CS229 Lecture Notes

Andrew Ng

updated by Tengyu Ma on April 21, 2019

Part V
Kernel Methods

1.1 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.

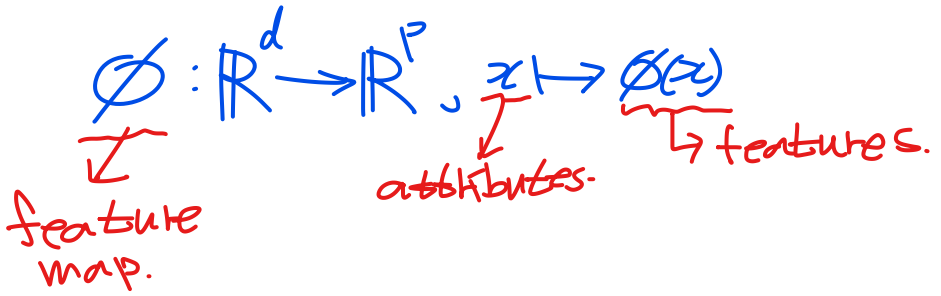
We start by considering fitting cubic functions $y = \theta_3 x^3 + \theta_2 x^2 + \theta_1 x + \theta_0$. It turns out that we can view the cubic function as a linear function over the a different set of feature variables (defined below). Concretely, let the function $\phi : \mathbb{R} \to \mathbb{R}^4$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

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

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.

1.2 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):

$$\begin{aligned} \theta &:= \theta + \alpha \sum_{i=1}^{n} \left(y^{(i)} - h_\theta(x^{(i)}) \right) x^{(i)} \\ &:= \theta + \alpha \sum_{i=1}^{n} \left(y^{(i)} - \theta^T x^{(i)} \right) x^{(i)}. \end{aligned} \tag{2}$$

Let $\phi : \mathbb{R}^d \to \mathbb{R}^p$ 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:

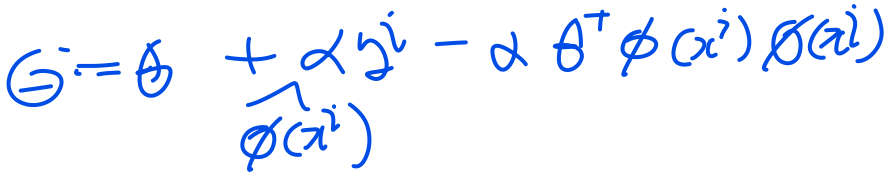
$$\theta := \theta + \alpha \sum_{i=1}^{n} \left(y^{(i)} - \theta^T \phi(x^{(i)}) \right) \phi(x^{(i)}) \tag{3}$$

Similarly, the corresponding stochastic gradient descent update rule is

$$\theta := \theta + \alpha \left(y^{(i)} - \theta^T \phi(x^{(i)}) \right) \phi(x^{(i)}) \tag{4}$$

1.3 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 (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

$$\phi(x) = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_1^2 \\ x_1 x_2 \\ x_1 x_3 \\ \vdots \\ x_2 x_1 \\ \vdots \\ x_1^3 \\ x_1^2 x_2 \\ \vdots \end{bmatrix}. \tag{5}$$

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 for ordinary least squares updates (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 (3). The main observation is that at any time, θ can be represented as a linear combination of the vectors $\phi(x^{(1)}), \ldots, \phi(x^{(n)})$. 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

$$\theta = \sum_{i=1}^{n} \beta_i \phi(x^{(i)}) \tag{6}$$

[1] Here, for simplicity, we include all the monomials with repetitions (so that, e.g., $x_1 x_2 x_3$ and $x_2 x_3 x_1$ both appear in $\phi(x)$). Therefore, there are totally $1 + d + d^2 + d^3$ entries in $\phi(x)$.

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

$$\begin{aligned}\theta &:= \theta + \alpha \sum_{i=1}^{n} \left(y^{(i)} - \theta^T \phi(x^{(i)})\right) \phi(x^{(i)}) \\ &= \sum_{i=1}^{n} \beta_i \phi(x^{(i)}) + \alpha \sum_{i=1}^{n} \left(y^{(i)} - \theta^T \phi(x^{(i)})\right) \phi(x^{(i)}) \\ &= \sum_{i=1}^{n} \underbrace{\left(\beta_i + \alpha \left(y^{(i)} - \theta^T \phi(x^{(i)})\right)\right)}_{\text{new } \beta_i} \phi(x^{(i)}) \end{aligned} \tag{7}$$

You may realize that our general strategy is to implicitly represent the p-dimensional vector θ by a set of coefficients $\beta_1, \ldots, \beta_n$. Towards doing this, we derive the update rule of the coefficients $\beta_1, \ldots, \beta_n$. Using the equation above, we see that the new β_i depends on the old one via

$$\beta_i := \beta_i + \alpha \left(y^{(i)} - \theta^T \phi(x^{(i)})\right) \tag{8}$$

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

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

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) (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:

$$\begin{aligned}\langle \phi(x), \phi(z) \rangle &= 1 + \sum_{i=1}^{d} x_i z_i + \sum_{i,j \in \{1,\ldots,d\}} x_i x_j z_i z_j + \sum_{i,j,k \in \{1,\ldots,d\}} x_i x_j x_k z_i z_j z_k \\ &= 1 + \sum_{i=1}^{d} x_i z_i + \left(\sum_{i=1}^{d} x_i z_i \right)^2 + \left(\sum_{i=1}^{d} x_i z_i \right)^3 \\ &= 1 + \langle x, z \rangle + \langle x, z \rangle^2 + \langle x, z \rangle^3 \end{aligned} \tag{9}$$

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 $\mathcal{X} \times \mathcal{X} \to \mathbb{R}$ satisfying: [2]

$$K(x, z) \triangleq \langle \phi(x), \phi(z) \rangle \tag{10}$$

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 (9) for all $i, j \in \{1, \ldots, n\}$. Set $\beta := 0$.
2. **Loop:**

$$\forall i \in \{1, \ldots, n\}, \beta_i := \beta_i + \alpha \left(y^{(i)} - \sum_{j=1}^{n} \beta_j K(x^{(i)}, x^{(j)}) \right) \tag{11}$$

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

[2] Recall 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

$$\theta^T\phi(x) = \sum_{i=1}^{n} \beta_i \phi(x^{(i)})^T \phi(x) = \sum_{i=1}^{n} \beta_i K(x^{(i)}, x) \tag{12}$$

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.

