Contents

1 Handcrafted Feature Expansion

We can make linear classifiers non-linear by applying basis function (feature transformations) on the input feature vectors. Formally, for a feature vector $\vec{X} \in \mathbb{R}^d$ (in terms of tabular data, this is a row with d features), we apply the transformation $\vec{X} \to \phi(\vec{X})$ where $\phi(\vec{X}) \in \mathbb{R}^D$. Usually $D \gg d$ because we add dimensions that capture non-linear interactions among the original features.

The advantage of expanding the feature space is that it is simple, and our problem stays convex and well behaved, i.e., we can still use our original gradient descent code, just with the higher dimensional representation. The disadvantage is that $\phi(\vec{X})$ might be very high dimensional. Consider the following:

$$\vec{X} = \begin{pmatrix} X_1 \\ X_2 \\ \vdots \\ X_d \end{pmatrix} \quad \longrightarrow \quad \phi(\vec{X}) = \begin{pmatrix} 1 \\ X_1 \\ \vdots \\ X_d \\ X_1X_2 \\ X_1X_3 \\ \vdots \\ X_{d-1}X_d \\ \vdots \\ X_1X_2\cdots X_d \end{pmatrix}$$

In this examples, the dimensionality of the feature vector $\phi(\vec{X})$ is 2^d. Each of the d feature values in the original feature vector $\vec{X}$ can either be multiplied by another feature value or not, so we have a binary decision. This new representation, $\phi(\vec{X}) \in \mathbb{R}^{2^d}$, is very expressive and allows for complicated non-linear decision boundaries— but the dimensionality is extremely high. This makes our algorithm unbearably (and quickly prohibitively) slow.

2 Kernelization

2.1 Gradient Descent with Squared Loss

The kernel is a technique to get around this computational dilemma by learning a function in the much higher dimensional space, without ever computing a single vector $\phi(\vec{X})$ or ever computing the full vector of coefficients or weights $\boldsymbol{\beta}$ in that high dimensional space. It is based on the following observation— if we use gradient descent with any one of the standard loss functions, the **gradient is a linear combination of the input samples**. For example, let us examine the squared loss:

$$\ell(\boldsymbol{\beta}) = \sum_{i=1}^{n} \left(\boldsymbol{\beta}^\top \vec{X}_i - Y_i\right)^2$$

Note this is a slightly different form compared to what is commonly denoted in other textbooks where the function is adjusted by the scale of $\frac{1}{n}$ or $\frac{1}{n-p}$. The gradient descent rule, with step-size or learning-rate $\eta > 0$, updates the vector $\boldsymbol{\beta}$ over time,

$$\boldsymbol{\beta}_{t+1} \leftarrow \boldsymbol{\beta}_t - \eta \left(\frac{\partial \ell}{\partial \boldsymbol{\beta}} \right) \text{ where: } \frac{\partial \ell}{\partial \boldsymbol{\beta}} = \sum_{i=1}^{n} \underbrace{2\left(\boldsymbol{\beta}^\top \vec{X}_i - Y_i\right)}_{\gamma_i\text{: function of } \vec{X}_i, Y_i} \vec{X}_i = \sum_{i=1}^{n} \gamma_i \vec{X}_i$$

We can express vector $\boldsymbol{\beta}$ as a linear combination of all input vectors,

$$\boldsymbol{\beta} = \sum_{i=1}^{n} \alpha_i \vec{X}_i$$

where $\vec{X}_i = \begin{bmatrix} 1 & X_{i1} & X_{i2} & \dots & X_{i,p-1} \end{bmatrix}^T$ for $1, 2, \dots, i, \dots, n$. Since the loss is convex, the final solution is independent of the initialization, and we can initialize vector $\boldsymbol{\beta}_0$ to be whatever we want. For convenience, we choose vector $\boldsymbol{\beta}_0 = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}$. For this initial choice of vector $\boldsymbol{\beta}_0$, the linear combination equation $\boldsymbol{\beta} = \sum_{i=1}^{n} \alpha_i \vec{X}_i$ is trivially satisfied $\alpha_1 = \cdots = \alpha_n = 0$. It is a homogeneous linear system. We now show that throughout the entire gradient descent optimization such coefficients $\alpha_1, \dots, \alpha_n$ must always exist, and we can re-write the gradient updates entirely in terms of updating the α_i coefficients:

