A Sparse Nonlinear Classifier Design Using AUC Optimization

Vishal Kakkar* Shirish Shevade* S Sundararajan† Dinesh Garg‡

Abstract

AUC (Area under the ROC curve) is an important performance measure for applications where the data is highly imbalanced. Efficient AUC optimization is a challenging research problem as the objective function is non-decomposable and non-continuous. Using a max-margin based surrogate loss function, AUC optimization problem can be approximated as a pairwise RankSVM learning problem. Batch learning algorithms for solving the kernelized version of this problem suffer from scalability issues. Therefore, recent years have witnessed an increased interest in the development of online or single-pass algorithms that design a nonlinear classifier by maximizing the AUC performance. However, on many real-world datasets, the AUC performance of these classifiers was observed to be inferior to that of the classifiers designed using batch learning algorithms. Further, many practical imbalanced data classification problems demand fast inference, which underlines the need for designing sparse nonlinear classifiers. Motivated by these observations, we design a scalable algorithm for maximizing the AUC performance by greedily adding the required number of basis functions into the classifier model. The resulting sparse classifier performs faster inference and its AUC performance is comparable with that of the classifier designed using batch mode. Our experimental results show that the level of sparsity achievable can be an order of magnitude larger than that achieved by the Kernel RankSVM model without significantly affecting the AUC performance.

1 Introduction.

In binary classification, a classifier is often trained by optimizing a performance measure such as accuracy. If the data is highly imbalanced, accuracy may not be a good measure to optimize. The all-positive or all-negative classifier may achieve good classification accuracy. But, this will result in misclassification of some important or rare events which typically belong to a minority class. Situations for which datasets are imbalanced are not uncommon in real-world applications and in such cases, classifiers are designed by optimizing measures other than accuracy [5].

Support Vector Machines (SVMs) have been very effective on several real-world problems. Standard SVM formulations for binary classification problem assume that misclassification costs are equal for both the classes. Therefore, SVMs are not suitable if the data is strongly imbalanced. Lin et al. [16] proposed a simple extension of SVMs by using different penalization of positive and negative examples. This approach is useful if misclassification costs are known, which is typically not the case in practice. It is thus necessary to use a different measure for learning from imbalanced data.

AUC (Area Under ROC Curve) [17] is an important performance measure and its optimization has been very effective, especially when class distributions are heavily skewed. However, computing the AUC is a costly operation as it is written as a sum of pairwise losses between examples from different classes, which is quadratic in the number of training set examples. Further, the AUC is not a continuous function on the training set. This makes the optimization of the AUC a challenging task.

More relevant to the work in this paper is the large-scale Kernel RankSVM algorithm proposed by Kuo et al. [14]. This batch learing algorithm, though designed for solving a ranking problem, can be extended to solve the AUC optimization problem. However, kernel evaluations are a bottleneck in training Kernel RankSVM. To alleviate this problem, it was proposed to store the full kernel matrix. Although this reduces repeated kernel evaluations, storage of the full kernel matrix is an issue if the dataset sizes are very large. Further, Kernel RankSVM algorithm may result in a classifier which uses a large number of support vectors, thereby incurring high inference cost. This makes kernel RankSVM unsuitable for practical imbalanced data classification problems which demand fast inference.

Many algorithms have been designed to optimize the AUC using surrogate loss functions (see, for example, Herschtal and Raskutti [9], Joachims [10], Rudin and Schapire [18], Kotlowski et al. [13], Zhao et al. [21]). Due to the high computational demands of the AUC or its variants, most of these algorithms are either one-pass algorithms or online algorithms which rely on sampling. Zhao et al. [21] proposed an online AUC algo-

*Indian Institute of Science, Bangalore, India.{vishal.kakkar, shirish}@csa.iisc.ernet.in

†Microsoft Research, India. (ssrajan@microsoft.com)

‡IIT Gandhinagar, India. (dgarg@iitgn.ac.in)

rithm (OAM) which is based on the idea of reservoir sampling. This idea helps to represent all the received examples by the examples stored in buffers of fixed size. Gao et al. [6] proposed a regression based algorithm for one-pass AUC (OPAUC) optimization. This algorithm maintains only the first and second order statistics of training data in memory, thereby resulting in a storage requirement which is independent of the training dataset size. Both these algorithms learn linear classifiers and are not directly suitable to design complex nonlinear decision boundaries, typically possible by using kernel classifiers.

Calders and Jaroszewicz [3] proposed the use of a polynomial approximation for the AUC, which can be computed in only one scan over the dataset. This approximation was used to design a linear classifier. Yang et al. [20] proposed an online learning algorithm to optimize the AUC by learning a nonlinear classifier via the kernel trick. This method, called online imbalanced learning with kernels (OILK), maintains a buffer to store the informative support vectors. Two buffer update policies, first-in-first-out and reservoir sampling were investigated. As the cost of determining the AUC is very large, most of these algorithms avoid the exact computation of the AUC and resort to online or one-pass approaches by making use of buffers to store the relevant information. Although the storage requirements are reduced for such methods, generalization AUC performance of the resulting classifiers is not comparable with that of the nonlinear classifiers designed using batch learning algorithms on many real world datasets.

Motivated by the above observations, we propose an algorithm to design a sparse nonlinear classifier by maximizing the AUC using a max-margin based surrogate loss function and greedily adding the required number of basis functions into the classifier model [7], [8]. The AUC performance of the resulting sparse classifier is comparable with that obtained by using a batch learning algorithm. Further, we observed that the level of sparsity achievable by this classifier can be an order of magnitude larger than that achieved by Kernel RankSVM without significant degradation in the AUC performance. This helps to achieve significant speed-up during inference. Due to the nature of our algorithm, parallelization is possible and we demonstrate that significant training speed-up is achievable by using a multi-core version of the algorithm. A supplementary material providing the details of efficient computations of different quantities used in our algorithm is available at http://drona.csa.iisc.ernet.in/~shirish/SDM2017/supplement.pdf.