1.4 Properties of kernels

Recap. 1.3

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 (12).)

Therefore, it would be tempted to define other kernel function $K(\cdot,\cdot)$ and run the algorithm (11). Note that the algorithm (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 (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.

$\phi(x)^T\phi(z)$에 대응하는 $k(x,z)$를 찾으면 K에 대한 식도 사용가능 (eq. 11)

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

$$K(x,z) = (x^Tz)^2. \quad (z^Tz)(z^Tz)$$

$$K(x,z) = \langle \phi(x), \phi(z) \rangle = \phi(x)^T\phi(z)$$

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 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 $K(x, z) = (x^Tz + c)^k$ corresponds to a feature mapping to an $\binom{d+k}{k}$ feature space, corresponding of all monomials of the form $x_{i_1}x_{i_2}\ldots x_{i_k}$ 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) $\{x^{(1)}, \ldots, x^{(n)}\}$, 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 $\{x^{(1)}, \ldots, x^{(n)}\}$, $(n < \infty)$, 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.

[3] Many 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."

Part VI
Support Vector Machines

This set of notes presents the Support Vector Machine (SVM) learning algorithm. SVMs are among the best (and many believe are indeed the best) "off-the-shelf" supervised learning algorithms. To tell the SVM story, we'll need to first talk about margins and the idea of separating data with a large "gap." Next, we'll talk about the optimal margin classifier, which will lead us into a digression on Lagrange duality. We'll also see kernels, which give a way to apply SVMs efficiently in very high dimensional (such as infinite-dimensional) feature spaces, and finally, we'll close off the story with the SMO algorithm, which gives an efficient implementation of SVMs.

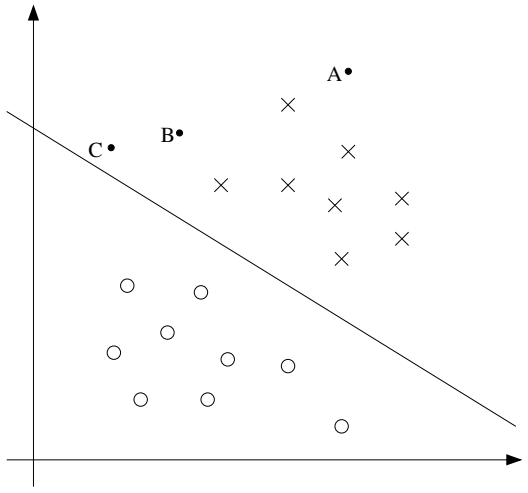
2 Margins: Intuition

We'll start our story on SVMs by talking about margins. This section will give the intuitions about margins and about the "confidence" of our predictions; these ideas will be made formal in Section 4.

Consider logistic regression, where the probability $p(y = 1|x; \theta)$ is modeled by $h_\theta(x) = g(\theta^T x)$. We then predict "1" on an input x if and only if $h_\theta(x) \geq 0.5$, or equivalently, if and only if $\theta^T x \geq 0$. Consider a positive training example ($y = 1$). The larger $\theta^T x$ is, the larger also is $h_\theta(x) = p(y = 1|x; \theta)$, and thus also the higher our degree of "confidence" that the label is 1. Thus, informally we can think of our prediction as being very confident that

$y = 1$ if $\theta^T x \gg 0$. Similarly, we think of logistic regression as confidently predicting $y = 0$, if $\theta^T x \ll 0$. Given a training set, again informally it seems that we'd have found a good fit to the training data if we can find θ so that $\theta^T x^{(i)} \gg 0$ whenever $y^{(i)} = 1$, and $\theta^T x^{(i)} \ll 0$ whenever $y^{(i)} = 0$, since this would reflect a very confident (and correct) set of classifications for all the training examples. This seems to be a nice goal to aim for, and we'll soon formalize this idea using the notion of functional margins.

For a different type of intuition, consider the following figure, in which x's represent positive training examples, o's denote negative training examples, a decision boundary (this is the line given by the equation $\theta^T x = 0$, and is also called the **separating hyperplane**) is also shown, and three points have also been labeled A, B and C.



Notice that the point A is very far from the decision boundary. If we are asked to make a prediction for the value of y at A, it seems we should be quite confident that $y = 1$ there. Conversely, the point C is very close to the decision boundary, and while it's on the side of the decision boundary on which we would predict $y = 1$, it seems likely that just a small change to the decision boundary could easily have caused out prediction to be $y = 0$. Hence, we're much more confident about our prediction at A than at C. The point B lies in-between these two cases, and more broadly, we see that if a point is far from the separating hyperplane, then we may be significantly more confident in our predictions. Again, informally we think it would be nice if, given a training set, we manage to find a decision boundary that allows us to make all correct and confident (meaning far from the decision boundary) predictions on the training examples. We'll formalize this later using the notion of geometric margins.

3 Notation

To make our discussion of SVMs easier, we'll first need to introduce a new notation for talking about classification. We will be considering a linear classifier for a binary classification problem with labels y and features x. From now, we'll use $y \in \{-1, 1\}$ (instead of $\{0, 1\}$) to denote the class labels. Also, rather than parameterizing our linear classifier with the vector θ, we will use parameters w, b, and write our classifier as

$$h_{w,b}(x) = g(w^T x + b).$$

Here, $g(z) = 1$ if $z \geq 0$, and $g(z) = -1$ otherwise. This "w, b" notation allows us to explicitly treat the intercept term b separately from the other parameters. (We also drop the convention we had previously of letting $x_0 = 1$ be an extra coordinate in the input feature vector.) Thus, b takes the role of what was previously θ_0, and w takes the role of $[\theta_1 \ldots \theta_d]^T$.

Note also that, from our definition of g above, our classifier will directly predict either 1 or -1 (cf. the perceptron algorithm), without first going through the intermediate step of estimating $p(y = 1)$ (which is what logistic regression does).

4 Functional and geometric margins

Let's formalize the notions of the functional and geometric margins. Given a training example $(x^{(i)}, y^{(i)})$, we define the **functional margin** of (w, b) with respect to the training example as

$$\hat{\gamma}^{(i)} = y^{(i)}(w^T x^{(i)} + b).$$

Note that if $y^{(i)} = 1$, then for the functional margin to be large (i.e., for our prediction to be confident and correct), we need $w^T x^{(i)} + b$ to be a large positive number. Conversely, if $y^{(i)} = -1$, then for the functional margin to be large, we need $w^T x^{(i)} + b$ to be a large negative number. Moreover, if $y^{(i)}(w^T x^{(i)} + b) > 0$, then our prediction on this example is correct. (Check this yourself.) Hence, a large functional margin represents a confident and a correct prediction.

