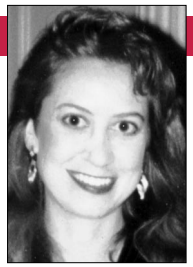


Support vector machines

By Marti A. Hearst
University of California, Berkeley
hearst@sims.berkeley.edu

My first exposure to Support Vector Machines came this spring when I heard Sue Dumais present impressive results on text categorization using this analysis technique. This issue's collection of essays should help familiarize our readers with this interesting new racehorse in the Machine Learning stable. Bernhard Schölkopf, in an introductory overview, points out that a particular advantage of SVMs over other learning algorithms is that it can be analyzed theoretically using concepts from computational learning theory, and at the same time can achieve good performance when applied to real problems. Examples of these real-world applications are provided by Sue Dumais, who describes the aforementioned text-categorization problem, yielding the best results to date on the Reuters collection, and Edgar Osuna, who presents strong results on application to face detection. Our fourth author, John Platt, gives us a practical guide and a new technique for implementing the algorithm efficiently.

–Marti Hearst

SVMs—a practical consequence of learning theory

Bernhard Schölkopf, GMD First

Is there anything worthwhile to learn about the new SVM algorithm, or does it fall into the category of "yet-another-algorithm," in which case readers should stop here and save their time for something more useful? In this short overview, I will try to argue that studying support-vector learning is very useful in two respects. First, it is quite satisfying from a theoretical point of view: SV learning is based on some beautifully simple ideas and provides a clear intuition of what learning from examples is about. Second, it can lead to high performances in practical applications.

In the following sense can the SV algorithm be considered as lying at the intersection of learning theory and practice: for certain simple types of algorithms, statistical learning theory can identify rather precisely the factors that need to be taken into account to learn successfully. Real-world applications, however, often mandate the use of more complex models and algorithms—such as neural networks—that are much harder to analyze theoretically. The SV algorithm achieves both. It constructs models that are complex enough: it contains a large class of neural nets, radial basis function (RBF) nets, and polynomial classifiers as special cases. Yet it is simple enough to be analyzed mathematically, because it can be shown to correspond to a *linear* method in a high-dimensional *feature space* nonlinearly related to input space. Moreover, even though we can think of it as a linear algorithm in a high-dimensional space, in practice, it does not involve any computations in that high-dimensional space. By the use of *kernels*, all necessary computations are performed directly in input space. This is the characteristic twist of SV methods—we are dealing with complex algorithms for nonlinear pattern recognition,[1] regression,[2] or feature extraction,[3] but for the sake of analysis and algorithmics, we can pretend that we are working with a simple linear algorithm.

I will explain the gist of SV methods by describing their roots in learning theory, the optimal hyperplane algorithm, the kernel trick, and SV function estimation. For details and further references, see Vladimir Vapnik's authoritative treatment,[2] the collection my colleagues and I have put together,[4] and the SV Web page at *http://svm.first.gmd.de*.

Learning pattern recognition from examples

For pattern recognition, we try to estimate a function $f:\mathcal{R}^N \to \{\pm 1\}$ using training data—that is, N-dimensional patterns $\mathbf{x}_i$ and class labels y_i,

$$(\mathrm{x}_1,y_1),\ \ldots,(\mathrm{x}_\ell, y_\ell) \in\ \mathcal{R}^N \times \{\pm 1\}, \qquad (1)$$

such that f will correctly classify new examples $(\mathbf{x},y)$—that is, $f(\mathbf{x}) = y$ for examples $(\mathbf{x},y)$, which were generated from the same underlying probability distribution $P(\mathbf{x},y)$ as the training data. If we put no restriction on the class of functions that we choose our estimate f from, however, even a function that does well on the training data—for example by satisfying $f(\mathbf{x}_i) = y_i$ (here and below, the index i is understood to run over $1, \ldots, \ell$)—need not generalize well to unseen examples. Suppose we know nothing additional about f (for example, about its smoothness). Then the values on the training patterns carry no information whatsoever about values on novel patterns. Hence learning is impossible, and minimizing the training error does not imply a small expected test error.

Statistical learning theory,[2] or VC (Vapnik-Chervonenkis) theory, shows that it is crucial to restrict the class of functions that the learning machine can implement to one with a *capacity* that is suitable for the amount of available training data.

Hyperplane classifiers

To design learning algorithms, we thus must come up with a class of functions whose capacity can be computed. SV classifiers are based on the class of hyperplanes

$$(\mathrm{w}\cdot\mathrm{x}) + b = 0\ \ \mathrm{w} \in \mathcal{R}^N, b \in R, \qquad (2)$$

corresponding to decision functions

$$f(\mathrm{x}) = \mathrm{sign}((\mathrm{w}\cdot\mathrm{x}) + b). \qquad (3)$$

We can show that the *optimal hyperplane*, defined as the one with the maximal margin of separation between the two classes (see Figure 1), has the lowest ca-

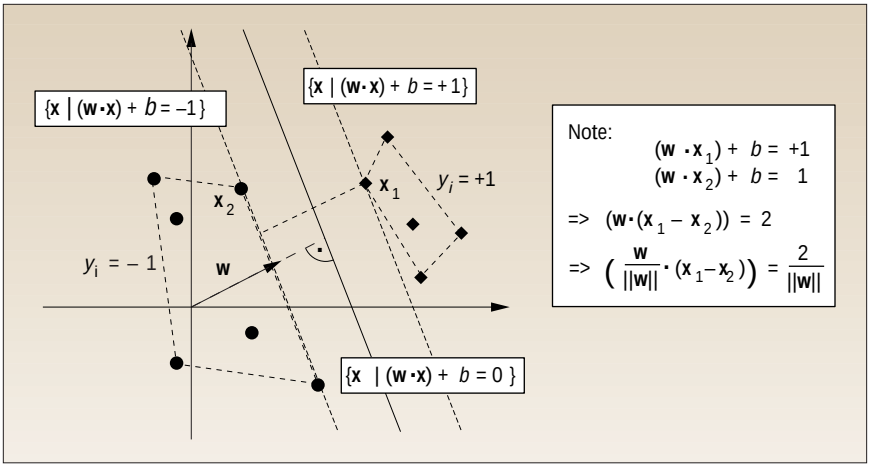


Figure 1. A separable classification toy problem: separate balls from diamonds. The optimal hyperplane is orthogonal to the shortest line connecting the convex hulls of the two classes (dotted), and intersects it half way. There is a weight vector $\mathbf{w}$ and a threshold b such that $y_i \cdot ((\mathbf{w}\cdot\mathbf{x}_i) + b) > 0$. Rescaling $\mathbf{w}$ and b such that the point(s) closest to the hyperplane satisfy $|(\mathbf{w}\cdot\mathbf{x}_i) + b| = 1$, we obtain a form $(\mathbf{w},b)$ of the hyperplane with $y_i((\mathbf{w}\cdot x_i) + b) \geq 1$. Note that the margin, measured perpendicularly to the hyperplane, equals $2/||\mathbf{w}||$. To maximize the margin, we thus have to minimize $|\mathbf{w}|$ subject to $y_i((\mathbf{w}\cdot\mathbf{x}_i) + b) \geq 1$.

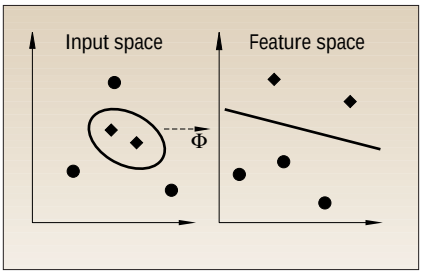


Figure 2. The idea of SV machines: map the training data nonlinearly into a higher-dimensional feature space via Φ, and construct a separating hyperplane with maximum margin there. This yields a nonlinear decision boundary in input space. By the use of a kernel function, it is possible to compute the separating hyperplane without explicitly carrying out the map into the feature space.

pacity. It can be uniquely constructed by solving a constrained quadratic optimization problem whose solution $\mathbf{w}$ has an expansion $\mathbf{w} = \sum_i v_i \mathbf{x}_i$ in terms of a subset of training patterns that lie on the margin (see Figure 1). These training patterns, called support vectors, carry all relevant information about the classification problem.

Omitting the details of the calculations, there is just one crucial property of the algorithm that we need to emphasize: both the quadratic programming problem and the final decision function $f(\mathbf{x}) = \text{sign}(\sum_i v_i(\mathbf{x}\cdot\mathbf{x}_i) + b)$ depend only on dot products between patterns. This is precisely what lets us generalize to the nonlinear case.

