

# Support Vector Machines for Natural Language Processing

CSCI 544 – Notes to Accompany the Lecture

Hal Daumé III

9 March 2004

## 1 Introduction

What have we seen so far?

Model	Strengths	Weaknesses
naïve bayes	easy to model, train	independence assumptions are often violated
decision trees	highly non-linear decision boundaries	segments data, tends to overfit
neural networks	linear or non-linear, intuitive	lots of parameters (number of layers, units, etc.)

In general, we would like something which has all of these strengths, but none of the weaknesses, i.e., is easy to model and train, has linear or non-linear decision boundaries, is intuitive, does not make unreasonable assumptions about the data, does not overfit, and does not have lots of (hyper) parameters. SVMs are supposed to have all these properties.

What is an SVM?

$$\text{SVM} = \underbrace{\text{linear NN}}_{\substack{\text{well-studied,} \\ \text{(simple decision rules)}}} + \underbrace{\text{regularization}}_{\text{(avoids overfitting)}} + \underbrace{\text{kernel trick}}_{\substack{\text{(allows non-linear} \\ \text{decision rules)}}$$

We will go through each of these topics (linear networks, regularization and the kernel trick) in order. For the most part, we will discuss the task of binary classification (distinguishing data points between two classes). We will call our classes +1 and -1 and will denote the class of the  $i$ th data point by  $y^{(i)}$ . Typically, our data will be drawn from  $\mathbb{R}^d$  where  $d$  is the “dimensionality” of the “input space” and will be called  $x^{(i)}$ . We will write  $x_j$  to refer to the  $j$ th component of a multi-dimensional vector. We will assume we have  $N$ -many training points.

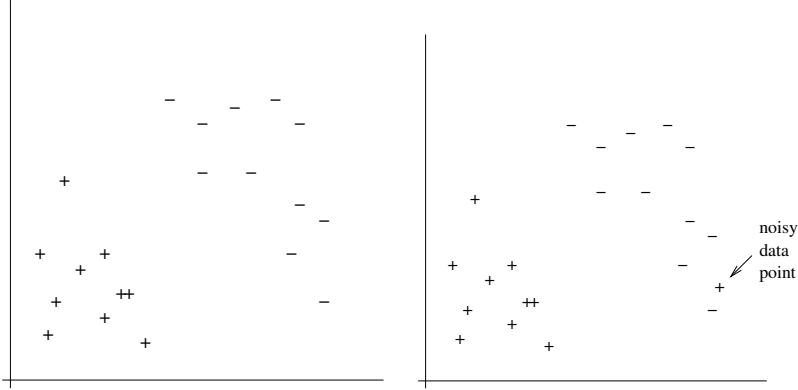


Figure 1: Left: linearly separable data; Right: linearly inseparable data

## 2 Linear Networks

A **linear function** of  $x$  is of the form:

$$f(x; w, b) = \sum_{i=1}^d w_i x_i + b = w \cdot x + b \quad (1)$$

where  $w$  and  $b$  are the open parameters of the model (i.e., the function).

We will say that the training data  $\{(x^{(i)}, dy_i) : 1 \leq i \leq N$  is **linearly separable** if we can find a  $w$  and  $b$  such that:

$$\text{sgn}(f(x^{(i)}; w, b)) = y^{(i)} \quad \forall 1 \leq i \leq N \quad (2)$$

Consider the two training sets shown in Figure 1. Of these, the left data set is linearly separable, while the right data set is not.

The goal of learning is then to find a *good* value for  $w$  and  $b$ . We can define good in (at least) two ways:

1. good = performs well on the *training data*
2. good = performs well on *all data*

Consider again the linearly separable data set depicted in Figure 1. Choosing  $w$  and  $b$  amounts to drawing a line somewhere on this graph to separate the +s from the -s. There are many (infinitely many) such lines one can draw, ranging from a vertical line just squeezing through to an angled line drawn evenly in between the data, to an angled line touching one of the classes. Intuitively, we would prefer the line that runs evenly between the data. This intuition is known as having the **maximal margin** (i.e., the margin space between the decision boundary and the nearest points in either class is maximized). This is the intuitive concept behind the regularization technique used for SVMs and will be made more formal shortly.

Before we officially move on the regularization, we will note that in the above definitions of “good”, the first criteria is known as “empirical error minimization” and the second is known as “true error minimization” or “generalization error minimization.” In general, it is the generalization error that we wish to minimize. In order to minimize the generalization error, though, we would need to know about all possible data sets. However, we will see that it is possible to put a bound on the generalization error using a notion of model complexity.

### 3 Regularization

When one performs non-regularized learning, one is almost always attempting to find a value for  $w^{opt}$  (and  $b$ ) that minimizes the empirical error:

$$w^{opt} = \arg \min_{w^*} \sum_{i=1}^N L(f(x^{(i)}; w), y^{(i)}) \quad (3)$$

where  $L$  is a “loss function” that penalizes  $f$  when it makes a mistake. Typically (and for the rest of these notes),  $L$  will be the “hinge-loss” (or “zero-one loss”) function, defined by:

$$L(\hat{y}, y) = \frac{1}{2}(y - \text{sgn}\hat{y}) \quad (4)$$

It is easy to see in this definition that if the sign of the predicted value  $\hat{y}$  is the same as  $y$ , the loss will be zero; however, if the sign is incorrect, the loss will be 1.

Solving such problems is very common; unfortunately, it can lead to poor results due to overfitting, since it is minimizing the empirical error, not the true error.

#### 3.1 Bounds on True Error

As mentioned before, we cannot calculate the true error, but we can *bound* it from above by:

$$\text{true error} \leq \text{emp. error} + \sqrt{\frac{1}{N} \left[ d \left( \log \frac{2N}{d} + 1 \right) - \log \eta 4 \right]} \quad (5)$$

with probability  $1 - \eta$ . This expression depends on a crucial value  $d$ , which is called the **VC dimension** of the model used to solve the problem (not to be confused with the dimensionality of the input space). For a bit of intuition, linearly separable models have VC dimension 1, quadratically separable models have a VC dimension of 2, and so on.

The statement Eq (5) makes is that we can say with, for example, 90% certainty ( $\eta = 0.1$ ) and 100 data points, that the true error is never more than  $\frac{1}{10} \sqrt{d(\log(200/d) + 1) + 1.6}$  more than the empirical error. When  $d = 1$ , this value is 0.281; when  $d = 2$ , this value is 0.357; when  $d = 10$ , this value is 0.645.

