

Making Linear Models Nonlinear via Kernel Methods

Piyush Rai

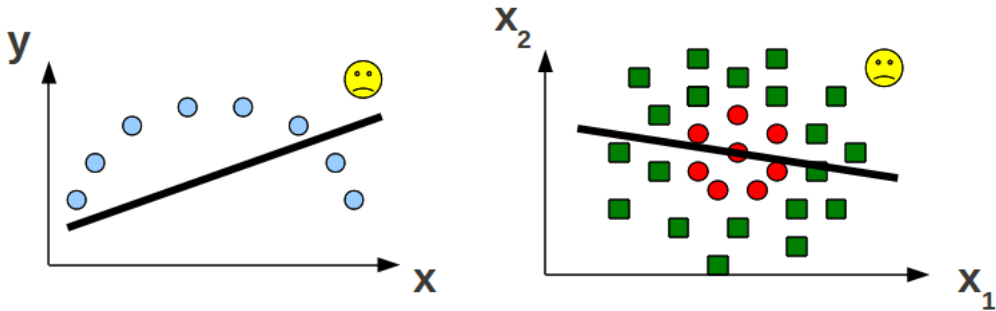
Introduction to Machine Learning (CS771A)

September 6, 2018



Linear Models

- Nice and interpretable but can't learn “difficult” nonlinear patterns



- So, are linear models useless for such problems?

Linear Models for Nonlinear Problems!

- Consider the following one-dimensional inputs from two classes

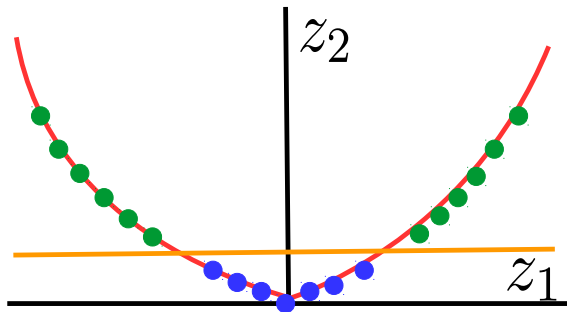


- Can't separate using a linear hyperplane



Linear Models for Nonlinear Problems!

- Consider mapping each x to two-dimensions as $x \rightarrow \mathbf{z} = [z_1, z_2] = [x, x^2]$

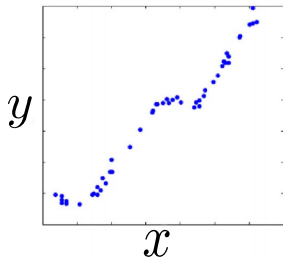


- Data now becomes linearly separable in the two-dimensional space



Linear Models for Nonlinear Problems!

- Consider this regression problem with one-dimensional inputs

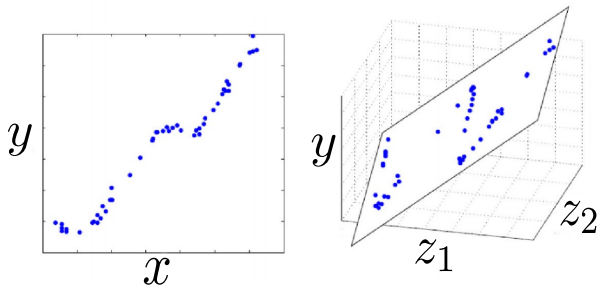


- Linear regression won't work well



Linear Models for Nonlinear Problems!

- Consider mapping each x to two-dimensions as $x \rightarrow \mathbf{z} = [z_1, z_2] = [x, \cos(x)]$



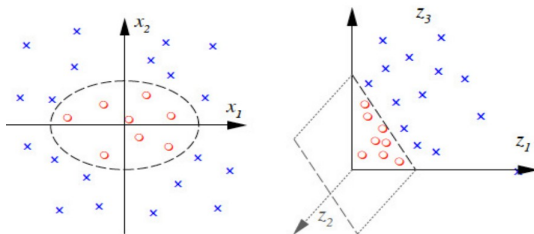
- Now we can fit a linear regression model in two-dimensional input space

Linear Models for Nonlinear Problems!