Feature spaces and kernels

Figure 2 shows the basic idea of SV machines, which is to map the data into some other dot product space (called the *feature space*) F via a nonlinear map

$$\Phi: \mathcal{R}^N \to F, \tag{4}$$

and perform the above linear algorithm in F. As I've noted, this only requires the evaluation of dot products.

$$k(\mathbf{x},\mathbf{y}) := (\Phi(\mathbf{x})\cdot\Phi(\mathbf{y})). \tag{5}$$

Clearly, if F is high-dimensional, the right-hand side of Equation 5 will be very expensive to compute. In some cases, however, there is a simple *kernel* k that can be evaluated efficiently. For instance, the polynomial kernel

$$k(\mathbf{x},\mathbf{y}) = (\mathbf{x}\cdot\mathbf{y})^d \tag{6}$$

can be shown to correspond to a map Φ into the space spanned by all products of exactly d dimensions of $\mathcal{R}^N$. For d=2 and $\mathbf{x}$, $\mathbf{y} \in \mathcal{R}^2$, for example, we have

$$\begin{aligned}(\mathbf{x}\cdot\mathbf{y})^2 &= \left(\begin{pmatrix}x_1\\x_2\end{pmatrix}\cdot\begin{pmatrix}y_1\\y_2\end{pmatrix}\right)^2 \\ &= \left(\begin{pmatrix}x_1^2\\\sqrt{2}x_1x_2\\x_2^2\end{pmatrix}\cdot\begin{pmatrix}y_1^2\\\sqrt{2}y_1y_2\\y_2^2\end{pmatrix}\right) \\ &= (\Phi(\mathbf{x})\cdot\Phi(\mathbf{y})),\end{aligned} \tag{7}$$

defining $\Phi(\mathbf{x}) = (x_1^2, \sqrt{2}x_1x_2, x_2^2)$. More generally, we can prove that for every kernel that gives rise to a positive matrix $(k(\mathbf{x}_i,\mathbf{x}_j))_{ij}$, we can construct a map Φ such that Equation 5 holds.

Besides Equation 6, SV practitioners use radial basis function (RBF) kernels such as

$$k(\mathbf{x},\mathbf{y}) = \exp(-\|\mathbf{x}-\mathbf{y}\|^2/(2\sigma^2)), \tag{8}$$

and sigmoid kernels (with gain κ and offset Θ)

$$k(\mathbf{x},\mathbf{y}) = \tanh(\kappa(\mathbf{x}\cdot\mathbf{y}) + \Theta). \tag{9}$$

SVMs

We now have all the tools to construct nonlinear classifiers (see Figure 2). To this end, we substitute $\Phi(\mathbf{x}_i)$ for each training example $\mathbf{x}_i$, and perform the optimal hyperplane algorithm in F. Because we are using kernels, we will thus end up with nonlinear decision function of the form

$$f(\mathbf{x}) = \text{sign}\left(\sum_{i=1}^{\ell} v_i \cdot k(\mathbf{x},\mathbf{x}_i) + b\right). \tag{10}$$

The parameters v_i are computed as the solution of a quadratic programming problem.

In input space, the hyperplane corresponds to a nonlinear decision function whose form is determined by the kernel (see Figures 3 and 4).

The algorithm I've described thus far has a number of astonishing properties:

- It is based on statistical learning theory,
- It is practical (as it reduces to a quadratic programming problem with a unique solution), and
- It contains a number of more or less heuristic algorithms as special cases: by the choice of different kernel functions, we obtain different architectures (Figure 4), such as polynomial classifiers (Equation 6), RBF classifiers (Equation 8 and Figure 3), and three-layer neural nets (Equation 9).

The most important restriction up to now has been that we were only considering the case of classification. However, a generalization to regression estimation—that is, to $y \in \mathcal{R}$, can be given. In this case, the algorithm tries to construct a linear function in the feature space such that the training points lie within a distance $\varepsilon > 0$. Similar to the pattern-recognition case, we can write this as a quadratic programming problem in terms of kernels. The nonlinear regression estimate takes the form

$$f(\mathbf{x}) = \sum_{i=1}^{\ell} v_i \cdot k(\mathbf{x}_i,\mathbf{x}) + b \tag{11}$$

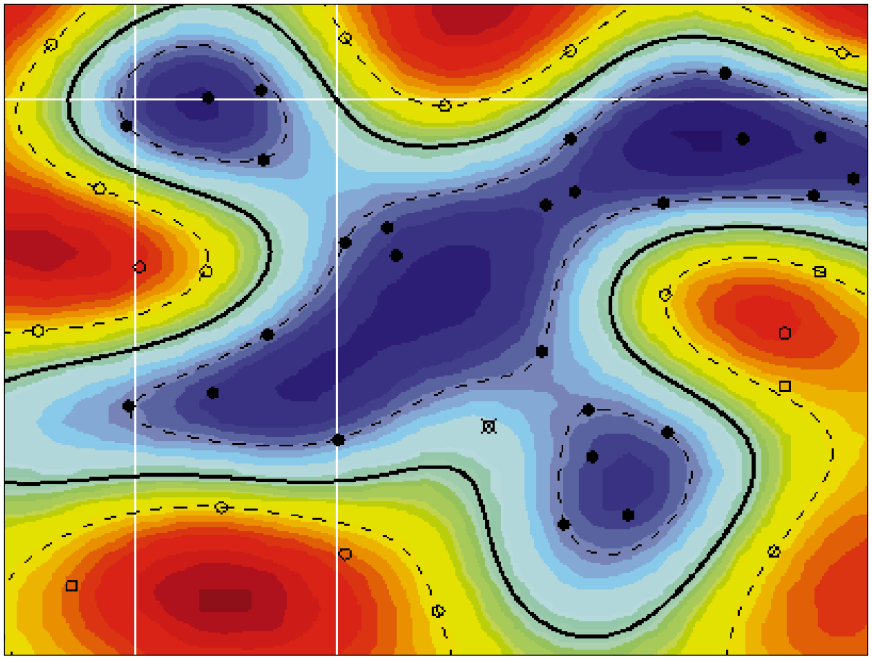


Figure 3. Example of an SV classifier found by using a radial basis function kernel (Equation 8). Circles and disks are two classes of training examples; the solid line is the decision surface; the support vectors found by the algorithm lie on, or between, the dashed lines. Colors code the modulus of the argument $\sum_i v_i \cdot k(\mathbf{x}, \mathbf{x}_i) + b$ of the decision function in Equation 10.

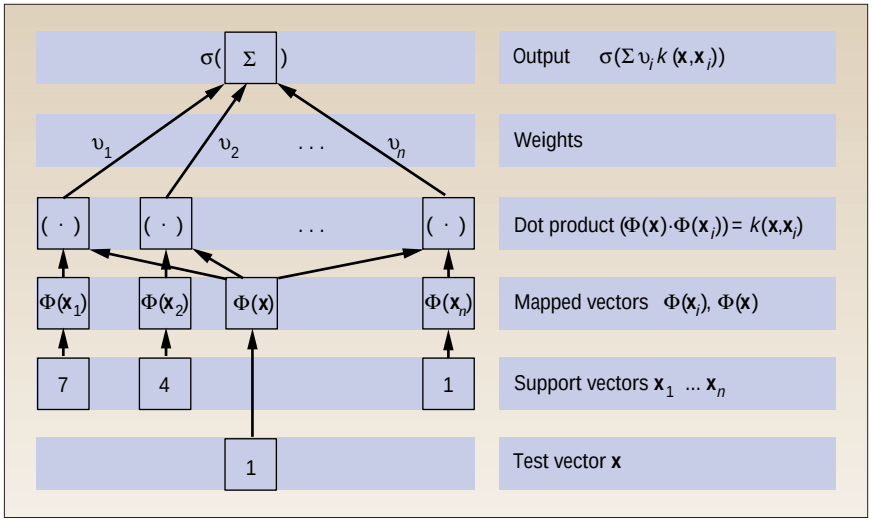


Figure 4. Architecture of SV methods. The input $\mathbf{x}$ and the support vectors $\mathbf{x}_i$ (in this example: digits) are nonlinearly mapped (by Φ) into a feature space F, where dot products are computed. By the use of the kernel k, these two layers are in practice computed in one single step. The results are linearly combined by weights v_i, found by solving a quadratic program (in pattern recognition, $v_i = y_i \alpha_i$; in regression estimation, $v_i = \alpha^*_i - \alpha_i$) [2] or an eigenvalue problem (in kernel PCA[3]). The linear combination is fed into the function σ (in pattern recognition, $\sigma(x) = \text{sign}(x + b)$; in regression stimation, $\sigma(x) = x + b$; in kernel PCA, $\sigma(x) = x$.

