9 | Kernel Methods

--

Learning Objectives:

- Explain how kernels generalize both feature combinations and basis functions.
- Contrast dot products with kernel products.
- Implement kernelized perceptron.
- Derive a kernelized version of regularized least squares regression.
- Implement a kernelized version of the perceptron.
- Derive the dual formulation of the support vector machine.

Dependencies:

Linear models are great because they are easy to understand and easy to optimize. They suffer because they can only learn very simple decision boundaries. Neural networks can learn more complex decision boundaries, but lose the nice convexity properties of many linear models.

One way of getting a linear model to behave non-linearly is to transform the input. For instance, by adding feature pairs as additional inputs. Learning a linear model on such a representation is convex, but is computationally prohibitive in all but very low dimensional spaces. You might ask: instead of *explicitly* expanding the feature space, is it possible to stay with our original data representation and do all the feature blow up *implicitly*? Surprisingly, the answer is often "yes" and the family of techniques that makes this possible are known as **kernel** approaches.

9.1 From Feature Combinations to Kernels

In Section 4.4, you learned one method for increasing the expressive power of linear models: explode the feature space. For instance, a "quadratic" feature explosion might map a feature vector $\boldsymbol{x} = \langle x_1, x_2, x_3, \ldots, x_D \rangle$ to an expanded version denoted $\phi(\boldsymbol{x})$:

$$\begin{aligned}\phi(\boldsymbol{x}) = \langle & 1, 2x_1, 2x_2, 2x_3, \ldots, 2x_D, \\ & x_1^2, x_1x_2, x_1x_3, \ldots, x_1x_D, \\ & x_2x_1, x_2^2, x_2x_3, \ldots, x_2x_D, \\ & x_3x_1, x_3x_2, x_3^2, \ldots, x_2x_D, \\ & \ldots, \\ & x_Dx_1, x_Dx_2, x_Dx_3, \ldots, x_D^2 \rangle \end{aligned} \tag{9.1}$$

(Note that there are repetitions here, but hopefully most learning algorithms can deal well with redundant features; in particular, the $2x_1$ terms are due to collapsing some repetitions.)

You could then train a classifier on this expanded feature space. There are two primary concerns in doing so. The first is computa-

tional: if your learning algorithm scales linearly in the number of features, then you've just squared the amount of computation you need to perform; you've also squared the amount of memory you'll need. The second is statistical: if you go by the heuristic that you should have about two examples for every feature, then you will now need quadratically many training examples in order to avoid overfitting.

This chapter is all about dealing with the *computational* issue. It will turn out in Chapter **??** that you can also deal with the statistical issue: for now, you can just hope that regularization will be sufficient to attenuate overfitting.

The key insight in kernel-based learning is that you can *rewrite* many linear models in a way that doesn't require you to ever *explicitly* compute $\phi(\boldsymbol{x})$. To start with, you can think of this purely as a computational "trick" that enables you to use the power of a quadratic feature mapping without actually having to compute and store the mapped vectors. Later, you will see that it's actually quite a bit deeper. Most algorithms we discuss involve a product of the form $\boldsymbol{w} \cdot \phi(\boldsymbol{x})$, after performing the feature mapping. The goal is to rewrite these algorithms so that they only ever depend on dot products between two examples, say $\boldsymbol{x}$ and $\boldsymbol{z}$; namely, they depend on $\phi(\boldsymbol{x}) \cdot \phi(\boldsymbol{z})$. To understand why this is helpful, consider the quadratic expansion from above, and the dot-product between two vectors. You get:

$$\begin{aligned}
\phi(\boldsymbol{x}) \cdot \phi(\boldsymbol{z}) &= 1 + x_1 z_1 + x_2 z_2 + \cdots + x_D z_D + x_1^2 z_1^2 + \cdots + x_1 x_D z_1 z_D + \\
&\quad \cdots + x_D x_1 z_D z_1 + x_D x_2 z_D z_2 + \cdots + x_D^2 z_D^2 && (9.2)\\
&= 1 + 2\sum_d x_d z_d + \sum_d \sum_e x_d x_e z_d z_e && (9.3)\\
&= 1 + 2\boldsymbol{x} \cdot \boldsymbol{z} + (\boldsymbol{x} \cdot \boldsymbol{z})^2 && (9.4)\\
&= (1 + \boldsymbol{x} \cdot \boldsymbol{z})^2 && (9.5)
\end{aligned}$$

Thus, you can compute $\phi(\boldsymbol{x}) \cdot \phi(\boldsymbol{z})$ in exactly the same amount of time as you can compute $\boldsymbol{x} \cdot \boldsymbol{z}$ (plus the time it takes to perform an addition and a multiply, about 0.02 nanoseconds on a circa 2011 processor).

