7 | Linear Models

> The essence of mathematics is not to make simple things complicated, but to make complicated things simple. – Stanley Gudder

Learning Objectives:

- Define and plot four surrogate loss functions: squared loss, logistic loss, exponential loss and hinge loss.
- Compare and contrast the optimization of 0/1 loss and surrogate loss functions.
- Solve the optimization problem for squared loss with a quadratic regularizer in closed form.
- Implement and debug gradient descent and subgradient descent.

Dependencies:

In Chapter 4, you learned about the perceptron algorithm for linear classification. This was both a *model* (linear classifier) and *algorithm* (the perceptron update rule) in one. In this section, we will separate these two, and consider general ways for optimizing linear models. This will lead us into some aspects of optimization (aka mathematical programming), but not very far. At the end of this chapter, there are pointers to more literature on optimization for those who are interested.

The basic idea of the perceptron is to run a particular algorithm until a linear separator is found. You might ask: are there better algorithms for finding such a linear separator? We will follow this idea and formulate a learning problem as an explicit optimization problem: find me a linear separator that is not too complicated. We will see that finding an "optimal" separator is actually computationally prohibitive, and so will need to "relax" the optimality requirement. This will lead us to a **convex** objective that combines a loss function (how well are we doing on the training data?) and a regularizer (how complicated is our learned model?). This learning framework is known as both **Tikhonov regularization** and **structural risk minimization**.

7.1 The Optimization Framework for Linear Models

You have already seen the perceptron as a way of finding a weight vector $\boldsymbol{w}$ and bias b that do a good job of separating positive training examples from negative training examples. The perceptron is a **model** and **algorithm** in one. Here, we are interested in *separating* these issues. We will focus on linear models, like the perceptron. But we will think about other, more generic ways of finding good parameters of these models.

The goal of the perceptron was to find a **separating hyperplane** for some training data set. For simplicity, you can ignore the issue of overfitting (but just for now!). Not all data sets are linearly sepa-

rable. In the case that your training data *isn't* linearly separable, you might want to find the hyperplane that makes the *fewest errors* on the training data. We can write this down as a formal mathematics **optimization problem** as follows:

$$\min_{w,b} \quad \sum_n \mathbf{1}[y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b) > 0] \tag{7.1}$$

In this expression, you are optimizing over two variables, $\boldsymbol{w}$ and b. The **objective function** is the thing you are trying to minimize. In this case, the objective function is simply the **error rate** (or **0/1 loss**) of the linear classifier parameterized by $\boldsymbol{w}, b$. In this expression, $\mathbf{1}[\cdot]$ is the **indicator function**: it is one when $(\cdot)$ is true and zero otherwise.

? You should remember the $y\boldsymbol{w} \cdot \boldsymbol{x}$ trick from the perceptron discussion. If not, re-convince yourself that this is doing the right thing.

We know that the perceptron algorithm is guaranteed to find parameters for this model if the data is linearly separable. In other words, if the optimum of Eq (7.1) is zero, then the perceptron will efficiently find parameters for this model. The notion of "efficiency" depends on the margin of the data for the perceptron.

You might ask: what happens if the data is *not* linearly separable? Is there an efficient algorithm for finding an optimal setting of the parameters? Unfortunately, the answer is *no.* There is no polynomial time algorithm for solving Eq (7.1), unless P=NP. In other words, this problem is NP-hard. Sadly, the proof of this is quite complicated and beyond the scope of this book, but it relies on a reduction from a variant of satisfiability. The key idea is to turn a satisfiability problem into an optimization problem where a clause is satisfied exactly when the hyperplane correctly separates the data.

You might then come back and say: okay, well I don't really need an *exact* solution. I'm willing to have a solution that makes one or two more errors than it has to. Unfortunately, the situation is really bad. Zero/one loss is NP-hard to even *appproximately minimize*. In other words, there is no efficient algorithm for even finding a solution that's a small constant worse than optimal. (The best known constant at this time is $418/415 \approx 1.007$.)

However, before getting too disillusioned about this whole enterprise (remember: there's an entire chapter about this framework, so it must be going somewhere!), you should remember that optimizing Eq (7.1) perhaps isn't even what you want to do! In particular, all it says is that you will get minimal *training error.* It says nothing about what your *test error* will be like. In order to try to find a solution that will *generalize* well to test data, you need to ensure that you do not overfit the data. To do this, you can introduce a **regularizer** over the parameters of the model. For now, we will be vague about what this regularizer looks like, and simply call it an arbitrary function $R(\boldsymbol{w}, b)$.

This leads to the following, **regularized objective**:

$$\min_{w,b} \quad \sum_n \mathbf{1}[y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b) > 0] + \lambda R(\boldsymbol{w}, b) \tag{7.2}$$

In Eq (7.2), we are now trying to optimize a *trade-off* between a solution that gives low training error (the first term) and a solution that is "simple" (the second term). You can think of the maximum depth hyperparameter of a decision tree as a form of regularization for trees. Here, R is a form of regularization for hyperplanes. In this formulation, λ becomes a **hyperparameter** for the optimization.

