Support Vector Machines with Sparse Binary High-Dimensional Feature Vectors

Kave Eshghi, Mehran Kafai

Hewlett Packard Labs
HPE-2016-30

Keyword(s):
SVM classification; Perceptron training; MinOver; CRO kernel

Abstract:
We introduce SparseMinOver, a maximum margin Perceptron training algorithm based on the MinOver algorithm that can be used for SVM training when the feature vectors are sparse, high-dimensional, and binary. Such feature vectors arise when the CRO feature map is used to map the input space to the feature space. We show that the training algorithm is efficient with this type of feature vector, while preserving the accuracy of the underlying SVM. We demonstrate the accuracy and efficiency of this technique on a number of datasets, including TIMIT, for which training a standard SVM with RBF kernel is prohibitively expensive. SparseMinOver relies on storing large indices and is particularly suited to large memory machines.
Support Vector Machines with Sparse Binary High-Dimensional Feature Vectors

Kave Eshghi
Hewlett Packard Labs
1501 Page Mill Rd.
Palo Alto, CA 94304
kave.eshghi@hpe.com

Mehran Kafai
Hewlett Packard Labs
1501 Page Mill Rd.
Palo Alto, CA 94304
mehran.kafai@hpe.com

ABSTRACT

We introduce SparseMinOver, a maximum margin Perceptron training algorithm based on the MinOver algorithm that can be used for SVM training when the feature vectors are sparse, high-dimensional, and binary. Such feature vectors arise when the CRO feature map is used to map the input space to the feature space. We show that the training algorithm is efficient with this type of feature vector, while preserving the accuracy of the underlying SVM. We demonstrate the accuracy and efficiency of this technique on a number of datasets, including TIMIT, for which training a standard SVM with RBF kernel is prohibitively expensive. SparseMinOver relies on storing large indices and is particularly suited to large memory machines.

Keywords

SVM classification; Perceptron training; MinOver; CRO kernel

1. INTRODUCTION

The accuracy of Support Vector Machine (SVM) classification with the RBF kernel has been shown to be superior to linear SVMs for many applications. For example, for the MNIST [11] handwritten digit recognition dataset, SVM with the RBF kernel achieves accuracy of 98.5%, whereas linear SVM can only achieve 92.7%. However, the theory behind kernel methods relies on a mapping between the input space and the feature space such that the inner product of the vectors in the feature space can be computed via the kernel function, aka the ‘kernel trick’. The kernel trick is used because a direct mapping to the feature space is expensive or, in the case of the RBF kernel, impossible, since the feature space is infinite dimensional.

For SVMs, the main drawback of the kernel trick is that both training and classification can be expensive. Training is expensive because the kernel function must be applied for each pair of the training samples, making the training task at least quadratic with the number of training samples. Classification is expensive because for each classification task the kernel function must be applied for each of the support vectors, whose number may be large. As a result, kernel SVMs are rarely used when the number of training instances is large or for online applications where classification must happen very fast. Many approaches have been proposed in the literature to overcome these efficiency problems with non-linear kernel SVMs. Section 2 mentions some of the related work.

In [2] we introduced a the kernel $\mathcal{K}_{\alpha}(A, B)$, called the CRO-kernel, that approximates the RBF kernel

\[ K_{\alpha} = \frac{\log(1 + |A - B|^2)}{\alpha} \]

on the unit sphere. We also introduced the randomized feature map $F_{\tau,q}(A)$, where $Q \in R^U \times U$ and $\tau = [\alpha U]$. We proved that when $Q$ is randomly chosen from a suitable distribution, as $U \to \infty$

\[ E_{U \to \infty} \left[ F_{\tau,q}(A) \cdot F_{\tau,q}(B) \right] = K_{\alpha}(A, B) \]  

The vectors generated by the feature map have $U$ elements. Of these, $\tau$ elements are 1, and the rest are 0. In all interesting cases when we want to emulate an RBF kernel, the ratio $\frac{\tau}{U}$ is very small, which means the vectors generated by the feature map are very sparse.

We use a highly efficient procedure described in [3, 8] to compute the feature map. The proposed kernel $K$ and feature map $F$ have interesting properties:

- The kernel approximates the RBF kernel on the unit sphere.
- The feature map is sparse, binary and high dimensional.
- The feature map can be computed very efficiently.

In this paper we introduce the SparseMinOver algorithm, which is an adaptation of the MinOver algorithm [9, 13] that is optimized for sparse, binary training vectors.

MinOver is a maximum margin Perceptron training algorithm that can be used for SVM training, as shown in [13]. SparseMinoOver can easily scale to millions of training vectors, while maintaining the accuracy of the underlying kernel SVM. We test our implementation on the TIMIT [4] and MNIST [11] datasets. On TIMIT, the size of the training set is 1,385,426 and the dimensionality of the input vectors is 792, which puts it beyond the pale for any standard RBF SVM trainer. With the CRO kernel and SparseMinOver, however, we can train the classifier in a few hours with state of the art accuracy. SparseMinOver relies on large indexes that need to be accessed in the inner loop, so it greatly benefits from large memory, flat address space architectures.

