Support Vector Machines

Northwestern CS 352 Spring 2019

Bryan Pardo

Support Vector Machines: High Level

- Classifiers that are good for linear and nonlinear classification
- Effective in high dimensional spaces (even infinite dimensional)
- Versatile: If you specify a Kernel (more on that later) you can apply them to lots of different kinds of data.
- The decision function uses a subset of the training data (called support vectors) to classify, so they are more memory efficient than K-nearest neighbor classifiers.

Some notes

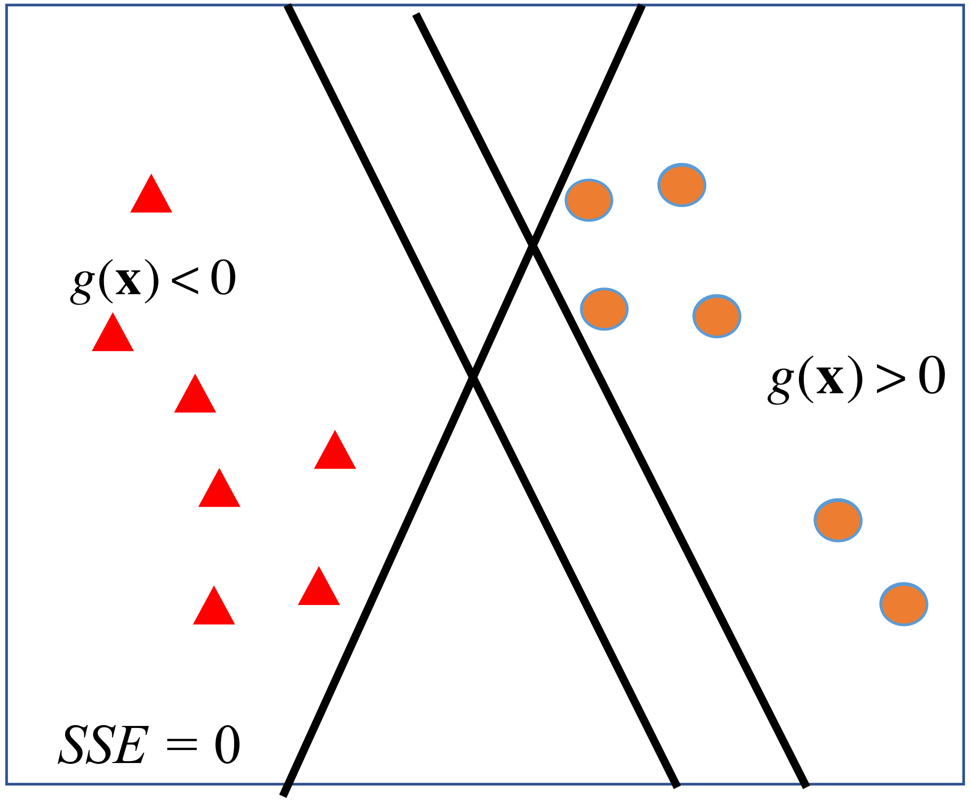
- The formulation in these slides comes from “A Tutorial on Support Vector Machines for Pattern Recognition”
- Chapter 9 of An Introduction to Statistical Learning presents the same material using a slightly different formulation (e.g. different variable names)

Three big advances over Perceptrons

- MAXIMUM MARGIN
  - They find the BEST linear separator (where best = maximum margin)
- SLACK VARIABLES
  - They can find a linear separator even when a little noise in the data means the data is not technically "linearly separable"
- THE KERNEL TRICK
  - They make it easy for the end user (software developer) to transform the data (like in polynomial regression) so that an inherently linear separator can learn non-linear decision surfaces.

Any separator is good to a Perceptron

The loss function is 0-1:
lose 0 points if you're right….even if just barely.
Lose 1 if you're wrong…..no matter how wrong.



Are all 3 lines equally good?