To apply the algorithm, we either specify ε a priori, or we specify an upper bound on the fraction of training points allowed to lie outside of a distance ε from the regression estimate (asymptotically, the number of SVs) and the corresponding ε is computed automatically.[5]

Current developments and open issues

Chances are that those readers who are still with me might be interested to hear how researchers have built on the above, applied the algorithm to real-world problems, and developed extensions. In this respect, several fields have emerged.

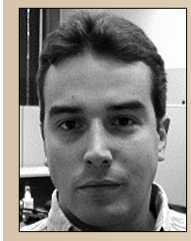
- Training methods for speeding up the quadratic program, such as the one described later in this installment of Trends & Controversies by John Platt.
- Speeding up the evaluation of the decision function is of interest in a variety of applications, such as optical-character recognition.[6]
- The choice of kernel functions, and hence of the feature space to work in, is of both theoretical and practical interest. It determines both the functional form of the estimate and, via the objective function of the quadratic program, the type of regularization that is used to constrain the estimate.[7,8] However, even though different kernels lead to different types of learning machines, the choice of kernel seems to be less crucial than it may appear at first sight. In OCR applications, the kernels (Equations 6, 9, and 8) lead to very similar performance and to strongly overlapping sets of support vectors.
- Although the use of SV methods in applications has only recently begun, application developers have already reported state-of-the-art performances in a variety of applications in pattern recognition, regression estimation, and time series prediction. However, it is probably fair to say that we are still missing an application where SV methods significantly outperform any other available algorithm or solve a problem that has so far been impossible to tackle. For the latter, SV methods for solving inverse problems are a promising candidate.[9] Sue Dumais and Edgar Osuna describe promising applications in this discussion.
- Using kernels for other algorithms emerges as an exciting opportunity for developing new learning techniques. The kernel method for computing dot products in feature spaces is not restricted to SV machines. We can use it to derive nonlinear generalizations of any algorithm that can be cast in terms of dot products. As a mere start, we decided to apply this idea to one of the most widely used algorithms for data analysis, *principal component analysis*. This leads to kernel PCA,[3] an algorithm that performs nonlinear PCA by carrying out linear PCA in feature space. The

method consists of solving a linear eigenvalue problem for a matrix whose elements are computed using the kernel function. The resulting feature extractors have the same architecture as SV machines (see Figure 4). A number of researchers have since started to "kernelize" various other linear algorithms.

References

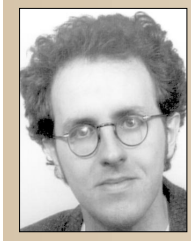
1. B.E. Boser, I.M. Guyon, and V.N. Vapnik, "A Training Algorithm for Optimal Margin Classifiers," *Proc. Fifth Ann. Workshop Computational Learning Theory*, ACM Press, New York, 1992, pp. 144–152.
2. V. Vapnik, *The Nature of Statistical Learning Theory*, Springer-Verlag, New York, 1995.
3. B. Schölkopf, A. Smola, and K.-R. Müller, "Nonlinear Component Analysis as a Kernel Eigenvalue Problem," *Neural Computation*, Vol. 10, 1998, pp. 1299–1319.
4. B. Schölkopf, C.J.C. Burges, and A.J. Smola, *Advances in Kernel Methods—Support Vector Learning*, to appear, MIT Press, Cambridge, Mass, 1998.
5. B. Schölkopf et al., "Support Vector Regression with Automatic Accuracy Control," to be published in *Proc. Eighth Int'l Conf. Artificial Neural Networks, Perspectives in Neural Computing*, Springer-Verlag, Berlin, 1998.
6. C.J.C. Burges, "Simplified Support Vector Decision Rules," *Proc. 13th Int'l Conf. Machine Learning*, Morgan Kaufmann, San Francisco, 1996, pp. 71–77.
7. A. Smola and B. Schölkopf, "From Regularization Operators to Support Vector Kernels," *Advances in Neural Information Processing Systems 10*, M. Jordan, M. Kearns, and S. Solla, eds., MIT Press, 1998.
8. F. Girosi, *An Equivalence between Sparse Approximation and Support Vector Machines*, AI Memo No. 1606, MIT, Cambridge, Mass., 1997.
9. J. Weston et al., *Density Estimation Using Support Vector Machines*, Tech. Report CSD-TR-97-23, Royal Holloway, Univ. of London, 1997.

Susan T. Dumais is a senior researcher in the Decision Theory and Adaptive Systems Group at Microsoft Research. Her research interests include algorithms and interfaces for improved information retrieval and classification, human-computer interaction, combining search and navigation, user modeling, individual differences, collaborative filtering, and organizational impacts of new technology. She received a BA in mathematics and psychology from Bates College and a PhD in cognitive psychology from Indiana University. She is a member of the ACM, the ASIS, the Human Factors and Ergonomic Society, and the Psychonomic Society. Contact her at Microsoft Research, 1 Microsoft Way, Redmond, WA 98052; sdumais@microsoft.com; http://research.microsoft.com/~sdumais.



Edgar Osuna has just returned to his native Venezuela after receiving his PhD in operations research from the Massachusetts Institute of Technology. His research interests include the study of different aspects and properties of Vapnik's SVM. He received his BS in computer engineering from the Universidad Simon Bolivar, Caracas, Venezuela. Contact him at IESA, POBA International #646, PO Box 02-5255, Miami, FL 33102-5255; eosuna@usb.ve.

John Platt is a senior researcher in the Signal Processing Group at Microsoft Research. Recently, he has concentrated on fast general-purpose machine-learning algorithms for use in processing signals. More generally, his research interests include neural networks, machine learning, computer vision, and signal processing. He received his PhD in computer science at Caltech in 1989, where he studied computer graphics and neural networks. His received his BS in chemistry from California State University at Long Beach. Contact him at Microsoft Research, 1 Microsoft Way, Redmond, WA 98005; jplatt@microsoft.com; http://research.microsoft.com/~jplatt.



Bernhard Schölkopf is a researcher at GMD First, Berlin. His research interests are in machine learning and perception, in particular using SVMs and Kernel PCA. He has an MSc in mathematics from the University of London, and a Diploma in physics and a PhD in computer science, both from the Max Planck Institute. Contact him at GMD-First, Rm. 208, Rudower Chaussee 5, D-12489 Berlin; bs@first.gmd.de; http://www.first.gmd.de/~bs.

Using SVMs for text categorization

Susan Dumais, Decision Theory and Adaptive Systems Group, Microsoft Research

As the volume of electronic information increases, there is growing interest in developing tools to help people better find, filter, and manage these resources. *Text categorization*—the assignment of natural-language texts to one or more predefined categories based on their content—is an important component in many information organization and management tasks. Machine-learning methods, including SVMs, have tremendous potential for helping people more effectively organize electronic resources.

Today, most text categorization is done by people. We all save hundreds of files, e-mail messages, and URLs in folders every day. We are often asked to choose keywords from an approved set of indexing terms for describing our technical publications. On a much larger scale, trained specialists assign new items to categories in large taxonomies such as the Dewey Decimal or Library of Congress subject headings, Medical Subject Headings (MeSH), or Yahoo!'s Internet directory. Between these two extremes, people organize objects into categories to support a wide variety of information-management tasks, including information routing/filtering/push, identification of objectionable materials or junk mail, structured search and browsing, and topic identification for topic-specific processing operations.

Human categorization is very time-consuming and costly, thus limiting its applicability—especially for large or rapidly changing collections. Consequently, interest is growing in developing technologies for (semi)automatic text categorization. Rule-based approaches similar to those employed in expert systems have been used, but they generally require manual construction of the rules, make rigid binary decisions about category membership, and are typically difficult to modify. Another strategy is to use *inductive-learning* techniques to automatically construct classifiers using labeled training data. Researchers have applied a growing number of learning techniques to text categorization, including multivariate regression, nearest-neighbor classifiers, probabilistic Bayesian models, decision trees, and neural networks.[1,2] Recently, my colleagues and I and others have used SVMs for text categorization with very promising results.[3,4] In this essay, I briefly describe the results of experiments in which we use SVMs to classify newswire stories from Reuters.[4]

Table 1. Break-even performance for five learning algorithms.