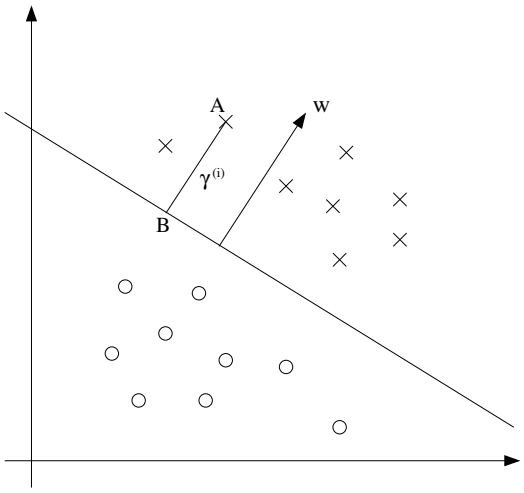
For a linear classifier with the choice of g given above (taking values in $\{-1, 1\}$), there's one property of the functional margin that makes it not a very good measure of confidence, however. Given our choice of g, we note that if we replace w with $2w$ and b with $2b$, then since $g(w^T x + b) = g(2w^T x + 2b)$,

this would not change $h_{w,b}(x)$ at all. I.e., g, and hence also $h_{w,b}(x)$, depends only on the sign, but not on the magnitude, of $w^T x + b$. However, replacing (w, b) with $(2w, 2b)$ also results in multiplying our functional margin by a factor of 2. Thus, it seems that by exploiting our freedom to scale w and b, we can make the functional margin arbitrarily large without really changing anything meaningful. Intuitively, it might therefore make sense to impose some sort of normalization condition such as that $||w||_2 = 1$; i.e., we might replace (w, b) with $(w/||w||_2, b/||w||_2)$, and instead consider the functional margin of $(w/||w||_2, b/||w||_2)$. We'll come back to this later.

Given a training set $S = \{(x^{(i)}, y^{(i)}); i = 1, \ldots, n\}$, we also define the function margin of (w, b) with respect to S as the smallest of the functional margins of the individual training examples. Denoted by $\hat{\gamma}$, this can therefore be written:

$$\hat{\gamma} = \min_{i=1,\ldots,n} \hat{\gamma}^{(i)}.$$

Next, let's talk about **geometric margins**. Consider the picture below:



The decision boundary corresponding to (w, b) is shown, along with the vector w. Note that w is orthogonal (at 90°) to the separating hyperplane. (You should convince yourself that this must be the case.) Consider the point at A, which represents the input $x^{(i)}$ of some training example with label $y^{(i)} = 1$. Its distance to the decision boundary, $\gamma^{(i)}$, is given by the line segment AB.

How can we find the value of $\gamma^{(i)}$? Well, $w/||w||$ is a unit-length vector pointing in the same direction as w. Since A represents $x^{(i)}$, we therefore find that the point B is given by $x^{(i)} - \gamma^{(i)} \cdot w/||w||$. But this point lies on

the decision boundary, and all points x on the decision boundary satisfy the equation $w^T x + b = 0$. Hence,

$$w^T \left(x^{(i)} - \gamma^{(i)} \frac{w}{||w||} \right) + b = 0.$$

Solving for $\gamma^{(i)}$ yields

$$\gamma^{(i)} = \frac{w^T x^{(i)} + b}{||w||} = \left(\frac{w}{||w||} \right)^T x^{(i)} + \frac{b}{||w||}.$$

This was worked out for the case of a positive training example at A in the figure, where being on the "positive" side of the decision boundary is good. More generally, we define the geometric margin of (w, b) with respect to a training example $(x^{(i)}, y^{(i)})$ to be

$$\gamma^{(i)} = y^{(i)} \left(\left(\frac{w}{||w||} \right)^T x^{(i)} + \frac{b}{||w||} \right).$$

Note that if $||w|| = 1$, then the functional margin equals the geometric margin—this thus gives us a way of relating these two different notions of margin. Also, the geometric margin is invariant to rescaling of the parameters; i.e., if we replace w with $2w$ and b with $2b$, then the geometric margin does not change. This will in fact come in handy later. Specifically, because of this invariance to the scaling of the parameters, when trying to fit w and b to training data, we can impose an arbitrary scaling constraint on w without changing anything important; for instance, we can demand that $||w|| = 1$, or $|w_1| = 5$, or $|w_1 + b| + |w_2| = 2$, and any of these can be satisfied simply by rescaling w and b.

Finally, given a training set $S = \{(x^{(i)}, y^{(i)}); i = 1, \ldots, n\}$, we also define the geometric margin of (w, b) with respect to S to be the smallest of the geometric margins on the individual training examples:

$$\gamma = \min_{i=1,\ldots,n} \gamma^{(i)}.$$

5 The optimal margin classifier

Given a training set, it seems from our previous discussion that a natural desideratum is to try to find a decision boundary that maximizes the (geometric) margin, since this would reflect a very confident set of predictions

on the training set and a good "fit" to the training data. Specifically, this will result in a classifier that separates the positive and the negative training examples with a "gap" (geometric margin).

For now, we will assume that we are given a training set that is linearly separable; i.e., that it is possible to separate the positive and negative examples using some separating hyperplane. How will we find the one that achieves the maximum geometric margin? We can pose the following optimization problem:

$$\begin{aligned} \max_{\gamma,w,b} \quad & \gamma \\ \text{s.t.} \quad & y^{(i)}(w^T x^{(i)} + b) \geq \gamma, \quad i = 1, \ldots, n \\ & ||w|| = 1. \end{aligned}$$

I.e., we want to maximize γ, subject to each training example having functional margin at least γ. The $||w|| = 1$ constraint moreover ensures that the functional margin equals to the geometric margin, so we are also guaranteed that all the geometric margins are at least γ. Thus, solving this problem will result in (w, b) with the largest possible geometric margin with respect to the training set.

