A Flexible Implementation for Support Vector Machines

Roland Nilsson
Johan Björkegren
Jesper Tegnér

Support vector machines (SVMs) are learning algorithms that have many applications in pattern recognition and nonlinear regression. Being very popular, SVM software is available in many versions. Still, existing implementations, usually in low-level languages such as C, are often difficult to understand and adapt to specific research tasks. In this article, we present a compact and yet flexible implementation of SVMs in *Mathematica*, traditionally named *MathSVM*. This software is designed to be easy to extend and modify, drawing on the powerful high-level language of *Mathematica*.

■ Background

A *pattern recognition problem* amounts to learning how to discriminate between data points x_i belonging to two *classes*, defined by class labels $y_i \in \{+1, -1\}$, when given only a set of examples (x_i, y_i) from each class. These problems are found in various applications, from automated handwriting recognition to medical expert systems, and pattern recognition or *machine learning* algorithms are routinely applied to solve them.

It may be helpful for newcomers to relate this to a more familiar problem: standard statistical hypothesis testing for one-dimensional x_i, such as Student's t-test [1, ch. 8], can be viewed as a very simple kind of pattern recognition problem. Here, the hypotheses H_0 and H_1 correspond to classes $+1$, -1, and the familiar

$$g(x) = \left\{ \begin{array}{ll} H_0, & \text{if } x < \overline{x} \\ H_1, & \text{if } x > \overline{x} \end{array} \right\},$$

where $\overline{x}$ is the mean of within-class means,

$$\overline{x} = \frac{1}{2}\left(\frac{1}{l_1}\sum\nolimits_{i:y_1=+1} x_i + \frac{1}{l_{-1}}\sum\nolimits_{i:y_1=-1} x_i\right),$$

$l_c = |\{i : y_1 = c\}|$ and $H_0 < H_1$, is called the *decision rule* or sometimes simply the *classifier*. We say that the decision rule g is induced from data x_i, in this case determined by computing $\overline{x}$.

However, real pattern recognition problems usually involve high-dimensional data (such as image data) and unknown underlying distributions. In this situation, it is nearly impossible to develop statistical tests like the preceding one. These problems are typically attacked with algorithms, such as artificial neural networks [2], decisions trees [3, ch. 18], Bayesian models [4], and recently SVMs [5], to which we will devote the rest of this article. Here we will only consider data that can be represented as vectors $x \in R^n$; other kinds of information can usually be changed to this form in some appropriate manner.

■ Support Vector Machines

SVMs attempt to find a hyperplane $\Pi_{w,b} = w.x + b = 0,\ x \in R^n$ that separates the data points x_i (meaning that all x_i in a given class are on the same side of the plane), corresponding to a decision rule

$$g(x) = \text{sign}(w.x + b).$$

In SVM literature, w is often referred to as the *weight vector*; b is called the *bias* (a term adopted from neural networks). This idea is not new; it dates back at least to R.A. Fisher and the theory of *linear discriminants* [6]. The novelty of SVMs lies in how this plane is determined: SVMs choose the separating hyperplane $w.x + b = 0$ that is furthest away from the data points x_i, that is, that has maximal *margin* (Figure 1). The underlying idea is that a hyperplane far from any observed data points should minimize the risk of making wrong decisions when classifying new data. To be precise, in SVMs we maximize the distance to the closest data points. We solve

$$\max_{(w,b)} (\min_i d(\Pi_{w,b}, x_i)), \tag{1}$$

where $d(\Pi_{w,b}, x_i) = |w.x_i + b| / \|w\|$ is the distance between data point i and the plane Π_w, subject to the constraint that this plane still separates the classes. The plane Π_w that solves (1) is called the *optimal separating hyperplane* and is unique [5]. *MathSVM* provides algorithms for determining this plane from data.