$$\boldsymbol{\beta}_1 = \boldsymbol{\beta}_0 - \eta \sum_{i=1}^{n} 2\left(\boldsymbol{\beta}_0^\top \vec{X}_i - Y_i\right) \vec{X}_i = \sum_{i=1}^{n} \alpha_i^0 \vec{X}_i - \eta \sum_{i=1}^{n} \gamma_i^0 \vec{X}_i = \sum_{i=1}^{n} \underbrace{(\alpha_i^0 - \eta\gamma_i^0)}_{\alpha_i^1} \vec{X}_i = \sum_{i=1}^{n} \alpha_i^1 \vec{X}_i \quad \left(\text{with } \alpha_i^1 = \alpha_i^0 - \eta\gamma_i^0\right)$$

$$\boldsymbol{\beta}_2 = \boldsymbol{\beta}_1 - \eta \sum_{i=1}^{n} 2\left(\boldsymbol{\beta}_1^\top \vec{X}_i - Y_i\right) \vec{X}_i = \sum_{i=1}^{n} \alpha_i^1 \vec{X}_i - \eta \sum_{i=1}^{n} \gamma_i^1 \vec{X}_i = \sum_{i=1}^{n} \underbrace{(\alpha_i^1 - \eta\gamma_i^1)}_{\alpha_i^2} \vec{X}_i = \sum_{i=1}^{n} \alpha_i^2 \vec{X}_i \quad \left(\text{with } \alpha_i^2 = \alpha_i^1 \vec{X}_i - \eta\gamma_i^1\right)$$

$$\boldsymbol{\beta}_3 = \boldsymbol{\beta}_2 - \eta \sum_{i=1}^{n} 2\left(\boldsymbol{\beta}_2^\top \vec{X}_i - Y_i\right) \vec{X}_i = \sum_{i=1}^{n} \alpha_i^2 \vec{X}_i - \eta \sum_{i=1}^{n} \gamma_i^2 \vec{X}_i = \sum_{i=1}^{n} \underbrace{(\alpha_i^2 - \eta\gamma_i^2)}_{\alpha_i^3} \vec{X}_i = \sum_{i=1}^{n} \alpha_i^3 \vec{X}_i \quad \left(\text{with } \alpha_i^3 = \alpha_i^2 - \eta\gamma_i^2\right)$$

$$\vdots \qquad\qquad\qquad\qquad \vdots$$

$$\vdots \qquad\qquad\qquad\qquad \vdots$$

$$\boldsymbol{\beta}_t = \boldsymbol{\beta}_{t-1} - \eta \sum_{i=1}^{n} 2\left(\boldsymbol{\beta}_{t-1}^\top \vec{X}_i - Y_i\right) \vec{X}_i = \sum_{i=1}^{n} \alpha_i^{t-1} \vec{X}_i - \eta \sum_{i=1}^{n} \gamma_i^{t-1} \vec{X}_i = \sum_{i=1}^{n} \underbrace{(\alpha_i^{t-1} - \eta\gamma_i^{t-1})}_{\alpha_i^t} \vec{X}_i = \sum_{i=1}^{n} \alpha_i^t \vec{X}_i$$

Formally, the argument is by induction. The vector $\boldsymbol{\beta}$ is trivially a linear combination of our feature vectors $\vec{X}_i$ for $\boldsymbol{\beta}_0$ (base case). If we apply the inductive hypothesis for $\boldsymbol{\beta}_t$, it follows for $\boldsymbol{\beta}_{t+1}$. Note that $\alpha_i^0 = 0$ since the linear combination equation $\boldsymbol{\beta} = \sum_{i=1}^{n} \alpha_i \vec{X}_i$ is trivially satisfied. The update-rule for α_i^t thus becomes

$$\begin{aligned}
\alpha_i^t &= \alpha_i^{t-1} - \eta\gamma_i^{t-1} \\
&= (\alpha_i^{t-2} - \eta\gamma_i^{t-2}) - \eta\gamma_i^{t-1} \\
&= \big((\alpha_i^{t-3} - \eta\gamma_i^{t-3}) - \eta\gamma_i^{t-2}\big) - \eta\gamma_i^{t-1} \\
&\vdots \\
&= \cancel{\alpha_i^0} - \eta\gamma_i^0 - \eta\gamma_i^1 - \eta\gamma_i^2 - \ldots - \eta\gamma_i^{t-3} - \eta\gamma_i^{t-2} - \eta\gamma_i^{t-1} \\
&= 0 - \eta\sum_{r=0}^{t-1}\gamma_i^r \\
&= -\eta\sum_{r=0}^{t-1}\gamma_i^r
\end{aligned}$$

