

# Multiresolution Approximation for Classification

Ilya Blayvas and Ron Kimmel {blayvas,ron}@cs.technion.ac.il  
CS Dept. Technion, Haifa, Israel, 32000

January 9, 2002

## Abstract

*We consider the classification problem as a problem of approximating a given training set. This approximation is constructed as a multi-resolution sparse grid, and organized in a tree-structure. It allows efficient training and query, both in constant time per training point, for low-dimensional classification problems ( $D \lesssim 10$ ) with large data sets. The memory required for such problems is usually substantially smaller than the size of the training set.*

## 1 Introduction

The classification problem can be defined as the problem of providing an answer  $y$ , associated with an arbitrary query  $\vec{x}$  on the basis of a given training set  $T$ . The training set consists of ordered pairs of queries and answers,

$$T = \{\vec{x}_i, y_i\}_{i=1}^M. \quad (1)$$

Any query belongs to the space of queries or *feature space*  $F : \vec{x} \in F$ , and any answer belongs to the space of answers or *class assignment space*  $Y : y \in Y$ . Obviously  $T \in F \times Y$ . An ordered pair  $\{\vec{x}_i, y_i\} \in T$  will be referred to as a *training point*.

This paper considers the classification problems with continuous feature spaces, prescaled into a  $D$  dimensional unit cube:  $\vec{x} \in F = [0, 1]^D$ . The classification decision  $y$  obtains values in the interval  $[-1, 1]$ .

The classification problem can be considered as the reconstruction problem of some unknown function  $f(\vec{x})$  that interpolates or approximates the points of the training set  $T = \{\vec{x}_i, y_i\}_{i=1}^M$ ,

$$\begin{aligned} f : F &\Rightarrow Y \\ f(\vec{x}_i) &\approx y_i, \forall i \in \{1, \dots, M\}. \end{aligned} \quad (2)$$

This problem is usually referred to as an ill-posed problem [1, 2, 3, 4]. In fact, there are infinitely many functions that interpolate the points of the training set and yet have different values at the other points. It is usually proposed to use regularization theory [5] in order to overcome this difficulty, and to convert the problem into the well posed one of minimizing of the functional,

$$H(f) = \sum_{i=1}^M (f(\vec{x}_i) - y_i)^2 + \lambda \Phi(f). \quad (3)$$

The first term in this functional is responsible for approximating the training set, while the second is the regularization term, favoring smoothness and ‘simplicity’ of the function  $f$ . The function  $f$  belongs to some pre-defined parametric family  $f(\vec{x}) = f(\vec{\alpha}_0, \vec{x}) \in \{f(\vec{\alpha}, \vec{x})\}$ . This problem is usually solved by minimization of  $H(f)$  in the space of  $\vec{\alpha}$ . Both Neural Networks [6] and Support Vector Machines [1, 2] are examples of such approaches. These methods were proved to be efficient by many successful applications. However, they suffer from some limitations:

- The search for optimal parameters  $\vec{\alpha}$  that minimize (3), usually requires  $O(M^2)$  operations for a training set of size  $|T| = M$  [7].
- Every training point is processed several (usually  $O(M)$ ) times during the training phase. Therefore, all the training points have to be stored until the end of the training, which is expensive for the large training sets.

- The class of functions  $f(\vec{\alpha})$  is often either too limited or too large for the training set, which leads to suboptimal performances of the classifier.

The reconstruction of an unknown function from the information in the training set (2) can be converted into a well posed problem by limiting the class of approximating functions. Let us consider the function  $f$  defined in some basis  $\{\phi_i(\vec{x})\}_{i=1}^{\infty}$  of  $L^2(F)$ ,

$$f = \sum_{i=1}^{\infty} c_i \phi_i(\vec{x}). \quad (4)$$

If the coefficients  $c_i$  can be uniquely determined from the training set  $c_i = c_i(T)$ , for an arbitrary training set, then the learning problem (2) becomes ‘well posed’. One can consider the regularization as a property of the basis and the procedure of calculating  $c_i$ .

We propose an approximation framework for the classification problem, in which the unknown function  $f$  is constructed as a set of coefficients in a given basis (4). The coefficients  $c_i$  are uniquely defined by the training set and are computed by a simple algebraic algorithm. This allows a fast construction of  $f$  in  $t = O(M)$  operations and efficient query in constant time. The memory required for the storage of  $f$  is  $O(M)$  in the worst case, but usually much smaller. The chosen set of functions  $\phi_i(\vec{x})$  can successfully learn and approximate a data set of arbitrary complexity, provided it has low dimensionality.