The rest of the practical challenge is to rewrite your algorithms so that they only depend on dot products between examples and not on any explicit weight vectors.

9.2 Kernelized Perceptron

Consider the original perceptron algorithm from Chapter 3, repeated in Algorithm 9.2 using linear algebra notation and using feature expansion notation $\phi(\boldsymbol{x})$. In this algorithm, there are two places where $\phi(\boldsymbol{x})$ is used explicitly. The first is in computing the activation

Algorithm 28 PERCEPTRONTRAIN($\mathbf{D}$, *MaxIter*)

```
1:  w ← 0, b ← 0                                   // initialize weights and bias
2:  for iter = 1 ... MaxIter do
3:      for all (x,y) ∈ D do
4:          a ← w · φ(x) + b                       // compute activation for this example
5:          if ya ≤ 0 then
6:              w ← w + y φ(x)                     // update weights
7:              b ← b + y                          // update bias
8:          end if
9:      end for
10: end for
11: return w, b
```

MATH REVIEW | SPANS AND NULL SPACES

reminder: if $\mathcal{U} = \{\boldsymbol{u}_i\}_i$ is a set of vectors in $\mathbb{R}^D$, then the span of $\mathcal{U}$ is the set of vectors that can be written as linear combinations of $\boldsymbol{u}_i$s; namely: $span(\mathcal{U}) = \{\sum_i a_i \boldsymbol{u}_i : a_1 \in \mathbb{R}, \ldots, a_I \in \mathbb{R}\}$.
the null space of $\mathcal{U}$ is everything that's left: $\mathbb{R}^D \backslash span(\mathcal{U})$.
TODO pictures

Figure 9.1:

(line 4) and the second is in updating the weights (line 6). The goal is to remove the explicit dependence of this algorithm on ϕ and on the weight vector.

To do so, you can observe that at any point in the algorithm, the weight vector $\boldsymbol{w}$ can be written as a linear combination of expanded training data. In particular, at any point, $\boldsymbol{w} = \sum_n \alpha_n \phi(\boldsymbol{x}_n)$ for some parameters α. Initially, $\boldsymbol{w} = \mathbf{0}$ so choosing $\boldsymbol{\alpha} = 0$ yields this. If the first update occurs on the nth training example, then the resolution weight vector is simply $y_n \phi(\boldsymbol{x}_n)$, which is equivalent to setting $\alpha_n = y_n$. If the second update occurs on the mth training example, then all you need to do is update $\alpha_m \leftarrow \alpha_m + y_m$. This is true, even if you make multiple passes over the data. This observation leads to the following **representer theorem**, which states that the weight vector of the perceptron lies in the **span** of the training data.

Theorem 11 (Perceptron Representer Theorem)**.** *During a run of the perceptron algorithm, the weight vector $\boldsymbol{w}$ is always in the span of the (assumed non-empty) training data, $\phi(\boldsymbol{x}_1), \ldots, \phi(\boldsymbol{x}_N)$.*

Proof of Theorem 11. By induction. Base case: the span of any non-empty set contains the zero vector, which is the initial weight vector. Inductive case: suppose that the theorem is true before the kth update, and suppose that the kth update happens on example n. By the inductive hypothesis, you can write $\boldsymbol{w} = \sum_i \alpha_i \phi(\boldsymbol{x}_i)$ before the update. The new weight vector is $[\sum_i \alpha_i \phi(\boldsymbol{x}_i)] + y_n \phi(\boldsymbol{x}_n) =$

Algorithm 29 KernelizedPerceptronTrain(**D**, *MaxIter*)

```
1:  α ← 0, b ← 0                                   // initialize coefficients and bias
2:  for iter = 1 ... MaxIter do
3:      for all (x_n, y_n) ∈ D do
4:          a ← Σ_m α_m φ(x_m) · φ(x_n) + b        // compute activation for this example
5:          if y_n a ≤ 0 then
6:              α_n ← α_n + y_n                    // update coefficients
7:              b ← b + y                          // update bias
8:          end if
9:      end for
10: end for
11: return α, b
```

$\sum_i (\alpha_i + y_n[i = n])\phi(\boldsymbol{x}_i)$, which is still in the span of the training data. □