Because of this, we can perform the entire gradient descent update rule without ever expressing $\boldsymbol{\beta}$ explicitly. We simply need to keep track of the n coefficients $\alpha_1, \ldots, \alpha_n$, which change at each step during training. Now that the vector $\boldsymbol{\beta}$ can be written as a linear combination of the training set, we can also express the inner-product of vector $\boldsymbol{\beta}$ with any input vector $\vec{X}_i$ purely in terms of inner-products between training inputs:

$$h(\vec{X}_j) = \hat{Y}_j = (\boldsymbol{\beta})^\top \vec{X}_j = (\sum_{i=1}^{n} \alpha_i \vec{X}_i)^\top \vec{X}_j = \sum_{i=1}^{n} \alpha_i \vec{X}_i^\top \vec{X}_j$$

To compute $\hat{Y}_j$, we do not actually compute the high dimensional vector $\boldsymbol{\beta}$; instead, we simply find the sum of n inner products between each of the $1, \ldots, n$ training examples and the j^{th} training example. Consequently, we can also re-write the squared-loss from $\ell(\boldsymbol{\beta}) = \sum_{i=1}^{n} \left(\boldsymbol{\beta}^\top \vec{X}_i - Y_i\right)^2$ entirely in terms of inner-product between training inputs:

$$\ell(\alpha) = \sum_{i=1}^{n} \left(\sum_{j=1}^{n} \alpha_j \vec{X}_j^\top \vec{X}_i - Y_i \right)^2$$

The i and j indices are chosen arbitrarily to distinguish the inner sum and the outer sum, and so flipping them would not affect the equation. During testing time, we also only need these α_i coefficients to make a prediction on a test input vector $\vec{X}_t^*$, and we can write the entire classifier in terms of inner-products between the test points and the training points:

$$h\left(\vec{X}_t^*\right) = \hat{Y}_t = \boldsymbol{\beta}^\top \vec{X}_t^* = \sum_{j=1}^{n} \alpha_j \vec{X}_j^\top \vec{X}_t^*$$

Simply put, the only information we ever need in order to learn a hyper-plane classifier with the squared-loss is the inner-products between all pairs of data vectors. Note that, at each step of the gradient descent r, these α_i coefficients depend on γ_i^r, which is a function of training example $\vec{X}_i$ and target value Y_i.

2.2 Inner Product Computation

How are these inner products computed? If we examine the previous handpicked feature expansion vector, the inner product $\vec{X} \cdot \vec{Z} = \vec{X}^\top \vec{Z} \longrightarrow \phi(\vec{X})^\top \phi(\vec{Z})$ can be formulated as:

$$\begin{aligned}
\phi(\vec{X}) \cdot \phi(\vec{Z}) &= \begin{pmatrix} 1 \\ X_1 \\ \vdots \\ X_d \\ X_1X_2 \\ X_1X_3 \\ \vdots \\ X_{d-1}X_d \\ \vdots \\ X_1X_2 \cdots X_d \end{pmatrix} \cdot \begin{pmatrix} 1 \\ Z_1 \\ \vdots \\ Z_d \\ Z_1Z_2 \\ Z_1Z_3 \\ \vdots \\ Z_{d-1}Z_d \\ \vdots \\ Z_1Z_2 \cdots Z_d \end{pmatrix} \\
&= \underbrace{1 \cdot 1 + X_1Z_1 + X_2Z_2 + \cdots + X_1X_2Z_1Z_2 + \cdots + X_{d-1}X_dZ_{d-1}Z_d + \cdots + (X_1X_2 \cdots X_d)(Z_1Z_2 \cdots Z_d)}_{2^d \text{terms}} \\
&= \prod_{k=1}^{d} (1 + X_kZ_k)
\end{aligned}$$

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 $O\left(2^d\right)$. We define the function

$$\underbrace{\mathbf{k}\left(\vec{X}_i, \vec{X}_j\right)}_{\text{kernel function}} = \phi\left(\vec{X}_i\right)^\top \phi\left(\vec{X}_j\right).$$

With a finite training set of n samples, the inner products, **which have been re-formulated as the product of fewer terms rather than the sum of many terms**, are often pre-computed and stored in a Kernel Matrix:

$$\mathrm{K}_{ij} = \phi\left(\vec{X}_i\right)^\top \phi\left(\vec{X}_j\right) \qquad \text{This inner product is a scalar cell in a matrix}$$

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. **This saves us from actually having to map our data vectors to high dimensional vector space and computing the inner products in that space**. The final classifier becomes:

$$\begin{aligned}
h\left(\vec{X}_t^*\right) &= \boldsymbol{\beta}^\top \vec{X}_t^* \\
&= \sum_{j=1}^{n} \alpha_j (\vec{X}_j^\top \vec{X}_t^*) \\
&= \sum_{j=1}^{n} \alpha_j \mathbf{k}\left(\vec{X}_j, \vec{X}_t^*\right)
\end{aligned}$$