The key remaining questions, given this formalism, are:

- How can we adjust the optimization problem so that there *are* efficient algorithms for solving it?
- What are good regularizers $R(\boldsymbol{w}, b)$ for hyperplanes?
- Assuming we can adjust the optimization problem appropriately, what algorithms exist for efficiently solving this regularized optimization problem?

We will address these three questions in the next sections.

? Assuming R does the "right thing," what value(s) of λ will lead to overfitting? What value(s) will lead to underfitting?

7.2 Convex Surrogate Loss Functions

You might ask: why is optimizing zero/one loss so hard? Intuitively, one reason is that small changes to $\boldsymbol{w}, b$ can have a large impact on the value of the objective function. For instance, if there is a positive training example with $\boldsymbol{w}, \boldsymbol{x} \cdot + b = -0.0000001$, then adjusting b upwards by 0.00000011 will decrease your error rate by 1. But adjusting it upwards by 0.00000009 will have no effect. This makes it really difficult to figure out good ways to adjust the parameters.

To see this more clearly, it is useful to look at plots that relate *margin* to *loss*. Such a plot for zero/one loss is shown in Figure 7.1. In this plot, the horizontal axis measures the margin of a data point and the vertical axis measures the loss associated with that margin. For zero/one loss, the story is simple. If you get a positive margin (i.e., $y(\boldsymbol{w} \cdot \boldsymbol{x} + b) > 0$) then you get a loss of zero. Otherwise you get a loss of one. By thinking about this plot, you can see how changes to the parameters that change the margin *just a little bit* can have an enormous effect on the overall loss.

You might decide that a reasonable way to address this problem is to replace the non-smooth zero/one loss with a smooth approximation. With a bit of effort, you could probably concoct an "S"-shaped function like that shown in Figure 7.2. The benefit of using such an S-function is that it is smooth, and potentially easier to optimize. The difficulty is that it is not **convex**.

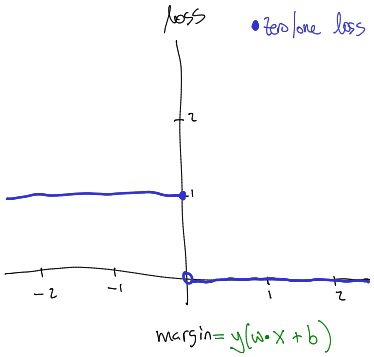

Figure 7.1: plot of zero/one versus margin

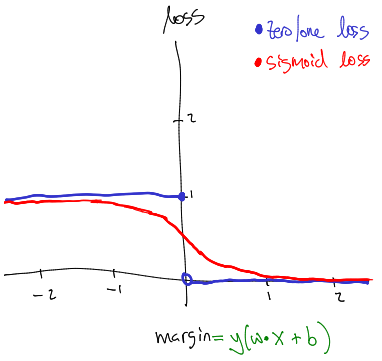

Figure 7.2: plot of zero/one versus margin and an S version of it

If you remember from calculus, a convex function is one that looks like a happy face ($\smile$). (On the other hand, a **concave** function is one that looks like a sad face ($\frown$); an easy mnemonic is that you can hide under a con**cave** function.) There are two equivalent definitions of a convex function. The first is that it's second derivative is always non-negative. The second, more geometric, defition is that any **chord** of the function lies above it. This is shown in Figure 7.3. There you can see a convex function and a non-convex function, both with two chords drawn in. In the case of the convex function, the chords lie above the function. In the case of the non-convex function, there are parts of the chord that lie below the function.

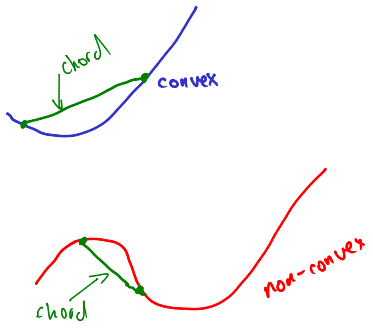

Figure 7.3: plot of convex and non-convex functions with two chords each

Convex functions are nice because they are *easy to minimize*. Intuitively, if you drop a ball anywhere in a convex function, it will eventually get to the minimum. This is not true for non-convex functions. For example, if you drop a ball on the very left end of the S-function from Figure 7.2, it will not go anywhere.

This leads to the idea of **convex surrogate loss functions**. Since zero/one loss is hard to optimize, you want to optimize something else, instead. Since convex functions are easy to optimize, we want to approximate zero/one loss with a convex function. This approximating function will be called a **surrogate loss**. The surrogate losses we construct will always be *upper bounds* on the true loss function: this guarantees that if you minimize the surrogate loss, you are also pushing down the real loss.