2. RELATED WORK

Reducing the training and classification cost of non-linear SVMs has attracted a great deal of attention in the literature. Joachims et.
al. [7] use basis vectors other than support vectors to find sparse solutions that speed up training and prediction. Segata et. al. [19] use local SVMs on redundant neighborhoods and choose the appropriate model at query time. In this way, they divide the large SVM problem into many small local SVM problems.

Tsang et. al. [21] re-formulate the kernel methods as minimum enclosing ball (MEB) problems in computational geometry, and solve them via an efficient approximate MEB algorithm, leading to the idea of core sets. Nandan et. al. [14] choose a subset of the training data, called the representative set, to reduce the training time. This subset is chosen using an algorithm based on convex hulls and extreme points.

A number of approaches compute approximations to the feature vectors and use linear SVM on these vectors. Chang et. al. [1] do so in time. This subset is chosen using an algorithm based on convex hulls and extreme points.

Weinberger et. al. [24] use hashing to reduce the dimensionality of the input vectors. Litayem et. al. [12] use hashing to speed up the prediction phase of linear SVM. Su et. al. [20] use sparse projection to reduce the dimensionality of the input vectors while preserving the kernel function.


Huang et. al. [6] apply kernel SVMs to the problem of phoneme classification for the TIMIT [4] dataset. The problem they address is similar to ours: a full RBF kernel SVM is impractical to train on this type of dataset. To ameliorate this, they try the random Fourier feature based method of [17], but they find that to achieve acceptable accuracy, the dimensionality of the feature space needs to be very high, in the order of 200,000, which makes it impractical to store and process the vectors in memory. They propose two ways of overcoming this problem: the first is to have an ensemble of weak learners, each working on a smaller dimensional feature space, and combine the results. The second approach is a scalable solver that does not require the storage of the whole matrix of the feature space vectors in memory; instead, it computes the feature vectors on demand as the training process unfolds.

By contrast, in our case even though the dimensionality of the vectors in the feature space is very high, the vectors are sparse, so they can be stored efficiently. What is more, the SparseMinOver training algorithm we present in this paper can efficiently train an SVM on data of this scale, with accuracy that is comparable with other state of the art techniques.

3. THE KERNEL AND FEATURE MAP

To better understand the rest of the paper we start with describing the notation.

3.1 Notation

3.1.1 $\Phi(x)$

We use $\Phi(x)$ to denote the CDF of the standard normal distribution $\mathcal{N}(0, 1)$, and $\phi(x)$ to denote its PDF.

$$\phi(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$$

$$\Phi(x) = \int_{-\infty}^{x} \phi(u) du$$

3.1.2 $\Phi_2(x, y, \rho)$

We use $\Phi_2(x, y, \rho)$ to denote the CDF of the standard bivariate normal distribution

$$\mathcal{N}\left(\left(\begin{array}{c} 0 \\ 0 \end{array}\right), \left(\begin{array}{cc} 1 & \rho \\ \rho & 1 \end{array}\right)\right)$$

and $\phi_2(x, y, \rho)$ to denote its PDF.

$$\phi_2(x, y, \rho) = \frac{1}{2\pi\sqrt{1-\rho^2}} e^{-\frac{x^2+y^2-2\rho xy}{2(1-\rho^2)}}$$

$$\Phi_2(x, y, \rho) = \int_{-\infty}^{x} \int_{-\infty}^{y} \phi_2(u, v, \rho) du dv$$

3.1.3 $R(x, A)$

We use $R(x, A)$ to denote the rank of $x$ in $A$, defined as follows:

**Definition 1.** For the scalar $x$ and vector $A \in \mathbb{R}^D$, $R(x, A)$ is the count of the of elements of $A$ which are less than or equal to $x$.

3.1.4 Vector Indexing

We use square brackets to index into vectors. For example $A[10]$ refers to the tenth element of the vector $A$. Indices start from one.

3.2 Concomitant Rank Order (CRO) Kernel

**Definition 2.** Let $A, B \in \mathbb{R}^D$. Let $0 \leq \alpha \leq 1$. Then the kernel $K_\alpha(A, B)$ is defined as:

$$K_\alpha(A, B) = \Phi_2(\Phi^{-1}(\alpha), \Phi^{-1}(\alpha), \cos(A, B))$$

In [2] we proved that $K_\alpha(A, B)$ satisfies the Mercer conditions, and is admissible as an SVM kernel. We also showed that when $A$ and $B$ are unit length,

$$K_\alpha(A, B) \approx \alpha e^{-\frac{\log(\alpha)}{2} \|A - B\|^2}$$

The right hand side of eq. (8) is the definition of an RBF kernel with parameter $\lambda = -\frac{\log(\alpha)}{2}$.

The purpose of the comparison with the RBF kernel is to give us an intuition about $K_\alpha(A, B)$, so that we can make meaningful comparisons with implementations that use the RBF kernel. Figure 1 shows the two sides of eq. (8) as $\cos(A, B)$ goes from 0 to 1. We ignore the case that $\cos(A, B)$ is negative, since in that case the values of both kernels are very close to zero.

We will now describe the feature map that allows us to use the kernel $K_\alpha(A, B)$ for tasks such as SVM training. This feature map is based on CRO hashes, so we describe these first.