- Essentially, can use some function ϕ to map/transform inputs to a “nice” space

$$\phi : \mathbb{R}^2 \rightarrow \mathbb{R}^3$$

$$(x_1, x_2) \mapsto (z_1, z_2, z_3) := (x_1^2, \sqrt{2}x_1x_2, x_2^2)$$

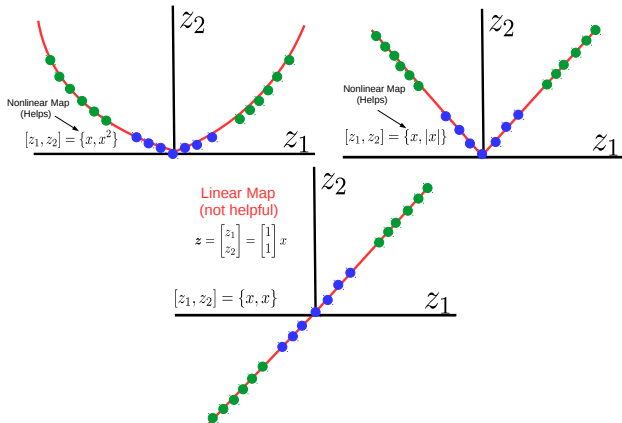


- .. and then happily apply a linear model in the new space!
- Linear in the new space but nonlinear in the original space!



Not Every Mapping is Helpful

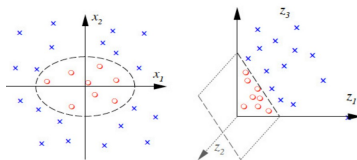
- Not every mapping helps in learning nonlinear patterns. Must at least be nonlinear!
- For the nonlinear classification problem we saw earlier, consider some possible mappings



How to get these “good” (nonlinear) mappings?

- Can try to **learn the mapping** from the data itself (e.g., using deep learning - later)
- There are also pre-defined “good” mappings (e.g., provided by kernel functions - today’s topic)

$$\phi : \mathbb{R}^2 \rightarrow \mathbb{R}^3$$
$$(x_1, x_2) \mapsto (z_1, z_2, z_3) := (x_1^2, \sqrt{2}x_1x_2, x_2^2)$$



- Looks like I have to compute these mapping using ϕ . That would be **quite expensive**!
- Thankfully, not always. For example, when using kernels, you get these for **(almost) free**
 - A kernel defines an “**implicit**” mapping for the data



Kernels as (Implicit) Feature Maps

- Consider two data points $\mathbf{x} = \{x_1, x_2\}$ and $\mathbf{z} = \{z_1, z_2\}$ (each in 2 dims)
- Suppose we have a **function** k which takes as inputs \mathbf{x} and \mathbf{y} and computes

$$\begin{aligned}k(\mathbf{x}, \mathbf{z}) &= (\mathbf{x}^\top \mathbf{z})^2 \\&= (x_1 z_1 + x_2 z_2)^2 \\&= x_1^2 z_1^2 + x_2^2 z_2^2 + 2x_1 x_2 z_1 z_2 \\&= (x_1^2, \sqrt{2}x_1 x_2, x_2^2)^\top (z_1^2, \sqrt{2}z_1 z_2, z_2^2) \\&= \phi(\mathbf{x})^\top \phi(\mathbf{z}) \quad (\text{an inner product})\end{aligned}$$

- k (known as “kernel function”) **implicitly** defines a mapping ϕ to a higher-dim space

$$\phi(\mathbf{x}) = \{x_1^2, \sqrt{2}x_1 x_2, x_2^2\}$$

.. and computes **inner-product based similarity** $\phi(\mathbf{x})^\top \phi(\mathbf{z})$ in that space

- We didn't need to pre-define/compute the mapping ϕ to compute $k(\mathbf{x}, \mathbf{z})$
- We can simply use the definition of the kernel - $(\mathbf{x}^\top \mathbf{z})^2$ in this case
- Also, evaluating $k(\mathbf{x}, \mathbf{z}) = \phi(\mathbf{x})^\top \phi(\mathbf{z})$ is almost **as fast as computing the inner product** $\mathbf{x}^\top \mathbf{z}$