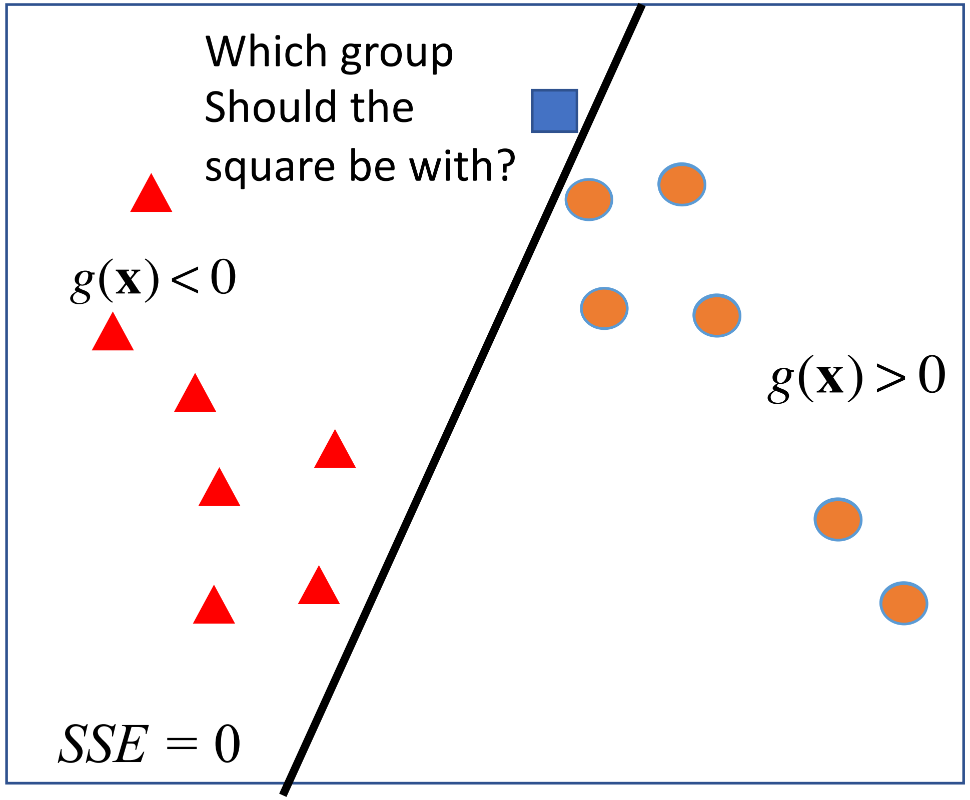
$$g(\mathbf{x}) = w_0 + w_1 x_1 + w_2 x_2 = 0$$ Decision surface

$$h(\mathbf{x}) = \begin{cases} 1 & \text{if } g(\mathbf{x}) > 0 \\ -1 & \text{otherwise} \end{cases}$$ Hypothesis function

$$SSE = \sum_i^n (y_i - h(\mathbf{x}_i))^2$$ 0-1 loss function

What if there is noise in the data?

A decision boundary with little margin to the nearest example may fail when new data is presented to it.



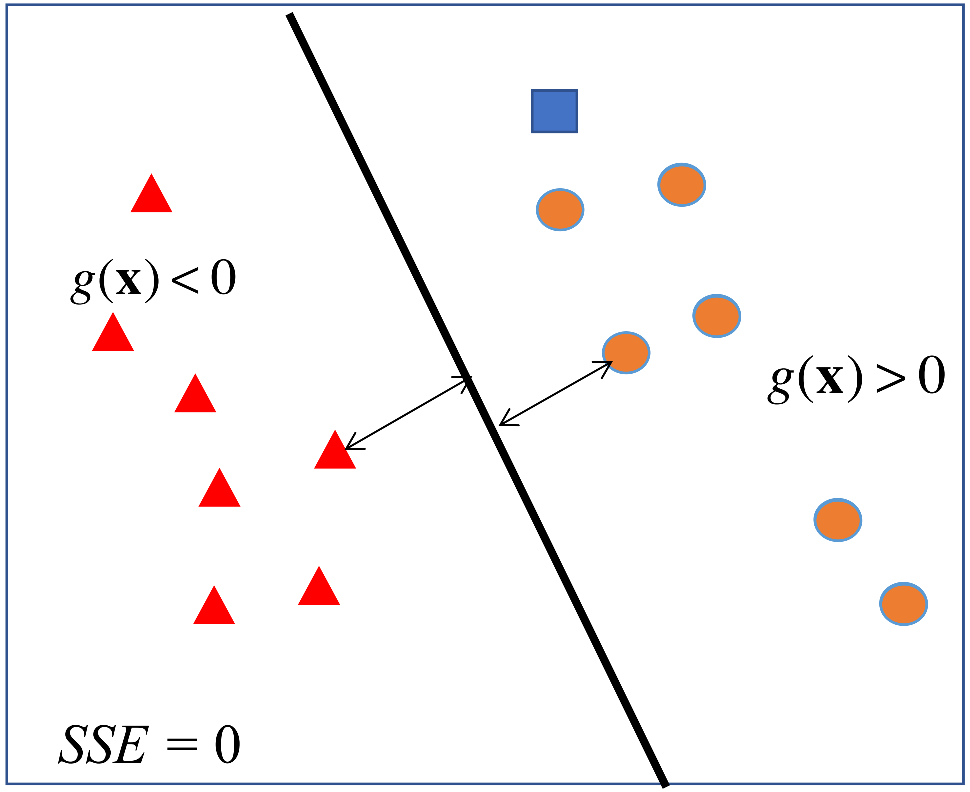
$$g(\mathbf{x}) = w_0 + w_1 x_1 + w_2 x_2 = 0$$ Decision surface

$$h(\mathbf{x}) = \begin{cases} 1 & \text{if } g(\mathbf{x}) > 0 \\ -1 & \text{otherwise} \end{cases}$$ Hypothesis function

$$SSE = \sum_{i}^{n} (y_i - h(\mathbf{x}_i))^2$$ 0-1 loss function

What if there is noise in the data?