4. THE CRO HASH FAMILY

The Concomitant Rank Order (CRO) hash family is a family of locality sensitive hash functions introduced in [3]. It was further developed and its collision probabilities analyzed in [2] and [8].
4.1 The CRO Feature Map

In this section we introduce the feature map

\[ \mathcal{F}_{\tau,Q}(A) : R^D \rightarrow \mathbb{Z}_2^U \]

where \( \mathbb{Z}_2 = \{0,1\} \).

\( \mathcal{F}_{\tau,Q}(A) \) is the function that maps vectors from the input space to sparse, high dimensional vectors in the feature space.

**Definition 5.** The feature map \( \mathcal{F}_{\tau,Q}(A) \) is defined as follows:

\[
\mathcal{F}_{\tau,Q}(A)[j] = \begin{cases} 
1 & \text{if } j \in H_{\tau,Q}(A) \\
0 & \text{if } j \notin H_{\tau,Q}(A)
\end{cases}
\]

Proposition 1 establishes the relationship between the feature map \( \mathcal{F}_{\tau,Q}(A) \) and the kernel \( K_\alpha(A,B) \).

**Proposition 1.** Let \( 0 < \alpha < 1 \). Let \( U \) be a positive integer. Let \( \tau = \lfloor \alpha U \rfloor \). Let \( M \) be CRO-admissible. Then

\[
\lim_{U \to \infty} \left[ E_{Q \in \mathcal{M}} \left[ \frac{\mathcal{F}_{\tau,Q}(A) \cdot \mathcal{F}_{\tau,Q}(B)}{U} \right] \right] = K_\alpha(A,B) \]

From eq. (8) and eq. (14) it is possible to derive that for unit vectors \( A,B \in R^D \)

\[
\mathcal{F}_{\tau,Q}(A) \cdot \mathcal{F}_{\tau,Q}(B) \approx \tau e^{-\frac{\log(\alpha)}{2} \|A-B\|^2} \tag{15}
\]

The right hand side of eq. (15) is an RBF kernel with kernel parameter \( \lambda = -\log(\alpha) \). Since \( \alpha = \frac{\tau}{U} \), it follows that

\[
\frac{\tau}{U} = e^{-\lambda} \tag{16}
\]

i.e. the sparsity of of the feature map, \( \frac{\tau}{U} \), is exponentially related to the kernel parameter \( \lambda \). Since \( \tau \) is the number of non-zero elements in the feature vector, even for moderately large kernel parameters, we end up with very sparse feature vectors.

4.2 Properties of The Feature Map

From definition 3 and definition 5 it follows that

- The total number of elements in \( \mathcal{F}_{\tau,Q}(A) \) is \( U \).
- The number of non-zero elements in \( \mathcal{F}_{\tau,Q}(A) \) is \( \tau \).
- All the non-zero elements of \( \mathcal{F}_{\tau,Q}(A) \) are 1.
- In the limit and using expected values,

\[
\mathcal{F}_{\tau,Q}(A) \cdot \mathcal{F}_{\tau,Q}(B) = UK_{\alpha}(A,B) \tag{17}
\]

where \( \alpha = \frac{\tau}{U} \). Moreover, \( K_{\alpha}(A,B) \) approximates the RBF kernel on the unit sphere.

We can interpret eq. (17) as follows: \( \mathcal{F}_{\tau,Q}(A) \) and \( \mathcal{F}_{\tau,Q}(B) \) are the projections of \( A \) and \( B \) into the feature space whose expected inner product is \( UK_{\alpha}(A,B) \). What is more, \( \mathcal{F}_{\tau,Q}(A) \) and \( \mathcal{F}_{\tau,Q}(B) \) are sparse and high dimensional. And as we will discuss in section 6, there is an efficient implementation of \( \mathcal{F}_{\tau,Q}(\cdot) \).

When the task is to train a support vector machine for classification, as long as the kernel \( K_{\alpha}(A,B) \) is effective (and this is the case whenever the RBF kernel on the unit sphere is effective), the SVM training and inference can be performed in the feature space. This is beneficial because:

- We can use linear SVM techniques in the feature space.

*Figure 1: Comparison of the two sides of eq. (8) for \( \alpha = \frac{1000}{2^4} \)*

**Definition 3.** Let \( U, \tau \) be positive integers, with \( \tau < U \). Let \( A \in R^D \) and \( Q \in R^{U \times N} \). Let

\[
P = QA
\]

Then

\[
H_{Q,\tau}(A) = \{ j : R(P[j], P) \leq \tau \}
\]

where \( R \) is the rank function defined in definition 1.

We call \( H_{Q,\tau}(A) \) the hash set of \( A \) with respect to \( Q \) and \( \tau \). As defined in eq. (10), the hash set of \( A \) with respect to \( Q \) and \( \tau \) is the set of indices of those elements of \( P \) whose rank is less than or equal to \( \tau \).

According to definition 3, the universe from which the hashes are drawn is \( 1 \ldots U \), since \( U \) is the number of rows of \( Q \). Assuming that there is no repetition in \( P \), the number of hashes is \( \tau \). This assumption is valid for all the projection matrices \( Q \) that we consider in this paper, in the sense that the probability that there is a repetition in \( P \) is very small.