For a dense sampling of a  $D$  dimensional feature space  $F \subset R^D$ , a training set of size  $M \gg 2^D$  is required. For the cases where  $M \lesssim 2^D$ , a modification of the basic algorithm improves the classification performances for sparse datasets, at the expense of the query time.

The rest of this paper is organized as follows: Section 2 briefly introduces and analyzes the interpolation and approximation classification algorithms. Section 3 presents the results of experimental evaluation of the proposed algorithms and compares them to existing techniques. Finally, Section 4 summarizes the results with some concluding remarks.

## 2 Interpolation and Approximation Algorithms for Classification

### 2.1 Previous Work

There are two frameworks that motivated our approach. Bernard [4] develops the wavelet approach for multiresolution interpolation in high-dimensional spaces for machine learning applications (see also Mallat [8]), Garcke et al., present sparse grid solution for data mining [3].

In this paper the machine learning problem is considered and treated as an interpolation or approximation of training set points. This approximation is done within the framework of multiresolution analysis, similar to [4]. However, the function is constructed in a different way in order to efficiently handle the sparse data, which is a typical case in classification problems. We span the feature space by multiresolution sparse grids as in [3], yet instead of a search for optimal coefficients by a minimization process, we determine and update coefficients by direct arithmetic calculations. Thereby, we end up with a shorter training time and smaller memory usage.

The proposed algorithms were used to efficiently construct a two dimensional approximation function of sparse scattered points. The constructed function was used as a threshold surface for image binarization in [9].

### 2.2 Description of the Algorithm

Let us first define the multi-resolution representation of the function  $f$ , and then present the algorithm for calculating the coefficients of this representation from the training set. In general, the training set is defined by,

$$T = \{\vec{x}_i, y_i\}_{i=1}^M \in [0, 1]^D \times [-1, 1]. \quad (5)$$

The unknown function  $f$  is constructed as follows:

$$f = \sum_{l=0}^{\infty} \sum_{j_1, \dots, j_D=0}^{2^l-1} c_{j_1, \dots, j_D}^l \phi_{j_1, \dots, j_D}^l(\vec{x}), \quad (6)$$

where

$$\phi^0(\vec{x}) = \begin{cases} 1 & \text{if } \vec{x} \in [0, 1]^D \\ 0 & \text{if } \vec{x} \notin [0, 1]^D \end{cases} \quad (7)$$

and

$$\phi_{j_1, \dots, j_D}^l(\vec{x}) = \phi^0(2^l(\vec{x} - \vec{x}_{j_1, \dots, j_D})). \quad (8)$$

Equations (6-8) define the approximation function in a basis which is a multidimensional generalization of the Haar basis [8], constructed by tensor products of the one-dimensional bases. The Haar basis is in  $L^2$ . Since an arbitrary finite training set can be interpolated by some function from  $L^2$ , basis (6-8) can interpolate an arbitrary finite training set.

In practice, the sum over  $l$  in (6) is truncated at some  $l = MaxLev$  and therefore it spans only a subspace of  $L^2$ . Such a truncated sum spans the feature space by the cells of size  $2^{-MaxLev \cdot D}$ . It will interpolate the training set everywhere except the cells of this size, which contain several training points with different values (an improbable case for reasonable  $MaxLev$  and  $D$ ).

The coefficient  $c_{j_1, \dots, j_D}^l$  relates to a cell  $C_{j_1, \dots, j_D}^l$  in the feature space. This cell is a unit cube  $[0, 1]^D$ , downscaled to size  $(2^{-l})^D$  and shifted to  $\vec{x}_{j_1, \dots, j_D} = 2^{-l}\{j_1, \dots, j_D\}$  in  $F$ , where  $j_k \in \{0, \dots, 2^l - 1\}$  defines a shift along the  $k$ -th coordinate. The coefficients  $c_{j_1, \dots, j_D}^l$  are determined by averaging all the residuals  $y_k^{(l)}$  of the training points in the corresponding cell (i.e.  $\vec{x}_k \in C_{j_1, \dots, j_D}^l$ ). The residuals are the remaining values of the training points after subtracting their approximation at the lower resolution levels.