A word about our notations. All vectors will be column vectors and the row vectors will be denoted by a superscript, ^{T}. The 2-norm of the vector $\boldsymbol{x}$ is denoted by $\|\boldsymbol{x}\|$. $|J|$ denotes the cardinality of the set J. $\boldsymbol{K}$ denotes the kernel matrix. $\boldsymbol{K}_{I,J}$ refers to the submatrix of $\boldsymbol{K}$ made of the rows indexed by I and the columns indexed by J. $\boldsymbol{K}_{i,\cdot}$ refers to the i-th row of $\boldsymbol{K}$ and $\boldsymbol{K}_{\cdot,j}$ denotes the j-th column of $\boldsymbol{K}$.

2 Problem Definition

Let the training data be denoted by $\mathcal{D} = P \cup N$, where $P = \{\boldsymbol{x}_i^+, +1\}_{i=1}^p$, $N = \{\boldsymbol{x}_j^-, -1\}_{j=1}^n$ and $\boldsymbol{x}_i^+, \boldsymbol{x}_j^- \in R^d$. Without loss of generality, we assume that $p \ll n$. We will denote the q^{th} training set example as $\boldsymbol{x}_q$. Let T denote the ordered index set of pairs of positive and negative examples in $\mathcal{D}$. Clearly, $|T| = pn$. Let $l = p + n$. We assume that the nonlinear decision function $f(\cdot)$ is an element of a Reproducing Kernel Hilbert Space (RKHS). That is, f is a linear combination of kernel functions,

(2.1)
$$f(\boldsymbol{x}) = \boldsymbol{w}^T\phi(\boldsymbol{x}) = \sum_{q=1}^{l}\beta_q\phi(\boldsymbol{x}_q)^T\phi(\boldsymbol{x}) = \sum_{q=1}^{l}\beta_q k(\boldsymbol{x}, \boldsymbol{x}_q),$$

where $\phi(\cdot)$ maps the data into a high dimensional space and $k(\cdot,\cdot)$ denotes a kernel function. The AUC score (or performance) of the function f on the dataset $\mathcal{D}$ is defined as

$$\begin{aligned}\text{AUC}(f) &= \frac{\sum_{i=1}^p\sum_{j=1}^n I(f(\boldsymbol{x}_i^+) > f(\boldsymbol{x}_j^-))}{pn} \\ &= 1 - \frac{\sum_{i=1}^p\sum_{j=1}^n I(f(\boldsymbol{x}_i^+) \le f(\boldsymbol{x}_j^-))}{pn} \qquad (2.2)\end{aligned}$$

where $I(\cdot)$ is the indicator function which outputs 1 if the argument is true and 0 otherwise. Thus maximizing AUC(f) is equivalent to minimizing $\sum_{i=1}^p\sum_{j=1}^n I(f(\boldsymbol{x}_i^+) \le f(\boldsymbol{x}_j^-))$. Writing $f(\boldsymbol{x}_i) = (\boldsymbol{K}\beta)_i$ and using a max-margin based surrogate loss function (a hinge or a squared hinge loss), we get the following two regularized formulations corresponding to the two loss functions:

$$\text{(2.3)}\quad \min_{\beta\in R^l} \frac{1}{2}\beta^T\boldsymbol{K}\beta + C\sum_{(i,j)\in T}\max(0, 1-(\boldsymbol{K}\beta)_i + (\boldsymbol{K}\beta)_j)$$

and

(2.4)
$$\min_{\beta\in R^l} \frac{1}{2}\beta^T\boldsymbol{K}\beta + \frac{C}{2}\sum_{(i,j)\in T}\max(0, 1-(\boldsymbol{K}\beta)_i + (\boldsymbol{K}\beta)_j)^2$$

where C is a hyperparameter that controls the loss.

In this work, we focus on problem (2.4) as it uses a continuously differentiable function and devise an efficient algorithm to solve it. Unlike typical classification problems where a loss function can be calculated for every single training set example, the second term in (2.4) involves losses defined over pairs of examples from different classes. This makes solving the problem (2.4) more challenging.

3 Related Work

We now briefly review some of the related works for the AUC optimization.

Many online algorithms have been proposed to learn a linear classifier by maximizing the AUC score. These algorithms include Online AUC Maximization (OAM) [21] and Adaptive Online AUC Maximization (AdaOAM) [4]. AUC optimization in online learning is a challenging task as the computation of the AUC score involves the sum of pairwise losses between instances from opposite classes. To tackle this challenge, online learning algorithms use the idea of buffer sampling [21] [11]. A fixed size buffer is used to represent all the observed data by storing some randomly sampled examples in it. Kar et al. [11] introduced the idea of stream subsampling with replacement as the buffer update strategy. Although these online algorithms have demonstrated good AUC performance by using simple online gradient descent approaches, they do not use the geometrical knowledge of the observed data. AdaOAM overcomes this limitation by employing an adaptive gradient method that exploits the knowledge of historical gradients. Its variant, SAdaOAM was proposed to design a sparse model in an online AUC maximization task. Gao et al. [6] proposed a one-pass optimization algorithm by considering squared error loss for the AUC optimization. Due to the use of squared error loss, the algorithm only needs to store the first and second order statistics for the observed data.

A main drawback of the online methods discussed above is that they learn a linear classifier and do not exploit the learning power of kernel methods. To address this issue, Yang et al. [20] investigated Online Imbalanced Learning with Kernels (OILK) where informative support vectors are stored in the buffer. Two buffer update strategies, First-In-First-Out (FIFO) and Reservoir Sampling (RS) were investigated. By conducting experiments on real-world datasets, it was demonstrated that the kernel methods for AUC maximization performed better than their linear classifier counterparts. The proposed method [20] is however an online algorithm and we observed that the generalization performance of the resulting classifier was not comparable with that of a batch learned nonlinear classifier on many real-world datasets.

Joachims [10] presented a structural SVM framework for optimizing the AUC in a batch mode. By formulating (2.3) as a 1-slack structural SVM problem, the dual problem was solved by a cutting plane method. The method, though initially designed for linear classifiers, can be easily extended to learn nonlinear classifiers. Numerical experiments showed that, for ranking learning problems, this method is slower than other state-of-the-art methods that solve (2.3) directly [14] [15].

