The Huller: a simple and efficient online SVM

Antoine Bordes[1 2] and Léon Bottou[2]

[1] Ecole Supérieure de Physique et de Chimie Industrielles, Paris, France
[2] NEC Labs America, Princeton, NJ, USA

Abstract. We propose a novel online kernel classifier algorithm that converges to the Hard Margin SVM solution. The same update rule is used to both add and remove support vectors from the current classifier. Experiments suggest that this algorithm matches the SVM accuracies after a single pass over the training examples. This algorithm is attractive when one seeks a competitive classifier with large datasets and limited computing resources.

1 Introduction

Support Vector Machines (SVMs) [1] are the successful application of the kernel idea [2] to large margin classifiers [3]. Early kernel classifiers [2] were derived from the perceptron [4], a simple and efficient online learning algorithm. Many authors have sought to replicate the SVM success by applying the large margin idea to such simple online algorithms [5–10].

This paper proposes a simple and efficient online kernel algorithm which combines several desirable properties:

- Continued iterations of the algorithm eventually converge to the exact Hard Margin SVM classifier.
- Like most SVM algorithms, and unlike most online kernel algorithms, it produces classifiers with a bias term. Removing the bias term is a known way to simplify the numerical aspects of SVMs. Unfortunately, this can also damage the classification accuracy [11].
- Experiments on a relatively clean dataset indicate that a single pass over the training set is sufficient to produce classifiers with competitive error rates, using a fraction of the time and memory required by state-of-the-art SVM solvers.

Section 2 reviews the geometric interpretation of SVMs. Section 3 presents a simple update rule for online algorithms that converge to the SVM solution. Section 4 presents a critical refinement and describes its relation with previous online kernel algorithms. Section 5 reports experimental results. Finally section 6 discusses the algorithm capabilities and limitations.

2 Geometrical formulation of SVMs

Figure 1 illustrates the geometrical formulation of SVMs [12, 13]. Consider a training set composed of patterns x_i and corresponding classes $y_i = \pm 1$. When

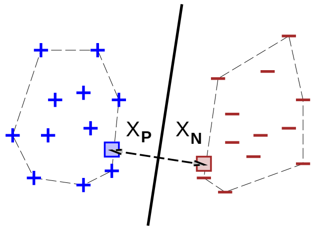

Fig. 1. Geometrical interpretation of Support Vector Machines

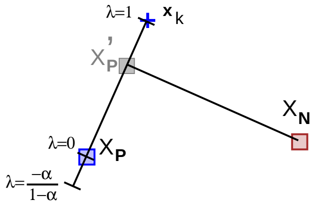

Fig. 2. Basic update of the HULLER.

the training data is separable, the convex hulls formed by the positive and negative examples are disjoint. Consider two points $\boldsymbol{X_P}$ and $\boldsymbol{X_N}$ belonging to each convex hull. Make them as close as possible without allowing them to leave their respective convex hulls. The median hyperplane of these two points is the maximum margin separating hyperplane.

The points $\boldsymbol{X_P}$ and $\boldsymbol{X_N}$ can be parametrized as

$$\begin{array}{lll} \boldsymbol{X_P} = \sum_{i\in\mathcal{P}} \alpha_i \boldsymbol{x}_i & \sum_{i\in\mathcal{P}} \alpha_i = 1 & \alpha_i \geq 0 \\ \boldsymbol{X_N} = \sum_{j\in\mathcal{N}} \alpha_j \boldsymbol{x}_j & \sum_{j\in\mathcal{N}} \alpha_j = 1 & \alpha_j \geq 0 \end{array} \tag{1}$$

where sets $\mathcal{P}$ and $\mathcal{N}$ respectively contain the indices of the positive and negative examples. The optimal hyperplane is then obtained by solving

$$\min_{\boldsymbol{\alpha}} \|\boldsymbol{X_P} - \boldsymbol{X_N}\|^2 \tag{2}$$

under the constraints of the parametrization (1). The separating hyperplane is then represented by the following linear discriminant function:

$$\hat{y}(\boldsymbol{x}) = (\boldsymbol{X_P} - \boldsymbol{X_N})\,\boldsymbol{x} + (\boldsymbol{X_N}\boldsymbol{X_N} - \boldsymbol{X_P}\boldsymbol{X_P})/2 \tag{3}$$

Since $\boldsymbol{X_P}$ and $\boldsymbol{X_N}$ are represented as linear combination of the training patterns, both the optimization criterion (2) and the discriminant function (3) can be expressed using dot products between patterns. Arbitrary non linear classifiers can be derived by replacing these dot products by suitable kernel functions. For simplicity, we discuss the simple linear setup and leave the general kernel framework to the reader.