Kernel Functions

- Every kernel function k implicitly defines a **feature mapping** ϕ
- ϕ takes input $\mathbf{x} \in \mathcal{X}$ (input space) and maps it to \mathcal{F} (new “feature space”)
- The kernel function k can be seen as taking **two points as inputs** and computing their inner-product based **similarity** in the \mathcal{F} space

$$\phi : \mathcal{X} \rightarrow \mathcal{F}$$

$$k : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}, \quad k(\mathbf{x}, \mathbf{z}) = \phi(\mathbf{x})^\top \phi(\mathbf{z})$$

- \mathcal{F} needs to be a *vector space* with a *dot product* defined on it
 - Also called a *Hilbert Space*
- Is *any* function k with $k(\mathbf{x}, \mathbf{z}) = \phi(\mathbf{x})^\top \phi(\mathbf{z})$ for some ϕ , a kernel function?
 - No. The function k must satisfy **Mercer's Condition**



Kernel Functions

- For k to be a kernel function
 - k must define a dot product for some Hilbert Space \mathcal{F}
 - Above is true if k is symmetric and **positive semi-definite (p.s.d.) function** (though there are exceptions; there are also “indefinite” kernels).
 - The function k is p.s.d. if the following holds

$$\int \int f(\mathbf{x}) k(\mathbf{x}, \mathbf{z}) f(\mathbf{z}) d\mathbf{x} d\mathbf{z} \geq 0 \quad (\forall f \in L_2)$$

.. for all functions f that are “square integrable”, i.e., $\int f(\mathbf{x})^2 d\mathbf{x} < \infty$

- This is the Mercer’s Condition
- Let k_1, k_2 be two kernel functions then the following are as well:
 - $k(\mathbf{x}, \mathbf{z}) = k_1(\mathbf{x}, \mathbf{z}) + k_2(\mathbf{x}, \mathbf{z})$: direct sum
 - $k(\mathbf{x}, \mathbf{z}) = \alpha k_1(\mathbf{x}, \mathbf{z})$: scalar product
 - $k(\mathbf{x}, \mathbf{z}) = k_1(\mathbf{x}, \mathbf{z}) k_2(\mathbf{x}, \mathbf{z})$: direct product
 - Kernels can also be constructed by composing these rules



Some Examples of Kernel Functions

- Linear (trivial) Kernel:

$$k(\mathbf{x}, \mathbf{z}) = \mathbf{x}^\top \mathbf{z} \text{ (mapping function } \phi \text{ is identity)}$$

- Quadratic Kernel:

$$k(\mathbf{x}, \mathbf{z}) = (\mathbf{x}^\top \mathbf{z})^2 \quad \text{or} \quad (1 + \mathbf{x}^\top \mathbf{z})^2$$

- Polynomial Kernel (of degree d):

$$k(\mathbf{x}, \mathbf{z}) = (\mathbf{x}^\top \mathbf{z})^d \quad \text{or} \quad (1 + \mathbf{x}^\top \mathbf{z})^d$$

- Radial Basis Function (RBF) of “Gaussian” Kernel:

$$k(\mathbf{x}, \mathbf{z}) = \exp[-\gamma \|\mathbf{x} - \mathbf{z}\|^2]$$

- γ is a hyperparameter (also called the **kernel bandwidth**)
- The RBF kernel corresponds to an **infinite dimensional** feature space \mathcal{F} (i.e., you can't actually write down or store the map $\phi(\mathbf{x})$ explicitly)
- Also called **“stationary kernel”**: only depends on the distance between \mathbf{x} and \mathbf{z} (translating both by the same amount won't change the value of $k(\mathbf{x}, \mathbf{z})$)
- Kernel hyperparameters (e.g., d, γ) need to be chosen via cross-validation



RBF Kernel = Infinite Dimensional Mapping

- We saw that the RBF/Gaussian kernel is defined as

$$k(\mathbf{x}, \mathbf{z}) = \exp[-\gamma \|\mathbf{x} - \mathbf{z}\|^2]$$

- Using this kernel corresponds to mapping data to infinite dimensional space
- This is explained below (assume \mathbf{x} and \mathbf{z} to be scalar and $\gamma = 1$):

$$\begin{aligned} k(x, z) &= \exp[-(x - z)^2] \\ &= \exp(-x^2) \exp(-z^2) \exp(2xz) \\ &= \exp(-x^2) \exp(-z^2) \sum_{k=1}^{\infty} \frac{2^k x^k z^k}{k!} \\ &= \phi(x)^\top \phi(z) \quad (\text{the constants } 2^k \text{ and } k! \text{ are subsumed in } \phi) \end{aligned}$$