Assuming that \( P \) does not have any repetitions, the following Matlab code returns the hash set:

```matlab
function hash_set = H(A, Q, tau)
    P = QA;
    [~, ix] = sort(P);
    hash_set = ix(1:tau);
end
```

**Definition 4.** Let \( 0 < \alpha < 1 \). Let \( U \) be a positive integer. Let \( \tau = \lfloor \alpha U \rfloor \). Let \( M \) be a set of unit matrices of iid \( N(0,1) \) random variables. Then \( M \) is CRO-admissible iff for all \( A,B \in R^D \),

\[
\lim_{U \to \infty} \left[ E_{Q \in \mathcal{M}} \left[ \frac{H_{Q,\tau}(A) \cdot H_{Q,\tau}(B)}{U} \right] \right] = K_{\alpha}(A,B) \tag{11}
\]

**Theorem 1.** Let \( M \) be the set of all instances of a \( U \times D \) matrix of iid \( N(0,1) \) random variables. Then \( M \) is CRO-admissible.

**Proof.** Follows from Theorem 1 in [2] with appropriate substitutions.

Computing \( P \) in eq. (9) using matrix multiplication takes \( O(UD) \) operations, which can be expensive when \( D \) is large. When \( Q \) is an instance of a matrix of iid normal random variables, as in theorem 1 above, this cannot be avoided. In section 6 we introduce other classes of CRO-admissible matrices that allow the computation of \( P \) in eq. (9) to be performed via the Fast Fourier Transform or the Walsh-Hadamard Transform, bringing down the cost of computing \( P \) to \( O(U \log(U)) \).
• The feature vectors are sparse, high dimensional and binary. We can take advantage of this to dramatically speed up the training and inference tasks.

In this paper we present SparseMinOver, an implementation of the maximum margin Perceptron training algorithm, MinOver, that takes advantage of the properties of the feature vectors (sparse, high dimensional, binary) to speed up the training process. We have implemented this algorithm, and run it on training sets with millions of items. We show the effectiveness of the overall scheme in terms of accuracy, and also in terms of computational efficiency.

5. MINOVER

The MinOver algorithm was introduced by Krauth et. al. [9] for spin-glass models of neural networks. It is a fairly simple modification of the Perceptron algorithm [18] that results in a maximum margin classifier [13]. We use the formulation of the MinOver algorithm presented in [13], where it was proved that the algorithm converges at rate $O(t^{-1})$ towards the optimal solution, where $t$ is the number of iterations. We ignore the bias term in the formulation of the algorithm, since the bias term is not needed when working in our feature space.

The MinOver algorithm as used here corresponds to a hard margin support vector machine. This is appropriate for our setting, since in the sparse high dimensional feature space that we are concerned with, there is always a separating plane and the hard margin SVM is effective.

5.1 The Problem Formulation

Let $X = \{x_1, x_2, \ldots, x_N \in R^D\}$ be a set of vectors, with the corresponding set of labels $Y = \{y_1, y_2, \ldots, y_N \in \{-1, 1\}\}$. The goal is to find the hyperplane which separates the two classes with maximum margin. This boils down to the following:

Find $w \in R^D$ that maximizes $\Delta$, defined as:

$$\Delta = \min_{i \in \{1, \ldots, N\}} (y_i w^T x_i) / ||w||$$  \hspace{1cm} (18)

With the constraint that

$$\forall i \in \{1, \ldots, N\}, y_i w^T x_i > 0$$  \hspace{1cm} (19)

For convenience in the description of the algorithm, we define the matrix $Z \in R^N \times D$ as follows:

$$Z_{i, \cdot} = y_i x_i^T$$

i.e. the $i^{th}$ row of $Z$ is $y_i x_i^T$.

5.2 The Basic Algorithm

The MinOver algorithm is an iterative algorithm that converges to the maximum margin classifier when the set of training vectors is linearly separable.

For notational convenience, for a positive integer $T$ we use $O_T$ to represent the vector of size $T$ all of whose elements are zero. In the following algorithm, $t_{\text{max}}$ is the number of iterations. $v \in R^D$ represents the separating vector after each iteration. $v$ is initialized to $O_D$. As the iterations proceed, $v$ converges to the maximal margin classifier.

The function $\min_{X}(g)$ returns the index of the minimal element of the vector $g$. If $g$ has more than one minimal element, it returns the index of the first minimal element.

It was proved in [13] that Algorithm 1 converges at rate $O(t^{-1})$ towards the optimal solution as $t$ increases.

Algorithm 1 The Basic MinOver Algorithm

1: $v := 0_D$ \hspace{1cm} $\triangleright$ Initialize $v$
2: for $t = 1 : t_{\text{max}}$ do
3: \hspace{0.5cm} $g := Zv$ \hspace{1cm} $\triangleright$ $g$ is an $N$ dimensional vector
4: \hspace{0.5cm} $m := \min_{X}(g)$
5: \hspace{0.5cm} $v := v + y_m x_m$
6: end for
7: $w := v$ \hspace{1cm} $\triangleright$ $w$ is the separating hyperplane

MinOver is very similar to the Perceptron algorithm. As with the Perceptron algorithm, we start with a vector $v$ initialized to $0_D$, and in each iteration we add to it one vector that is violated with the current value of $v$. The difference is that whereas the Perceptron algorithm chooses any violated vector, MinOver chooses the most badly violated vector. Moreover, with MinOver we continue the iterations even when all the training vectors are classified correctly, choosing as the vector to be added to $v$ the vector that is closest to the separating plane.