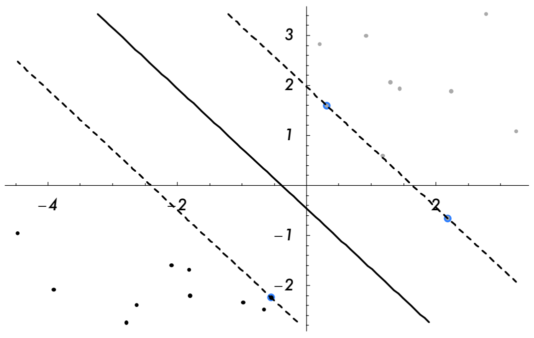


Figure 1. Two-class data (black and grey dots), their optimal separating hyperplane (continuous line), and support vectors (circled in blue). This is an example output of the `SVMPlot` function in *MathSVM*. The width of the "corridor" defined by the two dotted lines connecting the support vectors is the margin of the optimal separating hyperplane.

■ Solving the Optimization Problem

□ The Primal Problem

It turns out that the optimal separating hyperplane solving (1) can be found as the solution to the equivalent optimization problem

$$\begin{gathered}\min_{w,b} \frac{1}{2} \| w \|^2 \\ \text{subject to } y_i (w^T x_i + b) \geq 1,\end{gathered} \tag{2}$$

referred to as the *primal* problem. Typically, only a small subset of the data points will attain equality in the constraint; these are termed *support vectors* since they are "supporting" (constraining) the hyperplane (Figure 1). In fact, the solution (w, b) depends only on these specific points. Therefore, the method also is a scheme for data compression, in the sense that the support vectors contain all the information necessary to derive the decision rule.

□ The Dual Problem

For reasons that will become clear later, w often has very high dimension, which makes the primal problem (2) intractable. Therefore, we attack (2) indirectly, by solving the *dual problem*

$$\begin{gathered}\min_{\alpha} \alpha^T Q\alpha - \alpha \\ \text{subject to } \alpha_i \geq 0, \;\; y^T \alpha = 0,\end{gathered} \tag{3}$$

where $Q = (q_{i\,j}) = (y_i\, y_j\, x_j\, x_i)$. This is a quadratic programming (QP) problem, which has a unique solution whenever Q is positive semidefinite (as is the case here). It is solved numerically in *MathSVM* by the function `QPSolve`. (At this point we need to load the *MathSVM* package; see Additional Material.)

```
In[1]:=  << MathSVM`
         << Statistics`NormalDistribution`

In[3]:=  ? QPSolve
```

```
QPSolve[Q,p,a,b,c,y,τ] solves the quadratic programming problem min
  α.Q.α+p.α, subject to a≤α≤b and y.α=c. QPSolve uses the GSMO
  algorithm described by Keerthi et al. τ is a solution tolerance
  parameter (0.01 or so is usually good enough for SVMs). Q must
  be a positive semidefinite matrix to guarantee convergence.
```

The variable α has $\dim \alpha = l$, the number of data points, so the matrix Q has l^2 elements, which may be quite large for large problems. Therefore, `QPSolve` employs a divide-and-conquer approach [7] that allows for solving (3) efficiently without storing the full matrix Q in memory.

Having solved the dual problem for α using `QPSolve`, we obtain the optimal weight vector w and bias term b, that is, the solution to the primal problem (2), using the identities

$$w = \sum_i \alpha_i \, y_i \, x_i \tag{4}$$

$$b = -\frac{1}{2} \left(w^T x_+ + w^T x_-\right), \tag{5}$$

where in (5) x_+, x_- are any two support vectors belonging to class $+1$ and -1, respectively (there always exist at least two such support vectors) [5].