The loss function therefore becomes:

$$\ell(\alpha) = \sum_{i=1}^{n} \left(\sum_{j=1}^{n} \alpha_j \mathbf{k} \left(\vec{X}_i, \vec{X}_j \right) - Y_i \right)^2$$

During training in the new high dimensional space of $\phi(\vec{X}) \in \mathbb{R}^D$ we want to compute γ_i through kernels, without ever computing any $\phi\left(\vec{X}_i\right) \in \mathbb{R}^D$ or even vector $\boldsymbol{\beta} \in \mathbb{R}^D$. We previously established that

$$\boldsymbol{\beta} = \sum_{j=1}^{n} \alpha_j \phi\left(\vec{X}_j\right)$$

and

$$\gamma_i = 2\left(\boldsymbol{\beta}^\top \phi\left(\vec{X}_i\right) - Y_i\right)$$

It therefore follows that

$$\begin{aligned} \gamma_i &= 2\left(\boldsymbol{\beta}^\top \phi\left(\vec{X}_i\right) - Y_i\right) \\ &= 2\left(\left[\sum_{j=1}^{n} \alpha_j \phi\left(\vec{X}_j\right) \right]^\top \phi\left(\vec{X}_i\right) - Y_i \right) \\ &= 2\left(\sum_{j=1}^{n} \alpha_j \left[\phi\left(\vec{X}_j\right)^\top \phi\left(\vec{X}_i\right) \right] - Y_i \right) \\ &= 2\left(\sum_{j=1}^{n} \alpha_j \left[K_{ij} \right] - Y_i \right) \end{aligned}$$

The gradient update in iteration $t+1$ then becomes

$$\begin{aligned} \alpha_i^{t+1} &= \alpha_i^t - \eta \gamma_i^t \\ \alpha_i^{t+1} &= \alpha_i^t - 2\eta \left(\sum_{j=1}^{n} \alpha_j^t K_{ij} - Y_i \right) \end{aligned}$$

Since we have $i = 1, 2, ..., n$ such updates to do, the amount of work per gradient update in the transformed space is $O\left(n^2\right)$, which is far better than $O\left(2^d\right)$. Note that K_{ij} has been computed efficiently by reformulating the sum of many terms as the product of fewer terms; these inner products are stored in a matrix K and may be accessed during training.

3 Kernel Functions

Can any function $\mathrm{K}(\cdot, \cdot) \to \mathcal{R}$ be used as a kernel? No, the kernel matrix $\mathrm{K}\left(\vec{X}_i, \vec{X}_j\right)$ has to correspond to real inner-products after some transformation $\vec{X} \to \phi(\vec{X})$. This is the case if and only if K is positive semi-definite.

Definition 3.1. A matrix $A \in \mathbb{R}^{n \times n}$ is positive semi-definite iff $\mathbf{v}^\top A \mathbf{v} \geq 0 \quad \forall \quad \mathbf{v} \in \mathbb{R}^n$. It is positive definite if the inequality holds with equality only for vectors $\mathbf{v} = \vec{0}$.

Recall that the kernel matrix is $\mathrm{K}_{ij} = \phi\left(\vec{X}_i\right)^\top \phi\left(\vec{X}_j\right)$. So $\mathrm{K} = \Phi^\top \Phi$, where $\Phi = \left[\phi\left(\vec{X}_1\right), \ldots, \phi\left(\vec{X}_n\right)\right] \in \mathbb{R}^{D \times n}$. It follows that K is p.s.d., because $\mathbf{q}^\top \mathrm{K} \mathbf{q} = \left(\Phi^\top \mathbf{q}\right)^2 \geq 0$. Inversely, if any matrix $\mathbf{A}$ is p.s.d., it can be decomposed as $A = \Phi^\top \Phi$ for some realization of Φ. The columns $\phi\left(\vec{X}_1\right), \ldots, \phi\left(\vec{X}_n\right)$ of matrix Φ can be seen as vectors in real vector space. Then the entries of K are inner products (that is, dot products) of these vectors

$$K_{ij} = \left\langle \phi\left(\vec{X}_i\right), \phi\left(\vec{X}_j\right) \right\rangle$$

In other words, a matrix K is positive semi-definite if and only if it is the Gram matrix (whose entries are K_{ij}) of some vectors (training examples or rows of the data matrix) $\phi\left(\vec{X}_1\right), \ldots, \phi\left(\vec{X}_n\right)$. It is positive definite if and only if it is the Gram matrix of some linearly independent vectors. The fact that the entries of K are inner products is related to its positive semi-definiteness. This is because one of the properties of inner products $\langle \vec{v}, \vec{v} \rangle$ is that they are non-negative for all nonzero $\vec{v}$; it equals zero only if $\vec{v} = \vec{0}$.