As we can see from this example, as well as inspecting Eq (5) more closely, in order to minimize bound on the true error, we should simultaneously minimize the empirical error as well as the VC dimension of the model we are using.

### 3.2 Regularized Learners

Minimization of VC dimension is one of many possible regularization methods one can use. For instance, statisticians typically use Bayesian reasoning with appropriate prior distributions as a form of regularization. In general, regularized learning replaces Eq (3) with the following desiderata for our optimal parameters  $w$ :

$$w^{opt} = \arg \min_{w^*} \sum_{i=1}^N L(f(x^{(i)}; w), y^{(i)}) + \lambda R(w^*) \quad (6)$$

Again,  $L$  is a loss function.  $R$  is now a **regularization function**, which penalizes  $w$  that (in our case) result in models with too high of a VC dimension.  $\lambda$  is a scalar parameter that determines the trade-off between empirical error and expected generalization error. This learning framework is often called **structural risk minimization**, or SRM.

### 3.3 Large Margins

It is a fundamental result of SVM literature due to Vapnik that:

$$\text{Large Margins} \implies \text{small VC dimension}$$

the proof is relatively complicated and will be omitted; however, the basic idea is exactly what our intuition tells us: if we have two hypothesis models to choose from, and one is very fragile to small changes in the input data while the other is not, we should prefer the second one. Alternatively, the large margin principle can be motivated by leave-one-out generalization, in which we expect the solution to be relatively consistent when any one data point is removed.

This first result provides half of the necessary ingredients for motivating the support vector framework; the other necessary half is:

$$\text{Small } \|w\| \implies \text{Large Margins}$$

This can be seen by the following argument depicted in Figure 2.

In this figure, the size of the margin is the length of the vector  $m$ . We can compute this by noticing that the length of  $m$  is exactly the length of  $d_2$  minus the length of  $d_1$ .

Since  $w \cdot x^{(i)} + b \geq 1$  for all  $i$  such that  $y^{(i)} = 1$ , we can compute:

$$d_1 = |w \cdot 0 + b - 1| / \|w\| \quad (7)$$

Similarly, since  $w \cdot x^{(i)} + b \leq -1$  for all  $i$  such that  $y^{(i)} = -1$ , we can compute:

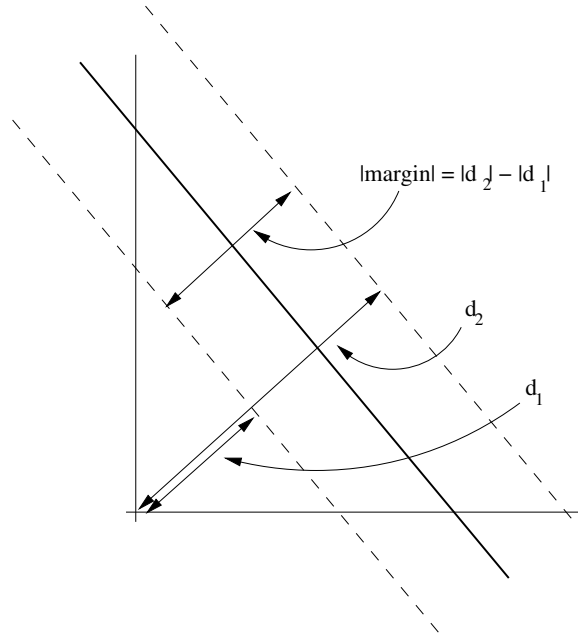


Figure 2: Pictorial description of how we can calculate the size of the margin.

$$d_2 = |w \cdot 0 + b + 1| / \|w\| \quad (8)$$

Combining Eq (7) and Eq (8), we obtain:

$$|m| = |d_2 - d_1| = \frac{2}{\|w\|} \quad (9)$$

From this, we can easily see that a small value for  $\|w\|$  will result in a large margin and, by the first result, therefore reduces the VC dimension and thus lowers the upper bound on expected generalization error.

## 4 Linear SVMs

In this section, we will discuss the support vector framework for finding linear separators.

### 4.1 Linearly Separable Data

Assuming for now that our training data is linearly separable, we know that we will be able to find  $w$  and  $b$  that separate the data. Among all such functions, we wish to find the one with minimum norm on  $w$ . We formulate this as:

$$\begin{aligned}
&\text{minimize:} && \frac{1}{2} \|w\|^2 \\
\text{subject to:} && w \cdot x^{(i)} + b \geq 1 && \forall i, y^{(i)} = 1 && (10) \\
&& w \cdot x^{(i)} + b \leq -1 && \forall i, y^{(i)} = -1 && (11)
\end{aligned}$$

Note that Eq (11) and Eq (10) can be condensed, leaving us with the following task:

$$\begin{aligned}
&\text{minimize:} && \frac{1}{2} \|w\|^2 && (12) \\
\text{subject to:} && y^{(i)}(w \cdot x^{(i)} + b) - 1 \geq 0 && \forall i
\end{aligned}$$

This problem can be solved using techniques from optimization theory. By introducing so-called Lagrange multipliers  $\alpha$ , we can rewrite problem (12) in its “dual formulation” as:

$$\text{maximize: } L(\alpha) = -\frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y^{(i)} y^{(j)} x^{(i)} \cdot x^{(j)} + \sum_{i=1}^N \alpha_i \quad (13)$$

The Karush-Kuhn-Tucker conditions tell us that at the optimum setting of the  $\alpha$ s, we get:

$$\alpha_i (y^{(i)}(w \cdot x^{(i)} + b) - 1) = 0 \quad \forall i \quad (14)$$

This means that  $\alpha_i \neq 0$  only if  $y^{(i)}(w \cdot x^{(i)} + b) - 1 = 0$ . The points for which  $\alpha_i \neq 0$  are rare and are those points sitting on the margin. They are called the **support vectors**. Given the  $\alpha$ s, we can write  $w$  as:

$$w = \sum_{i=1}^N \alpha_i y^{(i)} x^{(i)} \quad (15)$$

Given this, we can write our learned function as:

$$f(x; w, b) = w \cdot x + b = \sum_{i=1}^N \alpha_i y^{(i)} x^{(i)} \cdot x + b \quad (16)$$

As we can see from Eq (16), this expression depends only on the support vectors and we can thus forget all the other data points. The value of  $b$  can be calculated by computing  $f$  at any two support vectors.

