Linear classifiers are great, but what if there exists no linear decision boundary?

Handcrafted Feature Expansion

We can make linear classifiers non-linear by applying basis functions (feature transformations) on the input feature vectors.

Formally, for a data vector $\vec{x} \in \mathbb{R}^d$ we apply the transformation $\vec{x} \rightarrow \phi(\vec{x})$ where $\phi(\vec{x}) \in \mathbb{R}^D$. $D \gg d$ usually since we add dimensions that capture nonlinear features.

Advantage: It is simple, and your problem stays convex and well behaved

Disadvantage: $\phi(\vec{x})$ may be VERY high dimensional

Consider the following example:

$$\vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix} \quad \text{and define} \quad \phi(\vec{x}) = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_d \\ x_1 x_2 \\ \vdots \\ x_{d-1} x_d \\ \vdots \\ x_1 x_2 \cdots x_d \end{bmatrix}$$

This new representation, $\phi(\vec{x})$, is very expressive and allows for complicated non-linear decision boundaries - but the dimensionality is extremely high. This makes our algorithm unbearably slow.

The Kernel Trick

Gradient Descent with Squared Loss:

The kernel trick is a way to get around this dilemma by learning a function in the much higher dimensional space, w/o ever computing a single vector $\phi(\vec{x})$ or ever computing the full vector $\vec{w}$.

Observe the following:

If we use gradient descent with any of our standard loss functions, the gradient is a linear combination of the input samples

i.e for squared loss

$$L(\vec{w}) = \sum_{i=1}^{n} (\bar{w}^T \bar{x}_i - y_i)^2$$

The gradient descent rule, w/ learning rate $s>0$ updates $\bar{w}$ over time

$$\bar{w}_{t+1} \leftarrow \bar{w}_t - s \frac{\partial L}{\partial w}$$

where

$$\frac{\partial L}{\partial w} = \sum_{i=1}^{n} \underbrace{2(\bar{w}^T \bar{x}_i - y_i)}_{\gamma_i : \text{function of } x_i, y_i} \bar{x}_i = \sum_{i=1}^{n} \gamma_i \bar{x}_i$$

We now show that $\bar{w}$ can be expressed as a linear combination of input vectors,

$$\bar{w} = \sum_{i=1}^{n} \alpha_i \bar{x}_i$$

Since the loss is convex the final solution is independent of the initialization, we thus initialize $\bar{w}_0$ to whatever we want.

Let us pick

$$\bar{w}_0 = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

For this initial choice of $\bar{w}_0$, the linear combination in

$$\bar{w} = \sum_{i=1}^{n} \alpha_i \bar{x}_i$$

is trivially $\alpha_1 = \cdots = \alpha_n = 0$

We now show that throughout the entire gradient descent optimization such coefficients $\alpha_1, \ldots, \alpha_n$ must always exist, as we can re-write the gradient updates entirely in terms of updating the α_i coefficients:

$$\bar{w}_1 = \bar{w}_0 - s \sum_{i=1}^{n} 2(\bar{w}^T \bar{x}_i - y_i)\bar{x}_i = \sum_{i=1}^{n} \alpha_i^0 \bar{x}_i - s \sum_{i=1}^{n} \gamma_i^0 \bar{x}_i = \sum_{i=1}^{n} \alpha_i^1 \bar{x}_i \quad \left(\alpha_i^1 = \alpha_i^0 - s\gamma_i^0\right)$$

$$\bar{w}_2 = \bar{w}_1 - s \sum_{i=1}^{n} 2(\bar{w}^T \bar{x}_i - y_i)\bar{x}_i = \sum_{i=1}^{n} \alpha_i^1 \bar{x}_i - s \sum_{i=1}^{n} \gamma_i^1 \bar{x}_i = \sum_{i=1}^{n} \alpha_i^2 \bar{x}_i \quad \left(\alpha_i^2 = \alpha_i^1 - s\gamma_i^1\right)$$

$$\bar{w}_3 = \bar{w}_2 - s \sum_{i=1}^{n} 2(\bar{w}^T \bar{x}_i - y_i)\bar{x}_i = \sum_{i=1}^{n} \alpha_i^2 \bar{x}_i - s \sum_{i=1}^{n} \gamma_i^2 \bar{x}_i = \sum_{i=1}^{n} \alpha_i^3 \bar{x}_i \quad \left(\alpha_i^3 = \alpha_i^2 - s\gamma_i^2\right)$$