	FINDSIM (%)	NAIVE BAYES (%)	BAYESNETS (%)	TREES (%)	LINEARSVM (%)
earn	92.9	95.9	95.8	97.8	98.0
acq	64.7	87.8	88.3	89.7	93.6
money-fx	46.7	56.6	58.8	66.2	74.5
grain	67.5	78.8	81.4	85.0	94.6
crude	70.1	79.5	79.6	85.0	88.9
trade	65.1	63.9	69.0	72.5	75.9
interest	63.4	64.9	71.3	67.1	77.7
ship	49.2	85.4	84.4	74.2	85.6
wheat	68.9	69.7	82.7	92.5	91.8
corn	48.2	65.3	76.4	91.8	90.3
Avg. top 10	64.6	81.5	85.0	88.4	92.0
Avg. all	61.7	75.2	80.0	N/A	87.0

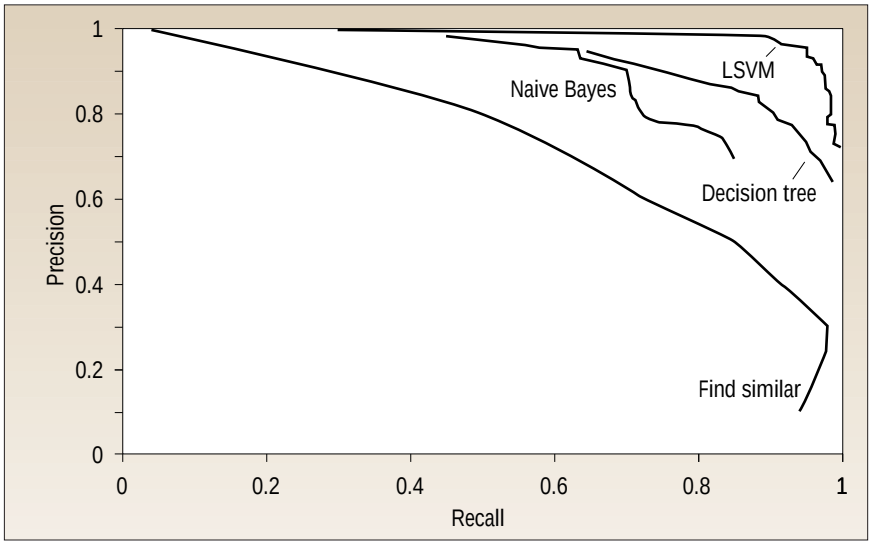


Figure 5. ROC curve.

Learning text categorizers

The goal of automatic text-categorization systems is to assign new items to one or more of a set of predefined categories on the basis of their textual content. Optimal categorization functions can be learned from labeled training examples.

Inductive learning of classifiers. A classifier is a function that maps an input attribute vector, $\vec{x} = (x_1, x_2, x_3, \ldots, x_n)$, to the confidence that the input belongs to a class—that is, $f(\vec{x}) = confidence(class)$. In the case of text classification, the attributes are words in the document and the classes are the categories of interest (for example, Reuters categories include "interest," "earnings," and "grain").

Example classifiers for the Reuters category interest are

- if (interest AND rate) OR (quarterly), then confidence ("interest" category) = 0.9
- confidence ("interest" category) = 0.3*interest + 0.4*rate + 0.7*quarterly

The key idea behind SVMs and other inductive-learning approaches is to use a training set of labeled instances to learn the classification function automatically. SVM classifiers resemble the second example above—a vector of learned feature weights. The resulting classifiers are easy to construct and update, depend only on information that is easy for people to provide (that is, examples of items that are in or out of categories), and allow users to smoothly trade off precision and recall depending on their task.

Text representation and feature selection. Each document is represented as a vector of words, as is typically done in information retrieval.[5] For most text-retrieval applications, the entries in the vector are weighted to reflect the frequency of terms in documents and the distribution of terms across the collection as a whole. For text classification, simpler binary feature values (a word either occurs or does not occur in a document) are often used instead.

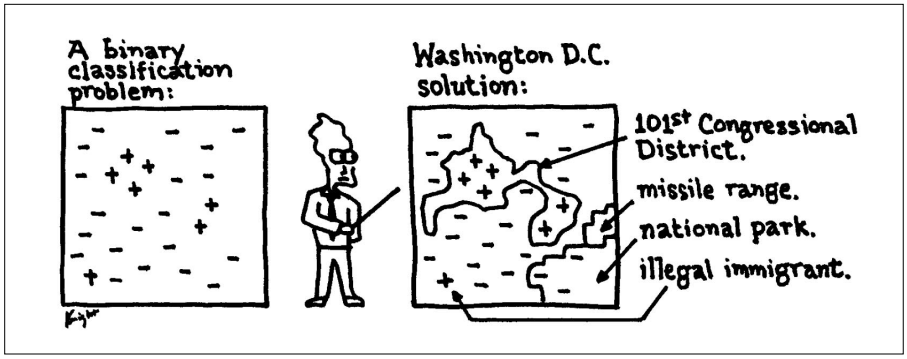
Text collections containing millions of unique terms are quite common. Thus, for both efficiency and efficacy, feature selection is widely used when applying machine-learning methods to text categorization. To reduce the number of features, we first remove features based on overall frequency counts, and then select a small number of features based on their fit to categories. We use the mutual information between each feature and a category to further reduce the feature space. These much smaller document descriptions then serve as input to the SVM.

Learning SVMs. We used simple linear SVMs because they provide good generalization accuracy and are fast to learn. Thorsten Joachims has explored two classes of nonlinear SVMs—polynomial classifiers and radial basis functions—and observed only small benefits compared to linear models.[3] We used John Platt's Sequential Minimal Optimization method[6] (described in a later essay) to learn the vector of feature weights, $\vec{w}$. Once the weights are learned, new items are classified by computing $\vec{x} \cdot \vec{w}$ where $\vec{w}$ is the vector of learned weights, and $\vec{x}$ is the binary vector representing a new document. We also learned two parameters of a sigmoid function to transform the output of the SVM to probabilities.

An example—Reuters

The Reuters collection is a popular one for text-categorization research and is publicly available at *http://www.research.att.com/~lewis/reuters21578.html*. We used the 12,902 Reuters stories that have been classified into 118 categories. Following the ModApte split, we used 75% of the stories (9,603 stories) to build classifiers and the remaining 25% (3,299 stories) to test the accuracy of the resulting models in reproducing the manual category assignments. Stories can be assigned to more than one category.

Text files are automatically processed to produce a vector of words for each document. Eliminating words that appear in only one document and then selecting the 300 words with highest mutual information with each category reduces the number of features. These 300-element binary feature vectors serve as input to the SVM. A separate



classifier ($\vec{w}$) is learned for each category. Using SMO to train the linear SVM takes an average of 0.26 CPU seconds per category (averaged over 118 categories) on a 266-MHz Pentium II running Windows NT. Other learning methods are 20 to 50 times slower. New instances are classified by computing a score for each document ($\vec{x} \cdot \vec{w}$) and comparing the score with a learned threshold. New documents exceeding the threshold are said to belong to the category.

The learned SVM classifiers are intuitively reasonable. The weight vector for the category "interest" includes the words prime (.70), rate (.67), interest (.63), rates (.60), and discount (.46), with large positive weights, and the words group (–.24), year (–.25), sees (–.33) world (–.35), and dlrs (–.71), with large negative weights.

As is typical in evaluating text categorization, we measure classification accuracy using the average of precision and recall (the so-called breakeven point). Precision is the proportion of items placed in the category that are really in the category, and recall is the proportion of items in the category that are actually placed in the category. Table 1 summarizes microaveraged breakeven performance for five learning algorithms explored by my colleagues and I explored for the 10 most frequent categories, as well as the overall score for all 118 categories.[4]

Linear SVMs were the most accurate method, averaging 91.3% for the 10 most frequent categories and 85.5% over all 118 categories. These results are consistent with Joachims' results in spite of substantial differences in text preprocessing, term weighting, and parameter selection, suggesting that the SVM approach is quite robust and generally applicable for text-categorization problems.[3]

Figure 5 shows a representative ROC curve for the category "grain." We generate this curve by varying the decision threshold to produce higher precision or higher recall, depending on the task. The advantages of the SVM can be seen over the entire recall-precision space.

