

Chapter 5

Kernel methods

5.1 Feature maps

Recall that in our discussion about linear regression, we considered the problem of predicting the price of a house (denoted by y) from the living area of the house (denoted by x), and we fit a linear function of x to the training data. What if the price y can be more accurately represented as a *non-linear* function of x ? In this case, we need a more expressive family of models than linear models.

We start by considering fitting cubic functions $y = \theta_3 x^3 + \theta_2 x^2 + \theta_1 x + \theta_0$. It turns out that we can view the cubic function as a linear function over the a different set of feature variables (defined below). Concretely, let the function $\phi : \mathbb{R} \rightarrow \mathbb{R}^4$ be defined as

$$\phi(x) = \begin{bmatrix} 1 \\ x \\ x^2 \\ x^3 \end{bmatrix} \in \mathbb{R}^4. \quad (5.1)$$

Let $\theta \in \mathbb{R}^4$ be the vector containing $\theta_0, \theta_1, \theta_2, \theta_3$ as entries. Then we can rewrite the cubic function in x as:

$$\theta_3 x^3 + \theta_2 x^2 + \theta_1 x + \theta_0 = \theta^T \phi(x)$$

Thus, a cubic function of the variable x can be viewed as a linear function over the variables $\phi(x)$. To distinguish between these two sets of variables, in the context of kernel methods, we will call the “original” input value the input **attributes** of a problem (in this case, x , the living area). When the

original input is mapped to some new set of quantities $\phi(x)$, we will call those new quantities the **features** variables. (Unfortunately, different authors use different terms to describe these two things in different contexts.) We will call ϕ a **feature map**, which maps the attributes to the features.

5.2 LMS (least mean squares) with features

We will derive the gradient descent algorithm for fitting the model $\theta^T \phi(x)$. First recall that for ordinary least square problem where we were to fit $\theta^T x$, the batch gradient descent update is (see the first lecture note for its derivation):

$$\begin{aligned}\theta &:= \theta + \alpha \sum_{i=1}^n (y^{(i)} - h_{\theta}(x^{(i)})) x^{(i)} \\ &:= \theta + \alpha \sum_{i=1}^n (y^{(i)} - \theta^T x^{(i)}) x^{(i)}.\end{aligned}\tag{5.2}$$

Let $\phi : \mathbb{R}^d \rightarrow \mathbb{R}^p$ be a feature map that maps attribute x (in \mathbb{R}^d) to the features $\phi(x)$ in \mathbb{R}^p . (In the motivating example in the previous subsection, we have $d = 1$ and $p = 4$.) Now our goal is to fit the function $\theta^T \phi(x)$, with θ being a vector in \mathbb{R}^p instead of \mathbb{R}^d . We can replace all the occurrences of $x^{(i)}$ in the algorithm above by $\phi(x^{(i)})$ to obtain the new update:

$$\theta := \theta + \alpha \sum_{i=1}^n (y^{(i)} - \theta^T \phi(x^{(i)})) \phi(x^{(i)})\tag{5.3}$$

Similarly, the corresponding stochastic gradient descent update rule is

$$\theta := \theta + \alpha (y^{(i)} - \theta^T \phi(x^{(i)})) \phi(x^{(i)})\tag{5.4}$$

5.3 LMS with the kernel trick

The gradient descent update, or stochastic gradient update above becomes computationally expensive when the features $\phi(x)$ is high-dimensional. For example, consider the direct extension of the feature map in equation (5.1) to high-dimensional input x : suppose $x \in \mathbb{R}^d$, and let $\phi(x)$ be the vector that

contains all the monomials of x with degree ≤ 3

$$\phi(x) = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_1^2 \\ x_1x_2 \\ x_1x_3 \\ \vdots \\ x_2x_1 \\ \vdots \\ x_1^3 \\ x_1^2x_2 \\ \vdots \end{bmatrix}. \quad (5.5)$$

The dimension of the features $\phi(x)$ is on the order of d^3 .¹ This is a prohibitively long vector for computational purpose — when $d = 1000$, each update requires at least computing and storing a $1000^3 = 10^9$ dimensional vector, which is 10^6 times slower than the update rule for ordinary least squares updates (5.2).

It may appear at first that such d^3 runtime per update and memory usage are inevitable, because the vector θ itself is of dimension $p \approx d^3$, and we may need to update every entry of θ and store it. However, we will introduce the kernel trick with which we will not need to store θ explicitly, and the runtime can be significantly improved.