5.3 The MinOver Algorithm with Sparse Binary Vectors

The most expensive part of Algorithm 1 is step 3, where

$$g := Zv$$  \hspace{1cm} (20)

is computed. This operation takes $O(U N)$ arithmetic operations, and it has to be done on each iteration.

When the vectors in $X$ are sparse and binary, there is a more efficient way of implementing MinOver. The key is that $g$ is not recomputed in each iteration. Instead, it is incrementally updated given the update vector $x_m$. To do this, we take advantage of the fact that the input vectors are binary and sparse. It turns out that with this type of input vector, the number of arithmetic operations to incrementally update $g$ in each iteration is much smaller than that of re-computing $g$ from scratch.

5.3.1 Sparse Binary Vectors

We call $x \in Z^D$ a sparse binary vector with sparsity $\tau$ iff:

• $\tau$ elements of $x$ are 1.
• All the other elements of $x$ are 0.
• $\tau \ll U$.

As discussed previously, all vectors generated by the feature map $F_{\tau, \cdot}(\cdot)$ are sparse binary vectors.

A sparse binary vector can be represented as the set of coordinates of its non-zero elements. For example, the vector

$$(0, 0, 1, 0, 1, 0)^T$$

can be represented as $\{3, 5\}$.

5.3.2 Data Structures Used With Sparse Binary Vectors

We use two underlying data structures to work with sparse binary vectors. The first data structure is an index that returns the set representation of the vector. This index is accessed via the $nz$ function. $nz(i)$ returns the set representation of $x_i$, i.e. the set of coordinates of the non-zero elements of $x_i$.

The second data structure we use is a reverse index, which returns the set of vectors all of which have a non-zero element at a given coordinate. The data structure is accessed via the function.
An occurs, which returns the set of vectors in which c occurs as a non-zero coordinate, i.e.

\[ \text{occurs}(c) = \{ i : c \in \text{nz}(i) \} \]

Thus we have the following:

\[ i \in \text{occurs}(c) \iff c \in \text{nz}(i) \]

5.3.3 MinOver with Sparse Binary Vectors

Algorithm 2 is the MinOver algorithm optimized for sparse binary vectors. Unlike Algorithm 1, in Algorithm 2 \( g \) is initialized outside the loop, and is incrementally updated inside an inner loop in line 8. \( v \) is also incrementally updated in line 6.

It is possible to prove that for the outer-loop, the following is a loop invariant:

\[ g = Zv \]

5.3.4 Computational Complexity of Algorithm 2

Line 8 in Algorithm 2 is in the inner loop of the algorithm, so the computational complexity of the algorithm is dominated by the number of times this line is visited.

It is clear that for all \( m \) the size of \( \text{nz}(m) \) is equal to \( \tau \). This is because the feature vectors all have exactly \( \tau \) non-zero elements. Let \( \kappa \) be the average size of \( \text{occurs}(c) \), i.e.

\[ \kappa = \frac{\sum_{c=1}^{U} |\text{occurs}(c)|}{U} \]

Let \( \gamma \) denote the average number of times line 8 is visited in each top level iteration, the average being taken over all possible inputs. Then, since \( \forall m \) \( \text{nz}(m) \) = \( \tau \),

\[ \gamma = \tau \kappa \]

From eq. (22) it is straightforward to show that

\[ \sum_{c=1}^{U} |\text{occurs}(c)| = \sum_{i=1}^{N} |\text{nz}(i)| \]

But since \( \forall i |\text{nz}(i)| = \tau \),

\[ \sum_{i=1}^{N} |\text{nz}(i)| = N\tau \]

Therefore, from eq. (24), eq. (27)

\[ \kappa = \frac{N\tau}{U} \]

Thus, from eq. (28), eq. (25) we get

\[ \gamma = \frac{N\tau^2}{U} \]

\( \gamma \) is the average number of arithmetic operations to update \( g \) in each iteration in Algorithm 2. Now, in all interesting cases for which we have used this feature map, \( \frac{\tau}{U} \) is a small constant (somewhere around 1 in value). Compare this with the cost of recomputing \( g \) in each iteration in Algorithm 1, which involves a matrix multiplication with complexity \( NU \).

When the input vectors are sparse, in Algorithm 1 we could take advantage of the sparsity of \( x_{m} \) to speed up the matrix multiplication. But this reduces the cost of recomputing \( g \) in step 3 of Algorithm 1 to \( N\tau \). So for sparse input vectors, the incremental algorithm is faster by a factor of \( \tau \).

Notice that the incremental update of \( g \) in line 8 of Algorithm 2 is only possible due to the sparsity of the input vectors. For non-sparse vectors, \( \text{occurs} \) returns the whole input set, thus making Algorithm 2 useless.