If we could solve the optimization problem above, we'd be done. But the "$||w|| = 1$" constraint is a nasty (non-convex) one, and this problem certainly isn't in any format that we can plug into standard optimization software to solve. So, let's try transforming the problem into a nicer one. Consider:

$$\begin{aligned} \max_{\hat{\gamma},w,b} \quad & \frac{\hat{\gamma}}{||w||} \\ \text{s.t.} \quad & y^{(i)}(w^T x^{(i)} + b) \geq \hat{\gamma}, \quad i = 1, \ldots, n \end{aligned}$$

Here, we're going to maximize $\hat{\gamma}/||w||$, subject to the functional margins all being at least $\hat{\gamma}$. Since the geometric and functional margins are related by $\gamma = \hat{\gamma}/||w|$, this will give us the answer we want. Moreover, we've gotten rid of the constraint $||w|| = 1$ that we didn't like. The downside is that we now have a nasty (again, non-convex) objective $\frac{\hat{\gamma}}{||w||}$ function; and, we still don't have any off-the-shelf software that can solve this form of an optimization problem.

Let's keep going. Recall our earlier discussion that we can add an arbitrary scaling constraint on w and b without changing anything. This is the key idea we'll use now. We will introduce the scaling constraint that the functional margin of w, b with respect to the training set must be 1:

$$\hat{\gamma} = 1.$$

Since multiplying w and b by some constant results in the functional margin being multiplied by that same constant, this is indeed a scaling constraint, and can be satisfied by rescaling w, b. Plugging this into our problem above, and noting that maximizing $\hat{\gamma}/||w|| = 1/||w||$ is the same thing as minimizing $||w||^2$, we now have the following optimization problem:

$$\begin{aligned} \min_{w,b} \quad & \frac{1}{2}||w||^2 \\ \text{s.t.} \quad & y^{(i)}(w^T x^{(i)} + b) \geq 1, \quad i = 1, \ldots, n \end{aligned}$$

We've now transformed the problem into a form that can be efficiently solved. The above is an optimization problem with a convex quadratic objective and only linear constraints. Its solution gives us the **optimal margin classifier**. This optimization problem can be solved using commercial quadratic programming (QP) code.[4]

While we could call the problem solved here, what we will instead do is make a digression to talk about Lagrange duality. This will lead us to our optimization problem's dual form, which will play a key role in allowing us to use kernels to get optimal margin classifiers to work efficiently in very high dimensional spaces. The dual form will also allow us to derive an efficient algorithm for solving the above optimization problem that will typically do much better than generic QP software.

6 Lagrange duality (optional reading)

Let's temporarily put aside SVMs and maximum margin classifiers, and talk about solving constrained optimization problems.

Consider a problem of the following form:

$$\begin{aligned} \min_{w} \quad & f(w) \\ \text{s.t.} \quad & h_i(w) = 0, \quad i = 1, \ldots, l. \end{aligned}$$

Some of you may recall how the method of Lagrange multipliers can be used to solve it. (Don't worry if you haven't seen it before.) In this method, we define the **Lagrangian** to be

$$\mathcal{L}(w, \beta) = f(w) + \sum_{i=1}^{l} \beta_i h_i(w)$$

[4]You may be familiar with linear programming, which solves optimization problems that have linear objectives and linear constraints. QP software is also widely available, which allows convex quadratic objectives and linear constraints.

Here, the β_i's are called the **Lagrange multipliers**. We would then find and set $\mathcal{L}$'s partial derivatives to zero:

$$\frac{\partial \mathcal{L}}{\partial w_i} = 0; \quad \frac{\partial \mathcal{L}}{\partial \beta_i} = 0,$$

and solve for w and β.

In this section, we will generalize this to constrained optimization problems in which we may have inequality as well as equality constraints. Due to time constraints, we won't really be able to do the theory of Lagrange duality justice in this class,[5] but we will give the main ideas and results, which we will then apply to our optimal margin classifier's optimization problem.

Consider the following, which we'll call the **primal** optimization problem:

$$\begin{aligned} \min_w \quad & f(w) \\ \text{s.t.} \quad & g_i(w) \leq 0, \quad i = 1, \ldots, k \\ & h_i(w) = 0, \quad i = 1, \ldots, l. \end{aligned}$$

To solve it, we start by defining the **generalized Lagrangian**

$$\mathcal{L}(w, \alpha, \beta) = f(w) + \sum_{i=1}^{k} \alpha_i g_i(w) + \sum_{i=1}^{l} \beta_i h_i(w).$$

Here, the α_i's and β_i's are the Lagrange multipliers. Consider the quantity

$$\theta_{\mathcal{P}}(w) = \max_{\alpha, \beta \,:\, \alpha_i \geq 0} \mathcal{L}(w, \alpha, \beta).$$

Here, the "$\mathcal{P}$" subscript stands for "primal." Let some w be given. If w violates any of the primal constraints (i.e., if either $g_i(w) > 0$ or $h_i(w) \neq 0$ for some i), then you should be able to verify that

$$\begin{aligned} \theta_{\mathcal{P}}(w) &= \max_{\alpha, \beta \,:\, \alpha_i \geq 0} f(w) + \sum_{i=1}^{k} \alpha_i g_i(w) + \sum_{i=1}^{l} \beta_i h_i(w) \qquad (13) \\ &= \infty. \qquad (14) \end{aligned}$$

Conversely, if the constraints are indeed satisfied for a particular value of w, then $\theta_{\mathcal{P}}(w) = f(w)$. Hence,

$$\theta_{\mathcal{P}}(w) = \begin{cases} f(w) & \text{if } w \text{ satisfies primal constraints} \\ \infty & \text{otherwise.} \end{cases}$$

[5] Readers interested in learning more about this topic are encouraged to read, e.g., R. T. Rockarfeller (1970), Convex Analysis, Princeton University Press.

Thus, $\theta_{\mathcal{P}}$ takes the same value as the objective in our problem for all values of w that satisfies the primal constraints, and is positive infinity if the constraints are violated. Hence, if we consider the minimization problem

$$\min_w \theta_{\mathcal{P}}(w) = \min_w \max_{\alpha,\beta\,:\,\alpha_i \geq 0} \mathcal{L}(w, \alpha, \beta),$$

we see that it is the same problem (i.e., and has the same solutions as) our original, primal problem. For later use, we also define the optimal value of the objective to be $p^* = \min_w \theta_{\mathcal{P}}(w)$; we call this the **value** of the primal problem.

Now, let's look at a slightly different problem. We define

$$\theta_{\mathcal{D}}(\alpha, \beta) = \min_w \mathcal{L}(w, \alpha, \beta).$$

Here, the "$\mathcal{D}$" subscript stands for "dual." Note also that whereas in the definition of $\theta_{\mathcal{P}}$ we were optimizing (maximizing) with respect to α, β, here we are minimizing with respect to w.