Summary

In summary, inductive learning methods offer great potential to support flexible, dynamic, and personalized information access and management in a wide variety of tasks.

References

1. D.D. Lewis and P. Hayes, special issue of *ACM Trans. Information Systems*, Vol. 12, No. 1, July 1994.
2. Y. Yang, "An Evaluation of Statistical Approaches to Text Categorization," to be published in *J. Information Retrieval*, 1998.
3. T. Joachims, "Text Categorization with Support Vector Machines: Learning with Many Relevant Features," to be published in *Proc. 10th European Conf. Machine Learning (ECML)*, Springer-Verlag, 1998; http://www-ai.cs.uni-dortmund.de/PERSONAL/joachims.html/Joachims_97b.ps.gz.
4. S. Dumais et al., "Inductive Learning Algorithms and Representations for Text Categorization, to be published in *Proc. Conf. Information and Knowledge Management*, 1998; http://research.microsoft.com/~sdumais/cikm98.doc.
5. G. Salton and M. McGill, *Introduction to Modern Information Retrieval*, McGraw Hill, New York, 1983.
6. J. Platt, "Fast Training of SVMs Using Sequential Minimal Optimization," to be published in *Advances in Kernel Methods—Support Vector Machine Learning*, B. Schölkpf, C. Burges, and A. Smola, eds., MIT Press, Cambridge, Mass., 1998.

Applying SVMs to face detection

Edgar Osuna, MIT Center for Biological and Computational Learning and Operations Research Center

This essay introduces an SMV application for detecting vertically oriented and unoccluded frontal views of human faces in gray-level images. This application handles faces over a wide range of scales and works under different lighting conditions, even with moderately strong shadows.

We can define the face-detection problem as follows. Given as input an arbitrary image, which could be a digitized video signal or a scanned photograph, determine whether there are any human faces in the image, and if there are, return an encoding of their location. The encoding in this system is to fit each face in a bounding box defined by the image coordinates of the corners.

Face detection as a computer-vision task has many applications. It has direct relevance to the face-recognition problem, because the first important step of a fully automatic human face recognizer is usually identifying and locating faces in an unknown image. Face detection also has potential application in human-computer interfaces, surveillance systems, and census systems, for example.

For this discussion, face detection is also interesting as an example of a natural and challenging problem for demonstrating and testing the potentials of SVMs. Many other real-world object classes and phenomena share similar characteristics—for example, tumor anomalies in MRI scans and structural defects in manufactured parts. A successful and general methodology for finding faces using SVMs should generalize well for other spatially well-defined pattern- and feature-detection problems.

Face detection, like most object-detection problems, is a difficult task because of the significant pattern variations that are hard to parameterize analytically. Some common sources of pattern variations are facial appearance, expression, presence or absence of common structural features such as glasses or a moustache, and light-source distribution.

This system works by testing candidate image locations for local patterns that appear like faces, using a classification procedure that determines whether a given local image pattern is a face. Therefore, our approach comes at the face-detection problem as a classification problem given by examples of two classes: faces and nonfaces.

Previous systems

Researchers have approached the face-detection problem with different techniques in the last few years, including neural networks,[1,2] detection of face features and use of geometrical constraints,[3] density estimation of the training data,[4] labeled graphs,[5] and clustering and distribution-based modeling.[6,7] The results of Kah-Kay Sung and

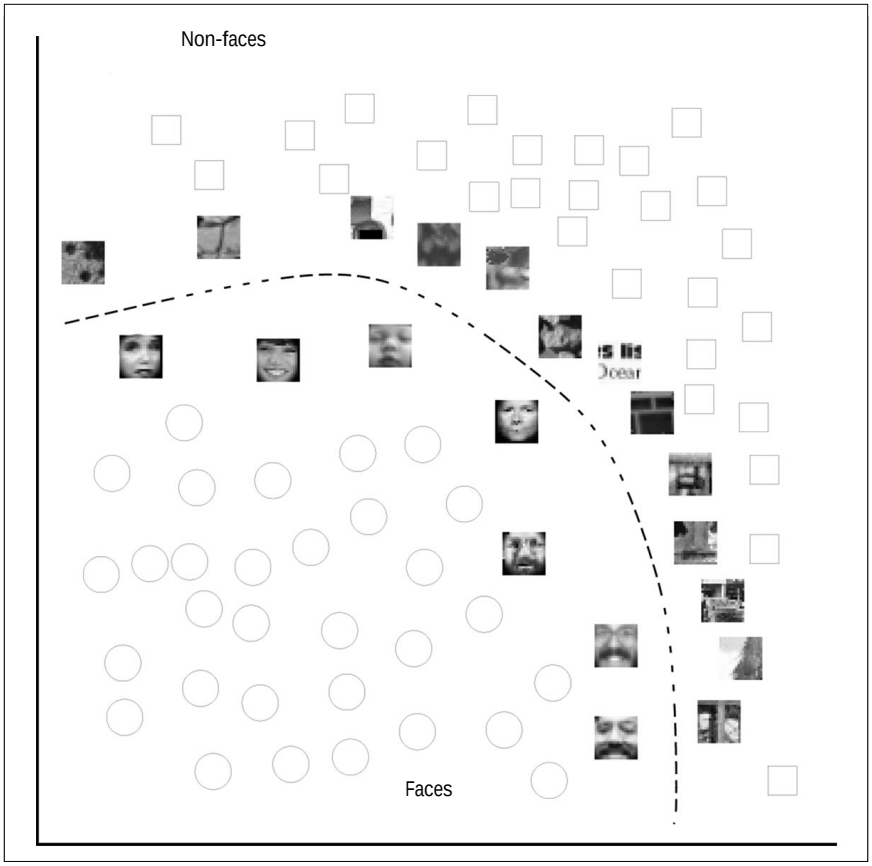


Figure 6. Geometrical interpretation of how the SVM separates the face and nonface classes. The patterns are real support vectors obtained after training the system. Notice the small number of total support vectors and the fact that a higher proportion of them correspond to nonfaces.

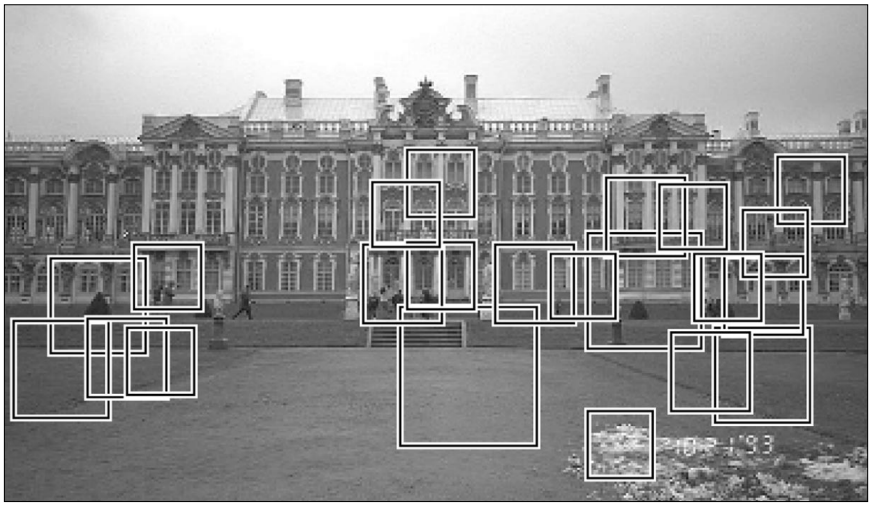


Figure 7. False detections obtained with the first version of the system. These false positives later served as nonface examples in the training process.

Tomaso Poggio[6,7] and Henry Rowley[2] reflect systems with very high detection rates and low false-positive detection rates.

Sung and Poggio use clustering and distance metrics to model the distribution of the face and nonface manifold and a neural network to classify a new pattern given the measurements. The key to the quality of their result is the clustering and use of combined Mahalanobis and Euclidean metrics to measure the distance from a new pattern and the clusters. Other important features of their approach are the use of nonface clusters and a bootstrapping technique to collect important nonface patterns. However, this approach does not provide a principled way to choose some important free parameters such as the number of clusters it uses.

Similarly, Rowley and his colleagues have used problem information in the design of a retinally connected neural network trained to classify face and nonface patterns. Their approach relies on training several neural networks emphasizing sets of the training data to obtain different sets of weights. Then, their approach uses different schemes of arbitration between them to reach a final answer.