Now that you know that you can always write $\boldsymbol{w} = \sum_n \alpha_n \phi(\boldsymbol{x}_n)$ for some α_is, you can additionall compute the activations (line 4) as:

$$\boldsymbol{w} \cdot \phi(\boldsymbol{x}) + b = \left(\sum_n \alpha_n \phi(\boldsymbol{x}_n) \right) \cdot \phi(\boldsymbol{x}) + b \qquad \text{definition of } \boldsymbol{w} \tag{9.6}$$

$$= \sum_n \alpha_n \Big[\phi(\boldsymbol{x}_n) \cdot \phi(\boldsymbol{x}) \Big] + b \qquad \text{dot products are linear} \tag{9.7}$$

This now depends only on dot-products between data points, and never explicitly requires a weight vector. You can now rewrite the entire perceptron algorithm so that it never refers explicitly to the weights and only ever depends on pairwise dot products between examples. This is shown in Algorithm 9.2.

The advantage to this "kernelized" algorithm is that you can perform feature expansions like the quadratic feature expansion from the introduction for "free." For example, for exactly the same cost as the quadratic features, you can use a **cubic feature map**, computed as $\phi(\boldsymbol{x})\phi(\boldsymbol{z}) = (1 + \boldsymbol{x} \cdot \boldsymbol{z})^3$, which corresponds to three-way interactions between variables. (And, in general, you can do so for any polynomial degree p at the same computational complexity.)

9.3 Kernelized K-means

For a complete change of pace, consider the K-means algorithm from Section **??**. This algorithm is for *clustering* where there is no notion of "training labels." Instead, you want to partition the data into coherent clusters. For data in $\mathbb{R}^D$, it involves randomly initializing K-many cluster means $\boldsymbol{\mu}^{(1)}, \ldots, \boldsymbol{\mu}^{(K)}$. The algorithm then alternates between the

following two steps until convergence, with $\boldsymbol{x}$ replaced by $\phi(\boldsymbol{x})$ since that is the eventual goal:

1. For each example n, set cluster label $z_n = \arg\min_k \left|\left|\phi(\boldsymbol{x}_n) - \boldsymbol{\mu}^{(k)}\right|\right|^2$.
2. For each cluster k, update $\boldsymbol{\mu}^{(k)} = \frac{1}{N_k} \sum_{n:z_n=k} \phi(\boldsymbol{x}_n)$, where N_k is the number of n with $z_n = k$.

The question is whether you can perform these steps without explicitly computing $\phi(\boldsymbol{x}_n)$. The **representer theorem** is more straightforward here than in the perceptron. The mean of a set of data is, almost by definition, in the span of that data (choose the a_is all to be equal to $1/N$). Thus, so long as you *initialize* the means in the span of the data, you are guaranteed always to have the means in the span of the data. Given this, you know that you can write each mean as an expansion of the data; say that $\boldsymbol{\mu}^{(k)} = \sum_n \alpha_n^{(k)} \phi(\boldsymbol{x}_n)$ for some parameters $\alpha_n^{(k)}$ (there are $N{\times}K$-many such parameters).

Given this expansion, in order to execute step (1), you need to compute norms. This can be done as follows:

$$z_n = \arg\min_k \left|\left|\phi(\boldsymbol{x}_n) - \boldsymbol{\mu}^{(k)}\right|\right|^2 \qquad \text{definition of } z_n \tag{9.8}$$

$$= \arg\min_k \left|\left|\phi(\boldsymbol{x}_n) - \sum_m \alpha_m^{(k)} \phi(\boldsymbol{x}_m)\right|\right|^2 \qquad \text{definition of } \boldsymbol{\mu}^{(k)} \tag{9.9}$$

$$= \arg\min_k ||\phi(\boldsymbol{x}_n)||^2 + \left|\left|\sum_m \alpha_m^{(k)} \phi(\boldsymbol{x}_m)\right|\right|^2 + \phi(\boldsymbol{x}_n) \cdot \left[\sum_m \alpha_m^{(k)} \phi(\boldsymbol{x}_m)\right] \qquad \text{expand quadratic term} \tag{9.10}$$

$$= \arg\min_k \sum_m \sum_{m'} \alpha_m^{(k)} \alpha_{m'}^{(k)} \phi(\boldsymbol{x}_m) \cdot \phi(\boldsymbol{x}_{m'}) + \sum_m \alpha_m^{(k)} \phi(\boldsymbol{x}_m) \cdot \phi(\boldsymbol{x}_n) + const \qquad \text{linearity and constant} \tag{9.11}$$