For simplicity, we assume the initialize the value $\theta = 0$, and we focus on the iterative update (5.3). The main observation is that at any time, θ can be represented as a linear combination of the vectors $\phi(x^{(1)}), \dots, \phi(x^{(n)})$. Indeed, we can show this inductively as follows. At initialization, $\theta = 0 = \sum_{i=1}^n 0 \cdot \phi(x^{(i)})$. Assume at some point, θ can be represented as

$$\theta = \sum_{i=1}^n \beta_i \phi(x^{(i)}) \quad (5.6)$$

¹Here, for simplicity, we include all the monomials with repetitions (so that, e.g., $x_1x_2x_3$ and $x_2x_3x_1$ both appear in $\phi(x)$). Therefore, there are totally $1 + d + d^2 + d^3$ entries in $\phi(x)$.

for some $\beta_1, \dots, \beta_n \in \mathbb{R}$. Then we claim that in the next round, θ is still a linear combination of $\phi(x^{(1)}), \dots, \phi(x^{(n)})$ because

$$\begin{aligned}
 \theta &:= \theta + \alpha \sum_{i=1}^n (y^{(i)} - \theta^T \phi(x^{(i)})) \phi(x^{(i)}) \\
 &= \sum_{i=1}^n \beta_i \phi(x^{(i)}) + \alpha \sum_{i=1}^n (y^{(i)} - \theta^T \phi(x^{(i)})) \phi(x^{(i)}) \\
 &= \sum_{i=1}^n \underbrace{(\beta_i + \alpha (y^{(i)} - \theta^T \phi(x^{(i)})))}_{\text{new } \beta_i} \phi(x^{(i)})
 \end{aligned} \tag{5.7}$$

You may realize that our general strategy is to implicitly represent the p -dimensional vector θ by a set of coefficients β_1, \dots, β_n . Towards doing this, we derive the update rule of the coefficients β_1, \dots, β_n . Using the equation above, we see that the new β_i depends on the old one via

$$\beta_i := \beta_i + \alpha (y^{(i)} - \theta^T \phi(x^{(i)})) \tag{5.8}$$

Here we still have the old θ on the RHS of the equation. Replacing θ by $\theta = \sum_{j=1}^n \beta_j \phi(x^{(j)})$ gives

$$\forall i \in \{1, \dots, n\}, \beta_i := \beta_i + \alpha \left(y^{(i)} - \sum_{j=1}^n \beta_j \phi(x^{(j)})^T \phi(x^{(i)}) \right)$$

We often rewrite $\phi(x^{(j)})^T \phi(x^{(i)})$ as $\langle \phi(x^{(j)}), \phi(x^{(i)}) \rangle$ to emphasize that it's the inner product of the two feature vectors. Viewing β_i 's as the new representation of θ , we have successfully translated the batch gradient descent algorithm into an algorithm that updates the value of β iteratively. It may appear that at every iteration, we still need to compute the values of $\langle \phi(x^{(j)}), \phi(x^{(i)}) \rangle$ for all pairs of i, j , each of which may take roughly $O(p)$ operation. However, two important properties come to rescue:

1. We can pre-compute the pairwise inner products $\langle \phi(x^{(j)}), \phi(x^{(i)}) \rangle$ for all pairs of i, j before the loop starts.
2. For the feature map ϕ defined in (5.5) (or many other interesting feature maps), computing $\langle \phi(x^{(j)}), \phi(x^{(i)}) \rangle$ can be efficient and does not

necessarily require computing $\phi(x^{(i)})$ explicitly. This is because:

$$\begin{aligned}
\langle \phi(x), \phi(z) \rangle &= 1 + \sum_{i=1}^d x_i z_i + \sum_{i,j \in \{1, \dots, d\}} x_i x_j z_i z_j + \sum_{i,j,k \in \{1, \dots, d\}} x_i x_j x_k z_i z_j z_k \\
&= 1 + \sum_{i=1}^d x_i z_i + \left(\sum_{i=1}^d x_i z_i \right)^2 + \left(\sum_{i=1}^d x_i z_i \right)^3 \\
&= 1 + \langle x, z \rangle + \langle x, z \rangle^2 + \langle x, z \rangle^3
\end{aligned} \tag{5.9}$$

Therefore, to compute $\langle \phi(x), \phi(z) \rangle$, we can first compute $\langle x, z \rangle$ with $O(d)$ time and then take another constant number of operations to compute $1 + \langle x, z \rangle + \langle x, z \rangle^2 + \langle x, z \rangle^3$.