A large-margin classifier tends to be more “robust”
(resistant to noise in the data, able to generalize)

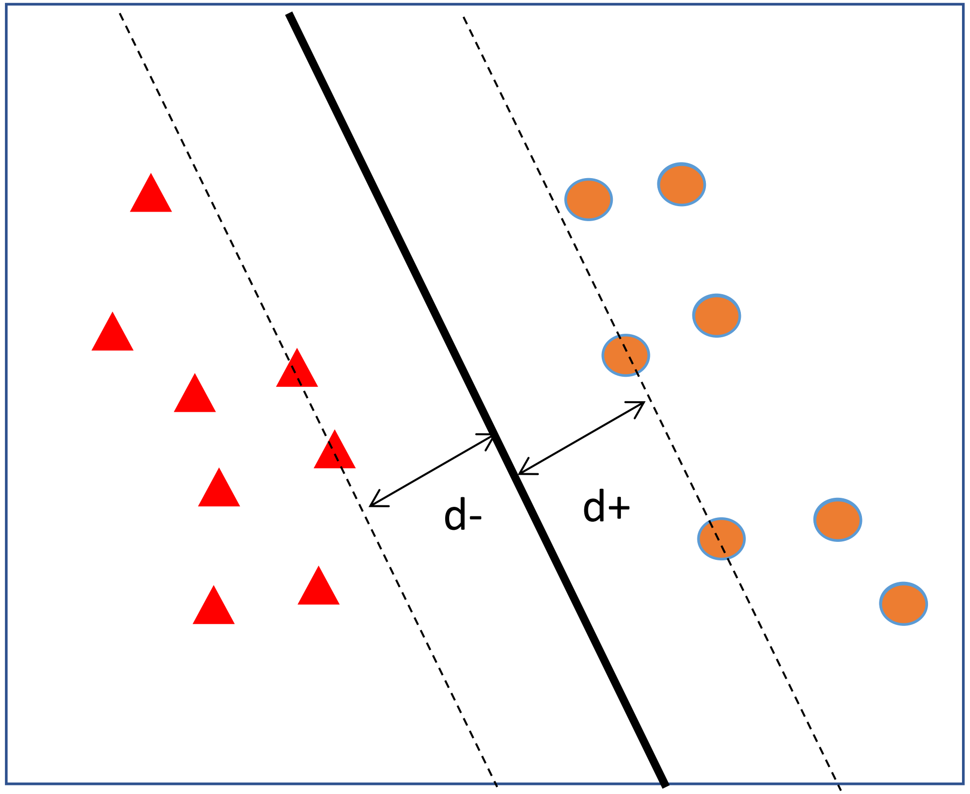


$$g(\mathbf{x}) = w_0 + w_1 x_1 + w_2 x_2 = 0$$ Decision surface

$$h(\mathbf{x}) = \begin{cases} 1 & \text{if } g(\mathbf{x}) > 0 \\ -1 & \text{otherwise} \end{cases}$$ Hypothesis function

