t$ at time t we thus perform the update given in figure 5 in the case that $\mathbf{x}_t$ is a margin error.

Complexity The last variant of the Tighter Budget Perceptron has a deletion step cost of $O(pq + qn)$ operations, where p is the cache size, q is the secondary cache size, and n is the input dimensionality. This should be compared to $O(p)$ for the Budget Perceptron, where clearly the deletion step is still less expensive.

In the case of relatively noise free problems with a reasonable cache size p, the deletion step occurs infrequently: by the time the cache becomes full, the perceptron performs well enough to make margin errors rare. The insertion step then dominates the computational cost. In the case of noisy problems, the cheaper deletion step of the Budget Perceptron performs too poorly to be considered a valid alternative. Moreover, as we shall see experimentally, the Tighter Budget Perceptron can achieve the same test error as the Budget Perceptron for smaller cache size p.

4 Experiments

4.1 2D Experiments - Online mode

Figure 6 shows a 2D classification problem of 1000 points from two classes separated by a very small margin. We show the decision rule found after one epoch of Perceptron, Budget Perceptron and Tighter Budget Perceptron training, using a linear kernel. Both Budget Perceptrons variants produce sparser solutions than the Perceptron, although the Budget Perceptron provides slightly less accurate solutions, even for larger cache sizes. Figure 7 shows a similar dataset, but with overlapping classes. The Perceptron algorithm will fail to converge with multiple epochs in this case. After one epoch a decision rule is obtained with a relatively large number of SVs. Most examples which are on the wrong side of the Bayes decision rule are SVs. [3] The Budget Perceptron fails to alleviate this problem. Although one can reduce the cache size to force more sparsity, the decision rule obtained is highly inaccurate. This is due to noise points which are far from the decision boundary

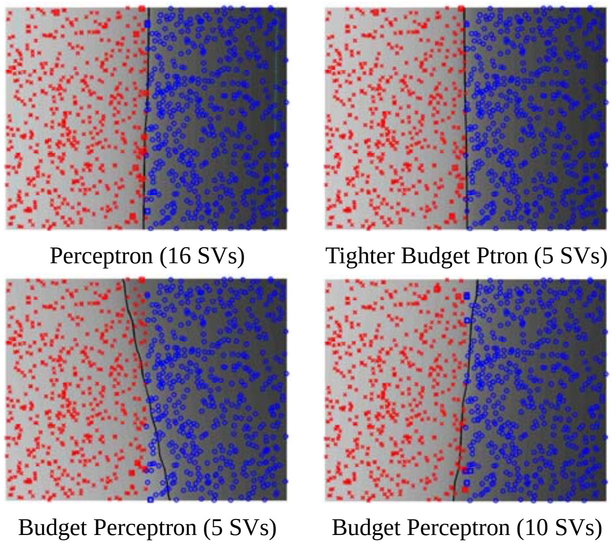


Figure 6: **Separable Toy Data in Online Mode.** The Budget Perceptron of (Crammer, Kandola and Singer, 2004) and our Tighter Budget Perceptron provide sparser solutions than the Perceptron algorithm, however the Budget Perceptron seems sometimes to provide slightly worse solutions.

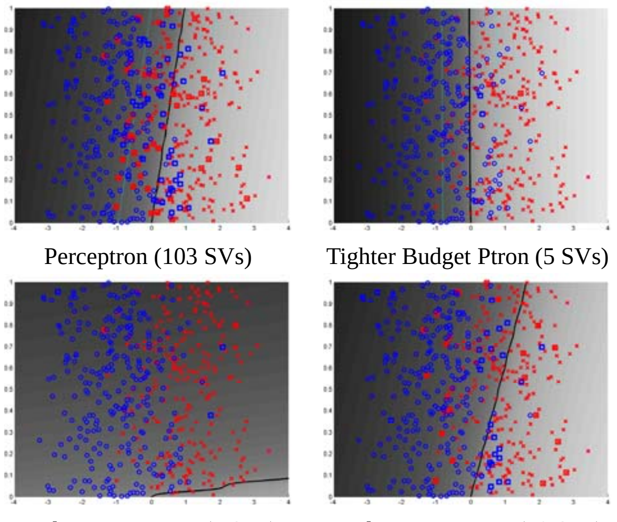
being the last vectors to be removed from the cache, as can be seen in the example with a cache size of 50.