As you will see, the inner products between the features $\langle \phi(x), \phi(z) \rangle$ are essential here. We define the **Kernel** corresponding to the feature map ϕ as a function that maps $\mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$ satisfying: ²

$$K(x, z) \triangleq \langle \phi(x), \phi(z) \rangle \tag{5.10}$$

To wrap up the discussion, we write the down the final algorithm as follows:

-
1. Compute all the values $K(x^{(i)}, x^{(j)}) \triangleq \langle \phi(x^{(i)}), \phi(x^{(j)}) \rangle$ using equation (5.9) for all $i, j \in \{1, \dots, n\}$. Set $\beta := 0$.
 2. **Loop:**

$$\forall i \in \{1, \dots, n\}, \beta_i := \beta_i + \alpha \left(y^{(i)} - \sum_{j=1}^n \beta_j K(x^{(i)}, x^{(j)}) \right) \tag{5.11}$$

Or in vector notation, letting K be the $n \times n$ matrix with $K_{ij} = K(x^{(i)}, x^{(j)})$, we have

$$\beta := \beta + \alpha(\vec{y} - K\beta)$$

With the algorithm above, we can update the representation β of the vector θ efficiently with $O(n)$ time per update. Finally, we need to show that

²Recall that \mathcal{X} is the space of the input x . In our running example, $\mathcal{X} = \mathbb{R}^d$

the knowledge of the representation β suffices to compute the prediction $\theta^T \phi(x)$. Indeed, we have

$$\theta^T \phi(x) = \sum_{i=1}^n \beta_i \phi(x^{(i)})^T \phi(x) = \sum_{i=1}^n \beta_i K(x^{(i)}, x) \quad (5.12)$$

You may realize that fundamentally all we need to know about the feature map $\phi(\cdot)$ is encapsulated in the corresponding kernel function $K(\cdot, \cdot)$. We will expand on this in the next section.

5.4 Properties of kernels

In the last subsection, we started with an explicitly defined feature map ϕ , which induces the kernel function $K(x, z) \triangleq \langle \phi(x), \phi(z) \rangle$. Then we saw that the kernel function is so intrinsic so that as long as the kernel function is defined, the whole training algorithm can be written entirely in the language of the kernel without referring to the feature map ϕ , so can the prediction of a test example x (equation (5.12).)

Therefore, it would be tempted to define other kernel function $K(\cdot, \cdot)$ and run the algorithm (5.11). Note that the algorithm (5.11) does not need to explicitly access the feature map ϕ , and therefore we only need to ensure the existence of the feature map ϕ , but do not necessarily need to be able to explicitly write ϕ down.

What kinds of functions $K(\cdot, \cdot)$ can correspond to some feature map ϕ ? In other words, can we tell if there is some feature mapping ϕ so that $K(x, z) = \phi(x)^T \phi(z)$ for all x, z ?

If we can answer this question by giving a precise characterization of valid kernel functions, then we can completely change the interface of selecting feature maps ϕ to the interface of selecting kernel function K . Concretely, we can pick a function K , verify that it satisfies the characterization (so that there exists a feature map ϕ that K corresponds to), and then we can run update rule (5.11). The benefit here is that we don't have to be able to compute ϕ or write it down analytically, and we only need to know its existence. We will answer this question at the end of this subsection after we go through several concrete examples of kernels.

Suppose $x, z \in \mathbb{R}^d$, and let's first consider the function $K(\cdot, \cdot)$ defined as:

$$K(x, z) = (x^T z)^2.$$

We can also write this as

$$\begin{aligned}
 K(x, z) &= \left(\sum_{i=1}^d x_i z_i \right) \left(\sum_{j=1}^d x_j z_j \right) \\
 &= \sum_{i=1}^d \sum_{j=1}^d x_i x_j z_i z_j \\
 &= \sum_{i,j=1}^d (x_i x_j) (z_i z_j)
 \end{aligned}$$

Thus, we see that $K(x, z) = \langle \phi(x), \phi(z) \rangle$ is the kernel function that corresponds to the feature mapping ϕ given (shown here for the case of $d = 3$) by

$$\phi(x) = \begin{bmatrix} x_1 x_1 \\ x_1 x_2 \\ x_1 x_3 \\ x_2 x_1 \\ x_2 x_2 \\ x_2 x_3 \\ x_3 x_1 \\ x_3 x_2 \\ x_3 x_3 \end{bmatrix}.$$

Revisiting the computational efficiency perspective of kernel, note that whereas calculating the high-dimensional $\phi(x)$ requires $O(d^2)$ time, finding $K(x, z)$ takes only $O(d)$ time—linear in the dimension of the input attributes.