3 Single example update

We now describe a first iterative algorithm that can be viewed as a simplification of the nearest point algorithms discussed in [14, 11]. The algorithm stores the position of points $\boldsymbol{X_P}$ and $\boldsymbol{X_N}$ using the parametrization (1). Each iteration considers a training pattern $\boldsymbol{x}_k$ and updates the position of $\boldsymbol{X_P}$ (when $y_k = +1$) or $\boldsymbol{X_N}$ (when $y_k = -1$.)

Figure 2 illustrates the case where $\boldsymbol{x}_k$ is a positive example (negative examples are treated similarly). The new point $\boldsymbol{X'_P}$ is *a priori* the point of segment

$[\boldsymbol{X_P}, \boldsymbol{x}_k]$ that minimizes the distance $\|\boldsymbol{X'_P} - \boldsymbol{X_N}\|^2$. The new point $\boldsymbol{X'_P}$ can be expressed as $\boldsymbol{X'_P} = (1-\lambda)\boldsymbol{X_P} + \lambda\boldsymbol{x}_k$ with $0 \leq \lambda \leq 1$.

This first algorithm is flawed: suppose that the current $\boldsymbol{X_P}$ contains a non zero coefficient α_k that in fact should be zero. The algorithm cannot reduce this coefficient by selecting example $\boldsymbol{x}_k$. It must instead select other positive examples and slowly erode the coefficient α_k by multiplying it by $(1-\lambda)$. A simple fix was proposed by Haffner [15]. If the coefficient α_k is strictly positive, we can safely let λ become slightly negative without leaving the convex hull. The revised constraints on λ are then $-\alpha_k/(1-\alpha_k) \leq \lambda \leq 1$.

The optimal value of λ can be computed analytically by first computing the unconstrained optimum λ_u. When $\boldsymbol{x}_k$ is a positive example, solving the orthogonality equation $(\boldsymbol{X_P} - \boldsymbol{X'_P})(\boldsymbol{X_N} - \boldsymbol{X'_P}) = 0$ for λ yields:

$$\lambda_\mathrm{u} = \frac{(\boldsymbol{X_P} - \boldsymbol{X_N})(\boldsymbol{X_P} - \boldsymbol{x}_k)}{(\boldsymbol{X_P} - \boldsymbol{x}_k)^2} = \frac{\boldsymbol{X_P}^2 - \boldsymbol{X_N}\boldsymbol{X_P} - \boldsymbol{X_P}\boldsymbol{x}_k + \boldsymbol{X_N}\boldsymbol{x}_k}{\boldsymbol{X_P}^2 + \boldsymbol{x}_k^2 - 2\boldsymbol{X_P}\boldsymbol{x}_k} \tag{4}$$

Similarly, when $\boldsymbol{x}_k$ is a negative example, we obtain:

$$\lambda_\mathrm{u} = \frac{(\boldsymbol{X_N} - \boldsymbol{X_P})(\boldsymbol{X_N} - \boldsymbol{x}_k)}{(\boldsymbol{X_N} - \boldsymbol{x}_k)^2} = \frac{\boldsymbol{X_N}^2 - \boldsymbol{X_N}\boldsymbol{X_P} - \boldsymbol{X_N}\boldsymbol{x}_k + \boldsymbol{X_P}\boldsymbol{x}_k}{\boldsymbol{X_N}^2 + \boldsymbol{x}_k^2 - 2\boldsymbol{X_N}\boldsymbol{x}_k} \tag{5}$$

A case by case analysis of the constraints shows that the optimal λ is:

$$\lambda = \min\left(1, \max\left(\frac{-\alpha_k}{1-\alpha_k},\ \lambda_\mathrm{u}\right)\right) \tag{6}$$