4.2 2D Experiments - Batch mode

Figure 8 shows a 2D binary classification problem with the decision surface found by the Tighter Budget Perceptron, Budget Perceptron, Perceptron, SVM, and two flavors of KMP when using the same Gaussian kernel. For the online algorithms we ran multiple epochs over the dataset until convergence. This example gives a simple demonstration of how the Budget and Tighter Budget Perceptrons can achieve a greater level of sparsity than the Perceptron, whilst choosing examples that are close to the margin, in constrast to the KMP algorithm.

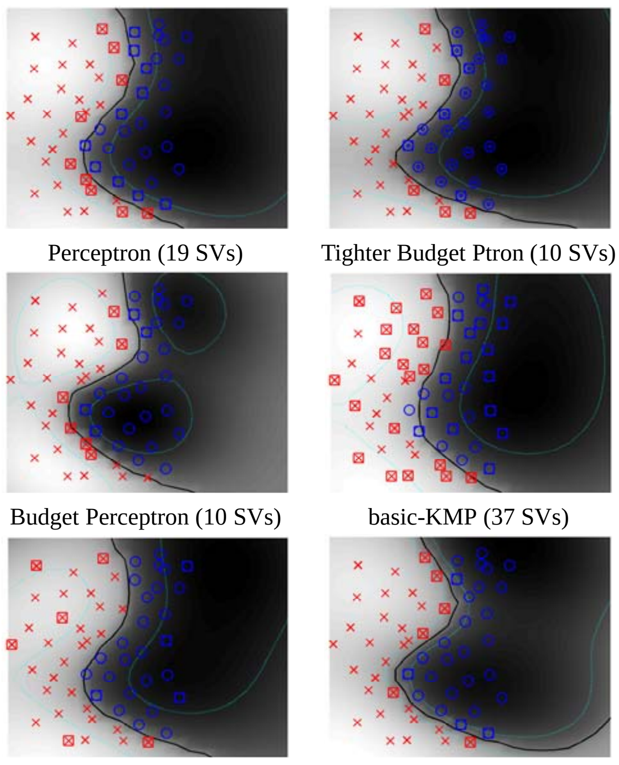
Where possible in the fixed cache algorithms, we fixed the cache sizes to 10 SVs (examples highlighted with squares), as a trained SVM uses this number. The Perceptron is not as sparse as SVM, and uses 19 SVs. However both the Budget Perceptron and the Tighter Budget Perceptron still separate the data with 10 SVs. The Perceptron required 14 epochs to converge, the Tighter Budget Perceptron required 22, the Budget Perceptron required 26 (however, we had to decrease the width of the Gaussian kernel for the last algorithm as it did not converge for larger widths). Backfitting-KMP provides as good sparsity as SVM. Basic-KMP does not give zero error even after 400 iterations, and by this time has used 37 SVs (it cycles around the same SVs many times). Note that all the algorithms except KMP choose

[3] Note that support vector machines, not shown, also suffer from a similar deficiency in terms of sparsity - all incorrectly classified examples are SVs.


Perceptron (103 SVs) Tighter Budget Ptron (5 SVs)

Budget Perceptron (5 SVs) Budget Perceptron (50 SVs)

Figure 7: **Noisy Toy Data in Online Mode.** The Perceptron and Budget Perceptron (independent of cache size) fail when problems contain noise, as demonstrated by a simple 2D problem with Gaussian noise in one dimension. The Tighter Budget Perceptron, however, finds a separation very close to the Bayes rule.


Perceptron (19 SVs) Tighter Budget Ptron (10 SVs)

Budget Perceptron (10 SVs) basic-KMP (37 SVs)

backfitting-KMP (10 SVs) SVM (10 SVs)

Figure 8: **Nonlinear Toy Data in Batch Mode.** We compare various algorithms on a simple nonlinear dataset following (Vincent and Bengio, 2000). See the text for more explanation.

SVs close to the margin.