## 4.2 Non-Linearly Separable Data

When the training data is not linearly separable, we introduce **slack parameters** that measure the misclassification rate of the noisy input vectors. A single slack parameter for a noisy input set is shown in Figure 3

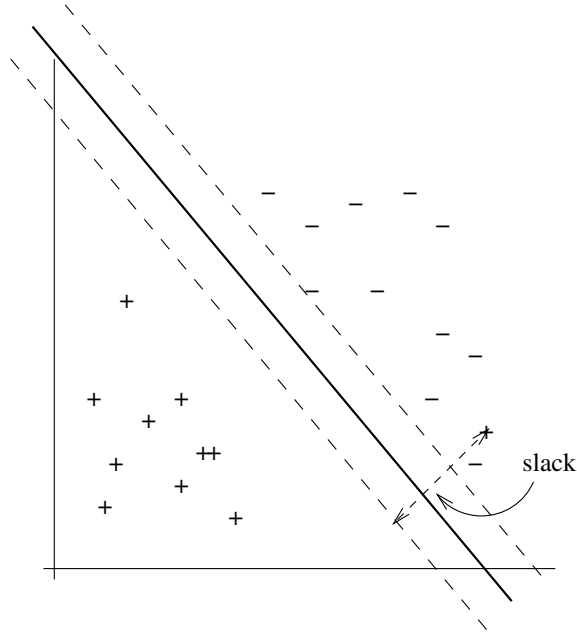


Figure 3: Depiction of slack variables for noisy data.

The variables  $\xi_i$  are introduced to account for the amount of misclassification that is allowed for the data point  $x^{(i)}$ . This changes our training criteria from problem (12) to:

$$\text{minimize: } \frac{1}{2} \|w\|^2 + C \sum_{i=1}^N \xi_i \quad (17)$$

$$\text{subject to: } y^{(i)}(w \cdot x^{(i)} + b) - 1 \geq 0 \quad \forall i$$

$$\xi_i \geq 0 \quad \forall i \quad (18)$$

Here,  $C$  is a hyperparameter that controls the amount of training error allowed. This formulation leads to a similar dual optimization problem, with the same decision function  $f$ . However, there will typically be more support vectors, since, in addition to the normal support vectors, any data point with nonzero  $\xi$  will also be a support vector.

A large value of  $C$  means that misclassifications are considered bad, which will result in smaller margins and thus less training error (but more expected generalization error). A small value of  $C$  results in more training error, but better expected generalization. We will discuss the setting of  $C$  later.

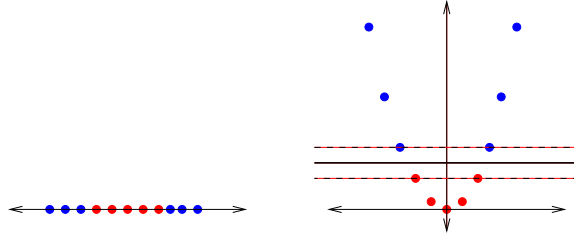


Figure 4: Linearly inseparable data in one dimension (left) that is separable in two dimensions (right).

## 5 The Kernel Trick

So far we have considered the task of finding separating hyperplanes. That is, when our input data is drawn from  $\mathbb{R}^d$ , we know how to find a “good” function  $f(x; w, b)$  that separates the two classes. However, in NLP, we are often faced with data that is not linearly separable, or that doesn’t even come from  $\mathbb{R}^d$ , but rather from some **discrete space** (such as the space of all words, or all strings, or trees or graphs, etc.). While it is often possible to embed such discrete spaces in  $\mathbb{R}^d$  for some  $d$ , doing so often results in loss of information for explosion of computational complexity (due to large  $d$ ). The kernel trick allows us to get around these problems.

### 5.1 A One-Dimensional Example

Consider the one-dimensional dataset depicted in the left side of Figure 4. This data is not linearly separable (in the one dimensional case, a linear separator is simply the choice of a point to the left of which one class is chosen and to the right of which the other class is chosen). However, if we map the input space from  $\mathbb{R}$  to  $\mathbb{R}^2$  by the feature function  $\Phi(x) = (x, x^2)$ , we obtain the data shown in the right side of Figure 4. Here, we can see that the data has immediately become linearly separable.

We can just as easily map the input from  $\mathbb{R}$  to  $\mathbb{R}^{100}$  by  $\Phi(x) = (x, x^2, \dots, x^{100})$ . The data will also be linearly separable in this case, but (a) the computational complexity will blow up exponentially and (b) overfitting becomes very easy. We will hope that regularization will solve (b). Issue (a) will be the subject of the remainder of this section.

### 5.2 Kernel Spaces

I have repeated here the dual formulation of the support vector learning problem as well as the learned function based on that problem – Eq (13) and Eq (16), respectively – for clarity.

$$\text{maximize: } L(\alpha) = -\frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y^{(i)} y^{(j)} x^{(i)} \cdot x^{(j)} + \sum_{i=1}^N \alpha_i$$

$$f(x; w, b) = w \cdot x + b = \sum_{i=1}^N \alpha_i y^{(i)} x^{(i)} \cdot x + b$$

Notice that in both of these equations, we never actually use the value of  $x^{(i)}$  by itself: it is always used in the context of a dot product with another training (or test) example,  $x^{(j)}$ . We can ask ourselves what will happen if we replace all occurrences of  $x$  with the image of  $x$  under some (perhaps non-linear) transformation  $\Phi$ . In this case, our two equations become:

$$\text{maximize: } L(\alpha) = -\frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y^{(i)} y^{(j)} \Phi(x^{(i)}) \cdot \Phi(x^{(j)}) + \sum_{i=1}^N \alpha_i \quad (19)$$

$$f(x; w, b) = w \cdot x + b = \sum_{i=1}^N \alpha_i y^{(i)} \Phi(x^{(i)}) \cdot \Phi(x) + b \quad (20)$$

For any two inputs  $x$  and  $x'$ , we will commonly denote the value  $\Phi(x) \cdot \Phi(x')$  by  $K_{\Phi}(x, x')$  or simply by  $X(x, x')$  when the feature mapping is clear from context. The value of  $K(x, x')$  can be thought of the value of the dot product between  $x$  and  $x'$  *after* they have been transformed into some other feature space, as we did early to get linear separation for Figure 4. It turns out that there are lots of functions  $K$  that correspond to taking dot products in larger (or even infinite) feature spaces.

A brief word on terminology is in order. We refer to the space from which the  $x$ s are originally drawn as the **input space** and the space in which they are classified after the  $\Phi$  mapping as the **feature space**.