Our SVM approach to the face-detection system uses no prior information to obtain the decision surface, this being an interesting property that can be exploited in using the same approach for detecting other objects in digital images.

The SVM face-detection system

This system detects faces by exhaustively scanning an image for face-like patterns at many possible scales, by dividing the original image into overlapping subimages and classifying them using an SVM to determine the appropriate class—face or nonface. The system handles multiple scales by examining windows taken from scaled versions of the original image.

Clearly, the major use of SVMs is in the classification step, which is the most critical part of this work. Figure 6 gives a geometrical interpretation of the way SVMs work in the context of face detection.

More specifically, this system works as follows. We train on a database of face and nonface 19×19 pixel patterns, assigned to classes +1 and –1, respectively, using the support vector algorithm. This process uses a second-degree homogeneous polynomial kernel function and an upper bound C = 200 to obtain a perfect training error.

To compensate for certain sources of image variation, we perform some preprocessing of the data:

- *Masking*: A binary pixel mask removes some pixels close to the window-pattern boundary, allowing a reduction in the dimensionality of the input space from $19 \times 19 = 361$ to 283. This step reduces background patterns that introduce unnecessary noise in the training process.
- *Illumination gradient correction*: The process subtracts a best-fit brightness plane from the unmasked window pixel values, allowing reduction of light and heavy shadows.

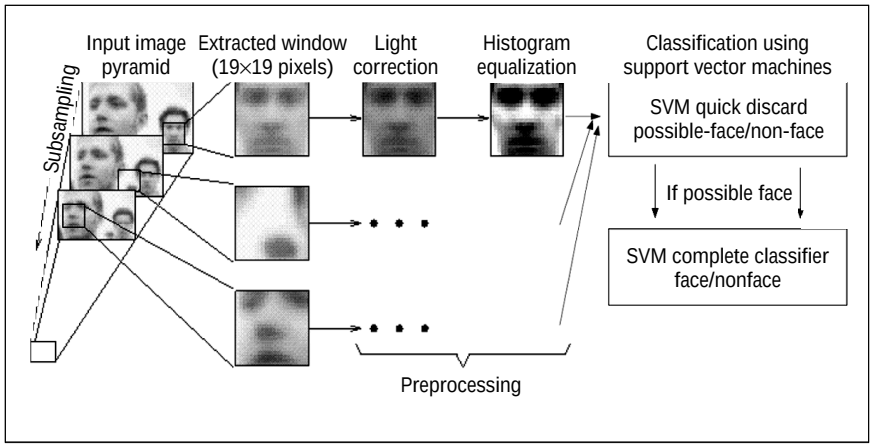


Figure 8. System architecture at runtime. (Used with permission.[2])

- *Histogram equalization*: Our process performs a histogram equalization over the patterns to compensate for differences in illumination brightness and different cameras' response curves, and so on.

Once the process obtains a decision surface through training, it uses the runtime system over images that do not contain faces, storing misclassifications so that they can be used as negative examples in subsequent training phases. Images of landscapes, trees, buildings, and rocks, for example, are good sources of false positives because of the many different *textured* patterns they contain. This bootstrapping step, which Sung and Poggio[6] successfully used, is very important in the context of a face detector that learns from examples:

- Although negative examples are abundant, negative examples that are useful from a learning standpoint are very difficult to characterize and define.
- By approaching the problem of object detection, and in this case of face detection, by using the paradigm of binary pattern classification, the two classes—object and nonobject—are not equally complex. The nonobject class is broader and richer, and therefore needs more examples to get an accurate definition that separates it from the object class. Figure 7 shows an image used for bootstrapping with some misclassifications that later served as negative examples.

After training the SVM, using an implementation of the algorithm my colleagues and I describe elsewhere,[8] we incorporate it as the classifier in a runtime system very similar to the one used by Sung and Poggio.[5,6] It performs the following operations:

- Rescale the input image several times;
- Cut 19×19 window patterns out of the scaled image;
- Preprocess the window using masking, light correction and histogran equalization;
- Classify the pattern using the SVM; and
- If the class corresponds to a face, draw a regtangle aroung the face in the output image.

Figure 8 reflects the system's architecture at runtime.

Experimental results on static images

To test the runtime system, we used two sets of images. Set A contained 313 high-quality images with the same number of faces. Set B contained 23 images of mixed quality, with a total of 155 faces. We tested both sets, first using our system and then the one by Sung and Poggio.[5,6] To give true meaning to the number of false positives obtained, note that set A involved 4,669,960 pattern windows, while set B involved 5,383,682. Table 2 compares the two systems.

Figure 9 presents some output images of our system, which were not used during the training phase of the system.

Extension to a real-time system

The system I've discussed so far spends approximately 6 seconds (SparcStation 20) on a 320×240 pixels gray-level image. Although this is faster than most previous systems, it is not fast enough for use as a runtime system. To build a runtime version of the system, we took the following steps:

- We ported the C code developed on the Sun environment to a Windows NT Pentium 200-MHz computer and added a Matrox RGB frame grabber and a Hitachi three-chip color camera. We used no special hardware to speed up the computational burden of the system.
- We collected several color images with faces, from which we extracted areas with skin and nonskin pixels. We collected a dataset of 6,000 examples.

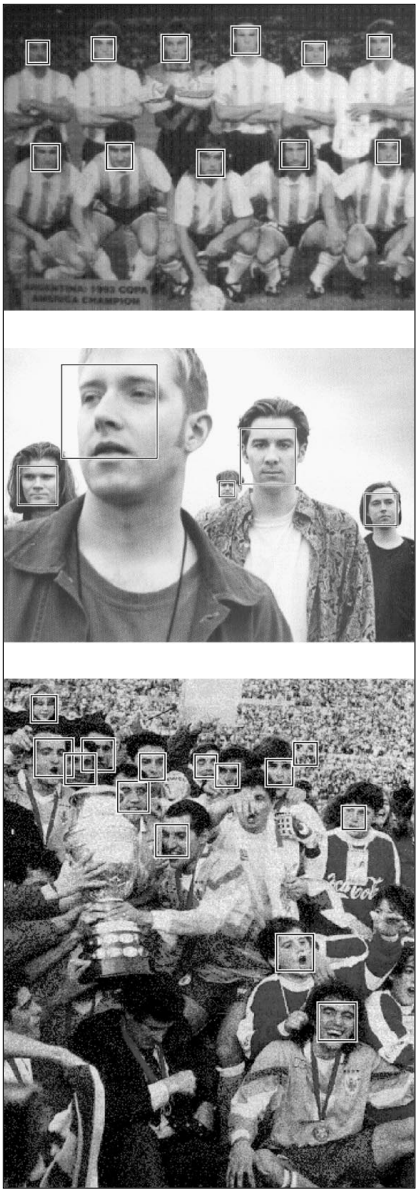


Figure 9. Results from our face-detection system.

Table 2. Performance of the SVM face-detection system.

	Test set A		Test set B	
	Detect rate (%)	False alarms	Detect rate (%)	False alarms
SVM	97.1	4	74.2	20
Sung	94.6	2	74.2	11

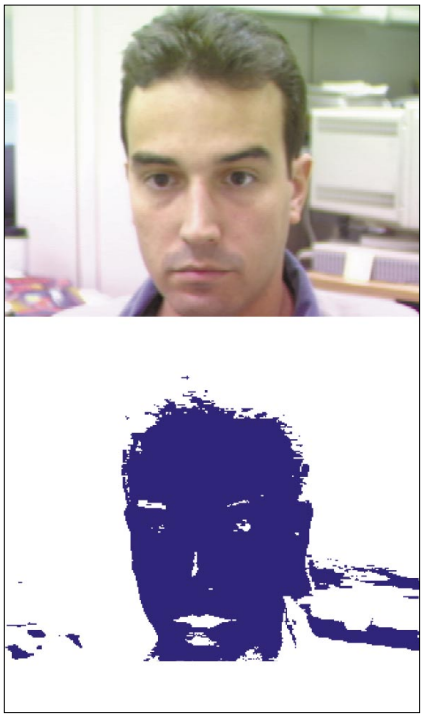


Figure 10. An example of the skin detection module implemented using SVMs.