$$SSE = \sum_i^n (y_i - h(\mathbf{x}_i))^2$$ 0-1 loss function

Maximizing the Margin

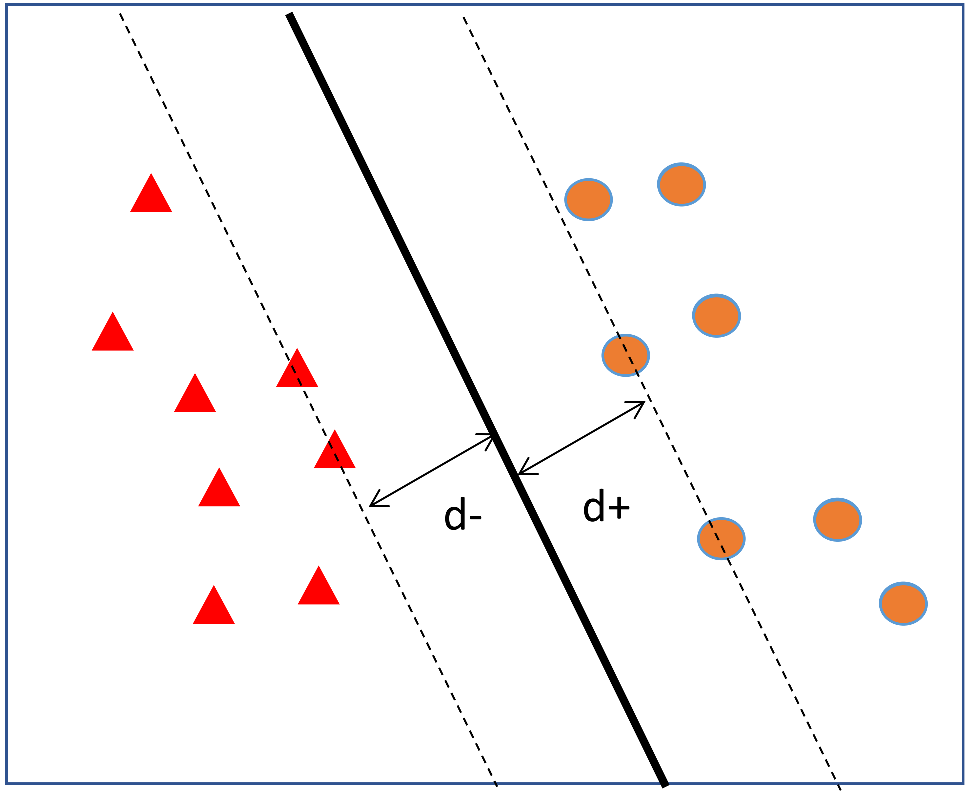


- d+ is the distance to the closest positive example.
- d- as the distance to the closest negative example
- Define the "margin", m as…

$$m = d^{+} + d^{-}$$

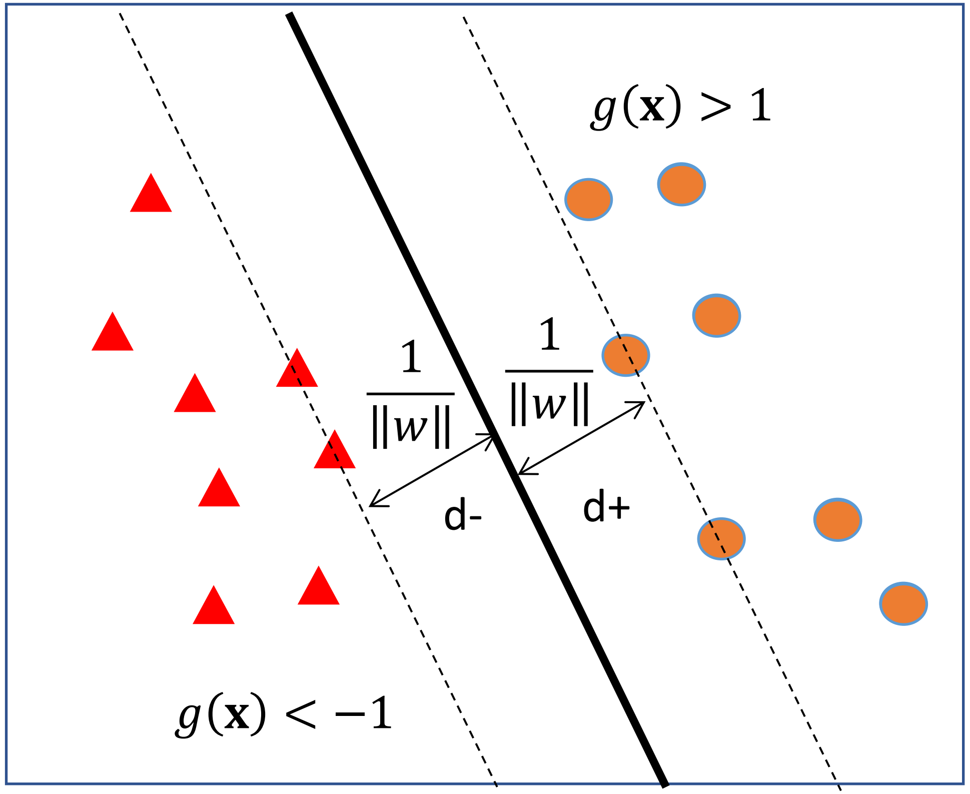
- Look for the largest margin

The Support Vectors



- The points that are within distance d of the classifier are the support vectors.
- Those are the ones on the dotted lines.
- These support vectors will become important later.

Scaling the data to simplify the math.