A necessary condition for a function  $K$  to be equivalent to a mapping into feature is that  $K$  needs to be a symmetric (i.e.,  $K(u, v) = K(v, u)$ ) positive semi-definite function. The latter condition means that for any choice of a function  $f$  (which is reasonably well behaved), it must hold that:

$$\int f(x) K(x, x') f(x') dx \geq 0 \quad (21)$$

Such functions are more formally known as **Mercer kernels**, to disambiguate them from other uses of the term kernel, which tends to be quite prevalent in different branches of mathematics.

It is known that Eq (21) is satisfied exactly when the following holds: given any set of training data  $x^{(i)}$ , it must be the case that the **kernel matrix** ( $K_{i,j} = K(x^{(i)}, x^{(j)})$ ) must be positive semi-definite. This can be a relatively difficult thing to check, but we will mostly concern ourselves with kernels that *other people* have found, rather than developing new ones ourselves.

## 6 Well (and not-so-well) Known Kernels

We will discuss several kernels, both for data falling into  $\mathbb{R}^d$  as well as discrete data.

### 6.1 Kernels over $\mathbb{R}^d$

The simplest kernel over  $\mathbb{R}^d$  is the **linear kernel**:

$$K(u, v) = u \cdot v \quad (22)$$

For the linear kernel, the feature space is exactly the same as the input space. A slight extension to the linear kernel gives us the **polynomial kernel**:

$$K(u, v) = (\alpha + \beta u \cdot v)^d \quad (23)$$

When  $d = \beta = 1$  and  $\alpha = 0$ , this reduces to the linear kernel. Setting  $d = 2$  results (nearly) in the mapping  $x \mapsto (x, x^2)$ , a similarly for  $d = 3$ , etc. This kernel has three parameters which must be set ahead of time,  $\alpha, \beta$  and  $d$ . We will discuss the setting of hyperparameters later.

A kernel derived from the work in the neural networks community on radial basis functions is the **RBF or Gaussian kernel**:

$$K(u, v) = \exp[-\gamma \|u - v\|^2] \quad (24)$$

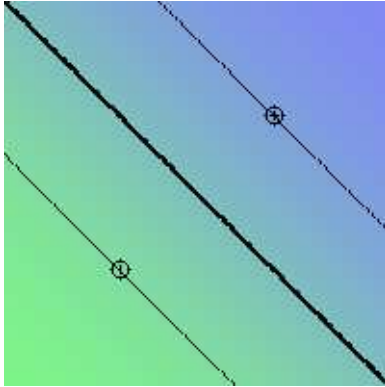
This kernel actually corresponds to an infinite feature space, though it can be thought of as drawing balls around the training vectors. The RBF kernel has one hyperparameter,  $\gamma$ , which corresponds to the size of these balls.

The final standard kernel over  $\mathbb{R}^d$  is the **tansig kernel**, which has the form:

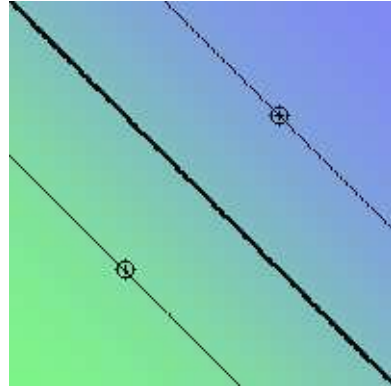
$$K(u, v) = \tanh(\alpha + \beta u \cdot v) \quad (25)$$

The tansig kernel should be used with caution: it is not a Mercer kernel for all values of  $\alpha$  and  $\beta$ , but is commonly provided for historical reasons.

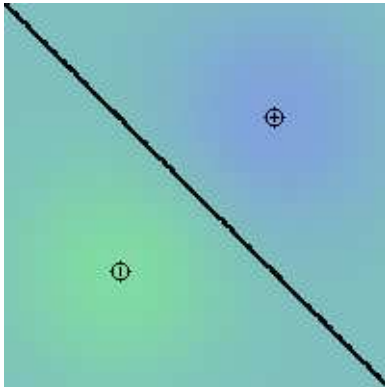
On the following pages we display the results of using various kernels with various parameters on various (small) datasets.



Linear



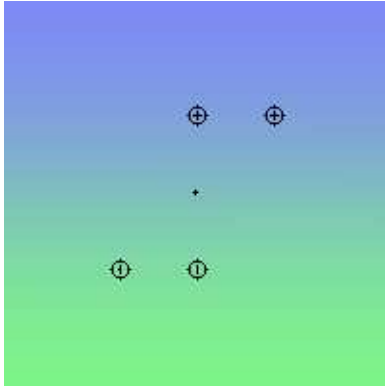
Poly 2



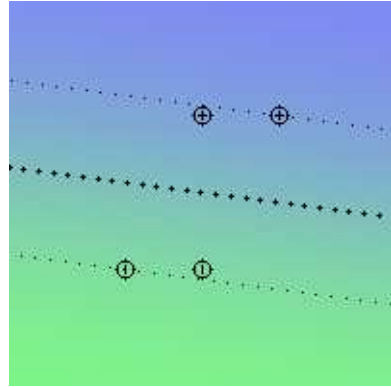
RBF

Separating two points, one positive and one negative. The decision boundary is shown darkened, the positive point with a plus and the negative point with a minus. Support vectors (both) are circled. The blue region specifies classifications of positive and the green region specifies negative classification. The thinner lines are the margin lines (note that the support vectors fall on them).

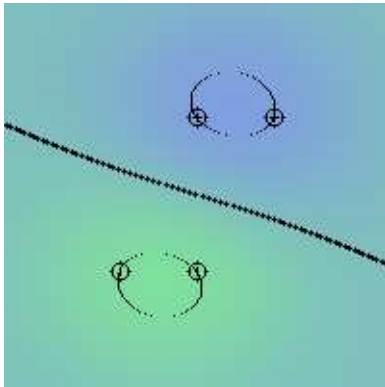
The upper left figure is using a linear classifier; the upper right is using a polynomial degree 2 classifier (note that it actually finds a linear decision boundary – this is because of the magic of regularization). The lower-left figure is using an RBF kernel; note the difference in the shape of the gradient.