Learning to rank is an important supervised learning problem and has applications in a variety of domains such as information retrieval and online advertising. By assigning the same query number to all the examples, the Kernel RankSVM problem, discussed in [14], is same as (2.4). Kuo et al. [14] used trust region Newton method to solve this problem. This batch learning method requires to store the full kernel matrix as repeated kernel evaluations are bottleneck in Kernel RankSVM. This method has two drawbacks: 1) It is not scalable as the memory requirement is prohibitively high for large datasets, and 2) The learned non-sparse model results in computationally expensive predictions.

4 Our Approach: Sparse Kernel AUC

Our aim is to design a sparse nonlinear classifier model for a binary classification problem with imbalanced data distributions for the two classes. We now discuss our approach to solve (2.4). A similar problem formulation was used in [14] to solve the problem of learning to rank and the algorithm designed there is also applicable to our setting. Kuo et al. [14] alleviated the difficulty of computing the loss term, which involves summation over preference pairs, by using order-statistic trees. Although the cost of computing the required quantities was reduced to $O(l \log l)$ from $O(l^2)$, the kernel evaluations amount to $O(ld)$ time, which can be reduced to $O(l)$ if the kernel matrix $\boldsymbol{K}$ is maintained throughout the optimization algorithm. In their implementation, Kuo et al. [14] store the full kernel matrix $\boldsymbol{K}$ which is a dense matrix of size $l \times l$. However, for large datasets it is impractical to store the full kernel matrix $\boldsymbol{K}$ in the main memory. Further, for such huge datasets, the resulting classifier may not be sparse, thereby making the inference slow. It is therefore desired to devise a different approach to solve (2.4) and design a sparse classifier.

Motivated by the success of the matching pursuit approach, presented by Keerthi et al. [12] to design sparse SVM classifiers, we propose a new and efficient algorithm to solve (2.4) using matching pursuit ideas. Starting with an empty model, the matching pursuit algorithm uses a greedy approach to add a desired (d_{max}) number of basis functions to the model, by making use of the objective function decrease for basis function selection. The algorithm requires to compute and maintain the kernel matrix of size $l \times d_{max}$ (where d_{max} is the user specified positive parameter whose value can be about $5 - 10\%$ of the dataset size l) which helps to reduce the memory requirement considerably. For the dataset with $l = 49,990$, we observed that $d_{max} \approx 200$ was sufficient to achieve very good AUC performance on the test set.

We also demonstrate that efficient computations of the objective function in (2.4), gradient and Hessian-vector product computations are done by using simple techniques like sorting, binary search and hashing and do not require the use of sophisticated data structures such as order-statistic trees. As our experimental results show, the proposed approach achieves comparable generalization performance using smaller number of basis functions.

We now discuss the key components of our proposed algorithm.

4.1 Reformulation: Borrowing the ideas presented in [12], we maintain a set of greedily chosen kernel basis functions to design a sparse nonlinear classifier. The maximum cardinality of this set is denoted by d_{max}, a user specified positive integer. Let J denote the index set of these basis functions. In our experiments, we choose $J \subseteq \{1, 2, \ldots, l\}$. Having defined the set J, the parameter vector $\boldsymbol{w}$ in (2.1) can be represented as

$$\boldsymbol{w} = \sum_{q \in J} \beta_q \phi(\boldsymbol{x}_q)$$

and the problem formulation in (2.4) can be written as

(4.5)
$$\min_{\beta_J \in R^{|J|}} \boldsymbol{E}(\beta_J) \equiv \frac{\beta_J^T \boldsymbol{K}_{J,J} \beta_J}{2} + \frac{C}{2} \sum_{(i,j) \in T} \max(0, 1 - \boldsymbol{K}_{i,J}\beta_J + \boldsymbol{K}_{j,J}\beta_J)^2$$

Note that the Kernel RankSVM algorithm [14] solves the following problem:

(4.6)
$$\min_{\beta \in R^l} \frac{1}{2} \beta^T \boldsymbol{K} \beta + \frac{C}{2} \sum_{(i,j) \in S} max(0, 1 - (\boldsymbol{K}\beta)_i + (\boldsymbol{K}\beta)_j)^2$$

where $S = \{(i,j) | q_i = q_j, y_i > y_j\}$ is the set of preference pairs for queries q. This problem requires either to store the full kernel matrix $\boldsymbol{K}$ or requires many kernel evaluations, which is computationally expensive for large datasets. On the other hand, the solution to our problem (4.5) requires to store the matrix of size $l \times d_{max}$, which makes our approach scalable.

As our aim is to design a sparse nonlinear classifier, we solve (4.5) using matching pursuit ideas [19], [12]. In this approach, starting with an empty model ($J = \phi$), a training example (also called a basis function) is greedily chosen, using some criterion, from the set $\{1, 2, \ldots, l\} \setminus J$ and included in the set J. The optimization problem (4.5) is then solved with respect to β_J. This procedure is repeated until $|J| = d_{max}$ holds true. Algorithm 1 gives the pseudo-code of this procedure. We now give details related to step 3 (basis function selection) and step 5 (optimization with respect to β_J) of this algorithm.

Algorithm 1: Sparse Classifier Design Algorithm

```
Input: $\mathcal{D} = \{\boldsymbol{x}_i^+, +1\}_{i=1}^p \cup \{\boldsymbol{x}_j^-, -1\}_{j=1}^n$, C, $d_{max}$
Output: $J$, $\beta_J$
1: $J = \phi$
2: while $|J| < d_{max}$ do
3:     Select a new basis function $j^* \in \{1, 2, \ldots, l\} \setminus J$
4:     $J := J \cup \{j^*\}$
5:     Solve (4.5) w.r.t $\beta_J$
6: end while
```

4.2 Basis Selection: We now discuss a systematic and efficient procedure to select basis functions. Given the basis function set J and the corresponding model parameters β_J, a new basis function $j^* \in \{1, 2, \ldots, l\} \setminus J$ is chosen such that its inclusion in the set J would result in a maximum improvement in the objective function in (4.5). A straightforward method is to choose every $q \in \{1, 2, \ldots, l\} \setminus J$, solve (4.5) completely w.r.t. (β_J, β_q) and calculate the improvement in the objective function, $\Delta \boldsymbol{E}_q$. Step 3 of Algorithm 1 is,

$$j^* = \text{argmin}_{q \in \{1,2,\ldots,l\} \setminus J} \, \Delta \boldsymbol{E}_q.$$