☐ A Simple SVM Example

Enough theory—let us generate some data and solve a simple SVM problem using *MathSVM*.

```
In[4]:=  len = 20;
         X = Join[
            RandomArray[NormalDistribution[-2, 1], {len / 2, 2}],
            RandomArray[NormalDistribution[2, 1], {len / 2, 2}]];
         y = Join[Table[1, {len / 2}], Table[-1, {len / 2}]];
```

For this elementary problem, we use the simple SVM formulation (2) provided in *MathSVM* by the `SeparableSVM` function.

```
In[7]:=  τ = 0.01;
         α = SeparableSVM[X, y, τ]

Out[8]=  {0.405345, 0, 0, 0, 0, 0, 0, 0, 0,
          0.102479, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.507824}
```

The returned vector α is the solution found by `QPSolve` for the dual formulation (3) of the SVM problem we just constructed. For this specific problem, the dual formulation used is exactly that described by (3), that is

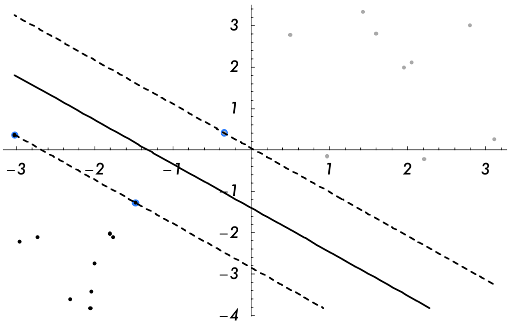
$$\begin{aligned}
&Q = (q_{i\,j}) = (y_i\, y_j\, x_j\, x_i) \\
&p = (-1, \ldots, -1) \\
&a = (0, \ldots, 0),\ b = (0, \ldots, 0) \\
&c = 0
\end{aligned}$$

The support vectors are immediately identifiable as the nonzero α_i. (4) and (5) are implemented as

```
In[9]:= WeightVector[α, X, y]

Out[9]= {-0.73531, -0.692296}

In[10]:= Bias[α, X, y]

Out[10]= -0.973496
```

A plot similar to Figure 1 is produced by the `SVMPlot` function. As in Figure 1, the solid line marks the optimal hyperplane, and dotted lines mark the width of the corridor that joins support vectors (highlighted in blue).

```
In[11]:= SVMPlot[α, X, y]
```



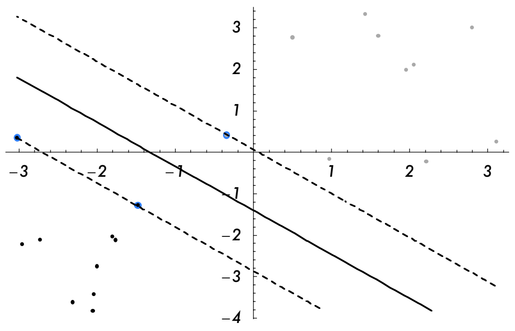
□ Nonseparable Data

Often the assumption of separable training data is not reasonable (it may fail even for the preceding simple example). In such cases, `NonseparableSVM` should be used. This SVM variant takes a parameter C that determines how hard points violating the constraint in (2) should be penalized. This parameter appears in the objective function of the primal problem, which now is formulated as [5]

$$\min_{w,b,\xi} \frac{1}{2} \| w.w \| + C \sum_i \xi_i$$
$$\text{subject to } y_i(x_i.w + b) \geq 1 - \xi_i,\ \xi_i \geq 0.$$

Large C means high penalty, and in the limit $C \to \infty$ we obtain the separable case.

```
In[12]:=  τ = 0.01;
          α = NonseparableSVM[X, y, 0.5, τ]

Out[13]=  {0.3991, 0, 0, 0, 0, 0, 0, 0, 0, 0.1009, 0, 0, 0, 0, 0, 0, 0, 0, 0.5}

In[14]:=  SVMPlot[α, X, y]
```