- There is some scaling of the data where…

$$d^+ = d^- = \frac{1}{\| w \|}$$

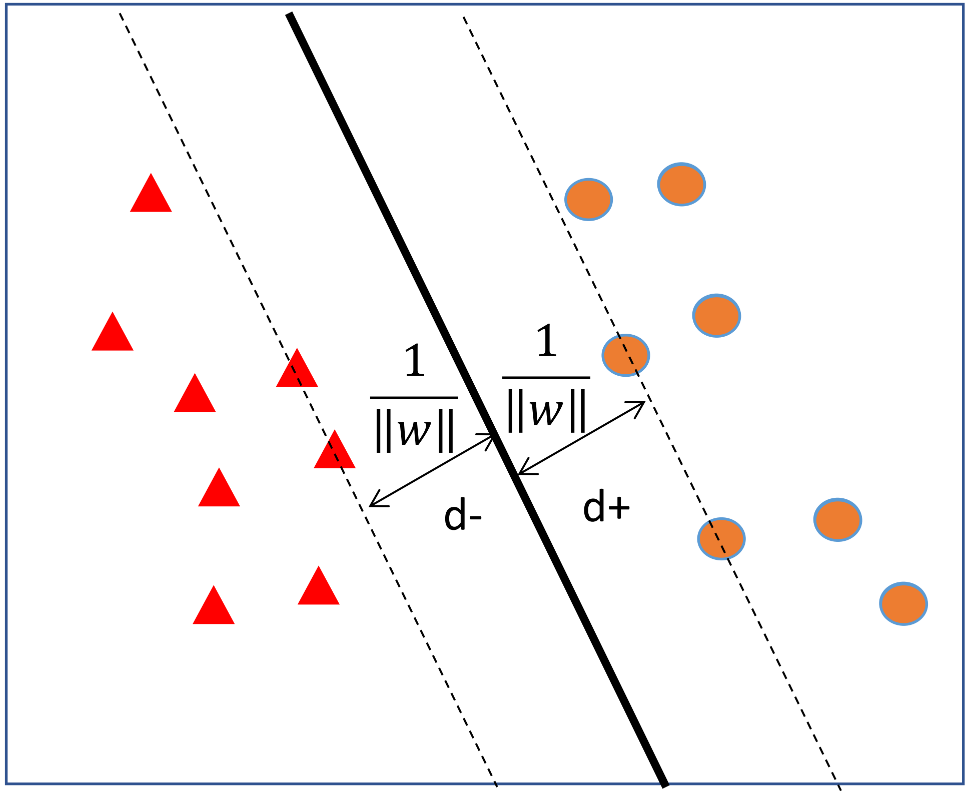
- Now, the decision boundary function will output a value with magnitude 1 or greater..

$$g(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b$$

Learned weights

Learned offset from origin

Optimizing to maximize the margin



- Maximize the margin $\frac{2}{\|\mathbf{w}\|}$

- ...such that, for every data point, the following equation holds.

$$y(\mathbf{w} \cdot \mathbf{x} + b) \geq 1,$$

True label drawn from {+1,-1}

Learned weights of decision boundary

Learned offset from origin

Making this an optimization problem

- Maximizing the margin means minimizing w.

- Introducing 1 Lagrangian multiplier α_i per data point lets us add the constraint that every data point be on the right side of the line into the formula to optimize.

$$L_p = \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{n} \alpha_i y_i(\mathbf{w} \cdot \mathbf{x}_i + b) + \sum_{i=1}^{n} \alpha_i$$

- Now solve for where the gradient of w vanishes, with respect to $\alpha_1, \ldots, \alpha_n$ (For this to work we require every $\alpha_i \geq 0$)

A dual formulation

- It turns out there is a *dual* formulation of the problem that will result in the same values for $\mathbf{w}, b, \alpha_1, \dots, \alpha_n$
- This time, maximize and require the gradient vanish with respect to $\mathbf{w}, b$
- That translates to putting these conditions on the maximization:

$$\mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i \qquad 0 = \sum_{i=1}^{n} \alpha_i y_i$$

A dual formulation, continued

- Substituting those formulae into the previous formula gives the following *dual* formulation, L_d.
- Training a linear SVM is done by maximizing L_d with respect to $\alpha_1 \ldots \alpha_n$
- The numbers that are learned here are the $\alpha_1 \ldots \alpha_n$
- Once you've trained, points where α_i>0 are the *support vectors*
- The support vectors are the data points that lie on the margin.

$$L_d = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{j=1}^{n} \sum_{i=1}^{n} \alpha_i \alpha_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j)$$

Getting the boundary from the support vectors

- Let *s* be the index of a support vector in the set of support vectors S.
- Get the decision boundary **w** from the support vectors like this:

$$\mathbf{w} = \sum_{s}^{S} \alpha_s y_s \mathbf{x}_s$$

- Use the line to classify a new point **z ,** just like a perceptron.
- Equivalently, we could directly use the support vectors to classify z.

$$h(\mathbf{z}) = sign(g(\mathbf{z})) \qquad g(\mathbf{z}) = \mathbf{w} \cdot \mathbf{z} + b = \sum_{s}^{S} \alpha_s y_s (\mathbf{x_s} \cdot \mathbf{z}) + b$$