4.3 Benchmark Datasets

We conducted experiments on three well-known datasets: the US Postal Service (USPS) Database, Waveform and Banana[4]. A summary of the datasets is given in Table 1. For all three cases (USPS,Waveform,Banana) we chose an RBF kernel, the width values were taken from (Vapnik, 1998) and (Rätsch, Onoda and Müller, 2001), and are chosen to be optimal for SVMs, the latter two by cross validation. We used the same widths for all other algorithms, despite that these may be suboptimal in these cases. For USPS we considered the two class problem of separating digit zero from the rest. We also constructed a second experiment by corrupting USPS with label noise. We randomly flipped the labels of 10% of the data in the training set to observe the performance effects on the algorithms tested (for SVMs, we report the test error with the optimal value of C chosen on the test set, in this case, $C = 10$).

We tested the Tighter Budget Perceptron, Budget Perceptron and Perceptron in an *online* setting by only allowing one pass through the training data. Obviously this puts these algorithms at a disadvantage compared to batch algorithms such as SVMs which can look at training points multiple times. Nevertheless, we compare with SVMs as our gold standard measure of performance. The results are given in figures 9-12. In all cases for the Perceptron variants we use $\beta = 0$ and for the Tighter Budget Perceptron we employ the algorithm given in figure 3 without the computational efficiency techniques given in section 3.2. We show the test error against different fixed cache sizes p resulting in p support vectors. We report averages over 5 runs for USPS and 10 runs for Waveform and Banana. The error bars indicate the maximum and minimum values over all runs.

The results show the Tighter Budget Perceptron yielding similar test error performance to the SVM but with considerably less SVs. The Budget Perceptron fares less well with

Name	Inputs	Train	Test	σ	C
USPS	256	7329	2000	128	1000
Waveform	21	4000	1000	3.16	1
Banana	2	4000	1300	0.7	316

Table 1: **Datasets used in the experiments.** The hyperparameters are for an SVM with RBF kernel, taken from (Vapnik, 1998) and (Rätsch, Onoda and Müller, 2001).

[4]USPS is available at `ftp://ftp.kyb.tuebingen.mpg.de/pub/bs/data`. Waveform and Banana are available at `http://mlg.anu.edu.au/˜raetsch/data/`.