$$\cdots \qquad \cdots \qquad \cdots$$

$$\bar{w}_t = \bar{w}_{t-1} - s \sum_{i=1}^{n} 2(\bar{w}^T \bar{x}_i - y_i)\bar{x}_i = \sum_{i=1}^{n} \alpha_i^{t-1} \bar{x}_i - s \sum_{i=1}^{n} \gamma_i^{t-1} \bar{x}_i = \sum_{i=1}^{n} \alpha_i^t \bar{x}_i \quad \left(\alpha_i^t = \alpha_i^{t-1} - s\gamma_i^{t-1}\right)$$

Formally the argument is by induction.

$\bar{w}$ is trivially a linear combination of our training vectors for $\bar{w}_0$ (base case).

If we apply the inductive hypothesis for $\bar{w}_t$, it follows for $\bar{w}_{t+1}$.

The update-rule for α_i^t is thus

$$\alpha_i^t = \alpha_i^{t-1} - s\gamma_i^{t-1} \quad \text{and we have} \quad \alpha_i^t = -s \sum_{r=0}^{t-1} \gamma_i^r$$

In other words, we can perform the entire gradient descent update rule without ever expressing $\bar{w}$ explicitly.

We just keep track of the n coefficients $\alpha_1, \ldots, \alpha_n$.

Now that $\bar{w}$ can be written as a linear combination of the training set, we can also express the inner product of $\bar{w}$ w/ any input $\bar{x}_i$ purely in terms of the inner product between training inputs:

$$\bar{w}^T \bar{x}_j = \sum_{i=1}^{n} \alpha_i \bar{x}_i^T \bar{x}_j$$

Consequently, we can also re-write the squared-loss from $\ell(\bar{w}) = \sum_{i=1}^{n} (\bar{w}^T \bar{x}_i - y_i)^2$ entirely in terms of inner-product between training inputs:

$$\ell(\alpha) = \sum_{i=1}^{n} \left(\sum_{j=1}^{n} \alpha_j \bar{x}_j^T \bar{x}_i - y_i \right)^2$$

During test-time we also only need these coefficients to make a prediction on a test input x_i and can write the entire classifier in terms of inner-products between the test points and training points:

$$h(\bar{x}_t) = \bar{w}^T \bar{x}_t = \sum_{j=1}^{n} \alpha_j \bar{x}_j^T \bar{x}_t$$

NOTICE THE THEME!

The only information we ever need in order to learn a hyperplane classifier with the squared loss is the inner-products between all pairs of data vectors.

Inner-Product Computation

Let's go back to the previous example,

$$\phi(\vec{x}) = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_d \\ x_1 x_2 \\ \vdots \\ x_{d-1} x_d \\ \vdots \\ x_1 x_2 \cdots x_d \end{bmatrix}$$

The inner product $\phi(\vec{x})^T \phi(\vec{z})$ can be formulated as:

$$\phi(\vec{x})^T \phi(\vec{z}) = 1 \cdot 1 + x_1 z_1 + x_2 z_2 + \cdots + x_1 x_2 z_1 z_2 + \cdots + x_1 \cdots x_d z_1 \cdots z_d = \prod_{k=1}^{d} (1 + x_k z_k)$$

The sum of 2^d terms becomes the product of d terms.

We can compute the inner-product from the above formula in $O(d)$ time instead of $O(2^d)$.

We define the function

$$\underbrace{k(x_i, x_j)}_{\text{kernel function}} = \phi(x_i)^T \phi(x_j)$$

With a finite training set of n samples, inner products are often pre-computed and stored in a kernel Matrix:

$$K_{ij} = \phi(x_i)^T \phi(x_j)$$

If we store matrix K, we only need to do simple inner-product lookups and low dimensional computations throughout the gradient descent algorithm.

The final classifier becomes:

$$h(\vec{x}_t) = \sum_{j=1}^{n} \alpha_j k(x_j, x_t)$$

During training in the new high dimensional space of $\phi(\vec{x})$ we want to compute γ_i through kernels without ever computing any $\phi(\vec{x}_i)$, or even $\vec{w}$.