Following is the procedure for calculating  $c_{j_1, \dots, j_D}^l$  (see also Figures 1-3,5):

1. At the first step, all  $M$  training points are approximated by their average value,

$$c^0 = \frac{1}{M} \sum_{i=1}^M y_i.$$

Next, the value of each training point is set to be the difference between its original value and the already interpolated part

$$y_i^{(1)} = y_i - c^0.$$

Here, superscript  $(1)$  denotes the first order residual, i.e. the remaining part of the training point value, after its approximation by  $c^0$  is subtracted off.

2. At the second step, the feature space  $F = [0, 1]^D$  is divided into  $2^D$  cubic cells with edges of size  $\frac{1}{2}$ . The cells are denoted by  $C_{j_1, \dots, j_D}^1$ , where the superscript 1 denotes the *first resolution level*,  $l = 1$ , and  $j_k \in \{0, 1\}$  denotes respectively the  $\{0, \frac{1}{2}\}$  position of the cell along the  $k$ -th coordinate of  $F$ . Some cells may not contain the training points and the value of  $f$  in such cells will not be refined. For each cell which contains some training points  $\{\vec{x}_k\}$ , their average in that cell is calculated and assigned to be the value of the cell,

$$c_{j_1, \dots, j_D}^1 = \frac{1}{K} \sum_{\vec{x}_k \in C_{j_1, \dots, j_D}^1} y_k^{(1)}.$$

Here  $K > 0$  is the number of training points inside the cell.

3. All the training points that contributed to the cell  $C_{j_1, \dots, j_D}^1$  are updated,

$$y_k^{(2)} = y_k^{(1)} - c_{j_1, \dots, j_D}^1.$$

4. Steps 2 and 3 are repeated at the higher resolution levels up to the pre-defined level  $MaxLev$ .

Figure 1 shows initial values of the training points,  $y_i^{(0)} \equiv y_i$  and  $c^0$  at resolution level  $l = 0$  as well as  $y_i^{(1)} = y_i^{(0)} - c^0$  and  $c_{\{0,1\}}^1$  at level 1. Figure 2 shows  $y_i^{(2)} = y_i^{(1)} - c^1$  and  $c_{\{0, \dots, 3\}}^2$  at level 2 as well as  $y_i^{(3)}$  and  $c_{\{0, \dots, 7\}}^3$  at level 3. Figure 3 shows the piece-wise constant interpolation function, obtained by summing over the cell values at all four resolution levels. Figure 5 illustrates the multiresolution decomposition of a two-dimensional feature space into cells at resolution levels  $l = 0, 1, 2$ . The point  $P$  belongs to  $C_{00}^0$  at  $l = 0$ ,  $C_{10}^1$  at  $l = 1$ , and  $C_{21}^2$  at  $l = 2$ .

### 2.3 Data Structures

In the  $D$  dimensional feature space, the number of cells at resolution level  $l$  is  $2^{D \cdot l}$ , which is usually much larger than the number of training points  $M$ . Therefore, the percentage of the non-empty cells is exponentially decreasing with the increase of the resolution level  $l$ .

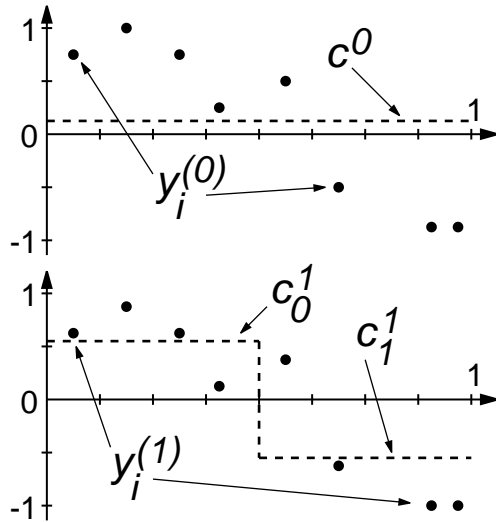


Figure 1: Resolution levels 0 and 1 for the one dimensional case.