- This shows that $\phi(x)$ and $\phi(z)$ are infinite dimensional vectors
 - But we didn't have to compute these to get $k(x, z)$

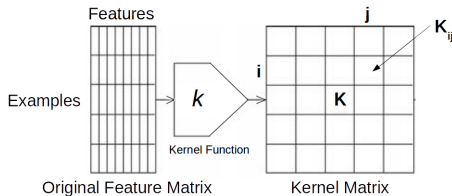


The Kernel Matrix

- The kernel function k defines the **Kernel Matrix** \mathbf{K} over the data
- Given N examples $\{\mathbf{x}_1, \dots, \mathbf{x}_N\}$, the (i, j) -th entry of \mathbf{K} is defined as:

$$K_{ij} = k(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i)^\top \phi(\mathbf{x}_j)$$

- K_{ij} : Similarity between the i -th and j -th example in the feature space \mathcal{F}
- \mathbf{K} : $N \times N$ matrix of **pairwise similarities** between examples in \mathcal{F} space



- \mathbf{K} is a symmetric and **positive definite matrix**
- For a P.D. matrix: $\mathbf{z}^\top \mathbf{K} \mathbf{z} > 0$, $\forall \mathbf{z} \in \mathbb{R}^N$ (also, all eigenvalues positive)
- The Kernel Matrix \mathbf{K} is also known as the **Gram Matrix**



Using Kernels

- Kernels can turn a linear model into a nonlinear one
- Recall: Kernel $k(\mathbf{x}, \mathbf{z})$ represents a dot product in some high dimensional feature space \mathcal{F}
- Any learning model in which, during training and test, **inputs only appear as dot products** ($\mathbf{x}_i^\top \mathbf{x}_j$) can be **kernelized** (i.e., non-linearized)
 - .. by replacing the $\mathbf{x}_i^\top \mathbf{x}_j$ terms by $\phi(\mathbf{x}_i)^\top \phi(\mathbf{x}_j) = k(\mathbf{x}_i, \mathbf{x}_j)$
- Most learning algorithms can be easily kernelized
 - Distance based methods, Perceptron, SVM, linear regression, etc.
 - Many of the unsupervised learning algorithms too can be kernelized (e.g., K -means clustering, Principal Component Analysis, etc. - will see later)
 - Let's look at two examples: Kernelized SVM and Kernelized Ridge Regression



Example 1:

Kernel (Nonlinear) SVM



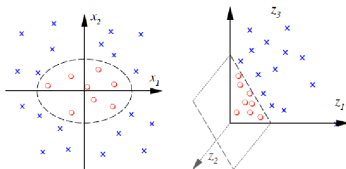
Kernelized SVM Training

- Recall the soft-margin SVM dual problem:

$$\text{Soft-Margin SVM: } \max_{\alpha \leq C} \mathcal{L}_D(\alpha) = \alpha^\top \mathbf{1} - \frac{1}{2} \alpha^\top \mathbf{G} \alpha$$

.. where we had defined $G_{mn} = y_m y_n \mathbf{x}_m^\top \mathbf{x}_n$

- Can simply replace $G_{mn} = y_m y_n \mathbf{x}_m^\top \mathbf{x}_n$ by $y_m y_n K_{mn}$
 - .. where $K_{mn} = k(\mathbf{x}_m, \mathbf{x}_n) = \phi(\mathbf{x}_m)^\top \phi(\mathbf{x}_n)$ for a suitable kernel function k
- The problem can be solved just like the linear SVM case
- The new SVM learns a **linear separator** in kernel-induced feature space \mathcal{F}
 - This corresponds to a **non-linear separator in the original space \mathcal{X}**