We can now pose the **dual** optimization problem:

$$\max_{\alpha,\beta\,:\,\alpha_i \geq 0} \theta_{\mathcal{D}}(\alpha, \beta) = \max_{\alpha,\beta\,:\,\alpha_i \geq 0} \min_w \mathcal{L}(w, \alpha, \beta).$$

This is exactly the same as our primal problem shown above, except that the order of the "max" and the "min" are now exchanged. We also define the optimal value of the dual problem's objective to be $d^* = \max_{\alpha,\beta\,:\,\alpha_i \geq 0} \theta_{\mathcal{D}}(w)$.

How are the primal and the dual problems related? It can easily be shown that

$$d^* = \max_{\alpha,\beta\,:\,\alpha_i \geq 0} \min_w \mathcal{L}(w, \alpha, \beta) \leq \min_w \max_{\alpha,\beta\,:\,\alpha_i \geq 0} \mathcal{L}(w, \alpha, \beta) = p^*.$$

(You should convince yourself of this; this follows from the "max min" of a function always being less than or equal to the "min max.") However, under certain conditions, we will have

$$d^* = p^*,$$

so that we can solve the dual problem in lieu of the primal problem. Let's see what these conditions are.

Suppose f and the g_i's are convex,[6] and the h_i's are affine.[7] Suppose further that the constraints g_i are (strictly) feasible; this means that there exists some w so that $g_i(w) < 0$ for all i.

[6]When f has a Hessian, then it is convex if and only if the Hessian is positive semi-definite. For instance, $f(w) = w^T w$ is convex; similarly, all linear (and affine) functions are also convex. (A function f can also be convex without being differentiable, but we won't need those more general definitions of convexity here.)

[7]I.e., there exists a_i, b_i, so that $h_i(w) = a_i^T w + b_i$. "Affine" means the same thing as linear, except that we also allow the extra intercept term b_i.

Under our above assumptions, there must exist w^*, α^*, β^* so that w^* is the solution to the primal problem, α^*, β^* are the solution to the dual problem, and moreover $p^* = d^* = \mathcal{L}(w^*, \alpha^*, \beta^*)$. Moreover, w^*, α^* and β^* satisfy the **Karush-Kuhn-Tucker (KKT) conditions**, which are as follows:

$$\begin{aligned}
\frac{\partial}{\partial w_i}\mathcal{L}(w^*, \alpha^*, \beta^*) &= 0, \quad i = 1, \ldots, d && (15)\\
\frac{\partial}{\partial \beta_i}\mathcal{L}(w^*, \alpha^*, \beta^*) &= 0, \quad i = 1, \ldots, l && (16)\\
\alpha_i^* g_i(w^*) &= 0, \quad i = 1, \ldots, k && (17)\\
g_i(w^*) &\leq 0, \quad i = 1, \ldots, k && (18)\\
\alpha^* &\geq 0, \quad i = 1, \ldots, k && (19)
\end{aligned}$$

Moreover, if some w^*, α^*, β^* satisfy the KKT conditions, then it is also a solution to the primal and dual problems.

We draw attention to Equation (17), which is called the KKT **dual complementarity** condition. Specifically, it implies that if $\alpha_i^* > 0$, then $g_i(w^*) = 0$. (I.e., the "$g_i(w) \leq 0$" constraint is **active**, meaning it holds with equality rather than with inequality.) Later on, this will be key for showing that the SVM has only a small number of "support vectors"; the KKT dual complementarity condition will also give us our convergence test when we talk about the SMO algorithm.

7 Optimal margin classifiers

Note: *The equivalence of optimization problem* (20) *and the optimization problem* (24)*, and the relationship between the primary and dual variables in equation* (22) *are the most important take home messages of this section.*

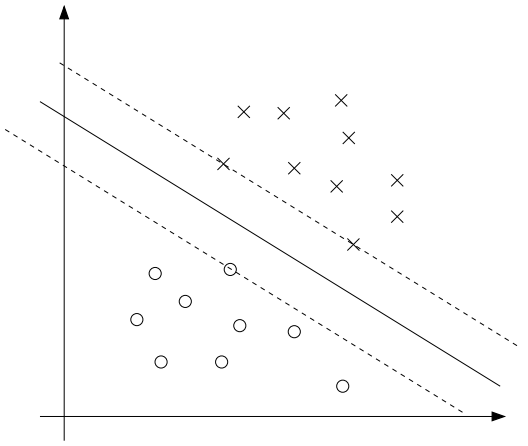
Previously, we posed the following (primal) optimization problem for finding the optimal margin classifier:

$$\begin{aligned}
\min_{w,b} \quad & \frac{1}{2}||w||^2 && (20)\\
\text{s.t.} \quad & y^{(i)}(w^T x^{(i)} + b) \geq 1, \quad i = 1, \ldots, n
\end{aligned}$$

We can write the constraints as

$$g_i(w) = -y^{(i)}(w^T x^{(i)} + b) + 1 \leq 0.$$

We have one such constraint for each training example. Note that from the KKT dual complementarity condition, we will have $\alpha_i > 0$ only for the training examples that have functional margin exactly equal to one (i.e., the ones corresponding to constraints that hold with equality, $g_i(w) = 0$). Consider the figure below, in which a maximum margin separating hyperplane is shown by the solid line.



The points with the smallest margins are exactly the ones closest to the decision boundary; here, these are the three points (one negative and two positive examples) that lie on the dashed lines parallel to the decision boundary. Thus, only three of the α_i's—namely, the ones corresponding to these three training examples—will be non-zero at the optimal solution to our optimization problem. These three points are called the **support vectors** in this problem. The fact that the number of support vectors can be much smaller than the size the training set will be useful later.

Let's move on. Looking ahead, as we develop the dual form of the problem, one key idea to watch out for is that we'll try to write our algorithm in terms of only the inner product $\langle x^{(i)}, x^{(j)} \rangle$ (think of this as $(x^{(i)})^T x^{(j)}$) between points in the input feature space. The fact that we can express our algorithm in terms of these inner products will be key when we apply the kernel trick.

When we construct the Lagrangian for our optimization problem we have:

$$\mathcal{L}(w, b, \alpha) = \frac{1}{2}||w||^2 - \sum_{i=1}^{n} \alpha_i \left[y^{(i)}(w^T x^{(i)} + b) - 1 \right]. \tag{21}$$

Note that there're only "α_i" but no "β_i" Lagrange multipliers, since the problem has only inequality constraints.