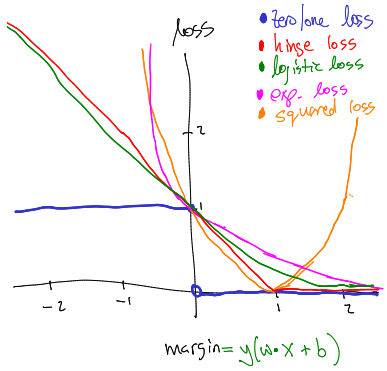

Figure 7.4: surrogate loss fns

There are four common surrogate loss functions, each with their own properties: **hinge loss**, **logistic loss**, **exponential loss** and **squared loss**. These are shown in Figure 7.4 and defined below. These are defined in terms of the true label y (which is just $\{-1, +1\}$) and the predicted value $\hat{y} = \boldsymbol{w} \cdot \boldsymbol{x} + b$.

$$\text{Zero/one:} \quad \ell^{(0/1)}(y, \hat{y}) = \mathbf{1}[y\hat{y} \leq 0] \tag{7.3}$$

$$\text{Hinge:} \quad \ell^{(\text{hin})}(y, \hat{y}) = \max\{0, 1 - y\hat{y}\} \tag{7.4}$$

$$\text{Logistic:} \quad \ell^{(\log)}(y, \hat{y}) = \frac{1}{\log 2} \log\left(1 + \exp[-y\hat{y}]\right) \tag{7.5}$$

$$\text{Exponential:} \quad \ell^{(\exp)}(y, \hat{y}) = \exp[-y\hat{y}] \tag{7.6}$$

$$\text{Squared:} \quad \ell^{(\text{sqr})}(y, \hat{y}) = (y - \hat{y})^2 \tag{7.7}$$

In the definition of logistic loss, the $\frac{1}{\log 2}$ term out front is there simply to ensure that $\ell^{(\log)}(y, 0) = 1$. This ensures, like all the other surrogate loss functions, that logistic loss upper bounds the zero/one loss. (In practice, people typically omit this constant since it does not affect the optimization.)

There are two big differences in these loss functions. The first difference is how "upset" they get by erroneous predictions. In the

case of hinge loss and logistic loss, the growth of the function as $\hat{y}$ goes negative is linear. For squared loss and exponential loss, it is super-linear. This means that exponential loss would rather get a few examples a little wrong than one example really wrong. The other difference is how they deal with very confident correct predictions. Once $y\hat{y} > 1$, hinge loss does not care any more, but logistic and exponential still think you can do better. On the other hand, squared loss thinks it's just as bad to predict $+3$ on a positive example as it is to predict -1 on a positive example.

7.3 Weight Regularization

In our learning objective, Eq (7.2), we had a term correspond to the zero/one loss on the training data, plus a **regularizer** whose goal was to ensure that the learned function didn't get too "crazy." (Or, more formally, to ensure that the function did not overfit.) If you replace to zero/one loss with a surrogate loss, you obtain the following objective:

$$\min_{\boldsymbol{w},b} \quad \sum_n \ell(y_n, \boldsymbol{w} \cdot \boldsymbol{x}_n + b) + \lambda R(\boldsymbol{w}, b) \tag{7.8}$$

The question is: what should $R(\boldsymbol{w}, b)$ look like?

From the discussion of surrogate loss function, we would like to ensure that R is convex. Otherwise, we will be back to the point where optimization becomes difficult. Beyond that, a common desire is that the components of the weight vector (i.e., the w_ds) should be small (close to zero). This is a form of **inductive bias**.

Why are small values of w_d good? Or, more precisely, why do small values of w_d correspond to *simple functions*? Suppose that we have an example $\boldsymbol{x}$ with label $+1$. We might believe that other examples, $\boldsymbol{x}'$ that are nearby $\boldsymbol{x}$ should also have label $+1$. For example, if I obtain $\boldsymbol{x}'$ by taking $\boldsymbol{x}$ and changing the first component by some small value ϵ and leaving the rest the same, you might think that the classification would be the same. If you do this, the difference between $\hat{y}$ and $\hat{y}'$ will be exactly ϵw_1. So if w_1 is reasonably small, this is unlikely to have much of an effect on the classification decision. On the other hand, if w_1 is large, this could have a large effect.

Another way of saying the same thing is to look at the derivative of the predictions as a function of w_1. The derivative of $\boldsymbol{w} \cdot \boldsymbol{x} + b$ with respect to w_1 is:

$$\frac{\partial \left[\boldsymbol{w} \cdot \boldsymbol{x} + b\right]}{\partial w_1} = \frac{\partial \left[\sum_d w_d x_d + b\right]}{\partial w_1} = x_1 \tag{7.9}$$

Interpreting the derivative as the rate of change, we can see that the rate of change of the prediction function is proportional to the

individual weights. So if you want the function to change slowly, you want to ensure that the weights stay small.

One way to accomplish this is to simply use the norm of the weight vector. Namely $R^{(\text{norm})}(\boldsymbol{w}, b) = ||\boldsymbol{w}|| = \sqrt{\sum_d w_d^2}$. This function is convex and smooth, which makes it easy to minimize. In practice, it's often easier to use the squared norm, namely $R^{(\text{sqr})}(\boldsymbol{w}, b) = ||\boldsymbol{w}||^2 = \sum_d w_d^2$ because it removes the ugly square root term and remains convex. An alternative to using the sum of squared weights is to use the sum of absolute weights: $R^{(\text{abs})}(\boldsymbol{w}, b) = \sum_d |w_d|$. Both of these norms are convex.