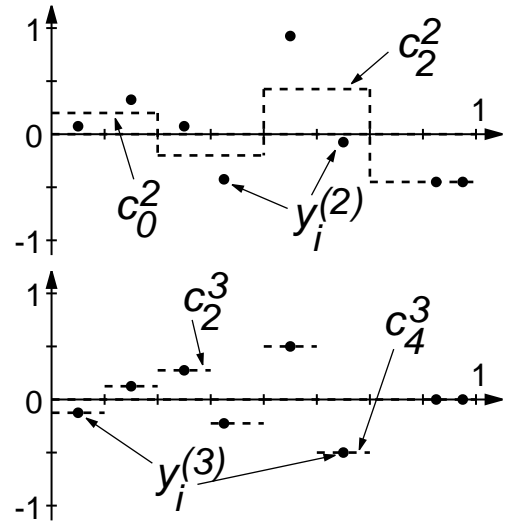


Figure 2: Resolution levels 2 and 3 for the one dimensional case.

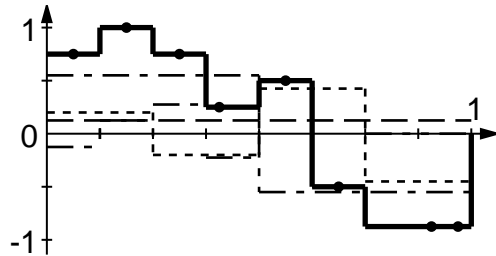


Figure 3: 'Interpolation' of the data points at the first four resolution levels.

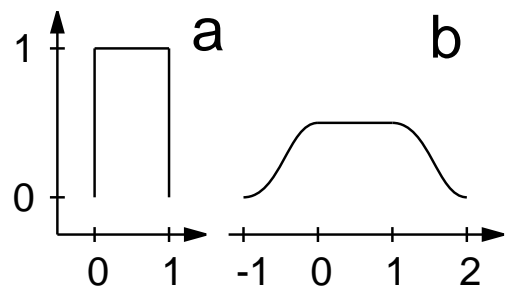


Figure 4: Source functions for piecewise continuous interpolation (a) and smooth approximation (b).

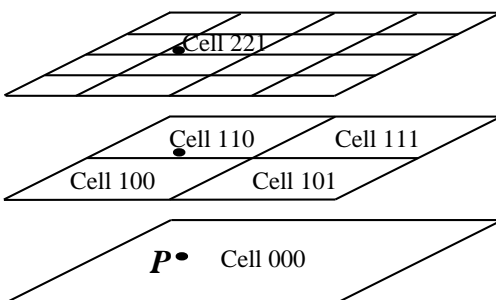


Figure 5: Multiresolution representation of a 2-dimensional feature space.

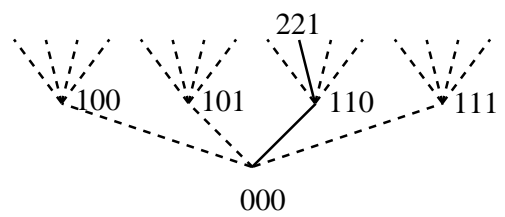


Figure 6: Tree structure of the cells of a 2-dimensional feature space.

For efficient storage and query, the non-empty cells are stored as a tree (Figure 6), where the root corresponds to the largest cell  $C^0$  and the descendants of the root correspond to non-empty smaller cells at higher resolution levels. The depth of the tree  $MaxLev$  should not exceed 5 for most practical cases, as it corresponds to the decomposition of the feature space into  $2^{D \cdot MaxLev}$  cells.

For an arbitrary point  $\vec{x}$ , there is exactly one cell, containing it at every resolution level (Figure 5). The value  $f(\vec{x})$  equals to the sum of the values of all the cells that contain  $\vec{x}$ .

Every tree node at level  $l$  corresponds to some cell  $C_{j_1, \dots, j_D}^l$  and is stored as a structure *treenode*:

```
struct treenode{ float cellval; int pnum;
                int sonnum; int level; sonstree* sonptrs; }
```

The value of  $C$  is stored in *cellval*, the number of points that are inside  $C$  in *pnum*, its level in *level* and the pointers to its non-empty sons at the level  $l + 1$  in *sonptrs*.

Figure 5 shows a two dimensional feature space divided into cells at the first three resolution levels. Figure 6 shows the organization of these cells into a tree structure.