But solving (4.5) completely by adding every possible basis function is computationally expensive as it requires to solve a $(|J|+1)$-dimensional optimization problem for every candidate basis function. Instead, it may be good idea to fix β_J and solve (4.5) using only β_q, to determine an approximate value of $\Delta \boldsymbol{E}_q$. This problem is easy to solve as it is a one-dimensional problem:

$$\min_{\beta_q} \frac{1}{2} \begin{pmatrix} \beta_J^T & \beta_q \end{pmatrix} \begin{pmatrix} \boldsymbol{K}_{J,J} & \boldsymbol{K}_{J,q} \\ \boldsymbol{K}_{q,J} & \boldsymbol{K}_{q,q} \end{pmatrix} \begin{pmatrix} \beta_J \\ \beta_q \end{pmatrix} +$$

(4.7)
$$\frac{C}{2} \sum_{(i,j) \in T} max(0, 1 - (\boldsymbol{K}_{i,J} - \boldsymbol{K}_{j,J})\beta_J - (\boldsymbol{K}_{i,q} - \boldsymbol{K}_{j,q})\beta_q)^2.$$

For practical purposes, as suggested in [12], one can do a few iterations of Newton-Raphson method on the derivative of the objective function to get a near optimal solution in $O(l^2)$ time.

If all $j \notin J$ are tried, then the complexity of selecting a new basis function is $O(l^3)$ which is very large. To speed up the basis function selection step, one can simply choose $\boldsymbol{\kappa}$ random basis functions as candidate basis functions. After some experiments, we found that $\boldsymbol{\kappa} = 100$ was a good choice.

4.3 Optimization: The function $\boldsymbol{E}(\beta_J)$ in (4.5) can be optimized using any second order optimization method. Each update step in the classical Newton Method requires the computation of the Hessian and its inverse, which are expensive both in terms of storage

Algorithm 2: Truncated Newton Method

```
Input: J, current $\beta_J$
Output: Optimized $\beta_J$
1: $\beta_J^0 = \beta_J$, k = 0
2: while stopping condition is not satisfied at $\beta_J^k$ do
3:     Compute a search direction $d^k$ by applying
       Conjugate-Gradient method to solve
       $\nabla^2 \boldsymbol{E}(\beta_J)d = -\nabla \boldsymbol{E}(\beta_J)$
4:     Find $\alpha^k$ satisfying Armijo backtracking
       conditions
5:     Update $\beta_J^{k+1} = \beta_J^k + \alpha^k d^k$
6:     $k := k + 1$
7: end while
```

and computations. Therefore, we resorted to Truncated Newton Method. Algorithm 2 gives the details.

This method does not require explicit knowledge of the Hessian matrix, $\nabla^2 \boldsymbol{E}(\beta_J)$. Rather, it is enough if we can provide matrix-vector products of the form $\nabla^2 \boldsymbol{E}(\beta_J)\boldsymbol{v}$ for any given vector $\boldsymbol{v}$. Thus, the speed of this method depends on the efficient computations of the objective function value $\boldsymbol{E}(\beta_J)$, its gradient $\nabla \boldsymbol{E}(\beta_J)$ and Hessian-vector product $\nabla^2 \boldsymbol{E}(\beta_J)\boldsymbol{v}$ for any vector $\boldsymbol{v} \in R^{|J|}$. If $\boldsymbol{A}$ denotes a pairwise indexing matrix and $\boldsymbol{A}_{\beta_J}$ denotes the indexing matrix of violating pairs which contribute to the loss function (details given in the supplementary material), then by defining

$$\boldsymbol{u}_{\beta_J} = \boldsymbol{A}_{\beta_J}^T \boldsymbol{A}_{\beta_J} \boldsymbol{K}_{\cdot,J}\beta_J, \tag{4.8}$$

the problem in (4.5) can be re-written as

$$\min_{\beta_J} \boldsymbol{E}(\beta_J) \equiv \frac{1}{2}\beta_J^T \boldsymbol{K}_{J,J}\beta_J + \frac{C}{2}(\beta_J^T \boldsymbol{K}_{\cdot,J}^T(\boldsymbol{u}_{\beta_J} - 2\boldsymbol{A}_{\beta_J}^T \boldsymbol{e}_{\beta_J}) + p_{\beta_J}). \tag{4.9}$$

where p_{β_J} is the number of violating pairs. This rewriting helps in computing $\boldsymbol{E}(\beta_J)$, $\nabla \boldsymbol{E}(\beta_J)$ and $\nabla^2 \boldsymbol{E}(\beta_J)\boldsymbol{v}$ efficiently as all of these quantities require the computation of $\boldsymbol{u}_{\beta_J}$. By defining

$$SV(\beta_J) = \{(i,j) \in T \mid 1 - \boldsymbol{K}_{i,J}\beta_J + \boldsymbol{K}_{j,J}\beta_J > 0\} \tag{4.10}$$

and

$$SV_i^+(\beta_J) \equiv \{j \mid (j,i) \in SV(\beta_J)\},\ l_i^+(\beta_J) \equiv |SV_i^+(\beta_J)|,$$
$$\gamma_i^+(\beta_J, \boldsymbol{v}) \equiv \textstyle\sum_{j \in SV_i^+(\beta_J)} \boldsymbol{K}_{j,J}^T \boldsymbol{v},$$
$$SV_i^-(\beta_J) \equiv \{j \mid (i,j) \in SV(\beta_J)\},\ l_i^-(\beta_J) \equiv |SV_i^-(\beta_J)|,$$
$$\gamma_i^-(\beta_J, \boldsymbol{v}) \equiv \textstyle\sum_{j \in SV_i^-(\beta_J)} \boldsymbol{K}_{j,J}^T \boldsymbol{v}.$$

one can compute $\boldsymbol{u}_{\beta_J}$ efficiently (details given in the supplementary material).

Lee et al. [15] and Airola et al. [1] used order-statistic trees to efficiently compute the $l_i^+(\beta_J)$ and $l_i^-(\beta_J)$ for Kernel RankSVM. The problem of maximizing the AUC does not require order-statistic trees. It is enough to use sorting, searching and hashing methods. The details are given in Algorithm 3.