Both expressions (4) and (5) depend on the quantities $\boldsymbol{X_P}\boldsymbol{X_P}$, $\boldsymbol{X_N}\boldsymbol{X_P}$, and $\boldsymbol{X_N}\boldsymbol{X_N}$ whose computation could be expensive. Fortunately there is a simple way to avoid this calculation: in addition to points $\boldsymbol{X_P}$ and $\boldsymbol{X_N}$, our algorithm also maintains three scalar variable containing the values of $\boldsymbol{X_P}\boldsymbol{X_P}$, $\boldsymbol{X_N}\boldsymbol{X_P}$, and $\boldsymbol{X_P}\boldsymbol{X_P}$. Their values are recursively updated after each iteration: when $\boldsymbol{x}_k$ is a positive example,

$$\begin{aligned} \boldsymbol{X'_P}\boldsymbol{X'_P} &= (1-\lambda)^2\boldsymbol{X_P}\boldsymbol{X_P} + 2\lambda(1-\lambda)\boldsymbol{X_P}\boldsymbol{x}_k + \lambda^2\boldsymbol{x}_k\boldsymbol{x}_k \\ \boldsymbol{X_N}\boldsymbol{X'_P} &= (1-\lambda)\boldsymbol{X_N}\boldsymbol{X_P} + \lambda\boldsymbol{X_N}\boldsymbol{x}_k \\ \boldsymbol{X_N}\boldsymbol{X_N} &= \boldsymbol{X_N}\boldsymbol{X_N} \end{aligned} \tag{7}$$

and similarly, when $\boldsymbol{x}_k$ is a negative example,

$$\begin{aligned} \boldsymbol{X_P}\boldsymbol{X_P} &= \boldsymbol{X_P}\boldsymbol{X_P} \\ \boldsymbol{X'_N}\boldsymbol{X_P} &= (1-\lambda)\boldsymbol{X_N}\boldsymbol{X_P} + \lambda\boldsymbol{x}_k\boldsymbol{X_P} \\ \boldsymbol{X'_N}\boldsymbol{X'_N} &= (1-\lambda)^2\boldsymbol{X_N}\boldsymbol{X_N} + 2\lambda(1-\lambda)\boldsymbol{X_N}\boldsymbol{x}_k + \lambda^2\boldsymbol{x}_k\boldsymbol{x}_k \end{aligned} \tag{8}$$

Figure 3 shows the resulting update algorithm. The cost of one update is dominated by the calculation of $\boldsymbol{X_P}\boldsymbol{x}_k$ and $\boldsymbol{X_N}\boldsymbol{x}_k$. This calculation requires the dot products between $\boldsymbol{x}_k$ and all the current support vectors, i.e. the training examples $\boldsymbol{x}_i$ with non zero coefficient α_i in the parametrization (1).

```
UPDATE($k$):
  - Compute $\boldsymbol{X_P x}_k$, $\boldsymbol{X_N x}_k$, and $\boldsymbol{x}_k\boldsymbol{x}_k$.
  - Compute $\lambda_{\mathrm{u}}$ using equations (4) or (5).
  - Compute $\lambda$ using equation (6)
  - $\alpha_i \leftarrow (1-\lambda)\alpha_i$ for all $i$ such that $y_i = y_k$.
  - $\alpha_k \leftarrow \alpha_k + \lambda$.
  - Update $\boldsymbol{X_P X_P}$, $\boldsymbol{X_N X_P}$ and $\boldsymbol{X_N X_N}$ using equation (7) or (8).
```

Fig. 3. Algorithm for the basic update.

```
HULLER:
  - Initialize $\boldsymbol{X_P}$ and $\boldsymbol{X_N}$ by averaging a few points.
    Compute initial $\boldsymbol{X_P X_P}$, $\boldsymbol{X_N X_P}$, and $\boldsymbol{X_N X_N}$.
  - Iterate:
      - Pick a random $p$ such that $\alpha_p = 0$
      - UPDATE($p$)
      - Pick a random $r$ such that $\alpha_r \neq 0$
      - UPDATE($r$)
```

Fig. 4. The HULLER algorithm.