We previously established that

$$\vec{w} = \sum_{j=1}^{n} \alpha_j \phi(\vec{x}_j)$$

and

$$\gamma_i = 2\left(\sum_{j=1}^{n} \alpha_j K_{i,j} - y_i\right)$$

The gradient update in iteration t+1 becomes

$$\alpha_i^{t+1} \leftarrow \alpha_i^t - 2s\left(\sum_{j=1}^{n} \alpha_j K_{i,j} - y_i\right)$$

As we have n such updates to do, the amount of work per gradient update in the transformed space is $O(n^2)$ ... **FAR** better than $O(2^d)$

General Kernels

Linear: $K(\vec{x}, \vec{z}) = \vec{x}^T \vec{z}$

Linear kernel is equivalent to using a linear classifier – but it can be faster to use a kernel matrix if dimensionality of data is high

Polynomial: $K(\vec{x}, \vec{z}) = (1 + \vec{x}^T \vec{z})^d$

Radial Basis Function (Gaussian Kernel): $K(\vec{x}, \vec{z}) = e^{-\frac{\|\vec{x} - \vec{z}\|^2}{\sigma^2}}$

The RBF is the most popular kernel! It is a universal approximator! Its corresponding feature vector is infinite dimensional and cannot be computed. However, very effective low dimensional approximations exist.

Exponential Kernel: $K(\vec{x}, \vec{z}) = e^{-\frac{\|\vec{x} - \vec{z}\|}{2\sigma^2}}$

Laplacian Kernel: $K(\vec{x}, \vec{z}) = e^{-\frac{|\vec{x} - \vec{z}|}{\sigma}}$

Sigmoid Kernel: $K(\vec{x}, \vec{z}) = \tanh(a\vec{x}^T + c)$

Kernel Functions

Can any function $\mathsf{K}(\cdot,\cdot) \to \mathcal{R}$ be used as a kernel?

No, the matrix $\mathsf{K}(\mathbf{x}_i, \mathbf{x}_j)$ has to correspond to real inner-products after some transformation $\mathbf{x} \to \phi(\mathbf{x})$. This is the case if and only if K is *positive semi-definite*.

> **Definition**: A matrix $A \in \mathbb{R}^{n \times n}$ is positive semi-definite iff $\forall \mathbf{q} \in \mathbb{R}^n$, $\mathbf{q}^\top A \mathbf{q} \geq 0$.

Remember $\mathsf{K}_{ij} = \phi(\mathbf{x}_i)^\top \phi(\mathbf{x}_j)$. So $\mathsf{K} = \Phi^\top \Phi$, where $\Phi = [\phi(\mathbf{x}_1), \ldots, \phi(\mathbf{x}_n)]$. It follows that K is p.s.d., because $\mathbf{q}^\top \mathsf{K} \mathbf{q} = (\Phi^\top \mathbf{q})^2 \geq 0$. Inversely, if any matrix $\mathbf{A}$ is p.s.d., it can be decomposed as $A = \Phi^\top \Phi$ for some realization of Φ.

You can even define kernels over sets, strings, graphs and molecules.