□ Getting QP Formulations

Sometimes it is interesting to examine what the QP problem looks like for a given SVM formulation. Using the option `FormulationOnly`, we can inspect the various parameters instead of actually solving the QP. This can, for example, be used to study expressions analytically.

```
In[15]:=  Clear[X, y, α, len]

In[16]:=  τ = 0.01;
          NonseparableSVM[Array[x, {2, 2}],
           Array[y, {2}], C, τ, FormulationOnly → True]

Out[17]=  {{{(x[1, 1]^2 + x[1, 2]^2) y[1]^2,
              (x[1, 1] x[2, 1] + x[1, 2] x[2, 2]) y[1] y[2]},
            {(x[1, 1] x[2, 1] + x[1, 2] x[2, 2]) y[1] y[2],
              (x[2, 1]^2 + x[2, 2]^2) y[2]^2}},
           {-1, -1}, {0, 0}, {C, C}, 0, {y[1], y[2]}, 0.01}
```

The parameters are given in the order corresponding to the arguments of `QPSolve`.

■ Feature Space and Kernels

In all of the preceding examples, the separating surface is assumed to be linear (a hyperplane). This is often a serious limitation, as many pattern recognition problems are inherently nonlinear in the input data and require nonlinear separating surfaces. To overcome this obstacle, for each specific problem we devise some appropriate transformation $\phi(x)$ from *input space* X (the domain of the original data) to a *feature space* H. The function ϕ is chosen so that a hyperplane in H corresponds to some desirable class of surfaces in X. (Choosing this function for a specific problem is something of an art, but we can always try a few different ϕ known to have been successful in similar problems and simply hope for the best.)

As an example, consider a second-degree polynomial surface in R^2, described by $w_1 + w_2\, x_1 + w_3\, x_2 + w_4\, x_1{}^2 + w_5\, x_1\, x_2 + w_6\, x_2{}^2 = 0$. We may represent such a surface by choosing a mapping $X \to H$ as

$$\phi(x_1, x_2) = (1, x_1, x_2, x_1{}^2, x_1\, x_2, x_2\, x_1, x_2{}^2) \tag{6}$$

and forming a hyperplane in H defined by $w.z + b = 0$, where $z = \phi(x)$. If we now solve the problem in the new variables z, we obtain a wide-margin hyperplane in H corresponding to a second-degree surface in X.

The problem with this approach is that nonlinear ϕ-transformations may have huge dimensions. Even for the simple quadratic surface considered here, we obtain $\dim H = 1 + n + n^2$ when $\dim X = n$. For increasing polynomial degree, this number grows quickly, and there are even nonlinear functions that result in infinite-dimensional H. This is why we do not solve the primal problem (2) directly: w may not be a finite vector, but α always is.

This problem is elegantly solved by *kernels*. It turns out that there are many functions ϕ for which we can compute scalar products $\phi(x).\phi(y)$ in H implicitly, without actually calculating ϕ. And in fact, this is all we need for solving the SVM formulation (1). Thus we define the kernel function as

$$K(x,\ y) = \phi(x).\phi(y)$$

In the case of example (6), it turns out that $K(x,\ y) = \phi(x).\phi(y) = (1 + x.y)^2$, which is why we chose that specific form of ϕ (explaining the two separate terms $x_1\, x_2$ and $x_2\, x_1$). It is not difficult to prove that this result holds for any polynomial degree d; therefore, using the polynomial kernel

$$K_d(x,\ y) = (1 + x.y)^d$$

we can obtain any polynomial separating surfaces.

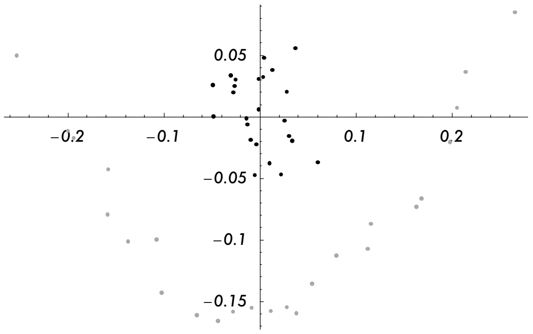
□ A Nonlinear Example: Using Kernels

Let us see how kernels are handled in *MathSVM* to solve nonlinear problems. The second-degree kernel in (6) is provided by

```
In[18]:=  PolynomialKernel[x, y, 2]

Out[18]=  (1 + x.y)^2
```