Returns +1 or -1

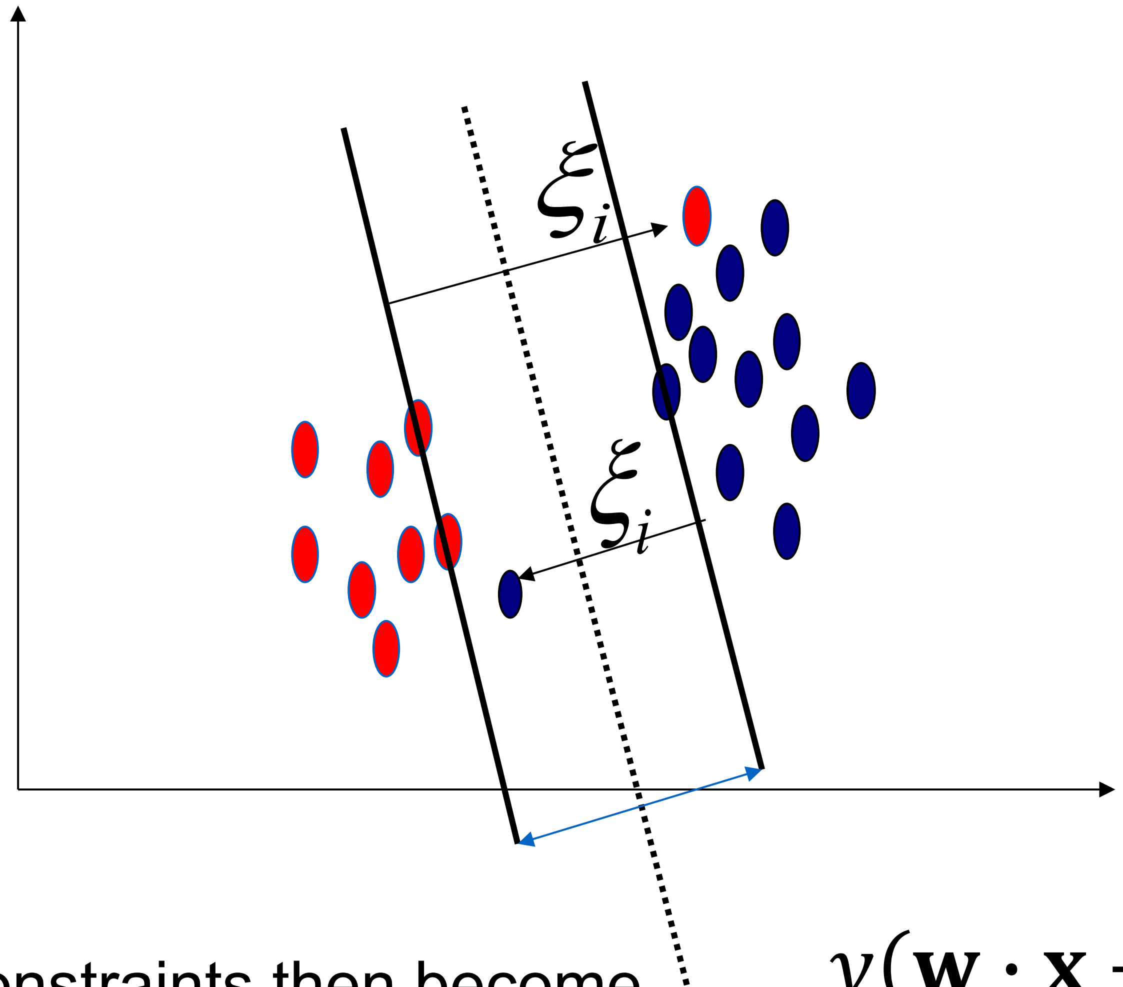
Three big advances over Perceptrons

- MAXIMUM MARGIN
  - They find the BEST linear separator (where best = maximum margin)

- SLACK VARIABLES
  - They can find a linear separator even when a little noise in the data means the data is not technically "linearly separable"

- THE KERNEL TRICK
  - They make it easy for the end user (software developer) to transform the data (like in polynomial regression) so that an inherently linear separator can learn non-linear decision surfaces.

Non-Linearly Separable Data



Allow some instances to fall within the margin, but penalize them

Introduce slack variables ξ (one per data point)

The constraints then become…

$$y(\mathbf{w} \cdot \mathbf{x} + b) \geq 1 - \xi \quad \forall \{\mathbf{x}, y\}$$

Our "Prime" Optimization, with slack

- Now we're trying to minimize **w** and also minimize the total "slack" , which is embodied by the slack variables $\xi_1 \; \dots \xi_n$
- Recall that each ξ_i captures how far over on the wrong side of the line data point $\mathbf{x}_i$ is.
- As I change C, I can increase or decrease the importance of the overall misclassification

$$minimize\ this\colon \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{n}\xi_i$$

Our "Prime" Optimization, with slack

- Let's put our constraints into the optimization formula, like we did before.
- Add a Lagrangian parameter μ_i for each slack variable ξ_i
- Require every $\alpha_i \geq 0$, every $\mu_i \geq 0$, and every $\xi_i \geq 0$

$$L_p = \underbrace{\frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n}\xi_i}_{\text{Our original minimization over W and } \xi} - \underbrace{\left(\sum_{i=1}^{n}\alpha_i y_i(\mathbf{w}\cdot\mathbf{x}_i + b) + \sum_{i=1}^{n}\mu_i\xi_i\right)}_{\text{Terms added to directly include constraints into the formula}}$$

Three big advances over Perceptrons

- MAXIMUM MARGIN
  - They find the BEST linear separator (where best = maximum margin)

- SLACK VARIABLES
  - They can find a linear separator even when a little noise in the data means the data is not technically "linearly separable"

- THE KERNEL TRICK
  - They make it easy for the end user (software developer) to transform the data (like in polynomial regression) so that an inherently linear separator can learn non-linear decision surfaces.