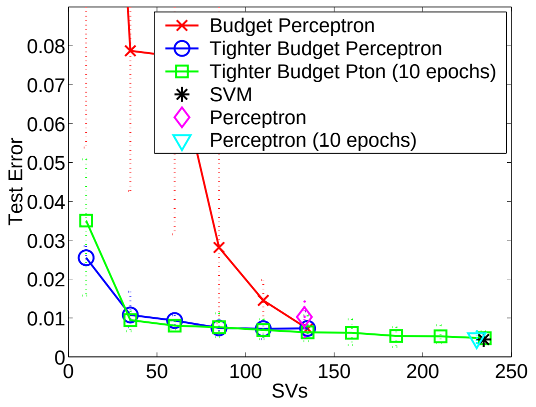


Figure 9: **USPS Digit 0 vs Rest.**

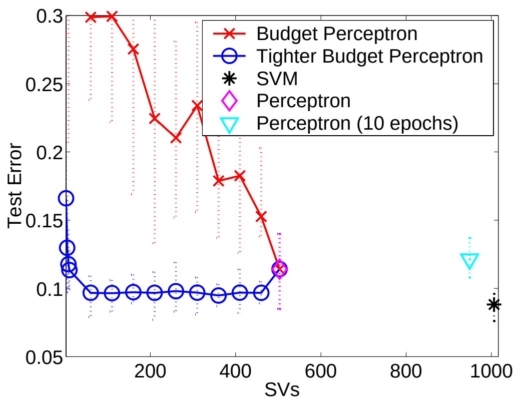


Figure 11: **Waveform Dataset.**

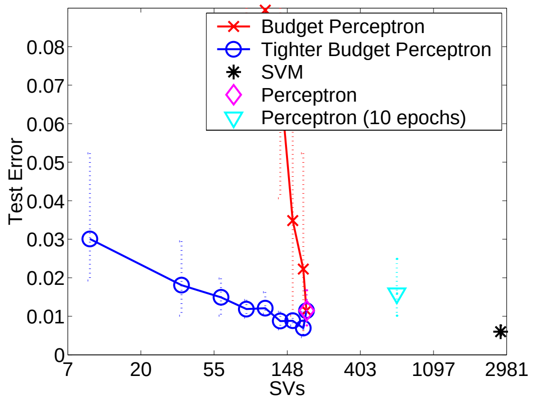


Figure 10: **USPS Digit 0 vs Rest + 10% noise.**

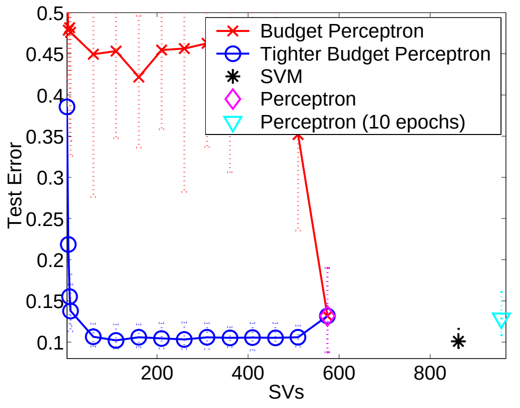


Figure 12: **Banana Dataset.**

the test error degrading considerably faster for decreasing cache size compared to the Tighter Budget Perceptron, particularly on the noisy problems. Note that if the cache size is large enough both the Budget and Tighter Budget Perceptrons converge to the Perceptron solution, hence the two curves meet at their furthest point. However, while the test error immediately degrades for the Budget Perceptron, for the Tighter Budget Perceptron the test error in fact *improves* over the Perceptron test error in both the noisy problems. This should be expected as the standard Perceptron cannot deal with overlapping classes.

In figure 9 we also show the Tighter Budget Perceptron with (up to) 10 epochs on USPS (typically the algorithm converges before 10 epochs). The performance is similar to only 1 epoch for small cache sizes. For larger cache sizes, clearly the maximum cache size converges to the same performance of a Perceptron with 10 epochs, which in this case gives slightly better performance than any cache size possible with 1 epoch.

4.4 Faster Error Computation

In this section we explore the error evaluation cache strategies described previously in section 3.2. We compared the following strategies to evaluate error rates: (i) using all points so far seen, (ii) using only the support vectors in the cache, i.e. equation (3), (iii) using a random cache of size q, i.e. equation (4), and (iv) using a cache of size q of the examples that flip label most often, i.e. figure 5.

Figure 13 compares these methods on the USPS dataset for fixed cache size of support vectors $p = 35$ and $p = 85$. The results are averaged over 40 runs (the error bars show the standard error).

We compare different amounts of evaluation vectors q. The results show that considerable computational speedup can be gained by any of the methods compared to keeping all training vectors. Keeping a cache of examples that change label most often performs better than a random cache for small cache sizes. Using the support vectors themselves also performs better than the random strategy for the same cache size. This makes sense as support vectors themselves are likely to be examples that can change label easily, making it similar to the cache of examples that most often change label. Nevertheless, it can be worthwhile to have a small number of support vectors for fast evaluation, but a larger set of error evaluation vectors when an error is encountered. We suggest to choose q and p such that a cache of the qp kernel calculations fits in memory at all times.

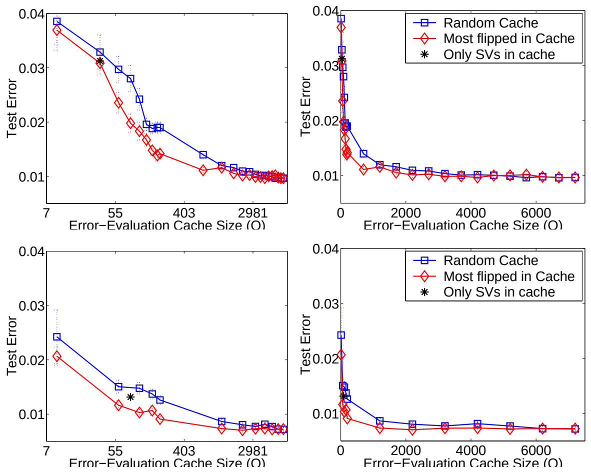


Figure 13: **Error Rates for Different Error Measure Cache Strategies on USPS, digit zero versus the rest.** The number of SVs is fixed to 35 in the top row, and 85 in the bottom row, the left-hand plots are log plots of the right-hand ones. The different strategies change the number of points used to evaluate the error rate for the SV cache deletion process.

