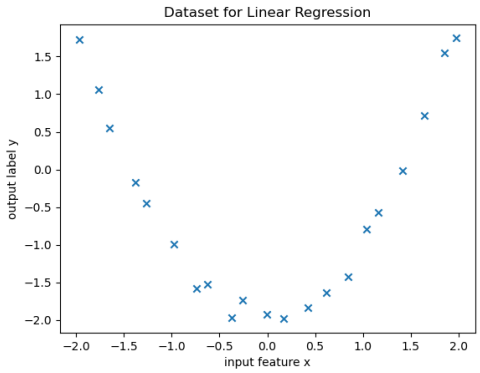
Kernels

Cornell CS 4/5780 — Spring 2022

Handcrafted Feature Expansion. Linear classifiers are great, but what if there exists no linear decision boundary? As it turns out, there is an elegant way to incorporate non-linearities into most linear classifiers. We can make linear classifiers non-linear by applying basis function (feature transformations) on the input feature vectors. Formally, for a data vector $\mathbf{x} \in \mathbb{R}^d$, we apply the transformation $\mathbf{x} \to \phi(\mathbf{x})$ where $\phi(\mathbf{x}) \in \mathbb{R}^D$. Usually $D \gg d$ because we add dimensions that capture non-linear interactions among the original features.

Question: In the figure to the right, what would be a good transformation to use? What would be a bad transformation?



Advantage: It is simple, and your problem stays convex and well behaved. (i.e. you can still use your original gradient descent code, just with the higher dimensional representation)

Disadvantage: $\phi(\mathbf{x})$ might be very high dimensional.

Consider the following example: $\mathbf{x} = [x_1 \quad x_2 \quad \cdots \quad x_d]^T$, and define $\phi(\mathbf{x}) = [1 \quad x_1 \quad \cdots \quad x_d \quad x_1x_2 \quad \cdots \quad x_{d-1}x_d \quad \cdots \quad x_1x_2\cdots x_d]^T$.

Quiz: What is the dimensionality of $\phi(\mathbf{x})$?

This new representation, $\phi(\mathbf{x})$, is very expressive and allows for complicated non-linear decision boundaries - but the dimensionality is extremely high. This makes our algorithm unbearable (and quickly prohibitively) 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, without ever computing a single vector $\phi(\mathbf{x})$ or ever computing the full vector $\mathbf{w}$. It is based on the following observation: If we use gradient descent with any one of our standard loss functions, the gradient is a linear combination of the input samples. For example, let us take a look at the squared loss: $\ell(\mathbf{w}) = \sum_{i=1}^n (\mathbf{w}^\top \mathbf{x}_i - y_i)^2$. The gradient descent rule, with step-size/learning-rate $s > 0$ (we denoted this as $\alpha > 0$ in our previous lectures), updates $\mathbf{w}$ over time,

$$w_{t+1} \leftarrow w_t - s(\frac{\partial \ell}{\partial \mathbf{w}}) \quad \text{where: } \frac{\partial \ell}{\partial \mathbf{w}} = \sum_{i=1}^n \underbrace{2(\mathbf{w}^\top \mathbf{x}_i - y_i)}_{\gamma_i \text{ : function of } \mathbf{x}_i, y_i} \mathbf{x}_i = \sum_{i=1}^n \gamma_i \mathbf{x}_i$$

We will now show that we can express $\mathbf{w}$ as a linear combination of all input vectors,

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

Since the loss is convex, the final solution is independent of the initialization, and we can initialize $\mathbf{w}^0$ to be whatever we want. For convenience, let us pick $\mathbf{w}_0 = 0$ the all-0 vector. For this initial choice of $\mathbf{w}_0$, the linear combination in $\mathbf{w} = \sum_{i=1}^n \alpha_i \mathbf{x}_i$ is trivially $\alpha_1 = \cdots = \alpha_n = 0$. 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. Formally, the argument is by induction. $\mathbf{w}$ is trivially a linear combination of our training vectors for $\mathbf{w}_0$ (base case). If we apply the inductive hypothesis for $\mathbf{w}_t$ it follows for $\mathbf{w}_{t+1}$.

The update-rule for α_i^t is thus $\alpha_i^t = \alpha_i^{t-1} - s\gamma_i^{t-1}$, and we have $\alpha_i^t = -s\sum_{r=0}^{t-1} \gamma_i^r$. In other words, we can perform the entire gradient descent update rule without ever expressing $\mathbf{w}$ explicitly. We just keep track of the n coefficients $\alpha_1, \ldots, \alpha_n$. Now that $\mathbf{w}$ can be written as a linear combination of the training set, we can also express the inner-product of $\mathbf{w}$ with any input $\mathbf{x}_i$ purely in terms of inner-products between training inputs:

$$\mathbf{w}^\top \mathbf{x}_j = \sum_{i=1}^n \alpha_i \mathbf{x}_i^\top \mathbf{x}_j.$$

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

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

Quiz: How would you write the test-time classifier in terms of these α parameters?

Inner-Product Computation

Let's go back to the previous example, of feature map ϕ consisting of polynomials of degree-up-to-d in x and degree at most 1 in each x_i. The inner product $\phi(\mathbf{x})^\top \phi(\mathbf{z})$ can be formulated as:

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

The sum of 2^d terms becomes the product of d terms. We can compute the inner-product from the above formula in time $O(d)$ instead of $O(2^d)$! We define the function

$$\underbrace{\mathsf{k}(\mathbf{x}_i, \mathbf{x}_j)}_{\text{this is called the } \textbf{kernel function}} = \phi(\mathbf{x}_i)^\top \phi(\mathbf{x}_j).$$

With a finite training set of n samples, inner products are often pre-computed and stored in a Kernel Matrix (a.k.a. Gram Matrix):

$$\mathsf{K}_{ij} = \phi(\mathbf{x}_i)^\top \phi(\mathbf{x}_j).$$

If we store the matrix K, we only need to do simple inner-product look-ups and low-dimensional computations throughout the gradient descent algorithm. During training in the new high dimensional space of $\phi(\mathbf{x})$ we want to compute α_i through kernels, without ever computing any $\phi(\mathbf{x}_i)$ or even $\mathbf{w}$.

How does this affect the computational cost of our linear regressor?

General Kernels

Below are some popular kernel functions:

Linear: $\mathsf{K}(\mathbf{x}, \mathbf{z}) = \mathbf{x}^\top \mathbf{z}$. (The linear kernel is equivalent to just using a good old linear classifier - but it can be faster to use a kernel matrix if the dimensionality d of the data is high.)

Polynomial: $\mathsf{K}(\mathbf{x}, \mathbf{z}) = (1 + \mathbf{x}^\top \mathbf{z})^d$.

Radial Basis Function (RBF) (aka Gaussian Kernel): $\mathsf{K}(\mathbf{x}, \mathbf{z}) = e^{\frac{-\|\mathbf{x}-\mathbf{z}\|^2}{\sigma^2}}$. The RBF kernel 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 (see this paper).

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

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

Sigmoid Kernel: $\mathsf{K}(\mathbf{x}, \mathbf{z}) = \tanh(\mathbf{a}\mathbf{x}^\top + c)$

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