This computation can replace the assignments in step (1) of K-means. The mean updates are more direct in step (2):

$$\boldsymbol{\mu}^{(k)} = \frac{1}{N_k} \sum_{n:z_n=k} \phi(\boldsymbol{x}_n) \quad \Longleftrightarrow \quad \alpha_n^{(k)} = \begin{cases} \frac{1}{N_k} & \text{if } z_n = k \\ 0 & \text{otherwise} \end{cases} \tag{9.12}$$

9.4 What Makes a Kernel

A **kernel** is just a form of generalized dot product. You can also think of it as simply shorthand for $\phi(\boldsymbol{x}) \cdot \phi(\boldsymbol{z})$, which is commonly written $K^\phi(\boldsymbol{x}, \boldsymbol{z})$. Or, when ϕ is clear from context, simply $K(\boldsymbol{x}, \boldsymbol{z})$.

This is often refered to as the kernel product between $\boldsymbol{x}$ and $\boldsymbol{z}$ (under the mapping ϕ).

In this view, what you've seen in the preceding two sections is that you can rewrite both the perceptron algorithm and the K-means algorithm so that *they only ever depend on kernel products between data points, and never on the actual datapoints themselves*. This is a very powerful notion, as it has enabled the development of a large number of non-linear algorithms essentially "for free" (by applying the so-called **kernel trick**, that you've just seen twice).

This raises an interesting question. If you have rewritten these algorithms so that they only depend on the data through a function $K : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$, can you stick *any* function K in these algorithms, or are there some K that are "forbidden?" In one sense, you "could" use any K, but the real question is: for what types of functions K do these algorithms retain the properties that we expect them to have (like convergence, optimality, etc.)?

One way to answer this question is to say that $K(\cdot, \cdot)$ is a valid kernel if it corresponds to the inner product between two vectors. That is, K is valid if there exists a function ϕ such that $K(\boldsymbol{x}, \boldsymbol{z}) = \phi(\boldsymbol{x}) \cdot \phi(\boldsymbol{z})$. This is a direct definition and it should be clear that if K satisfies this, then the algorithms go through as expected (because this is how we derived them).

You've already seen the general class of **polynomial kernels**, which have the form:

$$K_d^{(\text{poly})}(\boldsymbol{x}, \boldsymbol{z}) = \Big(1 + \boldsymbol{x} \cdot \boldsymbol{z}\Big)^d \tag{9.13}$$

where d is a hyperparameter of the kernel. These kernels correspond to polynomial feature expansions.

There is an alternative characterization of a valid kernel function that is more mathematical. It states that $K : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a kernel if K is **positive semi-definite** (or, in shorthand, **psd**). This property is also sometimes called **Mercer's condition**. In this context, this means the *for all* functions f that are square integrable (i.e., $\int f(\boldsymbol{x})^2 \mathrm{d}\boldsymbol{x} < \infty$), other than the zero function, the following property holds:

$$\iint f(\boldsymbol{x}) K(\boldsymbol{x}, \boldsymbol{z}) f(\boldsymbol{z}) \mathrm{d}\boldsymbol{x} \mathrm{d}\boldsymbol{z} > 0 \tag{9.14}$$

This likely seems like it came out of nowhere. Unfortunately, the connection is well beyond the scope of this book, but is covered well is external sources. For now, simply take it as a given that this is an equivalent requirement. (For those so inclined, the appendix of this book gives a proof, but it requires a bit of knowledge of function spaces to understand.)

The question is: why is this alternative characterization useful? It is useful because it gives you an alternative way to construct kernel

functions. For instance, using it you can easily prove the following, which would be difficult from the definition of kernels as inner products after feature mappings.

Theorem 12 (Kernel Addition). *If K_1 and K_2 are kernels, the K defined by $K(\boldsymbol{x}, \boldsymbol{z}) = K_1(\boldsymbol{x}, \boldsymbol{z}) + K_2(\boldsymbol{x}, \boldsymbol{z})$ is also a kernel.*

Proof of Theorem 12. You need to verify the positive semi-definite property on K. You can do this as follows:

$$\iint f(\boldsymbol{x}) K(\boldsymbol{x}, \boldsymbol{z}) f(\boldsymbol{z}) \mathrm{d}\boldsymbol{x}\mathrm{d}\boldsymbol{z} = \iint f(\boldsymbol{x}) \left[K_1(\boldsymbol{x}, \boldsymbol{z}) + K_2(\boldsymbol{x}, \boldsymbol{z})\right] f(\boldsymbol{z}) \mathrm{d}\boldsymbol{x}\mathrm{d}\boldsymbol{z} \qquad \text{definition of } K \tag{9.15}$$