Let's find the dual form of the problem. To do so, we need to first minimize $\mathcal{L}(w, b, \alpha)$ with respect to w and b (for fixed α), to get $\theta_{\mathcal{D}}$, which we'll do by setting the derivatives of $\mathcal{L}$ with respect to w and b to zero. We have:

$$\nabla_w \mathcal{L}(w, b, \alpha) = w - \sum_{i=1}^{n} \alpha_i y^{(i)} x^{(i)} = 0$$

This implies that

$$w = \sum_{i=1}^{n} \alpha_i y^{(i)} x^{(i)}. \tag{22}$$

As for the derivative with respect to b, we obtain

$$\frac{\partial}{\partial b} \mathcal{L}(w, b, \alpha) = \sum_{i=1}^{n} \alpha_i y^{(i)} = 0. \tag{23}$$

If we take the definition of w in Equation (22) and plug that back into the Lagrangian (Equation 21), and simplify, we get

$$\mathcal{L}(w, b, \alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} y^{(i)} y^{(j)} \alpha_i \alpha_j (x^{(i)})^T x^{(j)} - b \sum_{i=1}^{n} \alpha_i y^{(i)}.$$

But from Equation (23), the last term must be zero, so we obtain

$$\mathcal{L}(w, b, \alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} y^{(i)} y^{(j)} \alpha_i \alpha_j (x^{(i)})^T x^{(j)}.$$

Recall that we got to the equation above by minimizing $\mathcal{L}$ with respect to w and b. Putting this together with the constraints $\alpha_i \geq 0$ (that we always had) and the constraint (23), we obtain the following dual optimization problem:

$$\begin{aligned} \max_\alpha \quad & W(\alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} y^{(i)} y^{(j)} \alpha_i \alpha_j \langle x^{(i)}, x^{(j)} \rangle. \\ \text{s.t.} \quad & \alpha_i \geq 0, \quad i = 1, \ldots, n \\ & \sum_{i=1}^{n} \alpha_i y^{(i)} = 0, \end{aligned} \tag{24}$$

You should also be able to verify that the conditions required for $p^* = d^*$ and the KKT conditions (Equations 15–19) to hold are indeed satisfied in our optimization problem. Hence, we can solve the dual in lieu of solving

the primal problem. Specifically, in the dual problem above, we have a maximization problem in which the parameters are the α_i's. We'll talk later about the specific algorithm that we're going to use to solve the dual problem, but if we are indeed able to solve it (i.e., find the α's that maximize $W(\alpha)$ subject to the constraints), then we can use Equation (22) to go back and find the optimal w's as a function of the α's. Having found w^*, by considering the primal problem, it is also straightforward to find the optimal value for the intercept term b as

$$b^* = -\frac{\max_{i:y^{(i)}=-1} w^{*T} x^{(i)} + \min_{i:y^{(i)}=1} w^{*T} x^{(i)}}{2}. \tag{25}$$

(Check for yourself that this is correct.)

Before moving on, let's also take a more careful look at Equation (22), which gives the optimal value of w in terms of (the optimal value of) α. Suppose we've fit our model's parameters to a training set, and now wish to make a prediction at a new point input x. We would then calculate $w^T x + b$, and predict $y = 1$ if and only if this quantity is bigger than zero. But using (22), this quantity can also be written:

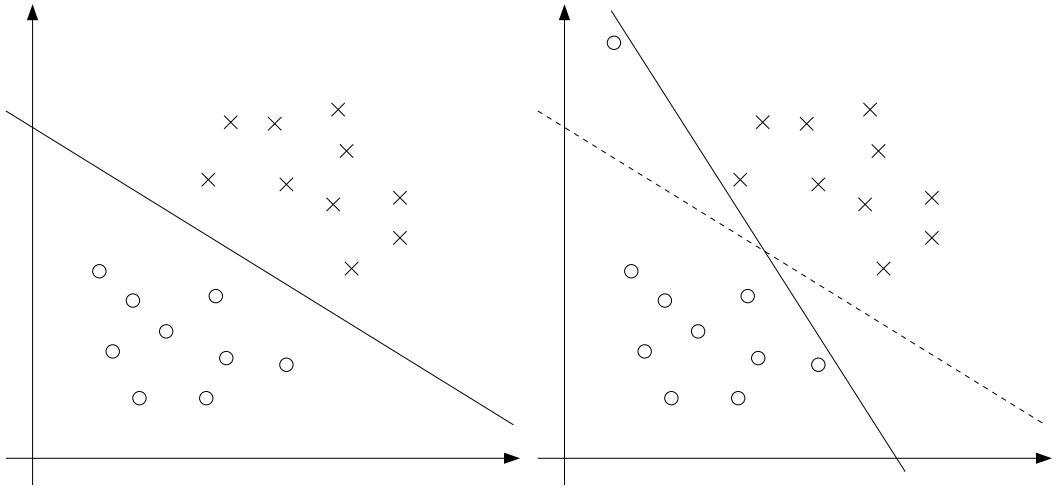
$$\begin{aligned} w^T x + b &= \left(\sum_{i=1}^{n} \alpha_i y^{(i)} x^{(i)} \right)^T x + b && (26) \\ &= \sum_{i=1}^{n} \alpha_i y^{(i)} \langle x^{(i)}, x \rangle + b. && (27) \end{aligned}$$

Hence, if we've found the α_i's, in order to make a prediction, we have to calculate a quantity that depends only on the inner product between x and the points in the training set. Moreover, we saw earlier that the α_i's will all be zero except for the support vectors. Thus, many of the terms in the sum above will be zero, and we really need to find only the inner products between x and the support vectors (of which there is often only a small number) in order calculate (27) and make our prediction.

By examining the dual form of the optimization problem, we gained significant insight into the structure of the problem, and were also able to write the entire algorithm in terms of only inner products between input feature vectors. In the next section, we will exploit this property to apply the kernels to our classification problem. The resulting algorithm, **support vector machines**, will be able to efficiently learn in very high dimensional spaces.

8 Regularization and the non-separable case (optional reading)

The derivation of the SVM as presented so far assumed that the data is linearly separable. While mapping data to a high dimensional feature space via ϕ does generally increase the likelihood that the data is separable, we can't guarantee that it always will be so. Also, in some cases it is not clear that finding a separating hyperplane is exactly what we'd want to do, since that might be susceptible to outliers. For instance, the left figure below shows an optimal margin classifier, and when a single outlier is added in the upper-left region (right figure), it causes the decision boundary to make a dramatic swing, and the resulting classifier has a much smaller margin.