? Why do we not regularize the bias term b?

In addition to small weights being good, you could argue that *zero* weights are better. If a weight w_d goes to zero, then this means that feature d is not used at all in the classification decision. If there are a large number of irrelevant features, you might want as many weights to go to zero as possible. This suggests an alternative regularizer: $R^{(\text{cnt})}(\boldsymbol{w}, b) = \sum_d \mathbf{1}[x_d \neq 0]$.

This line of thinking leads to the general concept of **p-norms**. (Technically these are called ℓ_p (or "ell p") norms, but this notation clashes with the use of ℓ for "loss.") This is a family of norms that all have the same general flavor. We write $||\boldsymbol{w}||_p$ to denote the p-norm of $\boldsymbol{w}$.

? Why might you not want to use $R^{(\text{cnt})}$ as a regularizer?

$$||\boldsymbol{w}||_p = \left(\sum_d |w_d|^p \right)^{\frac{1}{p}} \tag{7.10}$$

You can check that the 2-norm exactly corresponds to the usual Euclidean norm, and that the 1-norm corresponds to the "absolute" regularizer described above.

? You can actually identify the $R^{(\text{cnt})}$ regularizer with a p-norm as well. Which value of p gives it to you? (Hint: you may have to take a limit.)

When p-norms are used to regularize weight vectors, the interesting aspect is how they trade-off multiple features. To see the behavior of p-norms in two dimensions, we can plot their **contour** (or **level-set**). Figure 7.5 shows the contours for the same p norms in two dimensions. Each line denotes the two-dimensional vectors to which this norm assignes a total value of 1. By changing the value of p, you can interpolate between a square (the so-called "max norm"), down to a circle (2-norm), diamond (1-norm) and pointy-star-shaped-thing ($p < 1$ norm).

In general, smaller values of p "prefer" sparser vectors. You can see this by noticing that the contours of small p-norms "stretch" out along the axes. It is for this reason that small p-norms tend to yield weight vectors with many zero entries (aka **sparse** weight vectors). Unfortunately, for $p < 1$ the norm becomes non-convex. As you might guess, this means that the 1-norm is a popular choice for sparsity-seeking applications.

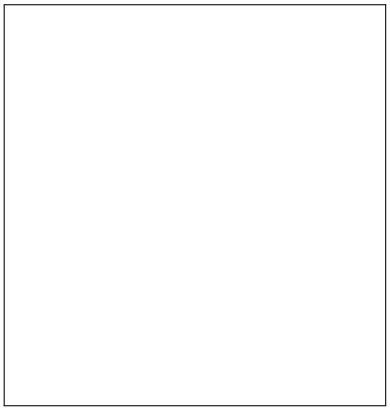

Figure 7.5: `loss:norms2d`: level sets of the same p-norms

? The max norm corresponds to $\lim_{p \to \infty}$. Why is this called the max norm?

MATH REVIEW | GRADIENTS

A gradient is a multidimensional generalization of a derivative. Suppose you have a function $f : \mathbb{R}^D \to \mathbb{R}$ that takes a vector $\boldsymbol{x} = \langle x_1, x_2, \ldots, x_D \rangle$ as input and produces a scalar value as output. You can differentite this function according to any one of the inputs; for instance, you can compute $\frac{\partial f}{\partial x_5}$ to get the derivative with respect to the fifth input. The **gradient** of f is just the vector consisting of the derivative f with respect to each of its input coordinates independently, and is denoted ∇f, or, when the input to f is ambiguous, $\nabla_{\boldsymbol{x}} f$. This is defined as:

$$\nabla_{\boldsymbol{x}} f = \left\langle \frac{\partial f}{\partial x_1} \ , \ \frac{\partial f}{\partial x_2} \ , \ \cdots \ , \ \frac{\partial f}{\partial x_D} \right\rangle \tag{7.11}$$

For example, consider the function $f(x_1, x_2, x_3) = x_1^3 + 5x_1x_2 - 3x_2x_3^2$. The gradient is:

$$\nabla_{\boldsymbol{x}} f = \left\langle 3x_1^2 + 5x_2 \ , \ 5x_1 - 3x_3^2 \ , \ -6x_2x_3 \right\rangle \tag{7.12}$$

Note that if $f : \mathbb{R}^D \to \mathbb{R}$, then $\nabla f : \mathbb{R}^D \to \mathbb{R}^D$. If you evaluate $\nabla f(\boldsymbol{x})$, this will give you the gradient *at* $\boldsymbol{x}$, a vector in $\mathbb{R}^D$. This vector can be interpreted as the direction of **steepest ascent**: namely, if you were to travel an infinitesimal amount in the direction of the gradient, you would go uphill (i.e., increase f) the most.