Linear

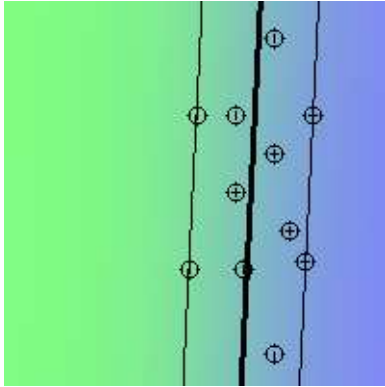


Poly 2

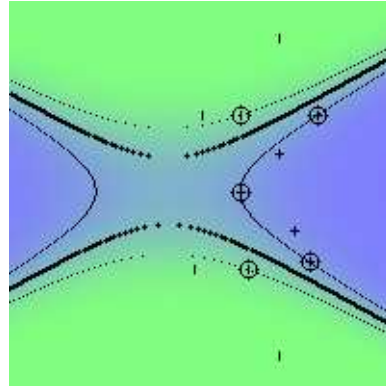


RBF

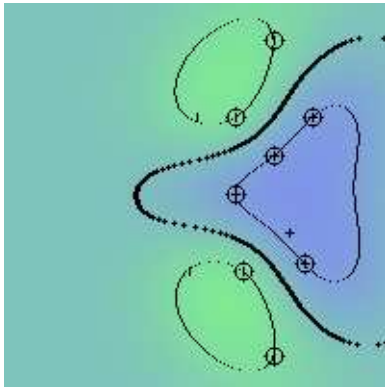
This is the same as the previous page, but with four training points instead of two. Note that the margin lines for the RBF kernel are ovular, though each machine finds approximately the same decision boundary.



Linear

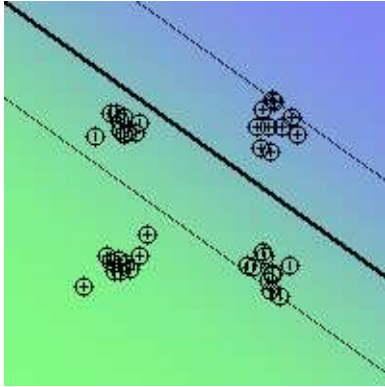


Poly 2

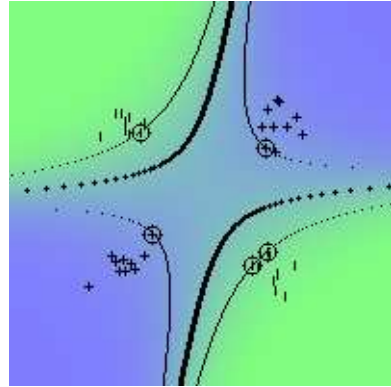


RBF

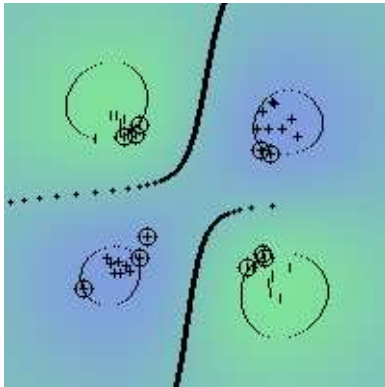
The figures are in the same order as before, but the data is now no longer linearly separable (it is however separable by a polynomial two kernel). We can see that the Poly 2 and RBF kernels found very different decision rules, both of which are non-linear and both of which correctly classify the data.



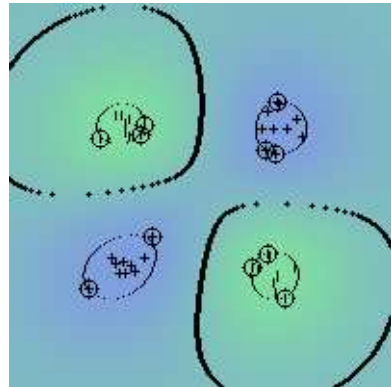
Linear



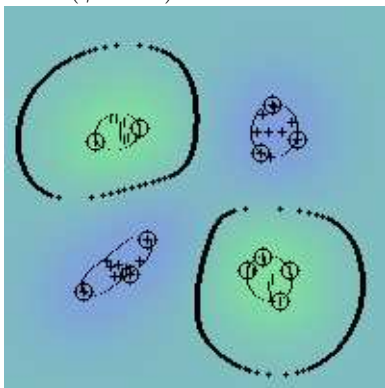
Poly 2



RBF ( $\gamma = 0.5$ )

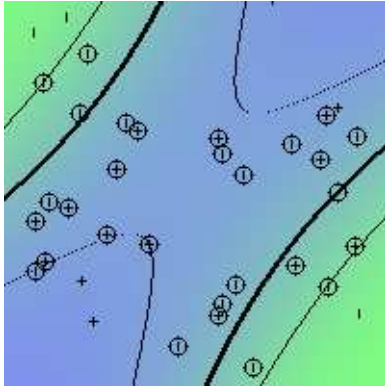


RBF ( $\gamma = 1$ )

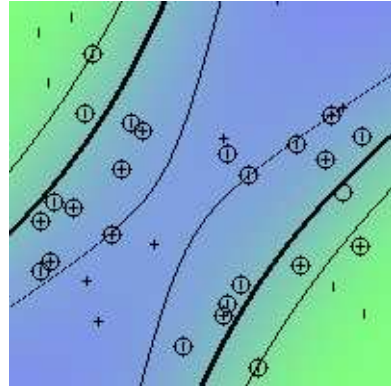


RBF ( $\gamma = 2$ )

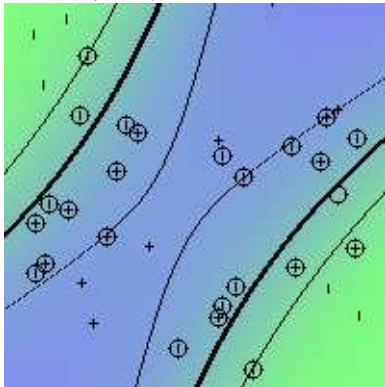
Data has been generated at  $(\pm 1, \pm 1)$  by gaussians with variance 0.2. The use of RBF kernels with different widths (value of  $\gamma$ ) are shown. Note that the larger  $\gamma$ , the tighter the circles. We can see that the linear SVM is unable to separate this data, while all the rest are.