Taking into account the computational cost of steps 4 and 6 of Algorithm 2, it is easy to show that the complexity of running the algorithm for \( t_{\text{max}} \) operations is to be

\[ t_{\text{max}} \mathcal{O} \left( \tau + N + \frac{N\tau^2}{U} \right) \]

6. Computing the Hash Set Using Orthogonal Transforms

Equation (9) requires the computation the projection vector \( P = QA \). Using matrix multiplication to do this computation requires \( U \times D \) operations, which can be expensive when \( D \) is large. This is unavoidable when \( Q \) is an instance of a matrix of iid normal variables, as in theorem 1.

Using the techniques explored in [8], we define a set of projection matrices \( \Theta_{U,D} \) that are CRO-admissible, and for which computing the projection can be achieved through fast orthogonal transforms. Thus we reduce the cost of computing the projection to \( \mathcal{O}(U \log(U)) \).

**Definition 6.** Let \( I_{D} \) denote the \( D \times D \) identity matrix and \( 0_{r,D} \) the \( r \times D \) zero matrix. Let \( \tau = U \mod 2D \) and \( d = U \div 2D \) (\( \div \) denotes integer division). Then the \( U \times D \) symmetric repetition matrix \( J_{U,D} \) is defined as follows:

\[ J_{U,D} = [I_{D}, -I_{D}, I_{D}, -I_{D}, \ldots, I_{D}, -I_{D}, 0_{r,D}] \]

where \( I_{D}, -I_{D} \) blocks are repeated \( d \) times.

**Definition 7.** Let \( T \) be a \( U \times U \) orthogonal transform matrix, \( J \) a \( U \times D \) symmetric repetition matrix, and \( \Pi \) the set of all \( U \times U \) permutation matrices. Then

\[ \Theta_{U,D} = \{ T \pi J : \pi \in \Pi \} \]

**Proposition 2.** \( \Theta_{U,D} \) is CRO-admissible.

**Proof.** Follows from Theorem 1 in [8] by using the same arguments used for proving Theorem 1 in [2].
As described previously, the CRO hash function maps input vectors to a set of hashes chosen from a universe $1 \ldots U$, where $U$ is a large integer. $\tau$ denotes the number of hashes that we require per input vector.

Let $A \in \mathbb{R}^{D\times D}$ be the input vector. The transform-based hash function takes as a second input a random permutation $\pi$ of $1 \ldots U$. It should be emphasized that the random permutation $\pi$ is chosen once and used for hashing all input vectors.

Table 1 shows the procedure for computing the CRO hash set using fast orthogonal transforms. Here we use $-A$ to represent the vector $A$ multiplied by $-1$. We use $A, B, C, \ldots$ to represent concatenation of vectors $A, B, C$ etc.

Table 2 presents an implementation of the transform-based CRO hash function in Matlab.

### Table 1: Computing the CRO hash set for input vector $A$ using orthogonal transforms

1. Let $\hat{A} = A, -A$
2. Create a repeated input vector $A'$ as follows:
   
   $A' = \hat{A}, \hat{A}, \ldots, \hat{A}_000$
   
   where $d = U/2D$ and $r = U \mod 2D$.
   
   Thus $|A'| = 2dD + r = U$.
3. Apply the random permutation $\pi$ to $A'$ to get permuted input vector $V$.
4. Compute the orthogonal transform of $V$ to get $S$. This could be the Fast Walsh-Hadamard Transform or the Fast DCT Transform.
5. Find the indices of the smallest $\tau$ members of $S$. These indices are the hash set of the input vector $A$.

### Table 2: Matlab code for the CRO hash function using orthogonal transform

```matlab
function hashes = CROHash(A,U,P,tau)
    % A is the input vector.
    % U is the size of the hash universe.
    % P is a random permutation of 1:U
    % tau is the desired number of hashes
    % E=zeros(1,U);
    % AHat = [A,-A];
    % D2=length(AHat);
    % d=floor(U/D2);
    % for i=0:d-1
    %     E(i+N2+1:(i+1)*N2)=AHat;
    % end
    % Q=E(P);
    % If an efficient implementation of
    % the Walsh-Hadamard transform is
    % available, we can use it instead, i.e.
    % S=fwht(Q);
    % S=dct(Q);
    [r,ix]=sort(S);
    hashes=ix(1:tau);
end
```

The experiments are performed on two publicly available datasets, MNIST [11] and TIMIT [4]. MNIST is a well known dataset commonly used for classification and handwritten digit recognition. The second dataset is the TIMIT acoustic-phonetic continuous speech corpus. The TIMIT corpus is widely used for the purpose of development and performance evaluation of speech recognition systems. The data from the TIMIT corpus includes recordings of 6300 utterances (5.4 hours) of eight dialects of American English (pronounced by 630 speakers, 10 sentences per each speaker). The standard train/test split is 4620 train and 1680 test utterances. The phonetic and word transcriptions are time-aligned and include a 16kHz, 16-bit waveform file for each utterance. All TIMIT data has been manually verified so that the corresponding labels match the phonemes. Due to the aforementioned attributes, TIMIT is especially suitable for phoneme classification.