Here is some data a linear classifier cannot possibly cope with.

```
In[19]:= len = 50;
         X = Join[
            RandomArray[NormalDistribution[0, 0.03], {len/2, 2}],
            Table[
             {Random[NormalDistribution[i/len - 1/4, 0.01]],
              Random[NormalDistribution[(2 i/len - 1/2)^2 - 1/6, 0.01]]},
             {i, len/2}]];
         y = Join[Table[1, {len/2}], Table[-1, {len/2}]];
         SVMDataPlot[X, y, PlotRange → All]
```

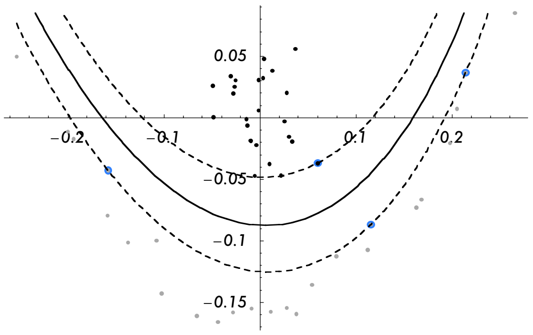


Let us solve this problem using the polynomial kernel. This is done as before, by supplying the desired kernel (which can be any function accepting two arguments) using the `KernelFunction` option.

```
In[23]:= τ = 0.01;
         pk = PolynomialKernel[#1, #2, 2] &;
         α = SeparableSVM[X, y, τ, KernelFunction → pk]

Out[25]= {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
          0, 0, 3826.52, 0, 0, 0, 0, 0, 0, 0, 1146.88, 0, 0, 0, 0, 0,
          0, 0, 0, 0, 0, 0, 0, 0, 0, 1644.97, 0, 0, 0, 0, 1034.67, 0}
```

When visualizing the results, `SVMPlot` can use the kernel functions to draw any nonlinear decision curves.

```
In[26]:= SVMPlot[α, X, y, KernelFunction → pk]
```



```
In[27]:= Clear[len, X, y, α, pk]
```

□ High-Dimensional Input Spaces

An interesting consequence of the kernel idea is that the dimensionality of the input space X does not matter for the time-complexity of the SVM algorithm. Since the solution is computed using only dot products between samples (in input space or some feature space), high-dimensional problems are solved equally fast (not considering the time used to precalculate the kernel matrix Q, which is usually not noticeable). As an example of this, consider a problem with dimension $n = 1000$.

```
In[28]:= len = 20; n = 1000;
         X = Join[
            RandomArray[NormalDistribution[-2, 1], {len / 2, n}],
            RandomArray[NormalDistribution[2, 1], {len / 2, n}]];
         y = Join[Table[1, {len / 2}], Table[-1, {len / 2}]];
```

The kernel matrix is still just $l \times l$.

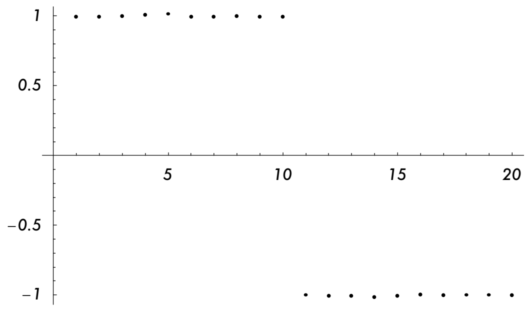
```
In[31]:= KernelMatrix[IdentityKernel, X] // Dimensions

Out[31]= {20, 20}
```

The SVM algorithm is still fast, although the problem is much harder due to extremely low sample density, which is reflected by more support vectors (the nonzero α_i).

```
In[32]:= τ = 0.01;
         (α = SeparableSVM[X, y, τ]) // Timing

Out[33]= {0.79 Second, {0.0000185576, 9.32288 × 10^-6,
           0, 0, 0, 0.000031162, 0.0000215259, 0, 0.0000267382,
           0.0000171359, 1.00351 × 10^-6, 0.0000297867, 0.0000293384, 0,
           0.0000131243, 0, 0.000024641, 0.000020647, 0, 5.90165 × 10^-6}}
```

The solution in this case may be viewed using the projection of data onto the weight vector w. The separation looks almost perfect, although there will be problems with *overfitting* (the solution may not work well when applied to new, unseen examples x). However, this problem is outside the scope of the present article.