Algorithm 3: Calculating $l_i^+(\beta_J)$, $l_i^-(\beta_J)$, $\gamma_i^+(\beta_J, \boldsymbol{v})$, and $\gamma_i^-(\beta_J, \boldsymbol{v})$

```
Input: $\boldsymbol{K}_{\cdot,J}$, $\beta_J$, $\boldsymbol{v}$, P, N
Output: $l_i^+(\beta_J)$, $l_i^-(\beta_J)$, $\gamma_i^+(\beta_J, \boldsymbol{v})$ and $\gamma_i^-(\beta_J, \boldsymbol{v})$
1: scoreP = $zeros(2, |P|)$, scoreN = $zeros(2, |N|)$
2: scoreP[1] = $\boldsymbol{K}_{iJ} * \beta_J$, for all $i \in P$
3: scoreP[2] = $\boldsymbol{K}_{iJ} * \boldsymbol{v}$, for all $i \in P$
4: sort scoreP w.r.t to first row
5: scoreN[1] = $\boldsymbol{K}_{jJ} * \beta_J$, for all $j \in N$
6: scoreN[2] = $\boldsymbol{K}_{jJ} * \boldsymbol{v}$, for all $j \in N$
7: sort scoreN w.r.t to first row
8: scorePsum = scoreP[2], scoreNsum = scoreN[2]
9: for $i = 2$ to $|P|$ do
10:    scorePsum[i] = scorePsum[i] + scorePsum[i-1]
11: end for
12: for $i = |N| - 1$ to 1 do
13:    scoreNsum[i] = scoreNsum[i] + scoreNsum[i+1]
14: end for
15: for $i = 1$ to $|P|$ do
16:    score = $(K_{i,J} * \beta_J)$ -1
17:    find the index k of scoreN using binary search
       s.t. $scoreN[k-1] < score \leq scoreN[k]$
18:    $l_i^-(\beta_J) = length(k : |N|)$
19:    $\gamma_i^-(\beta_J, \boldsymbol{v}) = scoreNsum[k]$
20: end for
21: for $j = 1$ to $|N|$ do
22:    score = $(K_{j,J} * \beta_J)$ +1
23:    find the index k of scoreP using binary search
       s.t. $scoreP[k] \leq score < scoreP[k+1]$
24:    $l_j^+(\beta_J) = k$
25:    $\gamma_j^+(\beta_J, \boldsymbol{v}) = scorePsum[k]$
26: end for
```

For a given β_J, we define the set of ordered pairs which contributes to the empirical loss of the objective function in (4.5) as $SV(\beta_J)$. For every example in the training set, by finding out the set of violating examples of the other class (SV^+ and SV^-) and the quantities γ^+ and γ^-, we can compute the empirical loss term in (4.5). These computations can be done efficiently by using sorting (Steps 1-7), hashing (Steps 9-14) and searching (Steps 15-26). The complexity of this algorithm is $O(l(d_{max} + \log l))^1$, which is better than naive computation of pairwise losses in (4.5). Further, in our experiments, we implemented steps 15-26 of Algorithm 3 in multi-core setting. This resulted in a significant speed up of our algorithm, which is evident

[1] ld_{max} is for computing $\boldsymbol{K}_{\cdot,J}$ and $l \log l$ is for sorting

from the empirical evaluation discussed in the next section.

We have not discussed the details of the Conjugate Gradient iteration (Step 3 of Algorithm 2). The details can be found in [2]. There are many variations around it, all of them rely on Hessian vector multiplications. In our implementation we used the **minres** function from MATLAB to get the direction in Step 3 of Algorithm 2.

4.4 Computational Complexity: Assuming that the kernel matrix $\boldsymbol{K}$ is stored in main memory, the computation of the loss term in (4.5) will require $O(l(d_{max} + \log l))$ computation time. On the other hand the corresponding term in (4.6), used by Kernel RankSVM, requires the computation time of $O(l^2)$. For large datasets it may not be feasible to store $\boldsymbol{K}$ in main memory. Therefore, for such datasets, Kernel RankSVM resorts to several block-wise computations of $\boldsymbol{K}$, which may result in increased training time. This problem does not arise in our approach, as the maximum sub-matrix of $\boldsymbol{K}$ that it needs to store is of size $l \times d_{max}$.

5 Empirical Evaluation

In this section, we discuss the experimental evaluations of the proposed algorithm for sparse nonlinear classifier design.[2] In particular, we demonstrate that the proposed Sparse Kernel AUC algorithm results in a sparser classifier and gives comparable generalization performance with the Kernel RankSVM algorithm. Further, we also observed that batch learning algorithms perform better (in terms of AUC performance) than online learning algorithms on majority of real world datasets.

In our experiments, we used the Gaussian kernel function, $K(x_i, x_j) = exp(-\frac{1}{2\sigma^2}\|x_i - x_j\|^2)$ where $\sigma > 0$, for all the experiments. The kernel parameter σ and regularization hyper-parameter C were tuned using cross-validation. For this, a grid of (C, σ) values, where $C \in \{10^{-5}, 10^{-4}, ..., 10^5\}$ and $\sigma \in \{2^{-5}, 2^{-4}, ..., 2^5\}$ was searched. The results reported correspond to the (C, σ) pair which gave the best validation set performance. The value of d_{max} was set to l. The proposed algorithm was terminated when $|J| = d_{max}$ was true or there was not a significant change in the validation set AUC performance. As our results demonstrate, the number of basis functions selected by the proposed algorithm was less than 10 % of the training data set size on most of the datasets. All the experiments were performed using MATLAB implementations on a Intel(R) Xeon(R) CPU E5620@2.40GHz machine with 16 cores and 16 GB main memory under Linux.

We compare the following methods: 1) Sparse Kernel AUC: our proposed sparse AUC optimization approach discussed in Section 4, 2) Kernel RankSVM: an extension of Kernel RankSVM method, discussed in [14], to the AUC optimization problem, 3) Online Imbalanced Learning with Kernels (OILK) [20], and 4) Adaptive Gradient Method for Online AUC Maximization (AdaOAM) [4]. The performance of these methods was compared in terms of the AUC score on the test set (if a test set is available). If the test set is not explicitly available, AUC score on validation set, averaged over 4 independent runs of five-fold splits of each dataset, is reported. Since the aim of this paper is to design a nonlinear sparse classifier model using AUC optimization, we report the number of basis functions present in the final model for batch learning methods: Sparse Kernel AUC and Kernel RankSVM. The other two methods use an online learning approach and it may not be fair to compare the number of basis functions obtained using them with those obtained using batch learning methods. CPU time comparison of batch learning methods, Sparse Kernel AUC and Kernel RankSVM, was not done as the implementations were done using MATLAB and C programming language respectively. The computational complexity of these two methods was discussed in Section 4.4.