Each utterance is converted to a sequence of 792-dimensional feature vectors of 25 Mel-frequency cepstral coefficients with cepstral mean subtraction (including Cepstral $C_0$ coefficient) plus delta and acceleration coefficients. Standard frames of length 25 ms shifted by 10 ms are used. The original set of 61 phonemes is collapsed into a smaller set of 39 phonemes. The following table summarize train/test datasets:

### Table 3: TIMIT corpus summary

<table>
<thead>
<tr>
<th>dataset</th>
<th># utterances</th>
<th># frames</th>
<th># phonemes</th>
</tr>
</thead>
<tbody>
<tr>
<td>train</td>
<td>4620</td>
<td>138542</td>
<td>39</td>
</tr>
<tr>
<td>test</td>
<td>1680</td>
<td>506113</td>
<td>39</td>
</tr>
</tbody>
</table>

The original set of 61 phonemes includes: ax-h, bcl, dcl, gcl, pcl, tcl, kcl, axr, pau, epi, eng, en, hh, hv, iy, ih, ey, ey, ee, ee, aw, ay, ah, ao, oy, ow, uh, uw, er, ax, ix, jh, dx, zh, th, dh, ng, em, en, h#, h, d, g, p, t, k, s, l, r, w, y, z, f, v, m, n, q.

The mapping from 61 to 39 phonemes is defined as:

- $aa \leftarrow ao$
- $ah \leftarrow ax$ $ax$ $h$
- $er \leftarrow axr$
- $hh \leftarrow hv$
- $ih \leftarrow ix$
- $l \leftarrow cl$

Therefore, the final set of phonemes becomes: aa, ao, ah, ax, ax-h, er, axr, hh, hv, ih, ix, l, el, m, em, n, en, nx, ng, eng, sh, zh, uw, ux, sil, pcl, tcl, kcl, bcl, dcl, gcl, h#, pau, epi, ey, eh, ey, ae, aw, ay, oy, ow, uh, jh, ch, b, d, g, p, t, k, dx, s, z, f, th, v, dh, r, w, y.

In this paper, we are concerned with phoneme classification, where each frame is classified independently of the frames preceding or following it. As a matter of fact, we permute the training inputs so all temporal relationship between phonemes is lost. This is in contrast with the phoneme recognition task, where the relationship between adjacent frames is taken into account, for example through a Hidden Markov Model.

We use the classification error rate defined as the number of incorrectly classified instance divided by the total number of test instances to evaluate the performance of the proposed SparseMi-nOver algorithm. A one-vs-all multi-class classification approach is used classify the test instances into one of the possible 39 classes.
In the one-vs-all approach we train 39 different binary classifiers. For the $k$th classifier, the positive instances belong to class $k$ and the negative instances are points not in class $k$.

All experiments were conducted on a Hewlett Packard Enterprise ProLiant DL580 with 4 sockets, 60 cores, and 1.5 TB of RAM. Each socket has 15 cores and 384 GB of RAM. The experiments focus mainly on three main parameters and how they impact the classification error rate, training time, and transformation time. These parameters include:

- $\alpha$: Sparsity of the feature map
- $\tau$: Number of non-zero elements in the feature map
- $t_{\text{max}}$: Number of iterations

### 7.1 TIMIT results

Table 4 and Table 5 present the classification error rate and training time for different values of $\tau$ when the sparsity $\alpha$ is equal to $2^{-12}$. Table 4 illustrates that increasing $\tau$ results in lower error rate, hence better classification accuracy. For this experiment the number of iterations $t_{\text{max}}$ is set to 1,000,000.

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>$\tau$: number of non-zero elements</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{-12}$</td>
<td>1000</td>
<td>2000</td>
<td>3000</td>
<td>4000</td>
</tr>
<tr>
<td>32.74%</td>
<td>31.47%</td>
<td>30.98%</td>
<td>30.75%</td>
<td></td>
</tr>
</tbody>
</table>

The timing results in Table 5 show that the training time increases linearly with the increase of $\tau$ when sparsity is constant. This can be explained by looking at the relationship between $\alpha$, $\tau$, and $U$. Recall that $\alpha = \frac{\tau}{U}$, so when $\alpha$ is constant, the increase of $\tau$ results in the increase of $U$. Both the increase of $\tau$ and $U$ result in the training time increase.

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>$\tau$: number of non-zero elements</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{-12}$</td>
<td>1000</td>
<td>2000</td>
<td>3000</td>
<td>4000</td>
</tr>
<tr>
<td>293</td>
<td>509</td>
<td>717</td>
<td>948</td>
<td></td>
</tr>
</tbody>
</table>

Table 6 presents the classification error rate when increasing the total number of iterations $t_{\text{max}}$ for sparsity values $2^{-8}$, $2^{-10}$, and $2^{-12}$. In this experiment $\tau$ has been chosen so that $\alpha \times \tau \approx 1$. Assuming that the vectors are linearly separable, as the number of iterations increases we expect that the SparseMinOver algorithm converges to the maximum margin classifier, resulting in a lower classification error rate. The results in Table 6 show that this is true for all experimented values of $\alpha$.