$$= \iint f(\boldsymbol{x}) K_1(\boldsymbol{x}, \boldsymbol{z}) f(\boldsymbol{z}) \mathrm{d}\boldsymbol{x}\mathrm{d}\boldsymbol{z} + \iint f(\boldsymbol{x}) K_2(\boldsymbol{x}, \boldsymbol{z}) f(\boldsymbol{z}) \mathrm{d}\boldsymbol{x}\mathrm{d}\boldsymbol{z} \qquad \text{distributive rule} \tag{9.16}$$

$$> 0 + 0 \qquad K_1 \text{ and } K_2 \text{ are psd} \tag{9.17}$$

□

More generally, any positive linear combination of kernels is still a kernel. Specifically, if $K_1, \ldots, K_M$ are all kernels, and $\alpha_1, \ldots, \alpha_M \geq 0$, then $K(\boldsymbol{x}, \boldsymbol{z}) = \sum_m \alpha_m K_m(\boldsymbol{x}, \boldsymbol{z})$ is also a kernel.

You can also use this property to show that the following **Gaussian kernel** (also called the **RBF kernel**) is also psd:

$$K_\gamma^{(\mathrm{RBF})}(\boldsymbol{x}, \boldsymbol{z}) = \exp\left[-\gamma \left|\left|\boldsymbol{x} - \boldsymbol{z}\right|\right|^2\right] \tag{9.18}$$

Here γ is a hyperparameter that controls the width of this Gaussian-like bumps. To gain an intuition for what the RBF kernel is doing, consider what prediction looks like in the perceptron:

$$f(\boldsymbol{x}) = \sum_n \alpha_n K(\boldsymbol{x}_n, \boldsymbol{x}) + b \tag{9.19}$$

$$= \sum_n \alpha_n \exp\left[-\gamma \left|\left|\boldsymbol{x}_n - \boldsymbol{z}\right|\right|^2\right] \tag{9.20}$$

In this computation, each training example is getting to "vote" on the label of the test point $\boldsymbol{x}$. The amount of "vote" that the nth training example gets is proportional to the negative exponential of the distance between the test point and itself. This is very much like an RBF neural network, in which there is a Gaussian "bump" at each training example, with variance $1/(2\gamma)$, and where the α_ns act as the weights connecting these RBF bumps to the output.

Showing that this kernel is positive definite is a bit of an exercise in analysis (particularly, integration by parts), but otherwise not difficult. Again, the proof is provided in the appendix.

So far, you have seen two bsaic classes of kernels: polynomial kernels ($K(\boldsymbol{x}, vz) = (1 + \boldsymbol{x} \cdot \boldsymbol{z})^d$), which includes the linear kernel ($K(\boldsymbol{x}, \boldsymbol{z}) = \boldsymbol{x} \cdot \boldsymbol{z}$) and RBF kernels ($K(\boldsymbol{x}, \boldsymbol{z}) = \exp[-\gamma ||\boldsymbol{x} - \boldsymbol{z}||^2]$). The former have a direct connection to feature expansion; the latter to RBF networks. You also know how to combine kernels to get new kernels by addition. In fact, you can do more than that: the product of two kernels is also a kernel.

As far as a "library of kernels" goes, there are many. Polynomial and RBF are by far the most popular. A commonly used, but technically *invalid* kernel, is the hyperbolic-tangent kernel, which mimics the behavior of a two-layer neural network. It is defined as:

$$K^{(\tanh)} = \tanh(1 + \boldsymbol{x} \cdot \boldsymbol{z}) \qquad \text{Warning: not psd} \qquad (9.21)$$

A final example, which is not very common, but is nonetheless interesting, is the all-subsets kernel. Suppose that your D features are all *binary*: all take values 0 or 1. Let $A \subseteq \{1, 2, \dots D\}$ be a subset of features, and let $f_A(\boldsymbol{x}) = \bigwedge_{d \in A} x_d$ be the conjunction of all the features in A. Let $\phi(\boldsymbol{x})$ be a feature vector over *all* such As, so that there are 2^D features in the vector ϕ. You can compute the kernel associated with this feature mapping as:

$$K^{(\text{subs})}(\boldsymbol{x}, \boldsymbol{z}) = \prod_d \Big(1 + x_d z_d\Big) \qquad (9.22)$$

Verifying the relationship between this kernel and the all-subsets feature mapping is left as an exercise (but closely resembles the expansion for the quadratic kernel).