We used 14 benchmark datasets to compare our proposed method, Sparse Kernel AUC, with the other three methods. The dataset details are given in Table 1. The datasets are available at UCI[3] or LIBSVM[4] dataset repositories. Some multi-class datasets (glass, vehicle and poker) were converted to class imbalanced binary datasets. For some datasets mentioned in Table 1, test set was not available.

Effect of retraining and $\boldsymbol{\kappa}$: To make Algorithm 1 efficient, it may be a good idea to perform optimization in step 5 only from time to time. We experimented with 3 retraining strategies where step 5 is executed after the addition of 1) every basis function (i.e always), 2) $|J| = \lfloor 2^{0.25} \rfloor$ basis functions and 3) $|J| = 2^j$, $j = 0, \ldots,$ basis functions. The results are presented in Figure 1. It is clear from this figure that, always retraining increases the training time. Similar generalization performance is achieved in other cases of retraining. We found that $\lfloor 2^{0.25} \rfloor$ was a good choice across many datasets and used it in our experiments.

As mentioned in Section 4.2, instead of choosing a possible basis function from $\{1, 2, \ldots, l\} \setminus J$, we chose a subset $\boldsymbol{\kappa}$ of examples from this set as possible candidates

[2]The MATLAB code for the proposed algorithm is available at https://www.dropbox.com/s/6e4fsj2hlq1b71n/Code.rar?dl=0

[3]https://archive.ics.uci.edu/ml/datasets.html

[4]https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/

Datasets	l	Test set size	d	n/p
sonar	208	-	60	1.144
glass	214	-	9	2.057
ionosphere	351	-	34	1.785
balance	625	-	4	11.755
australian	690	-	14	1.247
vehicle	846	-	18	3.251
fourclass	862	-	2	1.807
svmguide3	1,243	-	22	3.199
a2a	2,265	-	123	2.959
magic04	19,020	-	10	1.843
segment	210	2,100	19	6.000
satimage	4,435	2,000	36	9.279
ijcnn1	49,990	91,701	22	10.0
poker	25,010	1,000,000	11	20.0

Table 1: Details of datasets

for basis functions. Different values of $\boldsymbol{\kappa}$ (1, 10, and 100) were tried. The results are shown in Figure 2. Although these 3 values of $\boldsymbol{\kappa}$ resulted in similar steady state generalization performance, it was observed that for $\boldsymbol{\kappa} = 100$, steady state generalization performance was achieved faster. So, $\boldsymbol{\kappa} = 100$ was a good choice.

Discussion: From Tables 2 and 3, we observe that the generalization performance of the proposed Sparse Kernel AUC method is comparable with that of the Kernel RankSVM method. A small degradation in the performance of Sparse Kernel AUC method is due to reduced model complexity. Both these batch learning methods perform significantly better than the OILK method on ionosphere, fourclass and satimage datasets. The kernel based methods, Sparse Kernel AUC, Kernel RankSVM and $OILK$ perform better than linear classifier based method (AdaOAM) on majority of the datasets.

The proposed method required smaller number of basis functions than those required by Kernel RankSVM to achieve comparable AUC performance. Thus the proposed method is recommended for designing sparse classifiers for large datasets. Note that the reduction in the number of basis functions is two orders of magnitude in case of some large datasets (magic04, poker and ijcnn1).

Experiments in Multi-core Setting. To study the speed-up of our proposed algorithm in multi-core environment, we parallelized steps 15-26 of Algorithm 3. The speed-up was studied on three large datasets by gradually increasing the number of cores from 1 to 16. Figure 3 depicts the time comparison. It is clear from this figure that significant speed-up can be obtained by running our method in multi-core environment. The speed-up is noticeable on large datasets like ijcnn1. Detailed investigation is however needed to study the parallelization of the complete proposed algorithm.

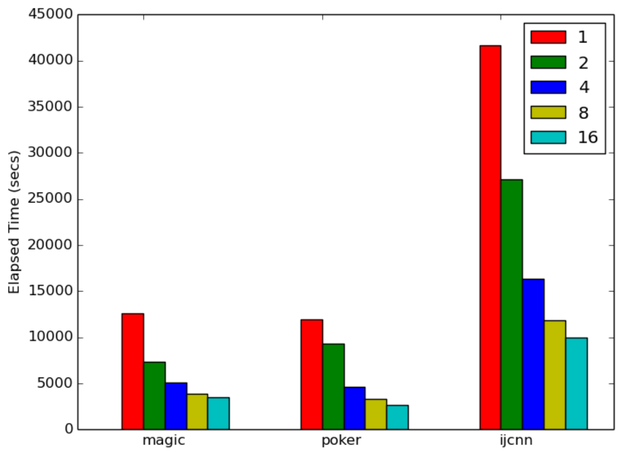

Figure 3: Effect of increasing the numbers of core on time is shown for 3 benchmark datasets.

6 Conclusion

This paper studied a new and efficient learning algorithm to design a sparse nonlinear classifier using AUC maximization. The algorithm tackles the challenge of larger training times of kernel methods by greedily adding the required number of basis functions in the model. We demonstrated that the resulting sparse classifier achieved comparable generalization performance with that achieved by Kernel RankSVM. On many large datasets, it was observed that the proposed algorithm results in using significantly small number of basis functions in the model. We also demonstrated that batch learning algorithms for AUC optimization perform better than online algorithms on many datasets. We are currently investigating the extension of these ideas to a distributed setting.

References

[1] Antti Airola, Tapio Pahikkala, and Tapio Salakoski. Training linear ranking SVMs in linearithmic time using red–black trees. *Pattern Recognition Letters*, 32(9):1328–1336, 2011.

[2] Richard Barrett, Michael W Berry, Tony F Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk Van der Vorst. *Templates for the solution of linear systems: building blocks for iterative methods*, volume 43. SIAM, 1994.