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>$t_{\text{max}}$: number of iterations</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{-8}$</td>
<td>62.5k 125k 250k 500k</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>55.14% 53.92% 53.25% 53.15%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$2^{-10}$</td>
<td>1024 36.89% 34.30% 33.57% 33.45%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4096</td>
<td>36.79% 34.02% 31.32% 30.83%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Clearly, as the number of iterations increases the time required to perform the process should increase. Table 7 shows the training time for results in Table 6. The timing results demonstrate a linear relationship between number of iterations and training time, which is expected. Recall from eq. (30) that each iteration has an average-case complexity of $O(\tau + N(1 + t_{\text{max}}^{\alpha}))$; hence explaining the linear relationship between the training time and number of iterations $t_{\text{max}}$.

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>$t_{\text{max}}$: number of iterations</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{-8}$</td>
<td>62.5k 125k 250k 500k</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>46% 93% 189% 369%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$2^{-10}$</td>
<td>1024 52% 105% 203% 422%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4096</td>
<td>64% 122% 248% 493%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 8 shows the classification error rate and corresponding training time for sparsity values of $2^{-12}$, $2^{-14}$, and $2^{-16}$, $\tau$ from 1000 to 4000, and $t_{\text{max}}$ equal to 1,000,000. We achieve the best result when $\alpha = 2^{-12}$, $\tau = 4000$, and $t_{\text{max}} = 1,000,000$.

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>$\tau$: number of iterations</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{-12}$</td>
<td>1000 32.74%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>293</td>
<td>509</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$2^{-14}$</td>
<td>1000 33.94%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>135</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$2^{-16}$</td>
<td>1000 36.15%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>94</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 9 shows the time required to perform the mapping from the input space to the sparse binary feature space under various parameter settings. The transformation time increases linearly with the increase of $\tau$. Also, higher sparsity results in higher transformation time.

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>$\tau$: number of iterations</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{-8}$</td>
<td>62.5k 125k 250k 500k</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>0.85%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$2^{-10}$</td>
<td>1024  2.6%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.9%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$2^{-12}$</td>
<td>1024  25.7%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 7.2 Comparison with related work on TIMIT

Hinton et. al. [5] use a feedforward neural network for frame-classification on TIMIT. The lowest classification error rate reported in [5] is more than 31%. Huang et. al. [6] (discussed in Section 2) also report frame-level classification results on the TIMIT dataset. They report that they could not train the full RBF classifier on the full training set, and instead trained it on a subset of 100,000 training samples, for which they obtained a classification error rate of 38.61%. Their best results are with an ensemble of 7 classifiers with feature space dimensionality of 60k, for which they achieved 33.5% error rate. They do not report the training time for this result.
With the CRO kernel and feature maps, combined with the Sparse-MinOver algorithm, however, we can train classifiers using millions of training samples, which makes our approach suitable for big-data learning tasks.

We achieve classification error rate of 30.83%, which is the best classification error rate reported on the TIMIT dataset using kernel methods, in 493 minutes of training time.

7.3 MNIST results

The standard setting for experiments on the MNIST dataset includes a training set of size 60,000 and a test set of size 10,000. The dimensionality of the feature vectors is equal to 780.

Figure 2 shows how increasing the number of iteration impacts the classification accuracy on the MNIST dataset. For this experiment $\alpha = 2^{-14}$ and $\tau = 16384$. As the number of the iterations reaches closer to 1000 we see that the algorithm converges to the maximum margin classifier and classification accuracy does not change much after that.

Figure 2: Accuracy on MNIST dataset with increasing number of iterations

Figure 3 depicts how increasing the number of non-zero elements $\tau$ impacts the classification accuracy. The parameters for this experiments are defined as $\alpha = 2^{-10}$ and $t_{max} = 1000$. As $\tau$ increases the classification accuracy also increases; however, there is a logarithmic relationship between $\tau$ and accuracy. Given that $\alpha$ is constant, when $\tau$ increases the universe size $U$ also increases.

Figure 3: Classification accuracy on MNIST dataset with increasing value of $\tau$

The complexity of each iteration is directly related to the value of $\tau$, thus, when $\tau$ has a large value the training time increases linearly. Figure 4 shows relationship between the training time and $\tau$ when $\alpha = 2^{-10}$ and $t_{max} = 1000$.

Figure 4: Training time on MNIST dataset with increasing value of $\tau$

Although all three parameter settings in Table 10 result in similar classification accuracy, the training time significantly differs. When sparsity is $2^{-14}$ the training time is significantly smaller when compared to $2^{-10}$. This is due to the relationship between the parameters defined in Section 5.3.4.

Table 10: Parameter settings resulting in high accuracy on MNIST dataset

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>$\tau$</th>
<th>$t_{max}$</th>
<th>acc</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{-10}$</td>
<td>15000</td>
<td>1000</td>
<td>98.48%</td>
</tr>
<tr>
<td>$2^{-12}$</td>
<td>15000</td>
<td>1000</td>
<td>98.50%</td>
</tr>
<tr>
<td>$2^{-14}$</td>
<td>16384</td>
<td>1000</td>
<td>98.49%</td>
</tr>
</tbody>
</table>

8. CONCLUSIONS

We present an approach to kernel SVM classification that is highly efficient and scalable, while achieving a level of accuracy that is on par with the RBF kernel SVM. To do this, we take advantage of the sparse binary feature space that lends itself to an efficient implementation of the classic MinOver algorithm. We demonstrate the efficacy of our approach on the TIMIT dataset, a large and significant benchmark for classifier training.

9. REFERENCES