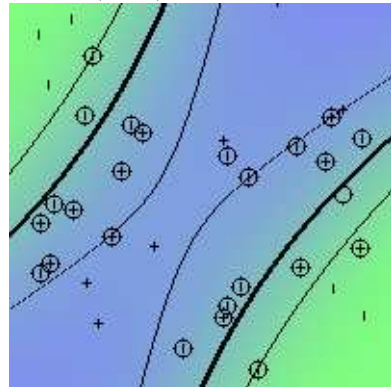
Poly 2 (C=0)



Poly 2 (C=1)

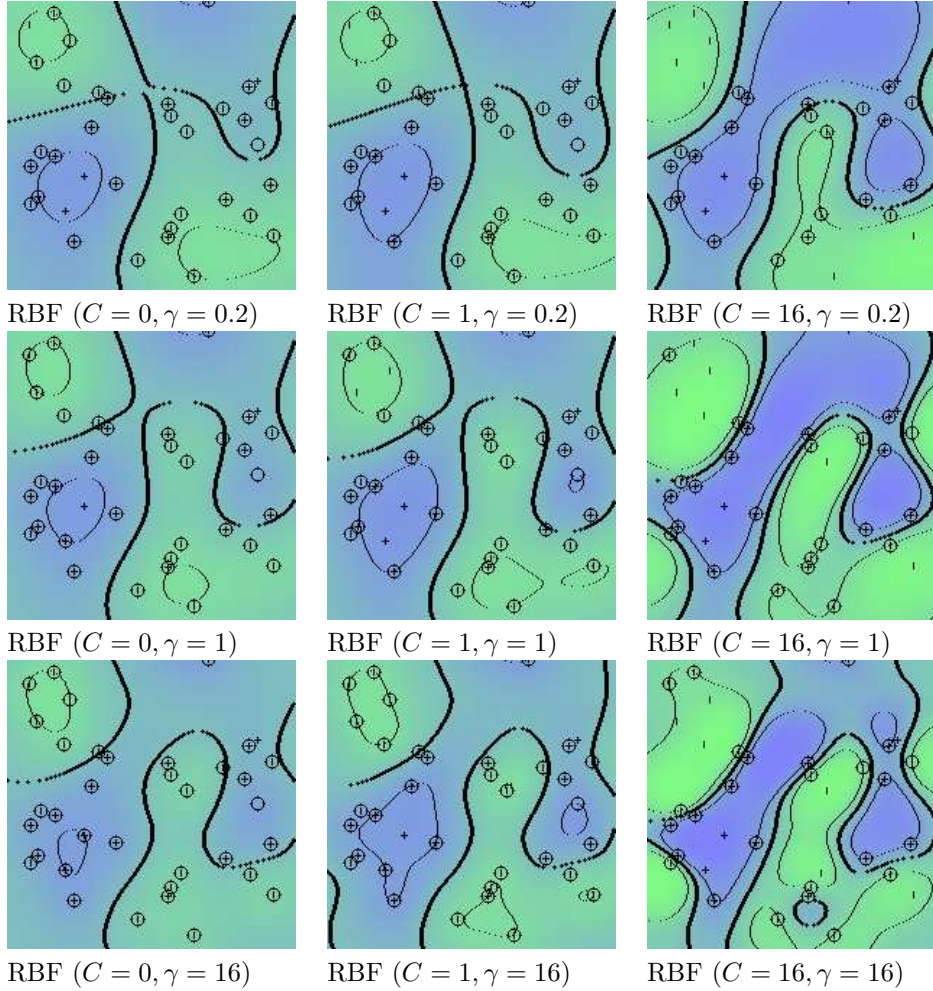


Poly 2 (C=4)



Poly 2 (C=16)

Data has been generated at  $(\pm 1, \pm 1)$  by gaussians with variance 1. Different values for  $C$  with a polynomial degree 2 kernel. Notice that the overall shape stays the same; it's just the intensity (the sharpness of the boundaries) that change.



Same data as before. Different values for  $C$  with an RBF kernel with different values of  $\gamma$ .

## 6.2 Kernels for Discrete Spaces

### 6.2.1 String Kernel

The most common discrete kernel is the **string kernel**, which is used to compute similarity between different strings. For the string kernel, we assume we have an alphabet  $\Sigma$  and write  $\Sigma^n$  for all sequences of length at most  $n$  from this alphabet. The feature space for a string kernel is  $\mathbb{R}^{\Sigma^n}$ . Each dimension in this space is indexed by a string  $u$  of length at most  $n$ , and has the form:

$$\Phi_u(s) = \sum_{i \subset \langle 1, 2, \dots, |s| \rangle, s[i]=u} \lambda^{|i|} \quad (26)$$

That is,  $\Phi_u$  is a feature function characterized by the string  $u$ . Its value at  $s$  is computed by identifying *all* substrings (indexed by  $i$ ) of  $s$  that are equal to  $u$ , and raising  $\lambda$  to the length of  $i$ .

For example, consider when  $u$  is **ab** and  $s$  is **abcacb**. There are three values of  $i$  for which  $s[i] = u$ , namely  $i = (1, 2)$ ,  $i = (1, 6)$  and  $i = (4, 6)$ . Typically  $\lambda < 1$ , so the string kernel function will react less to the far-separated index  $(1, 6)$  than to the nearby  $(1, 2)$ .

Without the kernel trick, doing learning in this space would be complete infeasible for all but tiny toy problems over small alphabets. However, given the derivation of the string kernel below:

$$\begin{aligned} K(s, s') &= \sum_{u \in \Sigma^n} \Phi(s) \cdot \Phi(s') \\ &= \sum_{u \in \Sigma^n} \sum_{i \subset \langle 1, 2, \dots, |s| \rangle, s[i]=u} \lambda^{|i|} \sum_{j \subset \langle 1, 2, \dots, |s'| \rangle, s'[j]=u} \lambda^{|j|} \\ &= \sum_{u \in \Sigma^n} \sum_{i \subset \langle 1, 2, \dots, |s| \rangle, s[i]=u} \sum_{j \subset \langle 1, 2, \dots, |s'| \rangle, s'[j]=u} \lambda^{|i|+|j|} \end{aligned} \quad (27)$$

This kernel value can be computed recursively using dynamic programming in time  $\mathcal{O}(d|s||s'|)$  or by using suffix trees in time  $\mathcal{O}(|s| + |s'|)$ .

The parameters for the string kernel are the maximum length  $n$  and the drop off rate,  $\lambda$ .

### 6.2.2 Tree Kernel

The tree kernel can be seen as a direct extension to the string kernel, but which works on trees (for instance, parse trees). Instead of defining a feature function  $\Phi_u(s)$  for a substring  $u$ , we define feature functions  $\Phi_u(t)$  over subtrees.

That is, for each possible tree  $u$  over a set of terminals and non-terminals, we define a feature function  $\Phi_u(t)$  by:

$$\Phi_u(t) = \sum_{s \in t, u \subseteq s} \lambda^{|s|} \quad (28)$$

That is, the sum is over all subtrees  $s$  of  $t$  such that  $u$  is a subtree of  $s$ . The value  $\lambda$  is raised to the size of the tree  $s$ .