[3] Toon Calders and Szymon Jaroszewicz. Efficient AUC optimization for classification. In *PKDD 2007*, pages 42–53. Springer, 2007.

Datasets	Sparse Kernel AUC	Kernel RankSVM	*OILK*	AdaOAM
sonar	0.914 ± 0.043 **(105)**	*0.951 ± 0.029* (167)	0.929 ± .039	-
glass	0.871 ± 0.054 **(150)**	*0.881 ± 0.051* (171)	-	0.816 ± 0.058
ionosphere	0.980 ± 0.017 **(182)**	*0.987 ± 0.014* (281)	0.954 ± 0.021	-
balance	*1.000 ± 0.000* **(6)**	*1.000 ± 0.000* (500)	-	0.579 ± 0.106
australian	0.913 ± 0.034 **(256)**	*0.930 ± 0.020* (552)	0.925 ± 0.021	0.927 ± 0.016
vehicle	0.977 ± 0.022 **(431)**	*0.995 ± 0.002* (677)	-	0.818 ± 0.026
fourclass	0.999 ± 0.000 **(108)**	*1.000 ± 0.000* (690)	0.829 ± 0.036	-
svmguide3	0.823 ± 0.027 **(216)**	*0.824 ± 0.026* (995)	-	0.734 ± 0.038
a2a	*0.880 ± 0.009* **(64)**	*0.880 ± 0.010* (1741)	-	0.873 ± 0.019
magic04	0.874 ± 0.023 **(182)**	*0.894 ± 0.007* (15124)	-	0.798 ± 0.007

Table 2: Validation set AUC Performance (mean ± s.d.) and maximum number of basis functions (in parenthesis) comparison of various methods. The AUC performance numbers for *OILK* and AdaOAM are reported from [4] and [20] respectively.

Datasets	Sparse Kernel AUC	Kernel RankSVM	*OILK*	AdaOAM
segment	0.996 **(91)**	*0.998* (210)	0.997 ± 0.003	-
satimage	0.961 **(431)**	*0.969* (4435)	0.896 ± 0.024	-
ijcnn1	0.995 **(182)**	*1.000* (49990)	-	-
poker	0.668 **(363)**	*0.674* (25010)	-	0.571 ± 0.007

Table 3: Test set AUC Performance and number of basis functions (in parenthesis) comparison of various methods. The AUC performance numbers for *OILK* and AdaOAM are reported from [4] and [20] respectively.

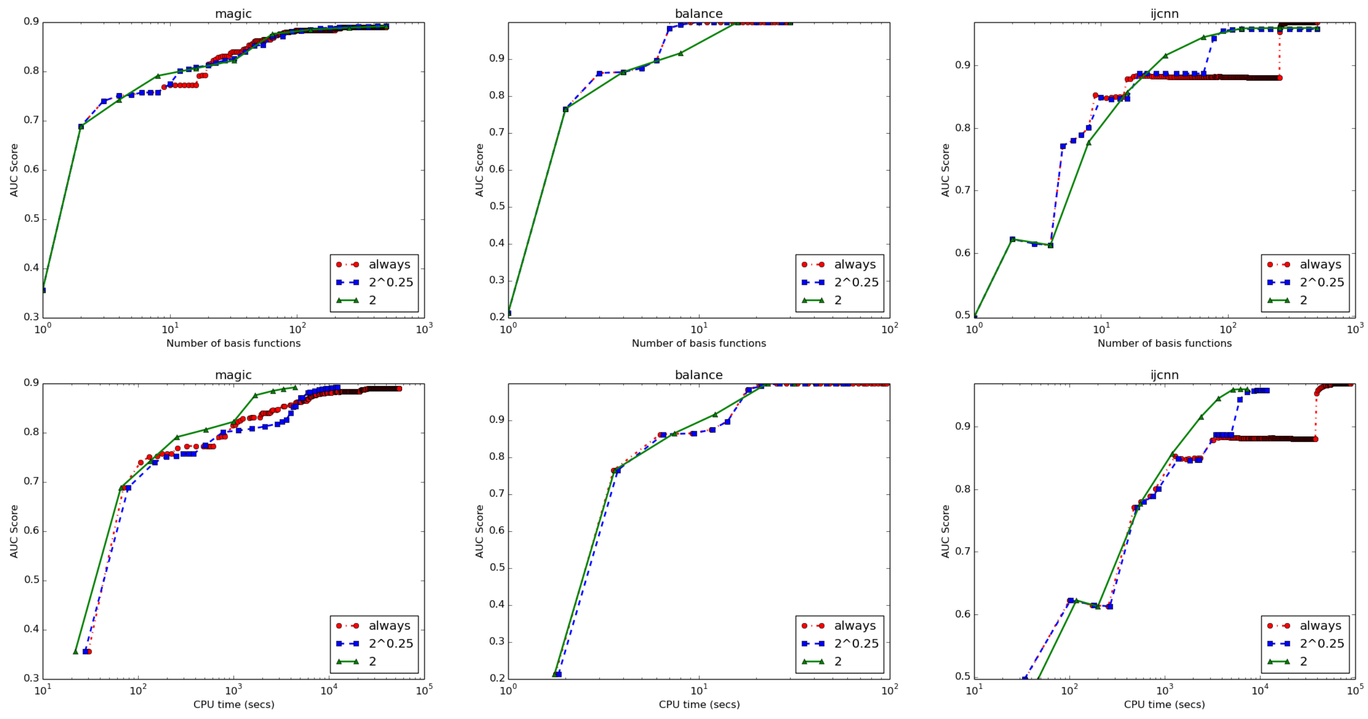

Figure 1: Three different retraining strategies showing a different trade-off between AUC and time, always retraining is too time consuming.