To make the algorithm work for non-linearly separable datasets as well as be less sensitive to outliers, we reformulate our optimization (using ℓ_1 **regularization**) as follows:

$$\begin{aligned} \min_{\gamma,w,b} \quad & \frac{1}{2}||w||^2 + C\sum_{i=1}^{n}\xi_i \\ \text{s.t.} \quad & y^{(i)}(w^T x^{(i)} + b) \geq 1 - \xi_i, \quad i = 1, \ldots, n \\ & \xi_i \geq 0, \quad i = 1, \ldots, n. \end{aligned}$$

Thus, examples are now permitted to have (functional) margin less than 1, and if an example has functional margin $1 - \xi_i$ (with $\xi > 0$), we would pay a cost of the objective function being increased by $C\xi_i$. The parameter C

controls the relative weighting between the twin goals of making the $||w||^2$ small (which we saw earlier makes the margin large) and of ensuring that most examples have functional margin at least 1.

As before, we can form the Lagrangian:

$$\mathcal{L}(w, b, \xi, \alpha, r) = \frac{1}{2} w^T w + C \sum_{i=1}^{n} \xi_i - \sum_{i=1}^{n} \alpha_i \left[y^{(i)}(x^T w + b) - 1 + \xi_i \right] - \sum_{i=1}^{n} r_i \xi_i.$$

Here, the α_i's and r_i's are our Lagrange multipliers (constrained to be ≥ 0). We won't go through the derivation of the dual again in detail, but after setting the derivatives with respect to w and b to zero as before, substituting them back in, and simplifying, we obtain the following dual form of the problem:

$$\begin{aligned} \max_\alpha \quad & W(\alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} y^{(i)} y^{(j)} \alpha_i \alpha_j \langle x^{(i)}, x^{(j)} \rangle \\ \text{s.t.} \quad & 0 \leq \alpha_i \leq C, \quad i = 1, \ldots, n \\ & \sum_{i=1}^{n} \alpha_i y^{(i)} = 0, \end{aligned}$$

As before, we also have that w can be expressed in terms of the α_i's as given in Equation (22), so that after solving the dual problem, we can continue to use Equation (27) to make our predictions. Note that, somewhat surprisingly, in adding ℓ_1 regularization, the only change to the dual problem is that what was originally a constraint that $0 \leq \alpha_i$ has now become $0 \leq \alpha_i \leq C$. The calculation for b^* also has to be modified (Equation 25 is no longer valid); see the comments in the next section/Platt's paper.

Also, the KKT dual-complementarity conditions (which in the next section will be useful for testing for the convergence of the SMO algorithm) are:

$$\alpha_i = 0 \quad \Rightarrow \quad y^{(i)}(w^T x^{(i)} + b) \geq 1 \tag{28}$$

$$\alpha_i = C \quad \Rightarrow \quad y^{(i)}(w^T x^{(i)} + b) \leq 1 \tag{29}$$

$$0 < \alpha_i < C \quad \Rightarrow \quad y^{(i)}(w^T x^{(i)} + b) = 1. \tag{30}$$

Now, all that remains is to give an algorithm for actually solving the dual problem, which we will do in the next section.

9 The SMO algorithm (optional reading)

The SMO (sequential minimal optimization) algorithm, due to John Platt, gives an efficient way of solving the dual problem arising from the derivation of the SVM. Partly to motivate the SMO algorithm, and partly because it's interesting in its own right, let's first take another digression to talk about the coordinate ascent algorithm.

9.1 Coordinate ascent

Consider trying to solve the unconstrained optimization problem

$$\max_{\alpha} W(\alpha_1, \alpha_2, \ldots, \alpha_n).$$

Here, we think of W as just some function of the parameters α_i's, and for now ignore any relationship between this problem and SVMs. We've already seen two optimization algorithms, gradient ascent and Newton's method. The new algorithm we're going to consider here is called **coordinate ascent**:

Loop until convergence: {

For $i = 1, \ldots, n$, {

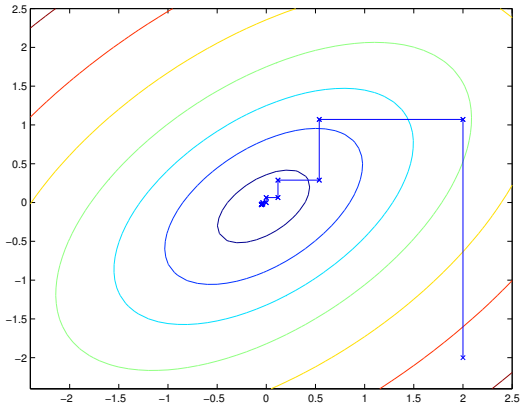
$\alpha_i := \arg\max_{\hat{\alpha}_i} W(\alpha_1, \ldots, \alpha_{i-1}, \hat{\alpha}_i, \alpha_{i+1}, \ldots, \alpha_n)$.

}

}

Thus, in the innermost loop of this algorithm, we will hold all the variables except for some α_i fixed, and reoptimize W with respect to just the parameter α_i. In the version of this method presented here, the inner-loop reoptimizes the variables in order $\alpha_1, \alpha_2, \ldots, \alpha_n, \alpha_1, \alpha_2, \ldots$. (A more sophisticated version might choose other orderings; for instance, we may choose the next variable to update according to which one we expect to allow us to make the largest increase in $W(\alpha)$.)

When the function W happens to be of such a form that the "arg max" in the inner loop can be performed efficiently, then coordinate ascent can be a fairly efficient algorithm. Here's a picture of coordinate ascent in action:



The ellipses in the figure are the contours of a quadratic function that we want to optimize. Coordinate ascent was initialized at $(2, -2)$, and also plotted in the figure is the path that it took on its way to the global maximum. Notice that on each step, coordinate ascent takes a step that's parallel to one of the axes, since only one variable is being optimized at a time.

9.2 SMO

We close off the discussion of SVMs by sketching the derivation of the SMO algorithm. Some details will be left to the homework, and for others you may refer to the paper excerpt handed out in class.

