Kernels
Cornell CS 4/5780 (Spring 2023)

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.

**Handcrafted Feature Expansion**

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 \( \mathbf{x} \rightarrow \phi(\mathbf{x}) \) 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.

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: \( \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix} \), and define \( \phi(\mathbf{x}) = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_d \\ x_1 x_2 \\ \vdots \\ x_{d-1} x_d \\ x_1 x_2 \cdots x_d \end{bmatrix} \).

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 a little magical.

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:

\[
t(\mathbf{w}) = \sum_{i=1}^{n} (\mathbf{w}^\top \mathbf{x}_i - y_i)^2
\]
The gradient descent rule, with step-size/learning-rate $s > 0$ (we denoted this as $\alpha > 0$ in our previous lectures), updates $w$ over time,

$$w_{t+1} \leftarrow w_t - s \frac{\partial \ell}{\partial w}$$

where:

$$\frac{\partial \ell}{\partial w} = \sum_{i=1}^{n} 2(y_i - w^\top x_i) x_i = \sum_{i=1}^{n} \gamma_i x_i$$

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

$$w = \sum_{i=1}^{n} \alpha_i x_i.$$ 

Since the loss is convex, the final solution is independent of the initialization, and we can initialize $w^0$ to be whatever we want. For convenience, let us pick $w_0 = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}$. For this initial choice of $w_0$, the linear combination in

$$w = \sum_{i=1}^{n} \alpha_i x_i$$

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 $\alpha_i$ coefficients:

$$\begin{align*}
\alpha_i &= \alpha_i^{t-1} - s \gamma_i^{t-1} \\
\gamma_i &= \gamma_i^{t-1} - s \sum_{r=0}^{t-1} \gamma_r.
\end{align*}$$

Formally, the argument is by induction. $w$ is trivially a linear combination of our training vectors for $w_0$ (base case). If we apply the inductive hypothesis for $w_t$ it follows for $w_{t+1}$.

The update-rule for $\alpha_i^t$ is thus

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

In other words, we can perform the entire gradient descent update rule without ever expressing $w$ explicitly. We just keep track of the $n$ coefficients $\alpha_1, \ldots, \alpha_n$.

Now that $w$ can be written as a linear combination of the training set, we can also express the inner-product of $w$ with any input $x_i$ purely in terms of inner-products between training inputs:

$$w^\top x_j = \sum_{i=1}^{n} \alpha_i x_i^\top x_j.$$ 

Consequently, we can also re-write the squared-loss from

$$\ell(w) = \sum_{i=1}^{n} (w^\top x_i - y_i)^2$$

entirely in terms of inner-products between training inputs:

$$\ell(\alpha) = \sum_{i=1}^{n} \left( \sum_{j=1}^{n} \alpha_j x_j^\top x_i - y_i \right)^2$$

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

$$h(x_t) = w^\top x_t = \sum_{j=1}^{n} \alpha_j x_j^\top x_t.$$ 

Do you notice a theme? The only information we ever need in order to learn a hyper-plane classifier with the squared-loss is inner-products between all pairs of data vectors.
Inner-Product Computation

Let’s go back to the previous example, \( \phi(x) = \begin{pmatrix} 1 \\ x_1 \\ \vdots \\ x_d \\ x_1x_2 \\ \vdots \\ x_{d-1}x_d \\ \vdots \\ x_1x_2 \cdots x_d \end{pmatrix} \).

The inner product \( \phi(x)^\top \phi(z) \) can be formulated as:

\[
\phi(x)^\top \phi(z) = 1 + x_1z_1 + x_2z_2 + \cdots + x_1x_2z_1z_2 + \cdots + x_1 \cdots x_dz_1 \cdots z_d = \prod_{k=1}^d (1 + :)
\]

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

\[
k(x_i, x_j) = \phi(x_i)^\top \phi(x_j).
\]

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

\[
K_{ij} = \phi(x_i)^\top \phi(x_j).
\]

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. The final classifier becomes:

\[
h(x_t) = \sum_{j=1}^n \alpha_j k(x_j, x_t).
\]

During training in the new high dimensional space of \( \phi(x) \) we want to compute \( \gamma_t \) through kernels, without ever computing any \( \phi(x_i) \) or even \( w \). We previously established that \( w = \sum_{j=1}^n \alpha_j \phi(x_j) \), and \( \gamma_t = 2(w^\top \phi(x_t) - y_t) \). It follows that \( \gamma_t = 2(\sum_{j=1}^n \alpha_j K_{ij} - y_t) \). The gradient update in iteration \( t+1 \) becomes

\[
\alpha_t^{t+1} \leftarrow \alpha_t^t - 2s(\sum_{j=1}^n \alpha_j^t K_{ij}) - y_t.
\]

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

Below are some popular kernel functions:

**Linear:** \( K(x, z) = x^\top z \).

(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:** \( K(x, z) = (1 + x^\top z)^d \).

**Radial Basis Function (RBF) (aka Gaussian Kernel):** \( K(x, z) = e^{-\frac{\|x-z\|^2}{\sigma^2}} \).
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:** \( K(x, z) = e^{-\frac{\|x-z\|^2}{2\sigma^2}} \)

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

**Sigmoid Kernel:** \( K(x, z) = \tanh(ax^T + c) \)

Kernel functions

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

No, the matrix \( K(x_i, x_j) \) has to correspond to real inner-products after some transformation \( x \rightarrow \phi(x) \). 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 \( \forall q \in \mathbb{R}^n, q^T A q \geq 0 \).

Remember \( K_{ij} = \phi(x_i)^T \phi(x_j) \). So \( K = \Phi^T \Phi \), where \( \Phi = [\phi(x_1), \ldots, \phi(x_n)] \). It follows that \( K \) is p.s.d., because \( q^T K q = (\Phi^T q)^T \Phi q \geq 0 \). Inversely, if any matrix \( A \) is p.s.d., it can be decomposed as \( A = \Phi^T \Phi \) for some realization of \( \Phi \).

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

![Figure 1](#)