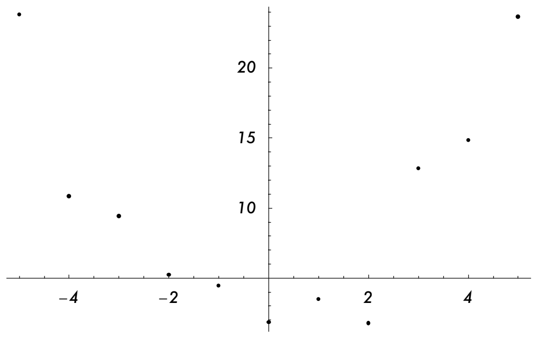
```
In[34]:= ListPlot[X.WeightVector[α, X, y]]
```



■ Regression Analysis with SVMs

So far we have considered SVMs as a tool for pattern recognition only. It is also possible to use the SVM framework for regression problems. Consider a function $y = f(x)$ to be approximated; for example, a quadratic.

```
In[35]:= X = Range[-5, 5]; len = Length[X];
         y = Map[#^2 + Random[NormalDistribution[0, 2]] &, X];
         ListPlot[Thread[{X, y}]]
```

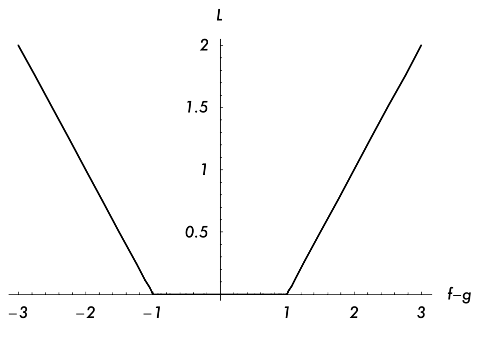


We can adapt the SVM method to the regression setting by using a *ϵ-insensitive loss function*

$$L_\epsilon(f(x), g(x)) = \begin{cases} 0, & \| f(x) - g(x) \| < \epsilon \\ \| f(x) - g(x) \| - \epsilon, & \text{otherwise} \end{cases}$$

where $g(x)$ is the SVM approximation to the regression function $f(x)$. This loss function determines how much a deviation from the true $f(x)$ is penalized; for deviations less than ϵ, no penalty is incurred. Here is what the loss function looks like.

```
In[38]:= Plot[If[Abs[x] < 1, 0, Abs[x] - 1], {x, -3, 3}, AxesLabel → {"f-g", "L"}]
```

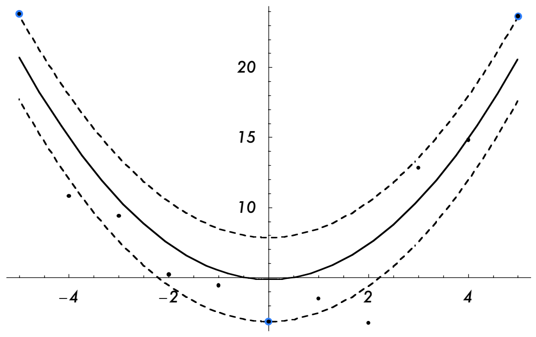


Using this idea, the regression problem is transformed to a classification problem: any x such that $L_\epsilon(f(x), g(x)) = 0$ may be considered "correctly classified." *MathSVM* solves such problems using the `RegressionSVM` function, parameterized by ϵ and a penalty constant C. Here we again try a polynomial kernel.

```
In[39]:= pk = PolynomialKernel[#1, #2, 2] &;
         ϵ = 3; c = 0.5;
         τ = 0.01;
         α = RegressionSVM[X, y, c, ϵ, τ, KernelFunction → pk]

Out[42]= {0, 0, 0, 0, 0, 0.0252908, 0, 0, 0, 0,
          0, 0.0133908, 0, 0, 0, 0, 0, 0, 0, 0, 0.0119}
```