9.5 Support Vector Machines

Kernelization predated support vector machines, but SVMs are definitely the model that popularized the idea. Recall the definition of the soft-margin SVM from Chapter 6.7 and in particular the optimization problem (6.36), which attempts to balance a large margin (small $||\boldsymbol{w}||^2$) with a small loss (small ξ_ns, where ξ_n is the **slack** on the nth training example). This problem is repeated below:

$$\begin{aligned} \min_{\boldsymbol{w}, b, \boldsymbol{\xi}} \quad & \frac{1}{2} ||\boldsymbol{w}||^2 + C \sum_n \xi_n & (9.23) \\ \text{subj. to} \quad & y_n (\boldsymbol{w} \cdot \boldsymbol{x}_n + b) \geq 1 - \xi_n & (\forall n) \\ & \xi_n \geq 0 & (\forall n) \end{aligned}$$

Previously, you optimized this by explicitly computing the slack variables ξ_n, given a solution to the decision boundary, $\boldsymbol{w}$ and b. However, you are now an expert with using Lagrange multipliers

to optimize constrained problems! The overall *goal* is going to be to rewrite the SVM optimization problem in a way that it no longer explicitly depends on the weights $\boldsymbol{w}$ and only depends on the examples $\boldsymbol{x}_n$ through kernel products.

There are $2N$ constraints in this optimization, one for each slack constraint and one for the requirement that the slacks are non-negative. Unlike the last time, these constraints are now *inequalities*, which require a slightly different solution. First, you rewrite all the inequalities so that they read as *something* ≥ 0 and then add corresponding Lagrange multipliers. The main difference is that the Lagrange multipliers are now constrained to be non-negative, and their sign in the augmented objective function matters.

The second set of constraints is already in the proper form; the first set can be rewritten as $y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b) - 1 + \xi_n \geq 0$. You're now ready to construct the Lagrangian, using multipliers α_n for the first set of constraints and β_n for the second set.

$$\mathcal{L}(\boldsymbol{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = \frac{1}{2}||\boldsymbol{w}||^2 + C\sum_n \xi_n - \sum_n \beta_n \xi_n \tag{9.24}$$

$$- \sum_n \alpha_n \left[y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b) - 1 + \xi_n \right] \tag{9.25}$$

The *new* optimization problem is:

$$\min_{\boldsymbol{w},b,\boldsymbol{\xi}} \max_{\boldsymbol{\alpha}\geq 0} \max_{\boldsymbol{\beta}\geq 0} \mathcal{L}(\boldsymbol{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) \tag{9.26}$$

The intuition is exactly the same as before. If you are able to find a solution that satisfies the constraints (eg., the purple term is properly non-negative), then the β_ns cannot do anything to "hurt" the solution. On the other hand, if the purple term *is* negative, then the corresponding β_n can go to $+\infty$, breaking the solution.

You can solve this problem by taking gradients. This is a bit tedious, but and important step to realize how everything fits together. Since your goal is to remove the dependence on $\boldsymbol{w}$, the first step is to take a gradient with respect to $\boldsymbol{w}$, set it equal to zero, and solve for $\boldsymbol{w}$ in terms of the other variables.

$$\nabla_{\boldsymbol{w}} \mathcal{L} = \boldsymbol{w} - \sum_n \alpha_n y_n \boldsymbol{x}_n = 0 \quad \iff \quad \boldsymbol{w} = \sum_n \alpha_n y_n \boldsymbol{x}_n \tag{9.27}$$

At this point, you should immediately recognize a similarity to the kernelized perceptron: the optimal weight vector takes *exactly* the same form in both algorithms.

You can now take this new expression for $\boldsymbol{w}$ and plug it back in to the expression for $\mathcal{L}$, thus removing $\boldsymbol{w}$ from consideration. To avoid subscript overloading, you should replace the n in the expression for

$\boldsymbol{w}$ with, say, m. This yields:

$$\mathcal{L}(b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = \frac{1}{2}\left|\left|\sum_m \alpha_m y_m \boldsymbol{x}_m\right|\right|^2 + C\sum_n \xi_n - \sum_n \beta_n \xi_n \tag{9.28}$$

$$- \sum_n \alpha_n \left[y_n \left(\left[\sum_m \alpha_m y_m \boldsymbol{x}_m \right] \cdot \boldsymbol{x}_n + b \right) - 1 + \xi_n \right] \tag{9.29}$$