Figure 7.6:

7.4 *Optimization with Gradient Descent*

Envision the following problem. You're taking up a new hobby: blindfolded mountain climbing. Someone blindfolds you and drops you on the side of a mountain. Your goal is to get to the peak of the mountain as quickly as possible. All you can do is feel the mountain where you are standing, and take steps. How would you get to the top of the mountain? Perhaps you would feel to find out what direction feels the most "upward" and take a step in that direction. If you do this repeatedly, you might hope to get the the top of the mountain. (Actually, if your friend promises always to drop you on purely concave mountains, you *will* eventually get to the peak!)

The idea of gradient-based methods of optimization is exactly the same. Suppose you are trying to find the maximum of a function $f(\boldsymbol{x})$. The optimizer maintains a current estimate of the parameter of interest, $\boldsymbol{x}$. At each step, it measures the **gradient** of the function it is trying to optimize. This measurement occurs *at* the current location, $\boldsymbol{x}$. Call the gradient $\boldsymbol{g}$. It then takes a step in the direction of the gradient, where the size of the step is controlled by a parameter η (eta). The complete step is $\boldsymbol{x} \leftarrow \boldsymbol{x} + \eta \boldsymbol{g}$. This is the basic idea of **gradient ascent**.

The opposite of gradient ascent is **gradient descent**. All of our

Algorithm 21 GRADIENTDESCENT($\mathcal{F}$, K, η_1, ...)

$z^{(0)} \leftarrow \langle 0, 0, \ldots, 0 \rangle$ // initialize variable we are optimizing
for $k = 1 \ldots K$ **do**
  $g^{(k)} \leftarrow \nabla_z \mathcal{F}|_{z^{(k-1)}}$ // compute gradient at current location
  $z^{(k)} \leftarrow z^{(k-1)} - \eta^{(k)} g^{(k)}$ // take a step down the gradient
end for
return $z^{(K)}$

learning problems will be framed as *minimization* problems (trying to reach the bottom of a ditch, rather than the top of a hill). Therefore, descent is the primary approach you will use. One of the major conditions for gradient ascent being able to find the true, **global minimum**, of its objective function is convexity. Without convexity, all is lost.

The gradient descent algorithm is sketched in Algorithm 7.4. The function takes as arguments the function $\mathcal{F}$ to be minimized, the number of iterations K to run and a sequence of learning rates $\eta_1, \ldots, \eta_K$. (This is to address the case that you might want to start your mountain climbing taking large steps, but only take small steps when you are close to the peak.)

The only real work you need to do to apply a gradient descent method is be able to compute derivatives. For concreteness, suppose that you choose exponential loss as a loss function and the 2-norm as a regularizer. Then, the regularized objective function is:

$$\mathcal{L}(\boldsymbol{w}, b) = \sum_n \exp\left[-y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b)\right] + \frac{\lambda}{2} ||\boldsymbol{w}||^2 \tag{7.13}$$

The only "strange" thing in this objective is that we have replaced λ with $\frac{\lambda}{2}$. The reason for this change is just to make the gradients cleaner. We can first compute derivatives with respect to b:

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial}{\partial b} \sum_n \exp\left[-y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b)\right] + \frac{\partial}{\partial b} \frac{\lambda}{2} ||\boldsymbol{w}||^2 \tag{7.14}$$

$$= \sum_n \frac{\partial}{\partial b} \exp\left[-y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b)\right] + 0 \tag{7.15}$$

$$= \sum_n \left(\frac{\partial}{\partial b} - y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b)\right) \exp\left[-y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b)\right] \tag{7.16}$$

$$= -\sum_n y_n \exp\left[-y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b)\right] \tag{7.17}$$

Before proceeding, it is worth thinking about what this says. From a practical perspective, the optimization will operate by updating $b \leftarrow b - \eta \frac{\partial \mathcal{L}}{\partial b}$. Consider positive examples: examples with $y_n = +1$. We would hope for these examples that the current prediction, $\boldsymbol{w} \cdot \boldsymbol{x}_n + b$, is as large as possible. As this value tends toward ∞, the term in the exp[] goes to zero. Thus, such points will not contribute to the step.

However, if the current prediction is small, then the exp[] term will be positive and non-zero. This means that the bias term b will be *increased*, which is exactly what you would want. Moreover, once all points are very well classified, the derivative goes to zero.

Now that we have done the easy case, let's do the gradient with respect to $\boldsymbol{w}$.

$$\nabla_{\boldsymbol{w}} \mathcal{L} = \nabla_{\boldsymbol{w}} \sum_n \exp\left[-y_n(\boldsymbol{w}\cdot\boldsymbol{x}_n + b)\right] + \nabla_{\boldsymbol{w}} \frac{\lambda}{2} ||\boldsymbol{w}||^2 \tag{7.18}$$

$$= \sum_n (\nabla_{\boldsymbol{w}} - y_n(\boldsymbol{w}\cdot\boldsymbol{x}_n + b)) \exp\left[-y_n(\boldsymbol{w}\cdot\boldsymbol{x}_n + b)\right] + \lambda \boldsymbol{w} \tag{7.19}$$

$$= -\sum_n y_n \boldsymbol{x}_n \exp\left[-y_n(\boldsymbol{w}\cdot\boldsymbol{x}_n + b)\right] + \lambda \boldsymbol{w} \tag{7.20}$$