The function `RegressionSVMPlot` provides convenient plotting of the resulting regression function. As with `SVMPlot`, the kernel type used is supplied as a parameter. Note how support vectors in this case are chosen as the data points that are furthest away from the regression line.

```
In[43]:= RegressionSVMPlot[α, X, y, ϵ, KernelFunction → pk]
```



```
In[44]:= RegressionBias[α, X, y, ϵ, KernelFunction → pk]

Out[44]= 4.83561
```

We can, of course, also obtain the analytical expression of the estimated regression function.

```
In[45]:= RegressionFunction[α, X, y, ϵ, x, KernelFunction → pk]
```

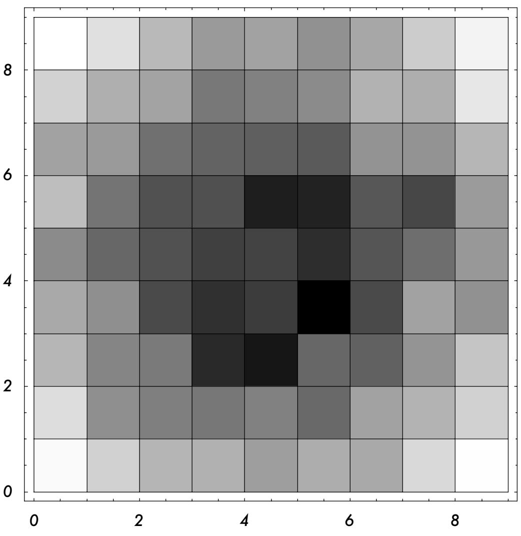
Out[45]= $4.81032 + 0.0133908\,(1 - 5\,x)^2 + 0.0119\,(1 + 5\,x)^2$

```
In[46]:= Clear[α, X, y, pk, ϵ, c, len]
```

□ Two-Dimensional Example

We can use SVM regression with domains of any dimension (that is the main advantage). Here is a simple two-dimensional example.

```
In[47]:= X = Table[{i, j}, {i, -4, 4}, {j, -4, 4}];
         y = Map[
            ((#[[1]]^2 + #[[2]]^2) / 5 + Random[NormalDistribution[0, 0.5]]) &, X, {2}];
         ListDensityPlot[y]
         X = Flatten[X, 1]; y = Flatten[y];
```



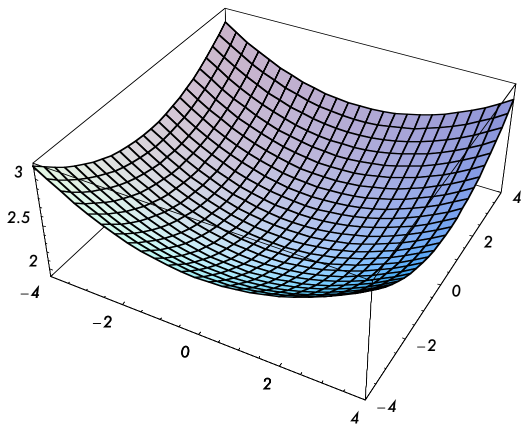
```
In[51]:= pk = PolynomialKernel[#1, #2, 2] &;
         ϵ = 3; c = 0.5;
         τ = 0.01;
         α = RegressionSVM[X, y, c, ϵ, τ, KernelFunction → pk]

Out[54]= {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
          0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.00231481, 0, 0, 0, 0, 0, 0, 0, 0,
          0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
          0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.00109909,
          0, 0, 0, 0, 0, 0, 0, 0.00059379, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
          0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
          0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
          0, 0, 0, 0, 0, 0, 0, 0, 0, 0.000621928, 0, 0, 0, 0, 0, 0, 0, 0}
```