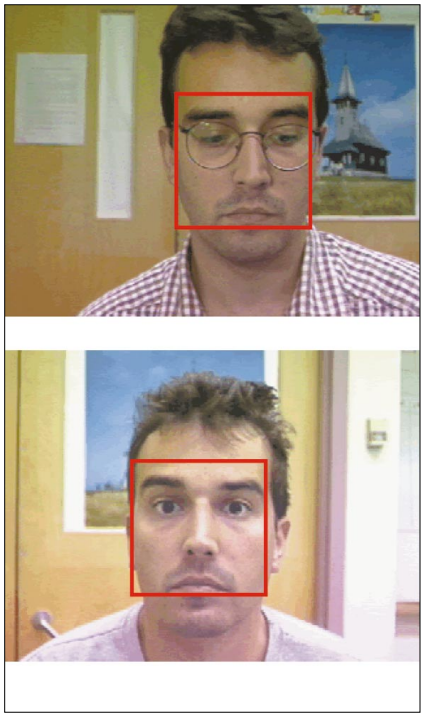


Figure 11. Face detection on the PC-based Color Real-Time system

- We trained a SVM classifier using the skin and nonskin data. The input variables were normalized green and red values—g/(r+g+b) and r/(r+g+b), respectively. Figure 10 presents an image captured by the system and its corresponding skin-detection output.
- We coded a very primitive motion detector based on thresholded frame differencing to identify areas of movement and use them as the focus of attention. Motion was not a requirement to be detected by the system because every so many frames (20 in the current implementation), we skipped this step and scanned the whole image.
- We put together a hierarchical system using as a first step the motion-detection module. We used the SVM skin-detection system as second layer to identify candidate locations of faces. We used the face/nonface SVM classifier described I described earlier over the gray-level version of the candidate locations.

The whole system achieves rates of 4 to 5 frames per second. Figure 11 presents a couple of images captured by our PC-based Color Real-Time face-detection system.

References

1. G. Burel and D. Carel, "Detection and Localization of Faces on Digital Images," *Pattern Recognition Letters*, Vol. 15, 1994, pp. 963–967.
2. H. Rowley, S. Baluja, and T. Kanade, *Human Face Detection in Visual Scenes*, Tech. Report 95–158, Computer Science Dept., Carnegie Mellon Univ., Pittsburgh, 1995.
3. G. Yang and T. Huang, "Human Face Detection in a Complex Background," *Pattern Recognition*, Vol. 27, 1994, pp. 53–63.
4. B. Moghaddam and A. Pentland, *Probabilistic Visual Learning for Object Detection*, Tech. Report 326, MIT Media Laboratory, Cambridge, Mass., 1995.
5. N. Krüger, M. Pötzsch, and C. v.d. Malsburg, *Determination of Face Position and Pose with Learned Representation Based on Labeled Graphs*, Tech. Report 96-03, Ruhr-Universität, 1996.
6. K. Sung, *Learning and Example Selection for Object and Pattern Detection*, PhD thesis, MIT AI Lab and Center for Biological and Computational Learning, 1995.
7. K. Sung and T. Poggio, *Example-Based Learning for View-Based Human Face Detection*, A.I. Memo 1521, C.B.C.L Paper 112, Dec. 1994.
8. E. Osuna, R. Freund, and F. Girosi, "An Improved Training Algorithm for Support Vector Machines," *Proc. IEEE Workshop on Neural Networks and Signal Processing*, IEEE Press, Piscataway, N.J., 1997.

How to implement SVMs

John Platt, Microsoft Research

In the past few years, SVMs have proven to be very effective in real-world classification tasks. [1] This installment of Trends & Controversies describes two of these tasks: face recognition and text categorization. However, many people have found the numerical implementation of SVMs to be intimidating. In this essay, I will attempt to demystify the implementation of SVMs. As a first step, if you are interested in implementing an SVM, I recommend reading Chris Burges' tutorial on SVMs,[2] available at *http://svm.research.bell-labs.com/SVMdoc.html*.

An SVM is a parameterized function whose functional form is defined before training. Training an SVM requires a labeled training set, because the SVM will fit the function from a set of examples. The training set consists of a set of N examples. Each example consists of an input vector, $\mathbf{x}_i$, and a label, y_i, which describes whether the input vector is in a predefined category. There are N free parameters in an SVM trained with N examples. These parameters are called α_i. To find these parameters, you must solve a quadratic programming (QP) problem:

$$\text{minimize } \frac{1}{2}\sum_{i,j=1}^{N}\alpha_i Q_{ij}\alpha_j - \sum_{i=1}^{N}\alpha_i;$$

$$\text{subject to } 0 \le \alpha_i \le C \text{ and } \sum_{i=1}^{N} y_i\alpha_i = 0$$

where Q is an $N \times N$ matrix that depends on the training inputs $\mathbf{x}_i$, the labels y_i, and the functional form of the SVM. We call this problem *quadratic programming* because the function to be minimized (called the *objective function*) depends on the α_i quadratically, while α_i only appears linearly in the constraints (see *http://www-c.mcs.anl.gov/home/otc/Guide/OptWeb/continuous/constrained/qprog*). Definitions and applications of $\mathbf{x}_i$, y_i, α_i, and Q appear in the tutorial by Burges.[2]

Conceptually, the SVM QP problem is to find a minimum of a bowl-shaped objective function. The search for the minimum is constrained to lie within a cube and on a plane. The search occurs in a high-dimensional space, so that the bowl is high dimensional, the cube is a hypercube, and the

plane is a hyperplane. For most typical SVM functional forms, the matrix **Q** has special properties, so that the objective function is either bowl-shaped (positive definite) or has flat-bottomed troughs (positive semidefinite), but is never saddle-shaped (indefinite). Thus, there is either a unique minimum or a connected set of equivalent minima. An SVM QP has a definite termination (or optimality) condition that describes these minima. We call these optimality conditions the Karush-Kuhn-Tucker (KKT) conditions, and they simply describe the set of α_i that are constrained minima.[3]

The values of α_i also have an intuitive explanation. There is one α_i for each training example. Each α_i determines how much each training example influences the SVM function. Most of the training examples do not affect the SVM function, so most of the α_i are 0.

Because of its simple form, you might expect the solution to the SVM QP problem to be quite simple. Unfortunately, for real-world problems, the matrix **Q** can be enormous: it has a dimension equal to the number of training examples. A training set of 60,000 examples will yield a **Q** matrix with 3.6 billion elements, which cannot easily fit into the memory of a standard computer.

We have at least two different ways of solving such gigantic QP problems. First, there are QP methods that use sophisticated data structures.[4] These QP methods do not require the storage of the entire **Q** matrix, because they do not need to access the rows or columns of **Q** that correspond to those α_i that are at 0 or at *C*. Deep in the inner loop, these methods only perform dot products between rows or columns of **Q** and a vector, rather than performing an entire matrix-vector multiplication.

Decomposing the QP problem

The other method for attacking the large-scale SVM QP problem is to decompose the large QP problem into a series of smaller QP problems. Thus, the selection of submatrices of **Q** happens outside of the QP package, rather than inside. Consequently, the decomposition method is compatible with standard QP packages.

Vapnik first suggested the decomposition approach in a method that has since been known as *chunking*.[1] The chunking algorithm exploits the fact that the value of the objective function is the same if you remove the rows and columns of the matrix **Q** that correspond to zero α_i. Therefore, the large QP problem can break down into a series of smaller QP problems, whose ultimate goal is to identify all of the nonzero α_i and discard all of the zero α_i. At every step, chunking solves a QP problem that consists of the following α_i: every nonzero α_i from the last step, and the α_i that correspond to the *M* worst violations of the KKT conditions, for some value of *M* (see Figure 12a). The size of the QP subproblem tends to grow with time. At the last step, the chunking approach has identified the entire set of nonzero α_i; hence, the last step solves the overall QP problem.

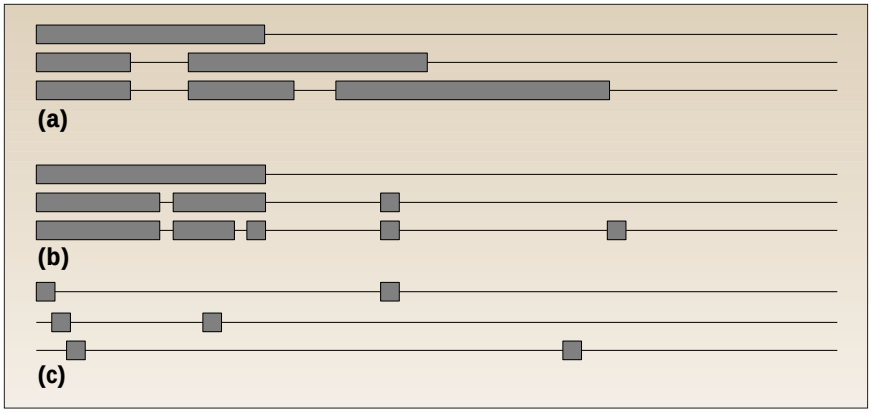


Figure 12. Three alternative methods for training SVMs: (a) Chunking, (b) Osuna's algorithm, and (c) SMO. There are three steps for each method. The horizontal thin line at every step represents the training set, while the thick boxes represent the α_i being optimized at that step. A given group of three lines corresponds to three training iterations, with the first iteration at the top.