Kernelized SVM Prediction

- Note that the SVM weight vector for the kernelized case can be written as

$$\mathbf{w} = \sum_{n=1}^N \alpha_n y_n \phi(\mathbf{x}_n)$$

- Prediction for a new test example \mathbf{x} (assume $b = 0$)

$$y = \text{sign}(\mathbf{w}^\top \phi(\mathbf{x})) = \text{sign}\left(\sum_{n=1}^N \alpha_n y_n \phi(\mathbf{x}_n)^\top \phi(\mathbf{x})\right) = \text{sign}\left(\sum_{n=1}^N \alpha_n y_n k(\mathbf{x}_n, \mathbf{x})\right)$$

- Note: \mathbf{w} can be stored explicitly as a vector only if the feature map $\phi(\cdot)$ can be explicitly written
- In general, kernelized SVMs have to store the training data (at least the support vectors for which α_n 's are nonzero) even at the test time
- Thus the prediction time cost of kernel SVM scales linearly in N
- For unkernelized version $\mathbf{w} = \sum_{n=1}^N \alpha_n y_n \mathbf{x}_n$ can be computed and stored as a $D \times 1$ vector. Thus training data need not be stored and the prediction cost is constant w.r.t. N ($\mathbf{w}^\top \mathbf{x}$ can be computed in $O(D)$ time).



Example 2:

Kernel (Nonlinear) Ridge Regression



Ridge Regression: Revisited

- Recall the ridge regression problem

$$\mathbf{w} = \arg \min_{\mathbf{w}} \sum_{n=1}^N (y_n - \mathbf{w}^\top \mathbf{x}_n)^2 + \lambda \mathbf{w}^\top \mathbf{w}$$

- The solution to this problem was

$$\mathbf{w} = \left(\sum_{n=1}^N \mathbf{x}_n \mathbf{x}_n^\top + \lambda \mathbf{I}_D \right) \left(\sum_{n=1}^N y_n \mathbf{x}_n \right) = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}_D)^{-1} \mathbf{X}^\top \mathbf{y}$$

- Inputs don't appear as inner-products here . They actually do! :-)
- Matrix inversion lemma: $(\mathbf{F}\mathbf{H}^{-1}\mathbf{G} - \mathbf{E})^{-1}\mathbf{F}\mathbf{H}^{-1} = \mathbf{E}^{-1}\mathbf{F}(\mathbf{G}\mathbf{E}^{-1}\mathbf{F} - \mathbf{H})^{-1}$
- The lemma allows us to rewrite \mathbf{w} as

$$\mathbf{w} = \mathbf{X}^\top (\mathbf{X}\mathbf{X}^\top + \lambda \mathbf{I}_N)^{-1} \mathbf{y} = \mathbf{X}^\top \boldsymbol{\alpha} = \sum_{n=1}^N \alpha_n \mathbf{x}_n$$

where $\boldsymbol{\alpha} = (\mathbf{X}\mathbf{X}^\top + \lambda \mathbf{I}_N)^{-1} \mathbf{y} = (\mathbf{K} + \lambda \mathbf{I}_N)^{-1} \mathbf{y}$ is an $N \times 1$ vector of dual variables, and $K_{nm} = \mathbf{x}_n^\top \mathbf{x}_m$

- Note: $\mathbf{w} = \sum_{n=1}^N \alpha_n \mathbf{x}_n$ is known as “dual” form of ridge regression solution. However, so far it is still a linear model. But now it is easily kernelizable.



Kernel (Nonlinear) Ridge Regression

- With the dual form $\mathbf{w} = \sum_{n=1}^N \alpha_n \mathbf{x}_n$, we can kernelize ridge regression
- Choosing some kernel k with an associated feature map ϕ , we can write

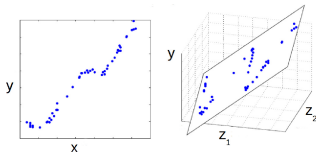
$$\mathbf{w} = \sum_{n=1}^N \alpha_n \phi(\mathbf{x}_n) = \sum_{n=1}^N \alpha_n k(\mathbf{x}_n, \cdot)$$

where $\alpha = (\mathbf{K} + \lambda \mathbf{I}_N)^{-1} \mathbf{y}$ and $K_{nm} = \phi(\mathbf{x}_n)^\top \phi(\mathbf{x}_m) = k(\mathbf{x}_n, \mathbf{x}_m)$