At this point, it's convenient to rewrite these terms; be sure you understand where the following comes from:

$$\mathcal{L}(b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = \frac{1}{2}\sum_n \sum_m \alpha_n \alpha_m y_n y_m \boldsymbol{x}_n \cdot \boldsymbol{x}_m + \sum_n (C - \beta_n)\xi_n \tag{9.30}$$

$$- \sum_n \sum_m \alpha_n \alpha_m y_n y_m \boldsymbol{x}_n \cdot \boldsymbol{x}_m - \sum_n \alpha_n \left(y_n b - 1 + \xi_n\right) \tag{9.31}$$

$$= -\frac{1}{2}\sum_n \sum_m \alpha_n \alpha_m y_n y_m \boldsymbol{x}_n \cdot \boldsymbol{x}_m + \sum_n (C - \beta_n)\xi_n \tag{9.32}$$

$$-b\sum_n \alpha_n y_n - \sum_n \alpha_n (\xi_n - 1) \tag{9.33}$$

Things are starting to look good: you've successfully removed the dependence on $\boldsymbol{w}$, *and* everything is now written in terms of dot products between input vectors! This might still be a difficult problem to solve, so you need to continue and attempt to remove the remaining variables b and $\boldsymbol{\xi}$.

The derivative with respect to b is:

$$\frac{\partial \mathcal{L}}{\partial b} = -\sum_n \alpha_n y_n = 0 \tag{9.34}$$

This doesn't allow you to *substitute* b with something (as you did with $\boldsymbol{w}$), but it does mean that the fourth term ($b\sum_n \alpha_n y_n$) goes to zero at the optimum.

The last of the original variables is ξ_n; the derivatives in this case look like:

$$\frac{\partial \mathcal{L}}{\partial \xi_n} = C - \beta_n - \alpha_n \quad \Longleftrightarrow \quad C - \beta_n = \alpha_n \tag{9.35}$$

Again, this doesn't allow you to substitute, but it does mean that you can rewrite the second term, which as $\sum_n (C - \beta_n)\xi_n$ as $\sum_n \alpha_n \xi_n$. This then cancels with (most of) the final term. However, you need to be careful to remember something. When we optimize, both α_n and β_n are constrained to be non-negative. What this means is that since we are dropping β from the optimization, we need to ensure that $\alpha_n \leq C$, otherwise the corresponding β will need to be negative, which is not

allowed. You finally wind up with the following, where $\boldsymbol{x}_n \cdot \boldsymbol{x}_m$ has been replaced by $K(\boldsymbol{x}_n, \boldsymbol{x}_m)$:

$$\mathcal{L}(\boldsymbol{\alpha}) = \sum_n \alpha_n - \frac{1}{2} \sum_n \sum_m \alpha_n \alpha_m y_n y_m K(\boldsymbol{x}_n, \boldsymbol{x}_m) \tag{9.36}$$

If you are comfortable with matrix notation, this has a very compact form. Let $\mathbf{1}$ denote the N-dimensional vector of all 1s, let $\boldsymbol{y}$ denote the vector of labels and let $\mathbf{G}$ be the $N \times N$ matrix, where $\mathbf{G}_{n,m} = y_n y_m K(\boldsymbol{x}_n, \boldsymbol{x}_m)$, then this has the following form:

$$\mathcal{L}(\boldsymbol{\alpha}) = \boldsymbol{\alpha}^\top \mathbf{1} - \frac{1}{2} \boldsymbol{\alpha}^\top \mathbf{G} \boldsymbol{\alpha} \tag{9.37}$$

The resulting optimization problem is to *maximize* $\mathcal{L}(\boldsymbol{\alpha})$ as a function of $\boldsymbol{\alpha}$, subject to the constraint that the α_ns are all non-negative and less than C (because of the constraint added when removing the β variables). Thus, your problem is:

$$\min_{\boldsymbol{\alpha}} \quad -\mathcal{L}(\boldsymbol{\alpha}) = \frac{1}{2} \sum_n \sum_m \alpha_n \alpha_m y_n y_m K(\boldsymbol{x}_n, \boldsymbol{x}_m) - \sum_n \alpha_n \tag{9.38}$$

$$\text{subj. to} \quad 0 \leq \alpha_n \leq C \qquad (\forall n)$$

One way to solve this problem is gradient descent on α. The only complication is making sure that the αs satisfy the constraints. In this case, you can use a **projected gradient** algorithm: after each gradient update, you adjust your parameters to satisfy the constraints by *projecting* them into the feasible region. In this case, the projection is trivial: if, after a gradient step, any $\alpha_n < 0$, simply set it to 0; if any $\alpha_n > C$, set it to C.