4 Insertion and Removal

Simply repeating this update for random examples $\boldsymbol{x}_k$ works poorly. Most of the updates do nothing because they involve examples that are not support vectors and have no vocation to become support vectors. A closer analysis reveals that the update operation has two functions:

– Performing an update for an example $\boldsymbol{x}_k$ such that $\alpha_k = 0$ represents an attempt to insert this example into the current set of support vectors. This occurs when the optimal λ is greater than zero, that is, when the point $\boldsymbol{x}_k$ violates the SVM margin conditions.
– Performing an update for an example $\boldsymbol{x}_k$ such that $\alpha_k \neq 0$ will optimize the current solution and possibly remove this example from the current set of support vectors. The removal occurs when the optimal λ reaches its (negative) lower bound.

Recent work on kernel perceptrons [10] also rely on two separate processes to insert and remove support vectors from the expression of the current separating hyperplane. This paper discusses a situation where both functions are implemented by the same update rule (figure 2). Picking the examples $\boldsymbol{x}_k$ randomly gives a disproportionate weight to the insertion function.

The HULLER algorithm, figure 4, corrects this imbalance by allocating an equivalent computing time to both functions. First, it picks a random example that is not a current support vector and attempts to insert it into the current set of support vectors. Second, it picks a random example that is a current support vector and attempts to remove it from the current set of support vectors. This simple modification has a dramatic effect on the convergence speed.

5 Experiments

The HULLER algorithm was implemented in C and benchmarked against the state-of-the-art SVM solver LIBSVM[3] on the well known MNIST[4] handwritten digit dataset. All experiments were run with a RBF kernel width parameter $\gamma = 0.005$. Both LIBSVM and the HULLER implementation use the same code to compute the kernel values and similar strategies to cache the frequently used kernel values. The cache size was initially set to 256MB.

Figure 5 reports the experimental results on the ten problems consisting of classifying each of the ten digit category against all other categories. The HULLER algorithm was run in epochs. Each epoch sequentially scans the randomly permuted MNIST training set and attempts to insert each example into the current set of support vectors (first update operation in figure 4). After each insertion attempt, the algorithm attempts to remove a random support vector (second update operation in figure 4.)

The HULLER×1 results were obtained after a single epoch, that is after processing each example once. The HULLER×2 results were obtained after two epochs. All results are averages over five runs.

The HULLER×2 test errors (top left graph in figure 5) closely match the LIBSVM solution. This is confirmed by counting the number of support vectors (bottom left graph), The HULLER×2 computing times usually are slightly shorter than the already fast LIBSVM computing times (top right graph).

The HULLER×1 test errors (top left graph in figure 5) are very close to both the HULLER×2 and LIBSVM test errors. Standard paired significance tests indicate that these small differences are not significant. This accuracy is achieved after less than half the LIBSVM running time, and, more importantly, after a single sequential pass over the training examples. The HULLER×1 always yields a slightly smaller number of support vectors (bottom left graph). We believe that a single HULLER epoch fails to insert a few examples that appear as support vectors in the SVM solution. A second epoch recaptures most missing examples.

Neither the HULLER×1 or HULLER×2 experiments yield the exact SVM solution. On this dataset, the HULLER typically reaches the SVM solution after five epochs. The corresponding computing times are not competitive with those achieved by LIBSVM.

These results should also be compared with results obtained with a theoretically justified kernel perceptron algorithm. Figure 5 contains results obtained with the AVERAGED PERCEPTRON [5] using the same kernel and cache size. The first epoch runs very quickly but does not produce competitive error rates. The AVERAGED PERCEPTRON approaches[5] the LIBSVM or HULLER×1 accuracies after ten epochs[6]. The corresponding training times stress the importance of the kernel cache size. When the cache can accomodate the dot products of all examples with all support vectors, additional epochs require very little compu-

[3] `http://www.csie.ntu.edu.tw/∼cjlin/libsvm`

[4] `http://yann.lecun.com/exdb/mnist`

[5] This is consistent with the empirical results reported in [5] (table 3).

[6] The Averaged Perceptron theoretical guarantees only hold for a single epoch.