- Prediction for a new test input \mathbf{x} will be

$$y = \mathbf{w}^\top \phi(\mathbf{x}) = \sum_{n=1}^N \alpha_n \phi(\mathbf{x}_n)^\top \phi(\mathbf{x}) = \sum_{n=1}^N \alpha_n k(\mathbf{x}_n, \mathbf{x})$$

- Thus, using the kernel, we effectively learn a nonlinear regression model



- Note: Just as in kernel SVM, prediction cost scales in N



Learning with Kernels: Some Aspects

- Choice of the right kernel is important
- Some kernels (e.g., RBF) work well for many problems but hyperparameters of the kernel function may need to be tuned via cross-validation
- There is a huge literature on **learning the right kernel** from data
 - Learning a combination of multiple kernels (**Multiple Kernel Learning**)
 - **Bayesian kernel methods** (e.g., Gaussian Processes) can learn the kernel hyperparameters from data (thus can be seen as learning the kernel)
- Various other alternatives to learn the “right” data representation
 - **Adaptive Basis Functions** (learn **basis functions** and **weights** from data)

$$f(\mathbf{x}) = \sum_{m=1}^M w_m \phi_m(\mathbf{x})$$

.. various methods can be seen as learning adaptive basis functions, e.g., (Deep) Neural Networks, mixture of experts, Decision Trees, etc.

- Kernel methods use a **“fixed” set of basis functions or “landmarks”**. The basis functions are the training **data points themselves**; also see the next slide.



Kernels: Viewed as Defining Fixed Basis Functions

- Consider each row (or column) of the $N \times N$ kernel matrix (it's symmetric)

$$K = \begin{bmatrix} k(\mathbf{x}_1, \mathbf{x}_1) & k(\mathbf{x}_1, \mathbf{x}_2) & \cdots & k(\mathbf{x}_1, \mathbf{x}_n) \\ k(\mathbf{x}_2, \mathbf{x}_1) & k(\mathbf{x}_2, \mathbf{x}_2) & \cdots & k(\mathbf{x}_2, \mathbf{x}_n) \\ \vdots & \vdots & \ddots & \vdots \\ k(\mathbf{x}_n, \mathbf{x}_1) & k(\mathbf{x}_n, \mathbf{x}_2) & \cdots & k(\mathbf{x}_n, \mathbf{x}_n) \end{bmatrix}$$

- For each input \mathbf{x}_n , we can define the following N dimensional vector

$$K(n, :) = [k(\mathbf{x}_n, \mathbf{x}_1) \ k(\mathbf{x}_n, \mathbf{x}_2) \ \dots \ k(\mathbf{x}_n, \mathbf{x}_N)]$$

- Can think of this as a **new feature vector** (with N features) for inputs \mathbf{x}_n . Each feature represents the similarity of \mathbf{x}_n with one of the inputs.
- Thus these new features are basically defined in terms of similarities of each input with a **fixed set of basis points** or “landmarks” $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N$
- In general, the set of basis points or landmarks can be **any set of points** (not necessarily the data points) and **can even be learned** (which is what Adaptive Basis Function methods basically do).

Learning with Kernels: Some Aspects (Contd.)

- Storage/computational efficiency can be a bottleneck when using kernels
- Need to store the kernel matrix even at test time. This can be an issue (remember that \mathbf{K} is an $N \times N$ matrix which can be very massive)
- Test time prediction can be slow because to make a prediction, we need to compute $\sum_{n=1}^N \alpha_n k(\mathbf{x}_n, \mathbf{x})$. Can be made faster if very few α_n 's are nonzero.
- There is a huge literature on **speeding up kernel methods**
 - Approximating the kernel matrix using a set of kernel-derived new features
 - Identifying a small set of landmark points in the training data
 - .. and a lot more



Kernels: Concluding Notes

- Kernels give a modular way to learn nonlinear patterns using linear models
 - All you need to do is replace the inner products with the kernel
- All the computations remain as efficient as in the original space
- A very general notion of similarity: Can define similarities between objects even though they can't be represented as vectors. Many kernels are tailor-made for specific types of data
 - Strings (string kernels): DNA matching, text classification, etc.
 - Trees (tree kernels): Comparing parse trees of phrases/sentences