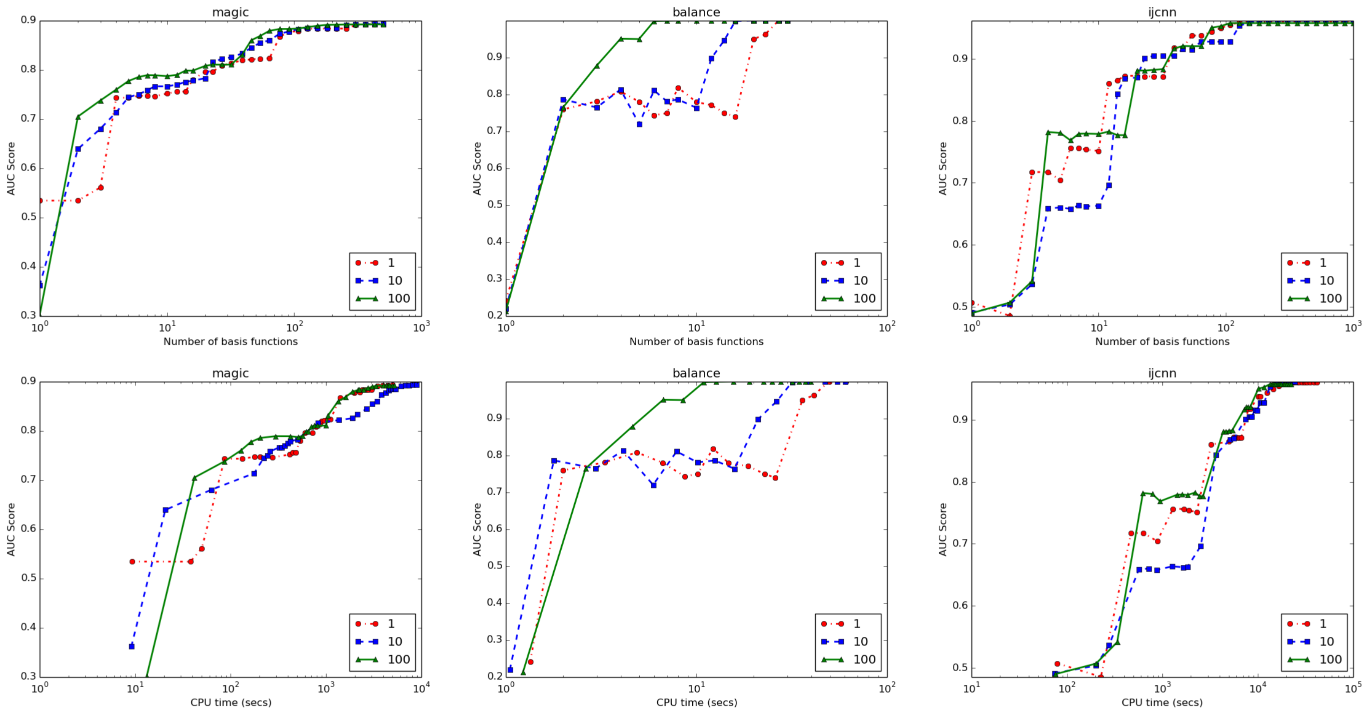

Figure 2: Influence of the parameter κ: performance is not much affected, but the computational cost is a bit larger when $\boldsymbol{\kappa}$=1. $\boldsymbol{\kappa} = 100$ seems a good choice.

[4] Yi Ding, Peilin Zhao, Steven CH Hoi, and Yew-Soon Ong. Adaptive subgradient methods for online AUC maximization. *arXiv:1602.00351*, 2016.

[5] Tom Fawcett. Using rule sets to maximize ROC performance. In *IEEE ICDM*, pages 131–138, 2001.

[6] Wei Gao, Rong Jin, Shenghuo Zhu, and Zhi-Hua Zhou. One-pass AUC optimization. In *ICML*, 2013.

[7] Jie Gui, Tongliang Liu, Dacheng Tao, Zhenan Sun, and Tieniu Tan. Representative vector machines: A unified framework for classical classifiers. *IEEE Transactions on Cybernetics*, 46(8):1877–1888, 2016.

[8] Jie Gui, Zhenan Sun, Shuiwang Ji, Dacheng Tao, and Tieniu Tan. Feature selection based on structured sparsity: A comprehensive study. *IEEE Transactions on Neural Networks and Learning Systems*, Digital Object Identifier 10.1109/TNNLS.2016.2551724, 2016.

[9] Alan Herschtal and Bhavani Raskutti. Optimising area under the ROC curve using gradient descent. In *ICML*, 2004.

[10] Thorsten Joachims. A support vector method for multivariate performance measures. In *ICML*, pages 377–384. ACM, 2005.

[11] Purushottam Kar, Bharath K Sriperumbudur, Prateek Jain, and Harish C Karnick. On the generalization ability of online learning algorithms for pairwise loss functions. *arXiv:1305.2505*, 2013.

[12] S Sathiya Keerthi, Olivier Chapelle, and Dennis DeCoste. Building support vector machines with reduced classifier complexity. *The Journal of Machine Learning Research*, 7:1493–1515, 2006.

[13] Wojciech Kotlowski, Krzysztof J Dembczynski, and Eyke Huellermeier. Bipartite ranking through minimization of univariate loss. In *ICML*, pages 1113–1120, 2011.

[14] Tzu-Ming Kuo, Ching-Pei Lee, and Chih-Jen Lin. Large-scale kernel ranksvm. In *SDM*, pages 812–820. SIAM, 2014.

[15] Ching-Pei Lee and Chuan-bi Lin. Large-scale linear ranksvm. *Neural computation*, 26(4):781–817, 2014.

[16] Yi Lin, Yoonkyung Lee, and Grace Wahba. Support vector machines for classification in nonstandard situations. *Machine learning*, 46(1-3):191–202, 2002.

[17] Charles E Metz. Basic principles of ROC analysis. *Seminars in Nuclear Medicine*, 8(4):283–298, 1978.

[18] Cynthia Rudin and Robert E Schapire. Margin-based ranking and an equivalence between adaboost and rankboost. *The Journal of Machine Learning Research*, 10:2193–2232, 2009.

[19] Pascal Vincent and Yoshua Bengio. Kernel matching pursuit. *Machine Learning*, 48(1-3):165–187, 2002.

[20] Haiqin Yang, Junjie Hu, Michael R Lyu, and Irwin King. Online imbalanced learning with kernels. In *NIPS Workshop on Big Learning*, 2013.

[21] Peilin Zhao, Rong Jin, Tianbao Yang, and Steven C Hoi. Online AUC maximization. In *ICML*, pages 233–240, 2011.