Reminder of where we are

- You train an SVM by optimizing on this (Yes, this is the dual formulation. Yes I'm leaving out slack. This is for simplicity of presentation.)

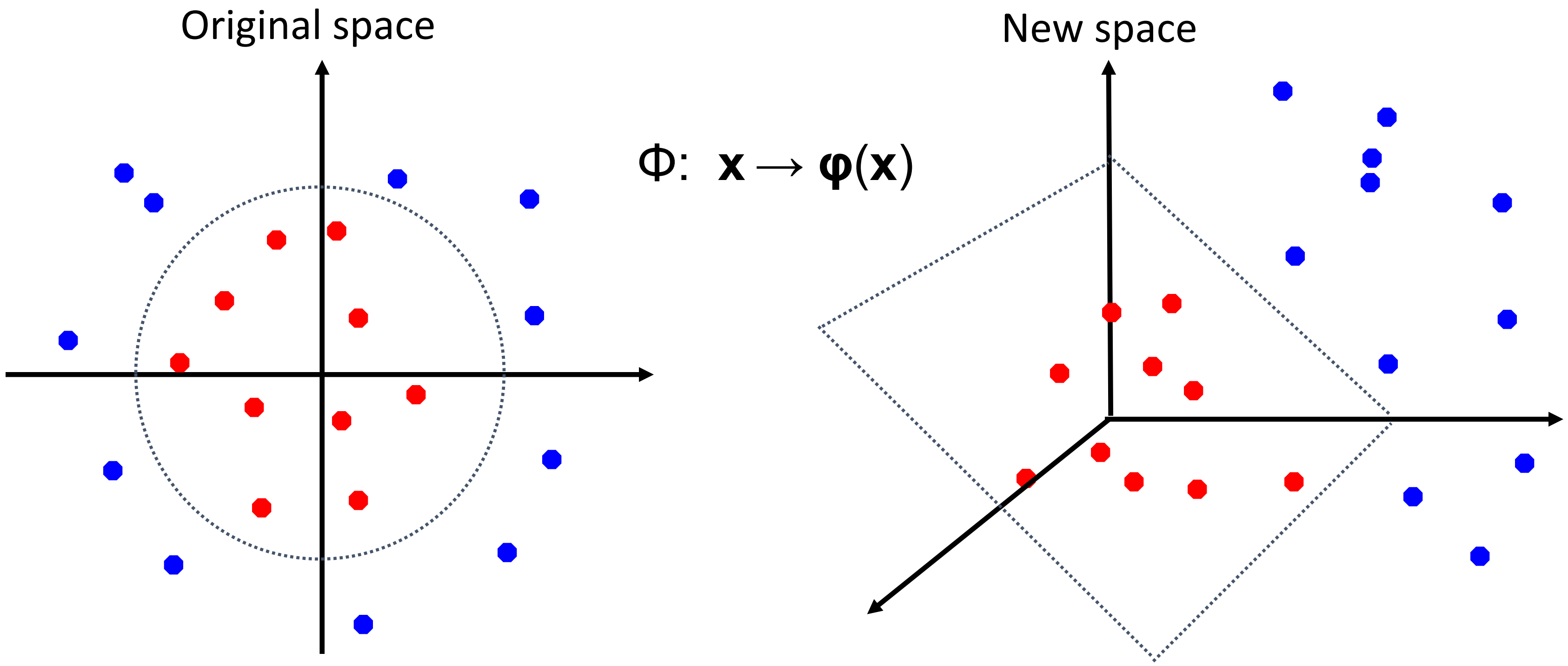
$$L_d = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{j=1}^{n} \sum_{i=1}^{n} \alpha_i \alpha_j y_i y_j \left(\mathbf{x}_i \cdot \mathbf{x}_j \right)$$

- Once you have your non-zero α values for the support vectors s, you use it to classify a new point **z** like this:

$$g(\mathbf{z}) = \sum_{s}^{S} \alpha_s y_s (\boldsymbol{x}_s \cdot \mathbf{z}) + b$$

Non-linear separation

- Map the original feature space to a higher-dimensional feature space where the training set is separable by a hyperplane. Call this mapping function $\phi(\cdot)$



Note: I can't remember which person's slides I adapted this image from, but it is one of these 3: Constantin F. Aliferis & Ioannis Tsamardinos, and Martin Law

With our non-linear mapping $\phi(\cdot)$

- You train an SVM by optimizing on this

$$L_d = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{j=1}^{n} \sum_{i=1}^{n} \alpha_i \alpha_j y_i y_j \left(\phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_j) \right)$$

- Once you have your non-zero α values for the support vectors s, you use it to classify a new point **z** like this:

$$\mathbf{g}(\mathbf{z}) = \sum_{s}^{S} \alpha_s y_s (\phi(\mathbf{x}_s) \cdot \phi(\mathbf{z})) + b$$