5 Summary

We have introduced a sparse online algorithm that is a variant of the Perceptron. It attempts to deal with some of the computational issues of using kernel algorithms in an online setting by restricting the number of SVs one can use. It also allows methods such as the Perceptron to deal with overlapping classes and noise. It can be considered as an improvement over the Budget Perceptron of (Crammer, Kandola and Singer, 2004) because it is empirically sparser than that method for the same error rate, and can handles noisy problems while that method cannot. Our method tends to keep only points that are close to the margin *and* that lie on the correct side of the Bayes decision rule. This occurs because other examples are less useful for describing a decision rule with low error rate, which is the quality measure we use for inclusion in to the cache.

However, the cost of this is that quality measure used to evaluate training points is more expensive to compute than for the Budget Perceptron (and in this sense the name ”Tighter Budget Perceptron” is slightly misleading). However, we believe there exist various approximations to speed up this method whilst retaining its useful properties. We explored some strategies in this vein by introducing a small secondary cache of evaluation vectors with positive results. Future work should investigate further ways to improve on this, some first suggestions being to only look at a subset of points to remove on each iteration, or to remove the worst n points every n iterations.

Acknowledgements

Part of this work was funded by NSF grant CCR-0325463.

References

Bakır, G., Bottou, L., and Weston, J. (2004). Breaking SVM Complexity with Cross-Training. In *Advances in Neural Information Processing Systems 17 (NIPS 2004)*. MIT Press, Cambridge, MA. to appear.

Boser, B. E., Guyon, I. M., and Vapnik, V. (1992). A Training Algorithm for Optimal Margin Classifiers. In Haussler, D., editor, *Proceedings of the 5th Annual ACM Workshop on Computational Learning Theory*, pages 144–152, Pittsburgh, PA. ACM Press.

Cortes, C. and Vapnik, V. (1995). Support Vector Networks. *Machine Learning*, 20:273–297.

Crammer, K., Kandola, J., and Singer, Y. (2004). Online Classification on a Budget. In Thrun, S., Saul, L., and Schölkopf, B., editors, *Advances in Neural Information Processing Systems 16*. MIT Press, Cambridge, MA.

Crammer, K. and Singer, Y. (2003). Ultraconservative Online Algorithms for Multiclass Problems. *Journal of Machine Learning Research*, 3:951–991.

Devijver, P. and Kittler, J. (1982). *Pattern Recogniton, A statistical approach*. Prentice Hall, Englewood Cliffs.

Gentile, C. (2001). A New Approximate Maximal Margin Classification Algorithm. *Journal of Machine Learning Research*, 2:213–242.

Kecman, V., Vogt, M., and Huang, T. (2003). On the Equality of Kernel AdaTron and Sequential Minimal Optimization in Classification and Regression Tasks and Alike Algorithms for Kernel Machines. In *Proceedings of European Symposium on Artificial Neural Networks, ESANN'2003*, pages 215–222, Evere, Belgium. D-side Publications.

Li, Y. and Long, P. (2002). The Relaxed Online Maximum Margin Algorithm. *Machine Learning*, 46:361–387.

Nair, P. B., Choudhury, A., and Keane, A. J. (2002). Some Greedy Learning Algorithms for Sparse Regression and Classification with Mercer Kernels. *Journal of Machine Learning Research*, 3:781–801.

Rätsch, G., Onoda, T., and Müller, K.-R. (2001). Soft Margins for AdaBoost. *Machine Learning*, 42(3):287–320. Also: NeuroCOLT Technical Report 1998-021.

Rosenblatt, F. (1957). The Perceptron: A perceiving and recognizing automaton. Technical Report 85-460-1, Project PARA, Cornell Aeronautical Lab.

Vapnik, V. and Lerner, A. (1963). Pattern Recognition using Generalized Portrait Method. *Automation and Remote Control*, 24:774–780.

Vapnik, V. N. (1998). *Statistical Learning Theory*. Wiley, New York.

Vincent, P. and Bengio, Y. (2000). Kernel Matching Pursuit. Technical Report 1179, Département d'Informatique et Recherche Opérationnelle, Université de Montréal. Presented at Snowbird'00.