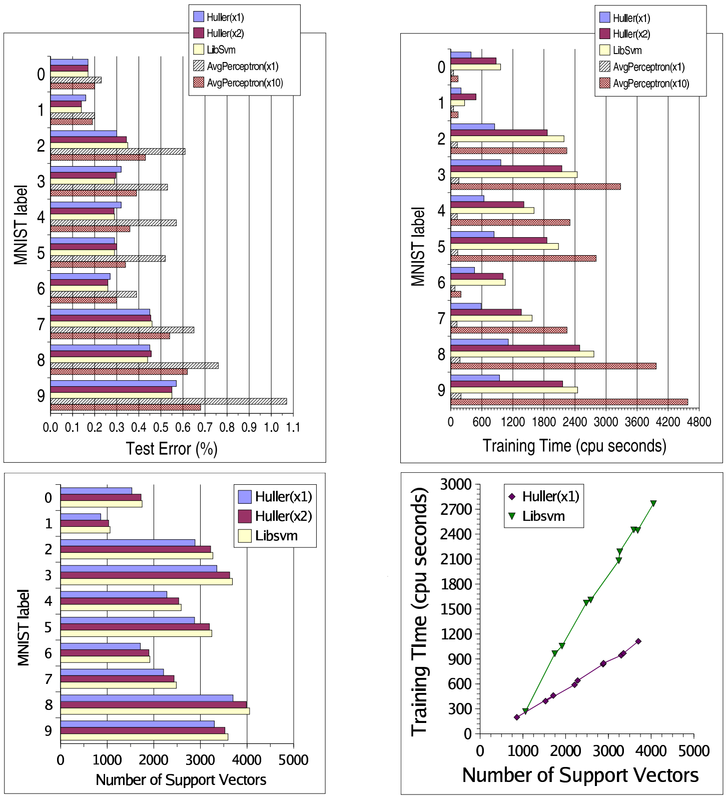

Fig. 5. MNIST results for the HULLER (one and two epochs), for LIBSVM, and for the AVERAGED PERCEPTRON (one and ten epochs). Top left: test error accuracies. Top right: training time. Bottom left: number of support vectors. Bottom right: training time as a function of the number of support vectors.

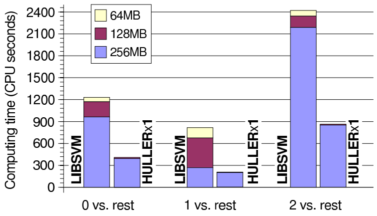

Fig. 6. Computing times with various cache sizes. Each color indicates the additional time required when reducing the cache size. The HULLER times remain virtually unchanged.

tation. When this is not the case, the AVERAGED PERCEPTRON times are not competitive.

Figure 6 shows how reducing the cache size affects the computing time. Whereas LIBSVM experiences significantly increased training times, the HULLER training times are essentially unchanged. The most dramatic case is the separation of digit “1” versus all other categories. The initial 256MB cache size is sufficient for holding all the kernel values required by LIBSVM. Under these condition, LIBSVM runs almost as quickly as the HULLER×1. Reducing the kernel cache size to 128MB doubles the LIBSVM training time and does not change the HULLER training times.

A detailled analysis of the algorithms indicate that LIBSVM runs best when the cache contains all the dot products involving a potential support vector and an arbitrary example: memory requirements grow with both the number of support vectors and the number of training examples. The HULLER runs best when the cache contains all the dot products involving two potential support vectors: the memory requirements grow with the number of support vectors only. This indicates that the HULLER is best suited for problems involving a large separable training set.

6 Discussion

Fast start versus deep optimization. The HULLER processes many more examples during the very first training stages. After processing the first pair of examples, the SMO core of LIBSVM must compute 120000 dot products to update the example gradients and choose the next pair. During the same time, the HULLER processes at least 500 examples. By the time LIBSVM has reached the fifth pair of examples, the HULLER has processed a minimum of 1500 fresh examples. Online kernel classifiers without removal step tend to slow down sharply because the number of support vectors increases quickly. The removal step ensures that the number of current support vectors does not significantly exceed the final number of support vectors.