Now you can repeat the previous exercise. The update is of the form $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \nabla_{\boldsymbol{w}} \mathcal{L}$. For well classified points (ones that tend toward $y_n \infty$), the gradient is near zero. For poorly classified points, the gradient points in the direction $-y_n \boldsymbol{x}_n$, so the update is of the form $\boldsymbol{w} \leftarrow \boldsymbol{w} + c y_n \boldsymbol{x}_n$, where c is some constant. This is just like the perceptron update! Note that c is large for very poorly classified points and small for relatively well classified points.

By looking at the part of the gradient related to the regularizer, the update says: $\boldsymbol{w} \leftarrow \boldsymbol{w} - \lambda \boldsymbol{w} = (1-\lambda)\boldsymbol{w}$. This has the effect of *shrinking* the weights toward zero. This is exactly what we expect the regulaizer to be doing!

The success of gradient descent hinges on appropriate choices for the step size. Figure 7.7 shows what can happen with gradient descent with poorly chosen step sizes. If the step size is too big, you can accidentally step over the optimum and end up oscillating. If the step size is too small, it will take way too long to get to the optimum. For a well-chosen step size, you can show that gradient descent will approach the optimal value at a fast *rate*. The notion of convergence here is that the *objective value* converges to the true minimum.

Theorem 8 (Gradient Descent Convergence). *Under suitable conditions*[1]*, for an appropriately chosen constant step size (i.e., $\eta_1 = \eta_2, \cdots = \eta$), the* **convergence rate** *of gradient descent is $\mathcal{O}(1/k)$. More specifically, letting $\boldsymbol{z}^*$ be the global minimum of $\mathcal{F}$, we have: $\mathcal{F}(\boldsymbol{z}^{(k)}) - \mathcal{F}(\boldsymbol{z}^*) \leq \frac{2||\boldsymbol{z}^{(0)} - \boldsymbol{z}^*||^2}{\eta k}$.*

The proof of this theorem is a bit complicated because it makes heavy use of some linear algebra. The key is to set the learning rate to $1/L$, where L is the maximum **curvature** of the function that is being optimized. The curvature is simply the "size" of the second derivative. Functions with high curvature have gradients that change

? This considered the case of positive examples. What happens with negative examples?

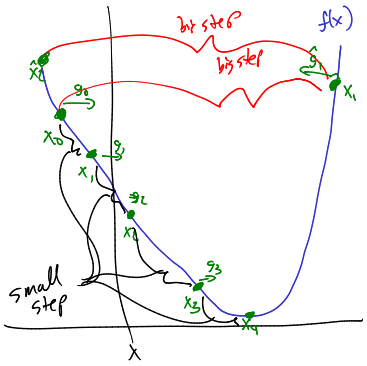

Figure 7.7: good and bad step sizes

[1] Specifically the function to be optimized needs to be **strongly convex**. This is true for all our problems, provided $\lambda > 0$. For $\lambda = 0$ the rate could be as bad as $\mathcal{O}(1/\sqrt{k})$.

? A naive reading of this theorem seems to say that you should choose huge values of η. It should be obvious that this cannot be right. What is missing?

quickly, which means that you need to take small steps to avoid overstepping the optimum.

This convergence result suggests a simple approach to deciding when to stop optimizing: wait until the objective function stops changing by much. An alternative is to wait until the *parameters* stop changing by much. A final example is to do what you did for perceptron: early stopping. Every iteration, you can check the performance of the current model on some held-out data, and stop optimizing when performance plateaus.

7.5 From Gradients to Subgradients

As a good exercise, you should try deriving gradient descent update rules for the different loss functions and different regularizers you've learned about. However, if you do this, you might notice that *hinge loss* and the 1-norm regularizer are not differentiable everywhere! In particular, the 1-norm is not differentiable around $w_d = 0$, and the hinge loss is not differentiable around $y\hat{y} = 1$.

The solution to this is to use **subgradient** optimization. One way to think about subgradients is just to not think about it: you essentially need to just ignore the fact that you forgot that your function wasn't differentiable, and just try to apply gradient descent anyway.