Here is the regression function.

```
In[55]:= rf = RegressionFunction[α, X, y, ϵ, {i, j}, KernelFunction → pk]

Out[55]= 1.94266 + 0.00109909 (1 - 4 i - 4 j)^2 + 0.000621928 (1 + 4 i - 4 j)^2 -
          0.00231481 (1 - i + j)^2 + 0.00059379 (1 - 4 i + 4 j)^2
```

There are no specialized 3D plots for regression in the *MathSVM* package. Here is the usual `Plot3D` visualization.

```
In[56]:=  Plot3D[rf, {i, -4, 4}, {j, -4, 4}]
```



■ Conclusion

In this article, we have demonstrated the utility of the *MathSVM* package for solving pattern recognition and regression problems. This is an area of very active research and these algorithms are evolving quickly. In a rapidly moving field such as this, it is important to have a clear, well documented, high-level approach to implementation to minimize confusion. *Mathematica* provides an excellent solution here, due to its high-level programming language and symbolic capabilities.

MathSVM is currently 100% native *Mathematica* code, written with the emphasis on clarity. This does incur penalties in terms of computational speed. Some parts of the QP algorithm are therefore being ported to Java at this time to improve performance. This should not impair the clarity of the software in any way, since the `QPSolve` function is easy separable from the other parts of *MathSVM* in a "black box" fashion.

The *MathSVM* software is still in its infancy and will no doubt expand rapidly, as our group is currently involved in many projects in pattern recognition and high-dimensional data analysis in general, as well as in a biomedical context. We hope that this contribution will initiate other efforts to bring understandable implementations of machine learning algorithms to the *Mathematica* community.

■ References

[1] G. Casella and R. L. Berger, *Statistical Inference*, 2nd ed., Belmont, CA: Duxbury Press, 2002.

[2] S. Haykin, *Neural Networks: A Comprehensive Foundation*, 2nd ed., Englewood Cliffs, NJ: Prentice Hall, 1999.

[3] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, Englewood Cliffs, NJ: Prentice Hall, 1995.

[4] N. Friedman, "Inferring Cellular Networks Using Probabilistic Graphical Models," *Science*, **303**, 2004 pp. 799–805.

[5] V. N. Vapnik, *Statistical Learning Theory,* New York: John Wiley & Sons, 1998.

[6] R. A. Fisher, "The Statistical Utilization of Multiple Measurements," *Annals of Eugenics*, **8**, 1938 pp. 376–386.

[7] S. S. Keerthi and E. G. Gilbert, "Convergence of a Generalized SMO Algorithm for SVM Classifier Design," *Machine Learning*, **46**, 2002 pp. 351–360.

■ Additional Material

MathSVM.nb
MathSVM.m

Available at www.mathematica-journal.com/issue/v10i1/download.

About the Authors

Roland Nilsson is a graduate student at Linköping University working with machine learning algorithms in analysis of high-dimensional biomedical data.

Johan Björkegren is an associate professor in molecular medicine at Karolinska Institutet, Sweden, and cofounder of Clinical Gene Networks, a biotechnology company involved in system-level analysis of biomedical data in cardiovascular disease.

Jesper Tegnér is a professor of computational biology at Linköping University, Sweden, and cofounder of Clinical Gene Networks.

Roland Nilsson
Computational Biology
Linköping University
SE-58183 Linköping, Sweden
rolle@ifm.liu.se

Johan Björkegren
Center for Genomics and Bioinformatics
Karolinska Institutet
SE-17177 Stockholm, Sweden
johan.bjorkegren@ks.se

Jesper Tegnér
Computational Biology
Linköping University
SE-58183 Linköping, Sweden
jespert@ifm.liu.se