9.6 Understanding Support Vector Machines

The prior discussion involved quite a bit of math to derive a representation of the support vector machine in terms of the Lagrange variables. This mapping is actually sufficiently standard that everything in it has a name. The original problem variables $(\boldsymbol{w}, b, \boldsymbol{\xi})$ are called the **primal variables**; the Lagrange variables are called the **dual variables**. The optimization problem that results after removing all of the primal variables is called the **dual problem**.

A succinct way of saying what you've done is: you found that after converting the SVM into its dual, it is possible to kernelize.

To understand SVMs, a first step is to peek into the dual formulation, Eq (9.38). The objective has two terms: the first depends on the data, and the second depends only on the dual variables. The first thing to notice is that, because of the second term, the αs "want" to

get as large as possible. The constraint ensures that they cannot exceed C, which means that the general tendency is for the αs to grow as close to C as possible.

To further understand the dual optimization problem, it is useful to think of the kernel as being a measure of *similarity* between two data points. This analogy is most clear in the case of RBF kernels, but even in the case of linear kernels, if your examples all have unit norm, then their dot product is still a measure of similarity. Since you can write the prediction function as $f(\hat{\boldsymbol{x}}) = \text{sign}(\sum_n \alpha_n y_n K(\boldsymbol{x}_n, \hat{\boldsymbol{x}}))$, it is natural to think of α_n as the "importance" of training example n, where $\alpha_n = 0$ means that it is not used at all at test time.

Consider two data points that have the same label; namely, $y_n = y_m$. This means that $y_n y_m = +1$ and the objective function has a term that looks like $\alpha_n \alpha_m K(\boldsymbol{x}_n, \boldsymbol{x}_m)$. Since the goal is to make this term small, then one of two things has to happen: either K has to be small, or $\alpha_n \alpha_m$ has to be small. If K is already small, then this doesn't affect the setting of the corresponding αs. But if K is large, then this *strongly* encourages at least one of α_n or α_m to go to zero. So if you have two data points that are very similar *and* have the same label, at least one of the corresponding αs will be small. This makes intuitive sense: if you have two data points that are basically the same (both in the $\boldsymbol{x}$ and y sense) then you only need to "keep" one of them around.

Suppose that you have two data points with different labels: $y_n y_m = -1$. Again, if $K(\boldsymbol{x}_n, \boldsymbol{x}_m)$ is small, nothing happens. But if it is large, then the corresponding αs are encouraged to be as large as possible. In other words, if you have two similar examples with different labels, you are strongly encouraged to keep the corresponding αs as large as C.

An alternative way of understanding the SVM dual problem is geometrically. Remember that the whole point of introducing the variable α_n was to ensure that the nth training example was correctly classified, modulo slack. More formally, the goal of α_n is to ensure that $y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b) - 1 + \xi_n \geq 0$. Suppose that this constraint it *not* satisfied. There is an important result in optimization theory, called the **Karush-Kuhn-Tucker conditions** (or **KKT conditions**, for short) that states that at the optimum, the product of the Lagrange multiplier for a constraint, and the value of that constraint, will equal zero. In this case, this says that at the optimum, you have:

$$\alpha_n \Big[y_n \left(\boldsymbol{w} \cdot \boldsymbol{x}_n + b \right) - 1 + \xi_n \Big] = 0 \tag{9.39}$$

In order for this to be true, it means that (at least) one of the following must be true:

$$\alpha_n = 0 \qquad or \qquad y_n \left(\boldsymbol{w} \cdot \boldsymbol{x}_n + b \right) - 1 + \xi_n = 0 \tag{9.40}$$

A reasonable question to ask is: under what circumstances will α_n be *non-zero?* From the KKT conditions, you can discern that α_n can be non-zero *only* when the constraint holds *exactly*; namely, that $y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b) - 1 + \xi_n = 0$. When does that constraint hold *exactly*? It holds exactly only for those points *precisely* on the margin of the hyperplane.

In other words, the *only* training examples for which $\alpha_n \neq 0$ are those that lie precisely 1 unit away from the maximum margin decision boundary! (Or those that are "moved" there by the corresponding slack.) These points are called the **support vectors** because they "support" the decision boundary. In general, the number of support vectors is far smaller than the number of training examples, and therefore you naturally end up with a solution that only uses a subset of the training data.

From the first discussion, you know that the points that wind up being support vectors are exactly those that are "confusable" in the sense that you have to examples that are nearby, but have different labels. This is a completely in line with the previous discussion. If you have a decision boundary, it will pass between these "confusable" points, and therefore they will end up being part of the set of support vectors.

9.7 Exercises

Exercise 9.1. TODO...