## 2.4 The Interpolation Algorithm

The classifier consists of two algorithms. The first gets a new training point  $\{\vec{x}, y\}$  and a pointer to the tree  $\mathcal{T}$ , and ‘learns’ the point by an appropriate update of the tree  $\mathcal{T}$ . The second algorithm gets a query  $\vec{x}$  and a pointer to the tree  $\mathcal{T}$ , and provides an answer  $y = y(\vec{x}, \mathcal{T})$  for this query.

The learning algorithm runs as follows:

Procedure *LearnPoint*( $tn, \vec{x}, y$ ) receives three arguments. The structure  $tn$  is a vertex of the tree  $\mathcal{T}$ , it contains the information about a cell  $C_{j_1, \dots, j_D}^l$ . Items  $\vec{x}$  and  $y$  are the coordinate and the residual value  $y^{(l)}$  of the training point to be learned.

The procedure calculates the value *delta* that will update the cell value  $tn \rightarrow val$ . The *delta* is calculated on the basis of the old value of the cell  $tn \rightarrow val$ , previous number of points that contributed to the cell  $tn \rightarrow pnum$  and the residual value of the training point  $y$ . Then, the residual  $y$  is updated to contain only the value which is not yet approximated. The number of the contributing points  $tn \rightarrow pnum$  is incremented to account for the new training point.

If  $tn$  corresponds to a cell that is not at the highest resolution level, then the sons of  $tn$ , that already exist, are updated by *UpdateSons*(), and the *LearnPoint*() procedure is repeated recursively for the son that contains  $\vec{x}$ . Procedure *GetSon1*() returns a pointer to such a son. If this son does not exist (there were no training points inside his cell, before) it is created by *GetSon1*() .

The procedure *UpdateSons*( $tn, (-delta)$ ) decrements by *delta* the value of each existing son of  $tn$ , in order to preserve the value of  $f$  inside the son’s cells.

1. **Procedure** *LearnPoint*( $tn, \vec{x}, y$ )
2.  $delta = \frac{y - (tn \rightarrow val)}{(tn \rightarrow pnum) + 1}$
3.  $(tn \rightarrow val) = (tn \rightarrow val) + delta$
4.  $y = y - (tn \rightarrow val)$
5.  $(tn \rightarrow pnum) = (tn \rightarrow pnum) + 1$
6. if  $((tn \rightarrow level) < MaxLev)$  then
7.     *UpdateSons*( $tn, (-delta)$ )
8.      $sn = GetSon1(tn, \vec{x})$
9.     *LearnPoint*( $sn, \vec{x}, y$ )
10. endif
11. return

The query algorithm simply sums up the values of cells that contain  $\vec{x}$  at all resolution levels,  $0 \leq l \leq MaxLev - 1$ :

1. **Procedure** *Query*( $tn, \vec{x}$ )
2.      $sn = GetSon2(tn, \vec{x})$

3. if ( $sn \neq NULL$ ) then
4.      $answer = Query(sn, \vec{x})$
5. endif
6.      $answer = answer + (tn \rightarrow val)$
7. return  $answer$

The  $Query(tn, \vec{x})$  starts at the root cell  $\mathcal{C}^0$  and proceeds recursively while there exist son cells, containing  $\vec{x}$ . The cell at level  $MaxLev$  has no sons by construction, however, it is possible that starting from some level  $l_{max} < MaxLev$ , a son cell does not exist if there were no training points contributing to it. In these cases  $GetSon2()$  returns  $NULL$ .

The learning and query algorithms, described above, implement construction and evaluation of the function (6) defined by (7) and (8). The resulting function interpolates the values of the training set, provided  $MaxLev$  is large enough, i.e., there is at most one training point in the cell of the highest resolution level or several training points with the same value [9].

## 2.5 Smooth Approximation

The interpolating function  $f$ , constructed by  $LearnPoint()$  and evaluated by  $Query()$  is discontinuous along the cell boundaries, which leads to suboptimal performance of the classifier. The simple way to make  $f$  continuous and smooth is to modify the  $Query()$  procedure. In the  $SmoothQuery()$ , not only the values of the cells that contain  $\vec{x}$  are summed up, but also the neighboring cells, with their contribution  $\phi(r)$  that depends on their distance  $r$  from  $\vec{x}$ .

This corresponds to the calculation of  $c_{j_1, \dots, j_D}^l$  in the basis defined by (7-8) and reconstruction of  $f$  in another basis, with overlapping (non-orthogonal) basis functions of neighboring cells (see Figure 4).