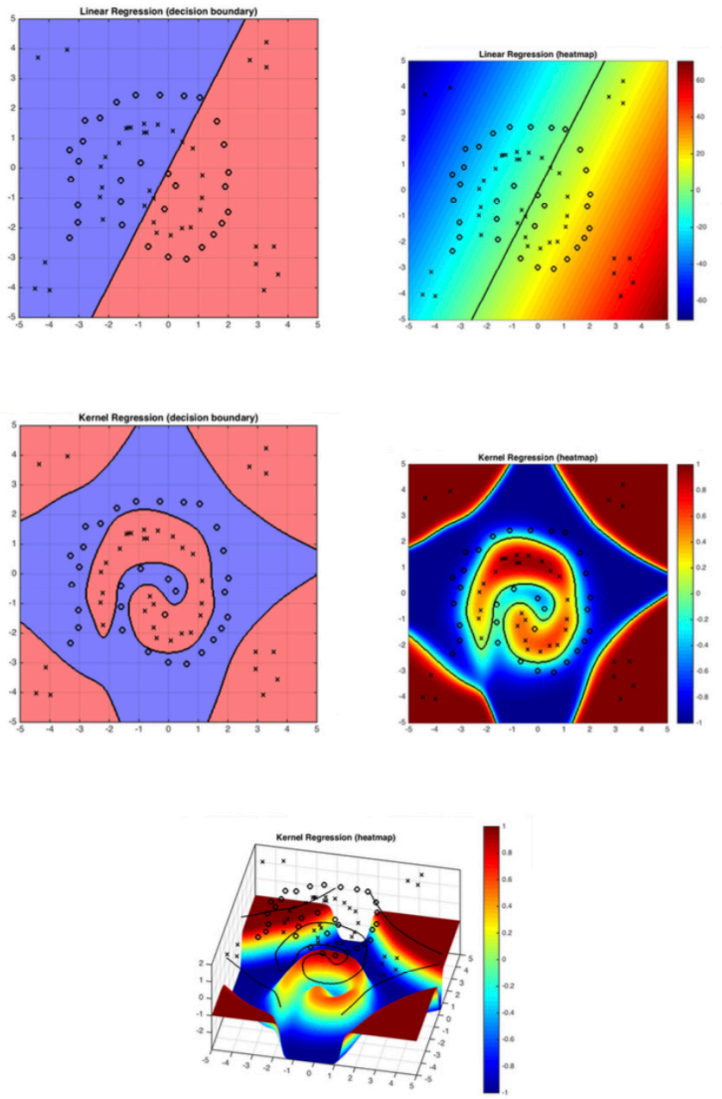

Figure 1: The demo shows how kernel function solves the problem linear classifiers can not solve. RBF works well with the decision boundary in this case.

Well-Defined Kernels

Kernels built by recursively combining one or more of the following rules are called **well-defined kernels:**

① $k(\vec{x}, \vec{z}) = \vec{x}^T \vec{z}$

② $k(\vec{x}, \vec{z}) = c k_1(x, z)$

③ $k(\vec{x}, \vec{z}) = k_1(\vec{x}, \vec{z}) + k_2(\vec{x}, \vec{z})$

④ $k(\vec{x}, \vec{z}) = g(k(x, z))$

⑤ $k(\vec{x}, \vec{z}) = k_1(\vec{x}, \vec{z}) k_2(\vec{x}, \vec{z})$

⑥ $k(\vec{x}, \vec{z}) = f(\vec{x}) k_1(\vec{x}, \vec{z}) f(\vec{z})$

⑦ $k(\vec{x}, \vec{z}) = e^{k_1(\vec{x}, \vec{z})}$

⑧ $k(\vec{x}, \vec{z}) = \vec{x}^T A \vec{z}$

where

k_1, k_2 are well-defined kernels

$c > 0$

g is a polynomial function with positive coeficients

f is any function

$A \succ 0$ is positive semi-definite

Kernel being well-defined is equivalent to the corresponding kernel matrix, K, being positive semi-definite, which is equivalent to any of the following statements:

① All eigenvalues of K are nonnegative

② $\exists$ real matrix P such that $K = P^T P$

③ $\forall$ real vector $\vec{x}$, $\vec{x}^T K \vec{x} \geq 0$

Theorem: The RBF kernel $k(\vec{x},\vec{z}) = e^{\frac{-\|\vec{x}-\vec{z}\|^2}{\sigma^2}}$ is a well-defined kernel matrix.

Proof:

$k_1(x,z) = \vec{x}^T\vec{z}$ well-defined by rule 1

$k_2(x,z) = \frac{2}{\sigma^2}k_1(x,z) = \frac{2}{\sigma^2}\vec{x}^T\vec{z}$ well-defined by rule 2

$k_3(x,z) = e^{k_2(\vec{x},\vec{z})} = e^{\frac{2}{\sigma^2}\vec{x}^T\vec{z}}$ well-defined by rule 7

$k_4(x,z) = e^{-\frac{\vec{x}^T\vec{x}}{\sigma^2}}k_3(x,z)\,e^{-\frac{\vec{z}^T\vec{z}}{\sigma^2}}$ well-defined by rule 6; $f(x) = e^{\frac{-\vec{x}^T\vec{x}}{\sigma^2}}$

$= e^{\frac{-\vec{x}^T\vec{x} + 2\vec{x}^T\vec{z} - \vec{z}^T\vec{z}}{\sigma^2}}$

$= e^{\frac{-(\vec{x}-\vec{z})^2}{\sigma^2}} = e^{\frac{-\|x-z\|^2}{\sigma^2}} = k_{RBF}(\vec{x},\vec{z})$

Q.E.D

You can even define kernels of sets, or strings, or molecules.