The kernel function

- If we combine the transformation function $\phi(\cdot)$ and the inner product, we call this a Kernel:

$$K\left(\mathbf{x_i}, \mathbf{x}_j\right) = \phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_i)$$

- Putting this into the optimization function gives…

$$L_d = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{j=1}^{n} \sum_{i=1}^{n} \alpha_i \alpha_j y_i y_j K\left(\mathbf{x_i}, \mathbf{x}_j\right)$$

- And using it to classify is done like this …

$$g(\mathbf{z}) = \sum_{s}^{S} \alpha_s y_s K(\mathbf{x_s}, \mathbf{z}\,) + b$$

The kernel trick

- If I know what the inner product of two transformed items is, then I can directly calculate the inner product without doing the transformation first.
- The simplest example: making a polynomial separator, where the polynomial exponent is 2 and the input x, z are each a scalar.
- Given the kernel below, we can directly calculate the inner product, without having to first apply $\phi(\cdot)$.

$$\phi(x) = [x, x^2]$$
$$\phi(z) = [z, z^2]$$

$$\boxed{K(x,z) = xz + x^2z^2} = \phi(x) \cdot \phi(z)$$

Just use this directly.

Why care about the kernel trick

- If you have a formula for $\phi(x) \cdot \phi(z)$, you can skip doing $\phi(\cdot)$
- This fact is used in one of the most popular kernels, the Gaussian Kernel, aka the Radial Basis Function Kernel (RBF)

$$K(\mathbf{x}, \mathbf{z}) = e^{-\|\mathbf{x}-\mathbf{z}\|^2/2\sigma^2}$$

- The RBF kernel, implicitly uses a $\phi(\cdot)$ which, if calculated as an explicit step, would expand the basis of the natural log e using the infinite Taylor series. This would result in an infinite dimensional vector.
- By using the Kernel trick you never explicitly use $\phi(\cdot)$ and never try to represent something infinite.

The kernel trick may force use of support vectors

- For a kernel like an RBF kernel, we can't ever calculate $\phi(x)$, since it would require an infinite series (the Taylor series) and that would mean an infinite dimensional vector.
- This means we can't directly represent the decision boundary **w**, since it would also have to be infinite dimensional.
- This means we optimize using the dual formulation, L_d
- This also forces us to classify a new point, z, by using the points on the margin (the support vectors) to classify the point.

$$h(\mathbf{z}) = sign(g(\mathbf{z}))$$

Returns +1 or -1

$$g(\mathbf{z}) = \sum_{s}^{S} \alpha_s y_s K(\mathbf{x_i}, \mathbf{z}) + b$$

You can build your own kernels

- If you create a kernel for a data type, you can apply a SVM to it.
- For example, let’s make a $\phi_t(\cdot)$ for text documents:
    1. Pick a dictionary (e.g. the Oxford English Dictionary, or OED)
    2. For any text document, create an n-dimensional binary vector where the nth dimension is 1 if the nth OED word is in the document and 0 otherwise
- Now we can turn any text document into a vector
- Define K(x,z) as the inner product of these two vectors

$$K(x, z) = \phi_t(\mathbf{x}) \cdot \phi_t(\mathbf{z})$$

- We’re done! We can run a SVM on text documents!
- Making a spam filter is now easy.

You can build kernels out of other kernels

- Once you have a set of kernels, you can compose new kernels from them.
- Let’s see how…

Definitions for the following slide

$k_1(x,x')$ and $k_2(x,x')$ are valid kernels on $\{\text{x,x'}\} \in S$

S is some set (of anything: emails, images, integers)

$c > 0$ is a constant

$f(\cdot)$ is any function

$q(\cdot)$ is a polynomial with non-negative coefficients

$\phi(\text{x})$ is a function from the ->$\mathbb{R}^m$

$k_3(\cdot,\cdot)$is a valid kernel in $\mathbb{R}^m$

A is a symmetric positive semidefinite matrix

$x = (x_a, x_b)$ ....essentially, x can be decomposed into subparts

...like scalars in a vector

$k_a(\cdot,\cdot), k_b(\cdot,\cdot)$ are valid kernels over their respective spaces

Techniques for Kernel Construction

Given valid kernels $k_1(x,x')$ and $k_2(x,x')$,

the following are also valid kernels

$$k(x,x') = ck_1(x,x')$$

$$k(x,x') = f(x)k_1(x,x')f(x')$$

$$k(x,x') = q(k_1(x,x'))$$

$$k(x,x') = \exp(k_1(x,x'))$$

$$k(x,x') = k_1(x,x') + k_2(x,x')$$

$$k(x,x') = k_3(\phi(x),\phi(x'))$$

$k(x,x') = x^T A x'$ This one assumes x,x' are vectors

$$k(x,x') = k_a(x_a,x'_a) + k_b(x_b,x'_b)$$

$$k(x,x') = k_a(x_a,x'_a)k_b(x_b,x'_b)$$