The smooth approximation query algorithm runs as follows:

1. **Procedure**  $SmoothQuery(tn, \vec{x})$
2.      $sonnum = GetSon3(tn, \vec{x}, sons)$
3.     for( $i = 0; i < sonnum; i++$ )
4.          $answer = SmoothQuery(sons[i], \vec{x})$
5.     endfor
6.      $answer = answer + (tn \rightarrow val) \cdot \phi(tn \rightarrow center, \vec{x}, tn \rightarrow level)$
7. return  $answer$

The function  $GetSon3(tn)$  finds, among the sons of  $tn$ , cells that are close enough to  $\vec{x}$  and assigns pointers to these sons in the array  $sons[]$ . The procedure  $\phi(tn \rightarrow center, \vec{x}, tn \rightarrow level)$  calculates the contribution of every such cell to the answer. This contribution is a function of the distance between the cells center, stored in  $tn \rightarrow center$  and the query coordinate  $\vec{x}$ , prescaled with respect to the resolution level  $l$ ,

$$\phi(\vec{x}, \vec{y}, l) = e^{-|\frac{\vec{x}-\vec{y}}{2^l}|^2 \cdot \frac{D}{2}}. \quad (9)$$

## 2.6 Algorithm Analysis

**Training time.** The procedure  $LearnPoint()$  performs the operations in lines 2–5 in constant time and the procedure  $GetSon()$  returns a pointer to the unique son (among up to  $2^D$  cells), that contains the training point, in  $O(\lg(2^D)) = O(D)$  operations. The call to procedure  $UpdateSons()$  in line 7 can take as much as  $O(\min\{2^D, M\})$  operations, which is the maximal number of sons. However, the number of sons at level  $l$  is on average less than  $\min\{\frac{M}{2^{l \times D}}, 2^D\}$ . The root has the largest number of sons, however, a slight modification of the  $LearnPoint()$  allows us to update them only once, at the end of the training phase. For this modified learning algorithm, the learning complexity of a single training point can be estimated by  $O(\min\{2^D, \frac{M}{2^D}\})$ . In both cases, the complexity of learning a training point is bounded by the constant  $2^D$ .

**Memory requirement.** In the worst case, every training point occupies  $MaxLev - 1$  independent cells. In this case the memory needed for storage of the tree is  $O(MaxLev \cdot M) = O(M)$ . However, in the

case of a redundant training set, where some training points are encountered several times or close training points have the same values, they occupy the same cells.

Figures 7 and 8 illustrate the decomposition of two dimensional feature spaces for two different training sets. One can see that the first, ‘simpler’ training set requires less coefficients than the second one.

**Query time.** The  $Query()$  calls to the  $GetSon2()$  procedure up to  $MaxLev$  times,  $GetSon2()$  returns the pointer to the relevant cell in the  $O(D)$  time. Besides this,  $Query()$  has one comparison (line 3) and one addition (line 6). Therefore, the  $Query()$  complexity is  $O(MaxLev \cdot D)$ , taking  $D = 10$  and  $MaxLev = 5$ , the number of processor operations per query is estimated to be  $MaxLev \cdot (2 + D) = 60$ . For most queries, the cells at the high resolution levels do not exist, and the query involves fewer operations (see line 3).

The query time of the  $SmoothQuery()$  procedure depends on the number of sons that are returned by the  $GetSon3()$  procedure. In the case where the number of the neighboring cells that are evaluated at every resolution level is bounded by a constant  $A$ , the query time is  $O(A \cdot MaxLev \cdot D)$ .

### 3 Experimental Results

The proposed method was implemented in VC++ 6.0 and run on ‘IBM PC 300 PL’ with 600MHZ Pentium III processor and 256MB RAM. It was tested on the Pima Indians Diabetes dataset [10], and a large artificial dataset generated with the DatGen program [11]. The results were compared to the Smooth SVM [12] and Sparse Grids [3].

#### 3.1 Pima Indians

The Pima Indians Diabetes training points were prescaled into a  $[0, 1]^8$  cube and the histogram was equalized along each coordinate. The classification performance was tested with a ‘leave-one-out’ procedure. The results for this dataset are shown in Table 1.