Theorem: The following kernel is well defined on any two sets $S_1, S_2 \subseteq \Omega$,

$$k(S_1, S_2) = e^{|S_1 \cap S_2|}$$

Proof: List out all possible samples Ω and arrange into a sorted list.

We define a vector $\vec{x}_S \in \{0,1\}^{|\Omega|}$, where each of its elements indicates whether a corresponding sample is included in set S.

It can be shown

$$k(S_1, S_2) = e^{\vec{x}_{S_1}^T \vec{x}_{S_2}}$$

which is a well-defined kernel by rules 1 and 7.

Q.E.D

Kernel Machines

In practice, an algorithm can be kernelized in 3 steps:

① Prove that the solution lies in the span of training points
i.e
$$\bar{w} = \sum_{i=1}^{n} \alpha_i \bar{x}_i \text{ for some } \alpha_i$$

② Rewrite the algorithm and the classifier so that all training or testing inputs $\bar{x}_i$ are only accessed in inner-products w/ other inputs, e.g. $x_i^T x_j$

③ Define a kernel function and substitute $k(x_i, x_j)$ for $x_i^T x_j$

Kernelized Linear Regression: An Example

Recall OLS Regression minimizes the following squared loss regression loss function,

$$\min_{w} \sum_{i=1}^{n} (\bar{w}^T \bar{x}_i - y_i)^2$$

to find the hyperplane $\bar{w}$.

The prediction at a test point is simply $h(\bar{x}_T) = \bar{w}^T \bar{x}_T$

If we let $X = [x_1, \ldots, x_n]$ and $\vec{y} = [y_1, \ldots, y_n]^T$, the solution of OLS is

$$\bar{w} = (XX^T)^{-1} Xy \qquad (*)$$

Kernilization

We begin by expressing the solution $\bar{w}$ as a linear combination of the training inputs

$$\bar{w} = \sum_{i=1}^{n} \alpha_i \bar{x}_i = X\vec{\alpha}$$

Such a vector $\vec{\alpha}$ must always exist by observing the gradient updates that occur in (*) are minimized w/ gradient descent and the initial vector is set to $\vec{w} = \vec{0}$ (Squared loss convex ⇒ solution independent of initialization)

Similarly, during testing a test point is only accessed through inner products w/ training inputs:

$$h(\vec{z}) = \bar{w}^T \vec{z} = \sum_{i=1}^{n} \alpha_i \vec{x}_i^T \vec{z}$$

We can immediately kernelize the algorithm by substituting $k(\vec{x}, \vec{z})$ for any inner product $x^T z$.

It remains to show we can solve for the values of α in closed form.

Theorem: Kernilized ordinary least squares has the solution $\vec{\alpha} = K^{-1}\vec{y}$

Proof:

$$X\vec{\alpha} = \bar{w} = (XX^T)^{-1}X\vec{y}$$

$$(X^TX)(X^TX)\vec{\alpha} = X^T(XX^T(XX^T)^{-1})X\vec{y}$$

Multiply from left

$$X^TXX^T$$

$$K^2\vec{\alpha} = Ky$$

$$K = X^TX$$

$$\vec{\alpha} = K^{-1}\vec{y}$$

Multiply from left

$$(K^{-1})^2$$

Q.E.D

Kernel regression can be extended to the kernelized version of ridge regression.

The solution then becomes

$$\vec{\alpha} = (K + \tau^2 I)^{-1}\vec{y}$$

In practice a small value of $\tau^2 > 0$ increases stability, especially if K is **NOT** invertible.