For another related example, also consider $K(\cdot, \cdot)$ defined by

$$\begin{aligned}
 K(x, z) &= (x^T z + c)^2 \\
 &= \sum_{i,j=1}^d (x_i x_j) (z_i z_j) + \sum_{i=1}^d (\sqrt{2c} x_i) (\sqrt{2c} z_i) + c^2.
 \end{aligned}$$

(Check this yourself.) This function K is a kernel function that corresponds

to the feature mapping (again shown for $d = 3$)

$$\phi(x) = \begin{bmatrix} x_1x_1 \\ x_1x_2 \\ x_1x_3 \\ x_2x_1 \\ x_2x_2 \\ x_2x_3 \\ x_3x_1 \\ x_3x_2 \\ x_3x_3 \\ \sqrt{2c}x_1 \\ \sqrt{2c}x_2 \\ \sqrt{2c}x_3 \\ c \end{bmatrix},$$

and the parameter c controls the relative weighting between the x_i (first order) and the x_ix_j (second order) terms.

More broadly, the kernel $K(x, z) = (x^T z + c)^k$ corresponds to a feature mapping to an $\binom{d+k}{k}$ feature space, corresponding of all monomials of the form $x_{i_1}x_{i_2}\dots x_{i_k}$ that are up to order k . However, despite working in this $O(d^k)$ -dimensional space, computing $K(x, z)$ still takes only $O(d)$ time, and hence we never need to explicitly represent feature vectors in this very high dimensional feature space.

Kernels as similarity metrics. Now, let's talk about a slightly different view of kernels. Intuitively, (and there are things wrong with this intuition, but nevermind), if $\phi(x)$ and $\phi(z)$ are close together, then we might expect $K(x, z) = \phi(x)^T \phi(z)$ to be large. Conversely, if $\phi(x)$ and $\phi(z)$ are far apart—say nearly orthogonal to each other—then $K(x, z) = \phi(x)^T \phi(z)$ will be small. So, we can think of $K(x, z)$ as some measurement of how similar are $\phi(x)$ and $\phi(z)$, or of how similar are x and z .

Given this intuition, suppose that for some learning problem that you're working on, you've come up with some function $K(x, z)$ that you think might be a reasonable measure of how similar x and z are. For instance, perhaps you chose

$$K(x, z) = \exp\left(-\frac{\|x - z\|^2}{2\sigma^2}\right).$$

This is a reasonable measure of x and z 's similarity, and is close to 1 when x and z are close, and near 0 when x and z are far apart. Does there exist

a feature map ϕ such that the kernel K defined above satisfies $K(x, z) = \phi(x)^T \phi(z)$? In this particular example, the answer is yes. This kernel is called the **Gaussian kernel**, and corresponds to an infinite dimensional feature mapping ϕ . We will give a precise characterization about what properties a function K needs to satisfy so that it can be a valid kernel function that corresponds to some feature map ϕ .

Necessary conditions for valid kernels. Suppose for now that K is indeed a valid kernel corresponding to some feature mapping ϕ , and we will first see what properties it satisfies. Now, consider some finite set of n points (not necessarily the training set) $\{x^{(1)}, \dots, x^{(n)}\}$, and let a square, n -by- n matrix K be defined so that its (i, j) -entry is given by $K_{ij} = K(x^{(i)}, x^{(j)})$. This matrix is called the **kernel matrix**. Note that we've overloaded the notation and used K to denote both the kernel function $K(x, z)$ and the kernel matrix K , due to their obvious close relationship.