To be more concrete, consider the hinge function $f(z) = \max\{0, 1 - z\}$. This function is differentiable for $z > 1$ and differentiable for $z < 1$, but not differentiable at $z = 1$. You can derive this using differentiation by parts:

$$\frac{\partial}{\partial z} f(z) = \frac{\partial}{\partial z} \begin{cases} 0 & \text{if } z > 1 \\ 1 - z & \text{if } z < 1 \end{cases} \tag{7.21}$$

$$= \begin{cases} \frac{\partial}{\partial z} 0 & \text{if } z > 1 \\ \frac{\partial}{\partial z} (1 - z) & \text{if } z < 1 \end{cases} \tag{7.22}$$

$$= \begin{cases} 0 & \text{if } z \geq 1 \\ -1 & \text{if } z < 1 \end{cases} \tag{7.23}$$

Thus, the derivative is zero for $z < 1$ and -1 for $z > 1$, matching intuition from the Figure. At the non-differentiable point, $z = 1$, we can use a **subderivative**: a generalization of derivatives to non-differentiable functions. Intuitively, you can think of the derivative of f at z as the tangent line. Namely, it is the line that touches f at z that is always below f (for convex functions). The subderivative, denoted ∂f, is the *set* of all such lines. At differentiable positions, this set consists just of the actual derivative. At non-differentiable positions, this contains all slopes that define lines that always lie under the function and make contact at the operating point. This is

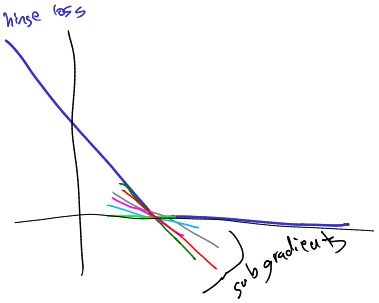

Figure 7.8: hinge loss with sub

Algorithm 22 HINGEREGULARIZEDGD($\mathbf{D}$, λ, *MaxIter*)

$\boldsymbol{w} \leftarrow \langle 0, 0, \ldots 0 \rangle$, $b \leftarrow 0$ // initialize weights and bias
for *iter* = $1 \ldots MaxIter$ **do**
  $\boldsymbol{g} \leftarrow \langle 0, 0, \ldots 0 \rangle$, $g \leftarrow 0$ // initialize gradient of weights and bias
  for all $(\boldsymbol{x}, y) \in \mathbf{D}$ **do**
    if $y(\boldsymbol{w} \cdot \boldsymbol{x} + b) \leq 1$ **then**
      $\boldsymbol{g} \leftarrow \boldsymbol{g} + y\,\boldsymbol{x}$ // update weight gradient
      $g \leftarrow g + y$ // update bias derivative
    end if
  end for
  $\boldsymbol{g} \leftarrow \boldsymbol{g} - \lambda \boldsymbol{w}$ // add in regularization term
  $\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta \boldsymbol{g}$ // update weights
  $b \leftarrow b + \eta g$ // update bias
end for
return $\boldsymbol{w}, b$

shown pictorally in Figure 7.8, where example subderivatives are shown for the hinge loss function. In the particular case of hinge loss, any value between 0 and -1 is a valid subderivative at $z = 0$. In fact, the subderivative is always a closed set of the form $[a, b]$, where a and b can be derived by looking at limits from the left and right.

This gives you a way of computing derivative-like things for non-differentiable functions. Take hinge loss as an example. For a given example n, the subgradient of hinge loss can be computed as:

$$\boldsymbol{\partial}_{\boldsymbol{w}} \max\{0, 1 - y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b)\} \tag{7.24}$$

$$= \boldsymbol{\partial}_{\boldsymbol{w}} \begin{cases} 0 & \text{if } y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b) > 1 \\ 1 - y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b) & \text{otherwise} \end{cases} \tag{7.25}$$

$$= \begin{cases} \boldsymbol{\partial}_{\boldsymbol{w}} 0 & \text{if } y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b) > 1 \\ \boldsymbol{\partial}_{\boldsymbol{w}} 1 - y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b) & \text{otherwise} \end{cases} \tag{7.26}$$

$$= \begin{cases} \mathbf{0} & \text{if } y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b) > 1 \\ -y_n \boldsymbol{x}_n & \text{otherwise} \end{cases} \tag{7.27}$$

If you plug this subgradient form into Algorithm 7.4, you obtain Algorithm 7.5. This is the **subgradient descent** for regularized hinge loss (with a 2-norm regularizer).

7.6 Closed-form Optimization for Squared Loss

Although gradient descent is a good, generic optimization algorithm, there are cases when you can do better. An example is the case of a 2-norm regularizer and squared error loss function. For this, you can actually obtain a *closed form* solution for the optimal weights. However, to obtain this, you need to rewrite the optimization problem in terms of matrix operations. For simplicity, we will only consider the

MATH REVIEW | MATRIX MULTIPLICATION AND INVERSION

If $\mathbf{A}$ and $\mathbf{B}$ are matrices, and $\mathbf{A}$ is $N \times K$ and $\mathbf{B}$ is $K \times M$ (the inner dimensions must match), then the matrix product $\mathbf{AB}$ is a matrix $\mathbf{C}$ that is $N \times M$, with $\mathbf{C}_{n,m} = \sum_k \mathbf{A}_{n,k}\mathbf{B}_{k,m}$. If $\boldsymbol{v}$ is a vector in $\mathbb{R}^D$, we will treat is as a *column vector*, or a matrix of size $D \times 1$. Thus, $\mathbf{A}\boldsymbol{v}$ is well defined if $\mathbf{A}$ is $D \times M$, and the resulting product is a vector $\boldsymbol{u}$ with $u_m = \sum_d \mathbf{A}_{d,m} v_d$.