Chunking reduces the **Q** matrix's dimension from the number of training examples to approximately the number of nonzero α_i. However, chunking still might not handle large-scale training problems, because even this reduced matrix might not fit into memory. Of course, we can combine chunking with the sophisticated QP methods described above, which do not require full storage of a matrix.

In 1997, Edgar Osuna and his colleagues suggested a new strategy for solving the SVM QP problem.[5] Osuna showed that the large QP problem can be broken down into a series of smaller QP subproblems. As long as at least one α_i that violates the KKT conditions is added to the previous subproblem, each step reduces the objective function and maintains all of the constraints. Therefore, a sequence of QP subproblems that always add at least one KKT violator will asymptotically converge.

Osuna suggests keeping a constant size matrix for every QP subproblem, which implies adding and deleting the same number of examples at every step[5] (see Figure 12b). Using a constant-size matrix allows the training of arbitrarily sized datasets. The algorithm in Osuna's paper suggests adding one example and deleting one example at every step. Such an algorithm converges, although it might not be the fastest possible algorithm. In practice, researchers add and delete multiple examples according to various unpublished heuristics. Typically, these heuristics add KKT violators at each step and delete those α_i that are either 0 or *C*. Joachims has published an algorithm for adding and deleting examples from the QP steps, which rapidly decreases the objective function.[6]

All of these decomposition methods require a numerical QP package. Such packages might be expensive for commercial users (see the "Where to get the programs" section). Writing your own efficient QP package is difficult without a numerical-analysis background.

Sequential minimal optimization

Sequential minimal optimization is an alternative method that can decompose the SVM QP problem without any extra matrix storage and without using numerical QP optimization steps.[3,7] SMO decomposes the overall QP problem into QP subproblems, identically to Osuna's method. Unlike the previous decomposition heuristics, SMO chooses to solve the smallest possible optimization problem at every step. For the standard SVM QP problem, the smallest possible optimization problem involves two elements of α_i, because the α_i must obey one linear equality constraint. At every step, SMO chooses two α_i to jointly optimize, finds the optimal values for these α_i, and updates the SVM to reflect the new optimal values (see Figure 12c).

Table 3. Five experiments comparing SMO to PCG chunking. The functional form of the SVM, training set size, CPU times, and scaling exponents are shown.

Experiment	Kernel used	Training set size	SMO training CPU time (sec.)	PCG chunking training CPU time (sec.)	SMO scaling exponent	PCG chunking scaling exponent
Adult linear	Linear	11,221	17.0	20,711.3	1.9	3.1
Web linear	Linear	49,749	268.3	17,164.7	1.6	2.5
Adult Gaussian	Gaussian	11,221	781.4	11,910.6	2.1	2.9
Web Gaussian	Gaussian	49,749	3,863.5	23,877.6	1.7	2.0
MNIST	Polynomial	60,000	29,471.0	33,109.0	N/A	N/A

SMO can solve for two α_i analytically, thus avoiding numerical QP optimization entirely. The inner loop can be expressed in a short amount of C code, rather than by invoking an entire QP library routine. Even though more optimization subproblems are solved in the course of the algorithm, each subproblem is so fast that the overall QP problem can be solved quickly.

Because there are so many possible combinations of QP packages, decomposition heuristics, code optimizations, data structures, and benchmark problems, it is very difficult to determine which SVM algorithm (if any) is the most efficient. SMO has been compared to the standard chunking algorithm suggested by Burges in his tutorial.[2,3] The QP algorithm used by this version of chunking is *projected conjugate gradient* (PCG). Table 3 compares the results for SMO versus PCG chunking. Both algorithms are coded in C++, share SVM evaluation code, are compiled with Microsoft Visual C++ version 5.0, and are run on a 266-MHz Pentium II with Windows NT and 128 Mbytes of memory. Both algorithms have inner loops that take advantage of input vectors that contain mostly zero entries (that is, sparse vectors).

For more details on this comparison, and for more experiments on synthetic datasets, please consult my upcoming publication.[3] The Adult experiment is an income-prediction task and is derived from the UCI machine-learning benchmark.[8] The Web experiment is a text-categorization task. The Adult and Web datasets are available at *http://www.research.microsoft.com/~jplatt/smo.html*. The MNIST experiment is an OCR benchmark available at *http://www.research.att.com/~yann/ocr/mnist*. The training CPU time is listed for both SMO and PCG chunking for the training set size shown in the table. The scaling exponent is the slope of a linear fit to a log-log plot of the training time versus the training set size. This scaling exponent varies with the dataset used. The empirical worst-case scaling for SMO is quadratic, while the empirical worst-case scaling for PCG chunking is cubic.

For a linear problem with sparse inputs, SMO can be more than 1,000 times faster than PCG chunking.

Joachims has compared his algorithm (SVM^{light} version 2) and SMO on the same datasets.[6] His algorithm and SMO have comparable scaling with training set size. The CPU time of Joachims' algorithm seems roughly comparable to SMO; different code optimizations make exact comparison between the two algorithms difficult.

Where to get the programs

The pseudocode for SMO is currently in a technical report available at *http://www.research.microsoft.com/~jplatt/smo.html*.[7] SMO can be quickly implemented in the programming language of your choice using this pseudocode. I would recommend SMO if you are planning on using linear SVMs, if your data is sparse, or if you want to write your own end-to-end code.

If you decide to use a QP-based system, be careful about writing QP code yourself—there are many subtle numerical precision issues involved, and you can find yourself in a quagmire quite rapidly. Also, be wary of freeware QP packages available on the Web: in my experience, such packages tend to run slowly and might not work well for ill-conditioned or very large problems. Purchasing a QP package from a well-known numerical analysis source is the best bet, unless you have an extensive numerical analysis background, in which case you can create your own QP package. Osuna and his colleagues use MINOS for their QP package, which has licensing information at *http://www-leland.stanford.edu/~saunders/brochure/brochure.html*.[5] LOQO is another robust, large-scale interior-point package suitable for QP and available for a fee at *http://www.princeton.edu/~rvdb*.

Finally, a program that implements Joachims' version of Osuna's algorithm,[6] called SVM^{light}, is available free, for scientific purposes only, at *http://www-ai.informatik.uni-dortmund.de/FORSCHUNG/VERFAHREN/SVM_LIGHT/svm_light.eng.html*. ■

References

1. V. Vapnik, *Estimation of Dependencies Based on Empirical Data*, Springer-Verlag, New York, 1982.
2. C.J.C. Burges, "A Tutorial on Support Vector Machines for Pattern Recognition," submitted to *Data Mining and Knowledge Discovery*, 1998.
3. J.C. Platt, "Fast Training of SVMs Using Sequential Minimal Optimization," to be published in *Advances in Kernel Methods—Support Vector Learning*, B. Schölkopf, C. Burges, and A. Smola, eds., MIT Press, Cambridge, Mass., 1998.
4. L. Kaufman, "Solving the Quadratic Programming Problem Arising in Support Vector Classification," to be published in *Advances in Kernel Methods—Support Vector Learning*, MIT Press, 1998.
5. E. Osuna, R. Freund, and F. Girosi, "An Improved Training Algorithm for Support Vector Machines," *Proc. IEEE Neural Networks for Signal Processing VII Workshop*, IEEE Press, Piscataway, N.J., 1997, pp. 276–285.
6. T. Joachims, "Making Large-Scale SVM Learning Practical," to be published in *Advances in Kernel Methods—Support Vector Learning*, MIT Press, 1998.
7. J.C. Platt, *Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines*, Microsoft Research Tech. Report MSR-TR-98-14, Microsoft, Redmond, Wash., 1998.
8. C.J. Merz and P.M. Murphy, *UCI Repository of Machine Learning Databases*, Univ. of California, Irvine, Dept. Information and Computer Science, Irvine, Calif.; http://www.ics.uci.edu/~mlearn/MLRepository.html.