Testing

Remember that we defined $\bar{w} = X\vec{\alpha}$. The prediction of a test point $\bar{z}$ then becomes

$$h(\bar{z}) = \bar{z}^T \bar{w} = \bar{z}^T \underbrace{X\vec{\alpha}}_{\bar{w}} = \underbrace{\vec{k}_*}_{\bar{z}^T X} \underbrace{(K + \tau^2 I)^{-1}\vec{y}}_{\vec{\alpha}} = k_*\vec{\alpha}$$

or, if everything is in closed form:

$$h(\bar{z}) = \vec{k}_* (K + \tau^2 I)^{-1} \vec{y},$$

where $\vec{k}_*$ is the kernel (vector) of the test point with the training points, i.e, the i^{th} dimension corresponds to $[k_*]_i = \phi(\bar{z})^T \phi(\bar{x}_i)$, the inner product between the test point $\bar{z}$ w/ the training point $\bar{x}_i$ after the mapping into the feature space through ϕ.

Nearest Neighbors

Let $D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$

Kernel SVM

The original, **primal** SVM is a quadratic programming problem:

$$\min_{w,b} \bar{w}^T\bar{w} + C\sum_{i=1}^{n} \xi_i \quad \text{such that} \quad \forall i,\ y_i(w^T x_i + b) \geq 1 - \xi_i\ ;\ \xi_i \geq 0$$

has the **dual form**

$$\min_{a_1, a_2, \ldots, a_n} \frac{1}{2}\sum_{i,j}\alpha_i\alpha_j y_i y_j K_{ij} - \sum_{i=1}^{n}\alpha_i \quad \text{such that} \quad 0 \leq \alpha_i \leq C \quad \sum_{i=1}^{n}\alpha_i y_i = 0$$

where

$$\bar{w} = \sum_{i=1}^{n}\alpha_i y_i \phi(x_i),$$

and

$$h(\bar{x}) = \text{sign}\left(\sum_{i=1}^{n}\alpha_i y_i k(x_i, x) + b\right).$$

Support Vectors

There's a nice interpretation of the dual problem in terms of support vectors.

For the primal formulation we know that only support vectors satisfy the constraint with equality:

$$y_i(\bar{w}^T\phi(\bar{x}_i) + b) = 1$$

In the dual, these same training inputs can be identified as their corresponding dual values satisfy $\alpha_i > 0$ (all other inputs have $\alpha_i = 0$).

For test time you only need to compute the sum in $h(\bar{x})$ over the support vectors and all inputs $\bar{x}_i$ with $\alpha_i = 0$ can be discarded after training.

Recovering b

One apparent problem with the dual version is that b is no longer a part of the optimization.

However, we need it to perform classification.

Luckily, we know that the primal solution and the dual solution are identical. In the dual, support vectors are those with $\alpha_i > 0$.

We can then solve the following equations for b

$$y_i(w^T \phi(x_i) + b) = 1$$

$$y_i\left(\sum_j y_j \alpha_j k(x_j, x_i) + b\right) = 1$$

$$y_i - \sum_j y_j \alpha_j k(x_j, x_i) = b$$

This allows us to solve for b from the support vectors.

Kernel SVM - The Smart Nearest Neighbor

In the k-NN algorithm, for binary classification problems ($y_i \in \{-1, +1\}$), we can write the decision function for a test point $\vec{z}$ as

$$h(\vec{z}) = \text{Sign}\left(\sum_{i=1}^{n} y_i \delta^{nn}(\vec{x}_i, \vec{z})\right)$$

where $\delta^{nn}(\vec{z}, \vec{x}_i) \in \{0, 1\}$ with $\delta^{nn}(\vec{z}, \vec{x}_i) = 1$ only if x_i is one of the k-nearest neighbors of test point $\vec{z}$.

The SVM decision function

$$h(\vec{z}) = \text{Sign}\left(\sum_{i=1}^{n} y_i \alpha_i k(x_i, z) + b\right)$$

is very similar, but instead of limiting the decision to the k-NN, it

considers ALL training points but the kernel function assigns more weight to those that are closer (large $k(z, x_i)$).

In some sense, you can view the RBF kernel as a soft nearest neighbour assessment, as the exponential decay w/ distance will assign almost no weight to all but the neighboring points of $\vec{z}$.

The kernel SVM algorithm also learns a weight $\alpha_i > 0$ for each training point and a bias b and it essentially "removes" useless training points by setting many $\alpha_i = 0$.