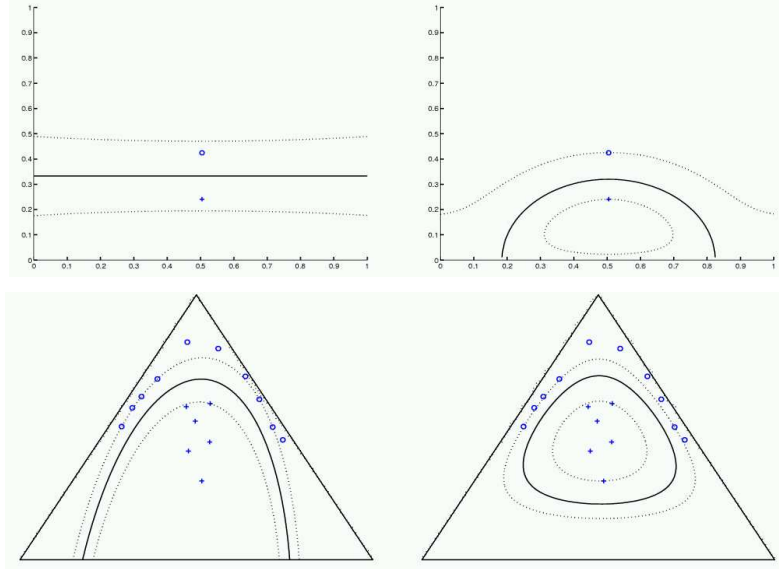


Figure 5: Top: The ID kernel (left) and RBF kernel (right) applied to training two points; Bottom: The ID kernel (left) and RBF kernel (right) on the 3-simplex.

### 6.2.3 Information Diffusion Kernel

The ID kernel has been used primarily for text categorization. In this context, we have a document over some vocabulary  $V$  and will typically represent this by vectors in  $\mathbb{R}^{|V|}$  where the feature functions are either term frequency (tf) values, tf-idf values, or normalized tf values (normalized so that the vectors have length 1 either in absolute value or squared value). After doing this, it is straightforward to use either a linear or RBF kernel to do text categorization.

However, if we normalize the data to fall into the  $|V| - 1$  simplex (vectors of length  $|V|$ , each component of which is positive and which sums to 1), then the following kernel, called the **information diffusion kernel**, tends to work quite well:

$$K(u, v) = (4\pi t)^{-(|V|-1)/2} \exp \left[ -\frac{1}{t} \arccos^2 \left( \sum_{i=1}^{|V|} \sqrt{u_i v_i} \right) \right] \quad (29)$$

The information diffusion kernel is based on an analysis of how the Fischer information metric behaves on statistical manifolds such as  $d$ -simplexes.

The ID kernel tends to produce similarly shaped decision boundaries as the RBF kernel, but not exactly the same. Some examples are shown in Figure 5.

### 6.2.4 Diffusion Kernels on Graphs

Graph diffusion kernels are models of how heat would flow around a graph. I'll discuss these only briefly since no publically available implementation exists, but these are very useful when the data come from categorical sources, rather than continuous sources. If each input vector is over  $n$ -many discrete features, the  $i$ th of which has  $A_i$  possible values, then the diffusion kernel has value:

$$K(u, v) = \prod_{i=1}^n \left( \frac{1 - \exp[-\beta A_i]}{1 + (A_i - 1) \exp[-\beta A_i]} \right)^{\delta_{u_i, v_i}} \quad (30)$$

where  $\delta$  is the Kronecker delta function.

## 7 Remaining Technical Issues

We have deferred many questions about how to use SVMs in practice. We'll discuss many of those issues here.

### 7.1 Choosing a Kernel

Your best bet in choosing which kernel to use is to use either your own prior knowledge or other people's results. If you have no reason to prefer one kernel over another a priori, a reasonable choice would be to first try a linear kernel and then also try an RBF kernel, since the RBF kernel only has one hyperparameter that needs to be set, unlike the polynomial kernel, which has 3.

### 7.2 Scaling of Data

It is typically advantageous to ensure that your training data (don't cheat!) lies either in  $[-1, 1]$  or  $[0, 1]$ . Simple scaling like this helps a lot, both in terms of generalization and efficiency of training.

### 7.3 Discrete Input Features

Arguably the best way to handle discrete input features is to use a discrete kernel, like the diffusion kernel on graphs. However, this is often not possible (due to the lack of implementations); or, perhaps, data contains both discrete and continuous variables, in which case diffusion kernels on graphs do not make sense.

Suppose there is an input feature which can take on  $k$  possible values. The typical way to encode this feature is to add  $k$ -many dimensions, each taking a value of either 0 or 1 exactly when that feature is active.

For example, suppose one of our features is hair color, which can take values BROWN, BLOND, RED or BLACK. In this case,  $k = 4$ , so we create four features, call them 1, 2, 3 and 4. If a person's hair is brown, then feature 1 has value one and the others have value 0; if a person's hair is blond, then feature 2 has value one and the others have value 0; etc.

## 7.4 Choosing Hyperparameters

This can be difficult and can have a large impact on your results. For instance, for the RBF kernel, you will need to set a value for  $C$  and  $\gamma$ .

If your data has been pre-scaled as suggested above, choosing  $C = \gamma = 1$  is a reasonable guess. Otherwise, cross-validation can be used to optimize the values. You can either perform a grid search with  $C = 2^n$  and  $\gamma = 2^m$  for  $n, m \in \{-4, \dots, 0, \dots, 4\}$ , or use a more intelligent search:

Start  $C = \gamma = 1$  and hold  $\gamma$  fixed while  $C$  is multiplied or divided by 2 until a maxima is reached. Then, hold  $C$  fixed there and perform the same optimization on  $\gamma$ . Finally, if so desired, hold  $\gamma$  fixed there and re-optimize  $C$ .

## 7.5 Probabilistic Classification

SVMs can be put into a probabilistic framework if so desired by assuming a probability of the form:

$$p(c = 1|x) = \frac{1}{1 + \exp(Af(x) + B)} \quad (31)$$

And learning the values for  $A$  and  $B$  through any unconstrained optimization algorithm.

## 7.6 Multiclass Classification

There are two standard multiclass classification techniques (over  $k$  classes) based on building several binary classifiers. It is also possible to attempt to do the full separation at once, but this is computationally expensive and does not perform any better in practice, provided the binary classifiers are well-tuned.