The table presents classification performance both for familiar training points (Train Performance) and the new queries (Test Performance), the training and the query times (per training point) and the memory required for the storage of the learning tree. The first column presents the results for the interpolation algorithm (procedure  $Query()$ ), while the second column for approximation (procedure  $SmoothQuery()$ ). One can see that approximation takes  $10^3$  longer time, since in the function evaluation not only the cells, containing the training point but also their neighbor cells are taken into account.

This training set is relatively small and cannot really benefit from the speed and memory efficiency of the proposed method. The training set size  $M = 768 - 1 = 767$  is comparable to the number of the cells at the first resolution level  $2^D = 256$  ( $M \lesssim 2^D$ ), therefore, this is the sparse case, what explains the advantage of  $SmoothQuery()$ .

The classification performance of 76.16%, achieved by  $SmoothQuery()$  is slightly lower than the best result for SSVM from [12] 78.12%. However our training time of  $24.8 \cdot 10^{-6} \cdot 768 \cdot 10 = 0.19$  sec for 10-fold cross-validation is better than 1.54 sec on 64 UltraSPARC II 250 MHz processors with 8 Gbyte RAM in [12]. This shows that our method is two orders of magnitude better in the training speed.

The best performance achieved in [3] was 77.47% at level 1 and 75.01% at level 2. The training time was 2-3 minutes for level 1 and  $\sim 30$  minutes for level 2, while the required memory was 250 MB for level 2 [13]. This shows the advantage of our method  $\sim 10^4$  times better in training speed and  $\sim 10^4$  times better in memory.

#### 3.2 Large Synthetic Dataset in 6D

Another example is the large artificially generated dataset in 6-dimensional feature space with up to 5 million training points. This dataset was generated with the *DatGen* program [11] using the command

Option	Interpolation	Approximation
Training time (sec)	$24.80 \cdot 10^{-6}$	$24.80 \cdot 10^{-6}$
Query time (sec)	$11.62 \cdot 10^{-6}$	$16.35 \cdot 10^{-3}$
Memory required, (kbyte)	20	20
Train performance (%)	100.00	77.60
Query performance (%)	70.5	76.16

Table 1: Experimental results for the Pima Indians dataset

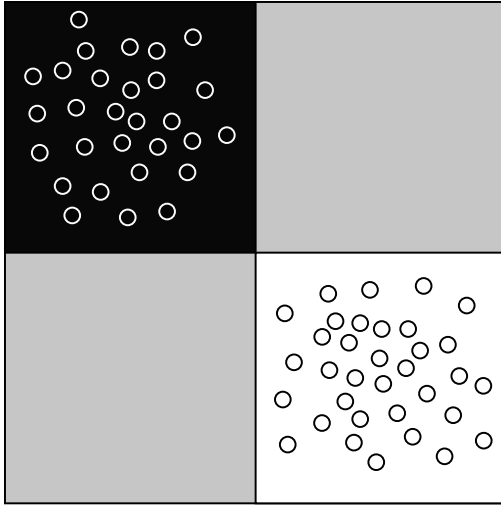


Figure 7: Partition of 2D feature space for a simple training set.

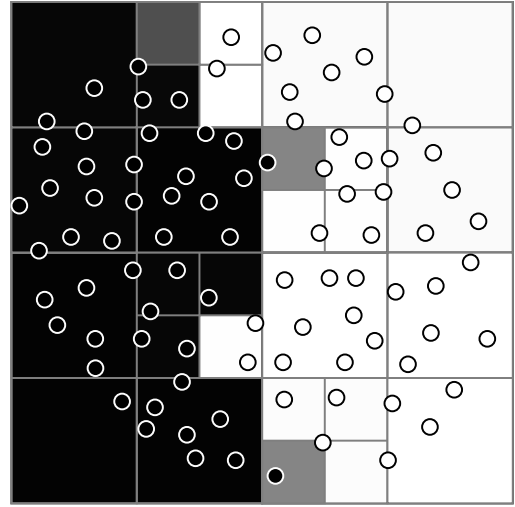


Figure 8: Partition of 2D feature space for a complex training set.

string:

```
sh >datgen -r1 -X0/100, R,O:0/100, R,O:0/100, R,O:0/100, R,O:0/100, R,O:0/200, R,O:0/200 -R2
-C2/4 -D2/5 -T10/60 -O5020000 -p -e0.15
```

In order to compare, the DatGen is used as in Section 3.2.2 of [3].