Here's the (dual) optimization problem that we want to solve:

$$\max_\alpha \quad W(\alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} y^{(i)} y^{(j)} \alpha_i \alpha_j \langle x^{(i)}, x^{(j)} \rangle. \tag{31}$$

$$\text{s.t.} \quad 0 \leq \alpha_i \leq C, \quad i = 1, \ldots, n \tag{32}$$

$$\sum_{i=1}^{n} \alpha_i y^{(i)} = 0. \tag{33}$$

Let's say we have set of α_i's that satisfy the constraints (32-33). Now, suppose we want to hold $\alpha_2, \ldots, \alpha_n$ fixed, and take a coordinate ascent step and reoptimize the objective with respect to α_1. Can we make any progress? The answer is no, because the constraint (33) ensures that

$$\alpha_1 y^{(1)} = -\sum_{i=2}^{n} \alpha_i y^{(i)}.$$

Or, by multiplying both sides by $y^{(1)}$, we equivalently have

$$\alpha_1 = -y^{(1)} \sum_{i=2}^{n} \alpha_i y^{(i)}.$$

(This step used the fact that $y^{(1)} \in \{-1, 1\}$, and hence $(y^{(1)})^2 = 1$.) Hence, α_1 is exactly determined by the other α_i's, and if we were to hold $\alpha_2, \ldots, \alpha_n$ fixed, then we can't make any change to α_1 without violating the constraint (33) in the optimization problem.

Thus, if we want to update some subject of the α_i's, we must update at least two of them simultaneously in order to keep satisfying the constraints. This motivates the SMO algorithm, which simply does the following:

Repeat till convergence {

1. Select some pair α_i and α_j to update next (using a heuristic that tries to pick the two that will allow us to make the biggest progress towards the global maximum).
2. Reoptimize $W(\alpha)$ with respect to α_i and α_j, while holding all the other α_k's $(k \neq i, j)$ fixed.

}

To test for convergence of this algorithm, we can check whether the KKT conditions (Equations 28-30) are satisfied to within some *tol*. Here, *tol* is the convergence tolerance parameter, and is typically set to around 0.01 to 0.001. (See the paper and pseudocode for details.)

The key reason that SMO is an efficient algorithm is that the update to α_i, α_j can be computed very efficiently. Let's now briefly sketch the main ideas for deriving the efficient update.

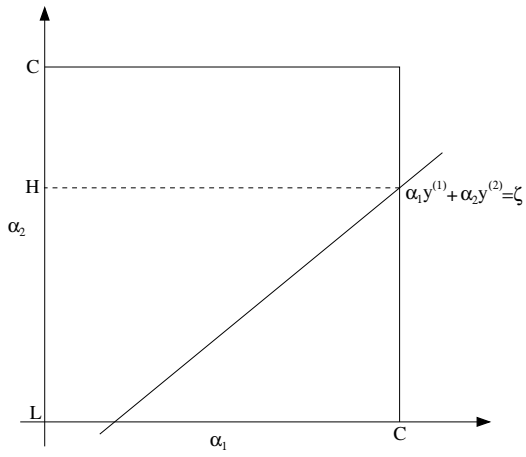
Let's say we currently have some setting of the α_i's that satisfy the constraints (32-33), and suppose we've decided to hold $\alpha_3, \ldots, \alpha_n$ fixed, and want to reoptimize $W(\alpha_1, \alpha_2, \ldots, \alpha_n)$ with respect to α_1 and α_2 (subject to the constraints). From (33), we require that

$$\alpha_1 y^{(1)} + \alpha_2 y^{(2)} = -\sum_{i=3}^{n} \alpha_i y^{(i)}.$$

Since the right hand side is fixed (as we've fixed $\alpha_3, \ldots \alpha_n$), we can just let it be denoted by some constant ζ:

$$\alpha_1 y^{(1)} + \alpha_2 y^{(2)} = \zeta. \tag{34}$$

We can thus picture the constraints on α_1 and α_2 as follows:



From the constraints (32), we know that α_1 and α_2 must lie within the box $[0, C] \times [0, C]$ shown. Also plotted is the line $\alpha_1 y^{(1)} + \alpha_2 y^{(2)} = \zeta$, on which we know α_1 and α_2 must lie. Note also that, from these constraints, we know $L \leq \alpha_2 \leq H$; otherwise, (α_1, α_2) can't simultaneously satisfy both the box and the straight line constraint. In this example, $L = 0$. But depending on what the line $\alpha_1 y^{(1)} + \alpha_2 y^{(2)} = \zeta$ looks like, this won't always necessarily be the case; but more generally, there will be some lower-bound L and some upper-bound H on the permissible values for α_2 that will ensure that α_1, α_2 lie within the box $[0, C] \times [0, C]$.

Using Equation (34), we can also write α_1 as a function of α_2:

$$\alpha_1 = (\zeta - \alpha_2 y^{(2)}) y^{(1)}.$$

(Check this derivation yourself; we again used the fact that $y^{(1)} \in \{-1, 1\}$ so that $(y^{(1)})^2 = 1$.) Hence, the objective $W(\alpha)$ can be written

$$W(\alpha_1, \alpha_2, \ldots, \alpha_n) = W((\zeta - \alpha_2 y^{(2)}) y^{(1)}, \alpha_2, \ldots, \alpha_n).$$

Treating $\alpha_3, \ldots, \alpha_n$ as constants, you should be able to verify that this is just some quadratic function in α_2. I.e., this can also be expressed in the form $a\alpha_2^2 + b\alpha_2 + c$ for some appropriate a, b, and c. If we ignore the "box" constraints (32) (or, equivalently, that $L \leq \alpha_2 \leq H$), then we can easily maximize this quadratic function by setting its derivative to zero and solving. We'll let $\alpha_2^{new,unclipped}$ denote the resulting value of α_2. You should also be able to convince yourself that if we had instead wanted to maximize W with respect to α_2 but subject to the box constraint, then we can find the resulting value optimal simply by taking $\alpha_2^{new,unclipped}$ and "clipping" it to lie in the

$[L, H]$ interval, to get

$$\alpha_2^{new} = \begin{cases} H & \text{if } \alpha_2^{new,unclipped} > H \\ \alpha_2^{new,unclipped} & \text{if } L \leq \alpha_2^{new,unclipped} \leq H \\ L & \text{if } \alpha_2^{new,unclipped} < L \end{cases}$$

Finally, having found the α_2^{new}, we can use Equation (34) to go back and find the optimal value of α_1^{new}.

There're a couple more details that are quite easy but that we'll leave you to read about yourself in Platt's paper: One is the choice of the heuristics used to select the next α_i, α_j to update; the other is how to update b as the SMO algorithm is run.