To attain the exact SVM solution with confidence, the HULLER also must compute all the dot products it did not compute in the early stages. On the other hand, when the kernel cache size is large enough, LIBSVM already knows these values and can use this rich local information to move more judiciously. This is why LIBSVM outperforms the huller in the final stages of the optimization. Nevertheless, the HULLER produces competitive classifiers well before reaching the point where it gets outpaced by state-of-the-art SVM optimization packages such as LIBSVM.

Noisy datasets. The HULLER addresses the hard margin SVM problem and therefore performs poorly on noisy datasets [16]. Most online kernel classifiers share this limitation. However, soft margin support vector machines *with quadratic slacks* [16] can be implemented as hard margin support vector machines with a modified kernel $K_C(\boldsymbol{x}_i, \boldsymbol{x}_j) = K(\boldsymbol{x}_i, \boldsymbol{x}_j) + \frac{1}{C}\delta_{ij}$. However, the resulting classifier is not directly comparable to the standard soft-margin SVM with linear slacks.

7 Conclusion

The HULLER is a novel online kernel classifier algorithm that converges to the Hard Margin SVM solution. Experiments suggest that it matches the SVM accuracies after a single pass over the training examples. Time and memory requirements are then modest in comparison to state-of-the-art SVM solvers.

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

References

1. Vapnik, V.N.: The Nature of Statistical Learning Theory. Springer Verlag, New York (1995)
2. Aizerman, M.A., Braverman, É..M., Rozonoér, L.I.: Theoretical foundations of the potential function method in pattern recognition learning. Automation and Remote Control **25** (1964) 821–837
3. Vapnik, V., Lerner, A.: Pattern recognition using generalized portrait method. Automation and Remote Control **24** (1963) 774–780
4. Rosenblatt, F.: The perceptron: A probabilistic model for information storage and organization in the brain. Psychological Review **65** (1958) 386–408
5. Freund, Y., Schapire, R.E.: Large margin classification using the perceptron algorithm. In Shavlik, J., ed.: Machine Learning: Proceedings of the Fifteenth International Conference, San Francisco, CA, Morgan Kaufmann (1998)
6. Frieß, T.T., Cristianini, N., Campbell., C.: The kernel Adatron algorithm: a fast and simple learning procedure for support vector machines. In Shavlik, J., ed.: 15th International Conf. Machine Learning, Morgan Kaufmann Publishers (1998) 188–196
7. Gentile, C.: A new approximate maximal margin classification algorithm. Journal of Machine Learning Research **2** (2001) 213–242
8. Li, Y., Long, P.: The relaxed online maximum margin algorithm. Machine Learning **46** (2002) 361–387
9. Crammer, K., Singer, Y.: Ultraconservative online algorithms for multiclass problems. Journal of Machine Learning Research **3** (2003) 951–991
10. Crammer, K., Kandola, J., Singer, Y.: Online classification on a budget. In Thrun, S., Saul, L., Schölkopf, B., eds.: Advances in Neural Information Processing Systems 16. MIT Press, Cambridge, MA (2004)
11. Keerthi, S.S., Shevade, S.K., Bhattacharyya, C., Murthy, K.R.K.: A fast iterative nearest point algorithm for support vector machine classifier design. Technical Report Technical Report TR-ISL-99-03, Indian Institute of Science, Bangalore (1999) http://guppy.mpe.nus.edu.sg/∼mpessk/npa_tr.ps.gz.
12. Bennett, K.P., Bredensteiner, E.J.: Duality and geometry in SVM classifiers. In Langley, P., ed.: Proceedings of the 17th International Conference on Machine Learning, San Francisco, California, Morgan Kaufmann (2000) 57–64
13. Crisp, D.J., Burges, C.J.C.: A geometric interpretation of ν-SVM classifiers. In Solla, S.A., Leen, T.K., Müller, K.R., eds.: Advances in Neural Information Processing Systems 12, MIT Press (2000)
14. Gilbert, E.G.: Minimizing the quadratic form on a convex set. SIAM J. Control **4** (1966) 61–79
15. Haffner, P.: Escaping the convex hull with extrapolated vector machines. In Dietterich, T., Becker, S., Ghahramani, Z., eds.: Advances in Neural Information Processing Systems 14, Cambridge, MA, MIT Press (2002) 753–760
16. Cortes, C., Vapnik, V.: Support vector networks. Machine Learning **20** (1995) 273–297