Table 2 shows the results of [3] and the corresponding results, obtained by our method. The results of [3] are shown in the upper part, while our results appear at the lower part of the table.

The processing times for our method do not include the loading time from a disk, which was 2.05 sec for 50k points and 20.5 sec for 500k points. One can see that both training and testing correctness are similar in both methods at resolution levels 1 and 2. However, our approach has an essential run-time advantage.

## 4 Summary

We proposed to approach the classification problem as a problem of constructing an approximation function  $f$  of the training set. This approximation is constructed with a multi-resolution sparse grid and organized in a tree-structure. For low dimensional training sets ( $D \lesssim 10$ ) this gives an advantage in both runtime and memory usage, while maintaining classification performances comparable to the best reported results of existing techniques.

## Acknowledgements

We would like to thank Alfred Bruckstein for the inspiring conversations, Irad Yavneh and Shaul Markovitz for the important references, and Israel Bar-David for his helpful comments.

## References

- [1] T. Evgeniou, M. Pontil, and T. Poggio. Regularization networks and support vector machines. *Advances in Computational Mathematics* 13(1), pages 1–50, 2000.
- [2] V. N. Vapnik. *Statistical Learning Theory*. John Wiley and Sons, 1998.
- [3] J. Garcke, M. Griebel, and M. Thess. Data mining with sparse grids. [citeseer.nj.nec.com/388914.html](http://citeseer.nj.nec.com/388914.html), 2000.
- [4] C. Bernard. *Wavelets and ill posed problems: optic flow and scattered data interpolation*. PhD thesis, 1999.
- [5] A. N. Tikhonov and V. Y. Arsenin. *Solutions of Ill-Posed Problems*. V. H. Winston & Sons, J. Wiley & Sons, Washington D.C., 1977.
- [6] F. Girosi, M. Jones, and T. Poggio. Regularization theory and neural networks architectures. *Neural Computation*, 7:219–269, 1995.
- [7] T. Joachims. Making large-scale SVM learning practical. *Advances in Kernel Methods: Support Vector Machines*, MIT Press, Cambridge, MA, 1998.

Ref. [3]	# of points	training correctness	testing correctness	runtime, sec		
Level 1	$5 \cdot 10^4$	87.9	88.1	158		
	$5 \cdot 10^5$	88.0	88.1	1570		
	$5 \cdot 10^6$	88.0	88.0	15933		
Level 2	$5 \cdot 10^4$	89.2	89.3	1155		
	$5 \cdot 10^5$	89.4	89.2	11219		
	$5 \cdot 10^6$	89.3	89.2	112656		
This Work	# of points train& test	training correctness	testing correctness	train time,sec	test time,sec	memory used
Level 1	50k & 50k	86.2	86.2	0.35	0.25	3.1k
	500k & 500k	86.4	86.5	3.4	2.4	3.1k
Level 2	50k & 50k	91.6	89.3	0.75	0.45	163k
	500k & 500k	90.7	90.5	9.5	4.5	197k
Level 3	50k & 50k	98.2	88.8	1.55	0.85	1.53M
	500k & 500k	95.4	90.6	16.5	9.5	5.40M
Level 4	50k & 50k	99.9	88.8	2.05	1.15	3.86M
	500k & 500k	99.5	90.3	21.5	12.5	24.7M
Level 5	50k & 50k	100.0	88.8	2.55	1.25	6.26M
	500k & 500k	100.0	90.3	28.5	15.5	48.6M

Table 2: Experimental results for a 6D dataset, work [3] and this work

- [8] S. G. Mallat. *A Wavelet Tour of Signal Processing*, pp.221-224. Academic Press, 1999.
- [9] I. Blayvas, A. Bruckstein, and R. Kimmel. Efficient threshold surfaces for image binarization. In *Computer Vision and Pattern Recognition*, 2001.
- [10] C. L. Blake and C. J. Merz. UCI repository of machine learning databases. <http://www.ics.uci.edu/~mllearn/MLRepository.html>, 1998. University of California, Irvine, Dept. of Information and Computer Sciences.
- [11] G. Melli. Datgen: A program that creates structured data. <http://www.datgen.com>.
- [12] Y. Lee and O. Mangasarian. SSVM: A smooth support vector machine for classification, 1999.
- [13] J. Garcke. Private communication.