Now, if K is a valid kernel, then $K_{ij} = K(x^{(i)}, x^{(j)}) = \phi(x^{(i)})^T \phi(x^{(j)}) = \phi(x^{(j)})^T \phi(x^{(i)}) = K(x^{(j)}, x^{(i)}) = K_{ji}$, and hence K must be symmetric. Moreover, letting $\phi_k(x)$ denote the k -th coordinate of the vector $\phi(x)$, we find that for any vector z , we have

$$\begin{aligned}
 z^T K z &= \sum_i \sum_j z_i K_{ij} z_j \\
 &= \sum_i \sum_j z_i \phi(x^{(i)})^T \phi(x^{(j)}) z_j \\
 &= \sum_i \sum_j z_i \sum_k \phi_k(x^{(i)}) \phi_k(x^{(j)}) z_j \\
 &= \sum_k \sum_i \sum_j z_i \phi_k(x^{(i)}) \phi_k(x^{(j)}) z_j \\
 &= \sum_k \left(\sum_i z_i \phi_k(x^{(i)}) \right)^2 \\
 &\geq 0.
 \end{aligned}$$

The second-to-last step uses the fact that $\sum_{i,j} a_i a_j = (\sum_i a_i)^2$ for $a_i = z_i \phi_k(x^{(i)})$. Since z was arbitrary, this shows that K is positive semi-definite ($K \geq 0$).

Hence, we've shown that if K is a valid kernel (i.e., if it corresponds to some feature mapping ϕ), then the corresponding kernel matrix $K \in \mathbb{R}^{n \times n}$ is symmetric positive semidefinite.

Sufficient conditions for valid kernels. More generally, the condition above turns out to be not only a necessary, but also a sufficient, condition for K to be a valid kernel (also called a Mercer kernel). The following result is due to Mercer.³

Theorem (Mercer). Let $K : \mathbb{R}^d \times \mathbb{R}^d \mapsto \mathbb{R}$ be given. Then for K to be a valid (Mercer) kernel, it is necessary and sufficient that for any $\{x^{(1)}, \dots, x^{(n)}\}$, $(n < \infty)$, the corresponding kernel matrix is symmetric positive semi-definite.

Given a function K , apart from trying to find a feature mapping ϕ that corresponds to it, this theorem therefore gives another way of testing if it is a valid kernel. You'll also have a chance to play with these ideas more in problem set 2.

In class, we also briefly talked about a couple of other examples of kernels. For instance, consider the digit recognition problem, in which given an image (16x16 pixels) of a handwritten digit (0-9), we have to figure out which digit it was. Using either a simple polynomial kernel $K(x, z) = (x^T z)^k$ or the Gaussian kernel, SVMs were able to obtain extremely good performance on this problem. This was particularly surprising since the input attributes x were just 256-dimensional vectors of the image pixel intensity values, and the system had no prior knowledge about vision, or even about which pixels are adjacent to which other ones. Another example that we briefly talked about in lecture was that if the objects x that we are trying to classify are strings (say, x is a list of amino acids, which strung together form a protein), then it seems hard to construct a reasonable, “small” set of features for most learning algorithms, especially if different strings have different lengths. However, consider letting $\phi(x)$ be a feature vector that counts the number of occurrences of each length- k substring in x . If we're considering strings of English letters, then there are 26^k such strings. Hence, $\phi(x)$ is a 26^k dimensional vector; even for moderate values of k , this is probably too big for us to efficiently work with. (e.g., $26^4 \approx 460000$.) However, using (dynamic programming-ish) string matching algorithms, it is possible to efficiently compute $K(x, z) = \phi(x)^T \phi(z)$, so that we can now implicitly work in this 26^k -dimensional feature space, but without ever explicitly computing feature vectors in this space.

³Many texts present Mercer's theorem in a slightly more complicated form involving L^2 functions, but when the input attributes take values in \mathbb{R}^d , the version given here is equivalent.

Application of kernel methods: We've seen the application of kernels to linear regression. In the next part, we will introduce the support vector machines to which kernels can be directly applied. dwell too much longer on it here. In fact, the idea of kernels has significantly broader applicability than linear regression and SVMs. Specifically, if you have any learning algorithm that you can write in terms of only inner products $\langle x, z \rangle$ between input attribute vectors, then by replacing this with $K(x, z)$ where K is a kernel, you can “magically” allow your algorithm to work efficiently in the high dimensional feature space corresponding to K . For instance, this kernel trick can be applied with the perceptron to derive a kernel perceptron algorithm. Many of the algorithms that we'll see later in this class will also be amenable to this method, which has come to be known as the “kernel trick.”