- **One-Versus-All:** In the OVA framework, we build  $k$  classifiers  $f_i$  for  $1 \leq i \leq k$ , where class  $i$  is positive and all other classes are negative for  $f_i$ . We then choose the class as the value  $i$  which has maximal value of  $f_i(x)$  for a datapoint  $x$ .
- **All-Versus-All:** In the AVA framework, we build  $\binom{k}{2}$  classifiers  $f_{ij}$  for  $1 \leq i < j \leq k$ , where class  $i$  is positive and  $j$  is negative. We then choose the class by:

$$\text{class} = \arg \max_i \sum_{j \neq i} \text{sgn}(f_{ij}(x)) \quad (32)$$

Both techniques work well in practice.

## 8 Existing Implementations

	linear, polynomial, rbf, tanhig kernels	ID kernel	string kernels	tree kernels	regression	ranking	cross-validation for hyperparameters	very fast	probabilistic classification	AVA support (built in)	most popular
SVM-light <sup>1</sup>	✓	?			✓	✓		✓			✓
LIBSVM <sup>2</sup>	✓				✓		✓	✓	✓	✓	
SVMseq <sup>3</sup>	✓	✓	✓	✓		✓			✓		
GINsvm <sup>4</sup>	✓		?		✓			✓	?		

1. <http://svmlight.joachims.org/>

2. <http://csie.ntu.edu.tw/~cjlin/libsvm/>

3. <http://www.isi.edu/~hdaume/SVMseq/>

4. <http://bach.ece.jhu.edu/svm/ginismv/>

## 9 Input Format

All of SVM-light, libsvm and SVMseq use the same input format. Each line in the input file consists of a class and then feature/value pairs, as in:

```
+1 1:2.0 2:9 5:-1.2
+1 4:1 3:3 5:0.112
-1 2:2 6:1
```

The first column is the class, one of  $\{+1, -1\}$  and then all feature/value pairs are space separated with a colon between the feature number and its value. Features with value 0 do not need to be included and typically feature numbers should be monotonically increasing.

Test data goes in the same format.

## 10 Incantations

Here are common incantations for the three learning systems:

### 10.1 SVM-light

The command line for training is:

```
svm_learn -c <C> -t <kernel> [params] <train> <model>
```

In this,  $\langle C \rangle$  is the  $C$  value used to control the cost of misclassifications for the slack parameters.  $\langle \text{train} \rangle$  is the name of the file containing the training

data and `<model>` is the name of the file to which the trained model should be written. The value of `<kernel>` should be 0 for a linear kernel, 1 for a polynomial kernel, 2 for an RBF kernel or 3 for a tansig kernel.

The `[params]` allows kernel parameters to be specified. There are none for the linear kernel; for the polynomial kernel, `-d d` specifies the degree, `-s beta` specifies the  $\beta$  value and `-r alpha` specifies the  $\alpha$  value; for RBF kernels, `-g gamma` specifies the  $\gamma$  value; the arguments for the tansig kernel are the same as for the polynomial kernel (without  $d$ ).

For performing prediction, you run:

```
svm_classify <test> <model> <pred>
```

Where `<test>` is the filename containing the test data, `<model>` is a trained model file and the resulting predictions are written to the file `<pred>`.

## 10.2 libsvm

libsvm is trained by running:

```
svm-train -c <C> -t <kernel> [params] <train> <model>
```

The kernel parameter is the same as SVM-light. The parameters for the RBF kernel are the same; for polynomial and tansig kernels, `-g` sets the  $\beta$  value and `-r` sets  $\alpha$ .

Prediction is run by:

```
svm-predict <test> <model> <pred>
```

## 10.3 SVMseq

The incantation of `SVMseqLearn` is identical to that of SVM-light, but also accepts options `--sigmoid` to train a probabilistic classifier; `--string-lambda` and `--string-maxn` to specify the  $\lambda$  and  $n$  values for the string kernel; and similar options for tree kernels. Additionally, `-t 4` uses the information diffusion kernel, where the  $t$  parameter is set with `-g` and the  $|V|$  parameter is set with `-d`.

Running `SVMseqClassify` is identical to all the other prediction programs.

## 11 Example Input File

In the lecture last Tuesday, we saw a decision tree algorithm applied to a binary classification problem of differentiating `medications` and `illegal_substances`. We will call the former class `+1` and the latter class `-1`. Recall that there were seven inputs: `NO_PRICES`  $\in \{0, 1, 2\}$ ; `NO_PRESCRIPTION`  $\in \{0, 1, 2\}$ ; `NO_ABUSE`  $\in \{0, 1, 2\}$ ; `NO_COCAINE`  $\in \{0, 1, 2\}$ ; `NO_BOOK`  $\in \{0, 1, 2\}$ ; `OCCURS_AT_END_SENTENCE`  $\in \{yes, no\}$ ; `OCCURS_AT_BEGINNING_SENTENCE`  $\in \{yes, no\}$ .

All of these features are discrete, so for each of the first 5 we introduce 3 binary features; for each of the last two, we introduce 2 binary features. The input file corresponding to the `WSD.data` file seen last time is:

```
+1 3:1 4:1 7:1 10:1 13:1 17:1 19:1
-1 2:1 4:1 8:1 10:1 13:1 17:1 19:1
+1 2:1 5:1 7:1 10:1 13:1 17:1 18:1
+1 1:1 5:1 7:1 10:1 14:1 17:1 19:1
+1 2:1 4:1 8:1 10:1 14:1 17:1 19:1
-1 1:1 4:1 8:1 10:1 13:1 17:1 19:1
+1 1:1 6:1 7:1 11:1 13:1 17:1 19:1
+1 1:1 5:1 7:1 10:1 14:1 17:1 19:1
-1 1:1 4:1 8:1 11:1 13:1 17:1 19:1
-1 1:1 4:1 7:1 12:1 13:1 17:1 19:1
-1 1:1 4:1 7:1 12:1 14:1 17:1 19:1
+1 1:1 4:1 7:1 10:1 13:1 17:1 19:1
```

## A References

C.J.C. Burges, *A Tutorial on Support Vector Machines for Pattern Recognition*, KDD'98, pp. 121–167.

M. Collins and N. Diffy, *Convolution Kernels for Natural Language*, NIPS14.

J.C. Platt, *Probabilities for SV Machines*, Advances in Large Margin Classifiers, 1999.

G. Lebanon and J. Lafferty, *Information Diffusion Kernels*, NIPS15.

R.I. Kensor and J. Lafferty, *Diffusion Kernels on Graphs and other Discrete Input Spaces*, ICML 2002.

S.V.N. Vishwanathan and A.J. Smola, *Fast Kernels for String and Tree Matching*, in *Kernels and Bioinformatics*, 2003.