Aside from matrix product, a fundamental matrix operation is inversion. We will often encounter a form like $\mathbf{A}\boldsymbol{x} = \boldsymbol{y}$, where $\mathbf{A}$ and $\boldsymbol{y}$ are known and we want to solve for $\mathbf{A}$. If $\mathbf{A}$ is square of size $N \times N$, then the inverse of $\mathbf{A}$, denoted $\mathbf{A}^{-1}$, is also a square matrix of size $N \times N$, such that $\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}_N = \mathbf{A}^{-1}\mathbf{A}$. I.e., multiplying a matrix by its inverse (on either side) gives back the identity matrix. Using this, we can solve $\mathbf{A}\boldsymbol{x} = \boldsymbol{y}$ by multiplying both sides by $\mathbf{A}^{-1}$ on the left (recall that order matters in matrix multiplication), yielding $\mathbf{A}^{-1}\mathbf{A}\boldsymbol{x} = \mathbf{A}^{-1}\boldsymbol{y}$ from which we can conclude $\boldsymbol{x} = \mathbf{A}^{-1}\boldsymbol{y}$. Note that not all square matrices are invertible. For instance, the all zeros matrix does not have an inverse (in the same way that $1/0$ is not defined for scalars). However, there are other matrices that do not have inverses; such matrices are called **singular**.

Figure 7.9:

unbiased version, but the extension is Exercise **??**. This is precisely the **linear regression** setting.

You can think of the training data as a large matrix $\mathbf{X}$ of size $N \times D$, where $X_{n,d}$ is the value of the dth feature on the nth example. You can think of the labels as a column ("tall") vector $\boldsymbol{Y}$ of dimension N. Finally, you can think of the weights as a column vector $\boldsymbol{w}$ of size D. Thus, the matrix-vector product $\boldsymbol{a} = \mathbf{X}\boldsymbol{w}$ has dimension N. In particular:

$$a_n = [\mathbf{X}\boldsymbol{w}]_n = \sum_d \mathbf{X}_{n,d} w_d \tag{7.28}$$

This means, in particular, that $\boldsymbol{a}$ is actually the predictions of the model. Instead of calling this $\boldsymbol{a}$, we will call it $\hat{\boldsymbol{Y}}$. The squared error says that we should minimize $\frac{1}{2}\sum_n (\hat{Y}_n - Y_n)^2$, which can be written in vector form as a minimization of $\frac{1}{2}\left|\left|\hat{\boldsymbol{Y}} - \boldsymbol{Y}\right|\right|^2$.

? Verify that the squared error can actually be written as this vector norm.

This can be expanded visually as:

$$\underbrace{\begin{bmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,D} \\ x_{2,1} & x_{2,2} & \dots & x_{2,D} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N,1} & x_{N,2} & \dots & x_{N,D} \end{bmatrix}}_{\mathbf{X}} \underbrace{\begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_D \end{bmatrix}}_{\boldsymbol{w}} = \underbrace{\begin{bmatrix} \sum_d x_{1,d} w_d \\ \sum_d x_{2,d} w_d \\ \vdots \\ \sum_d x_{N,d} w_d \end{bmatrix}}_{\hat{\boldsymbol{Y}}} \approx \underbrace{\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}}_{\boldsymbol{Y}} \tag{7.29}$$

So, compactly, our optimization problem can be written as:

$$\min_{\boldsymbol{w}} \quad \mathcal{L}(\boldsymbol{w}) = \frac{1}{2}\,||\mathbf{X}\boldsymbol{w} - \boldsymbol{y}||^2 + \frac{\lambda}{2}\,||\boldsymbol{w}||^2 \tag{7.30}$$

If you recall from calculus, you can minimize a function by setting its derivative to zero. We start with the weights $\boldsymbol{w}$ and take gradients:

$$\nabla_{\boldsymbol{w}}\mathcal{L}(\boldsymbol{w}) = \mathbf{X}^\top(\mathbf{X}\boldsymbol{w} - \boldsymbol{y}) + \lambda\boldsymbol{w} \tag{7.31}$$

$$= \mathbf{X}^\top\mathbf{X}\boldsymbol{w} - \mathbf{X}^\top\boldsymbol{y} + \lambda\boldsymbol{w} \tag{7.32}$$

$$= \left(\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}\right)\boldsymbol{w} - \mathbf{X}^\top\boldsymbol{y} \tag{7.33}$$

We can equate this to zero and solve, yielding:

$$\left(\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}\right)\boldsymbol{w} - \mathbf{X}^\top\boldsymbol{y} = 0 \tag{7.34}$$

$$\iff \left(\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}_D\right)\boldsymbol{w} = \mathbf{X}^\top\boldsymbol{y} \tag{7.35}$$

$$\iff \boldsymbol{w} = \left(\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}_D\right)^{-1}\mathbf{X}^\top\boldsymbol{y} \tag{7.36}$$

Thus, the *optimal* solution of the weights can be computed by a few matrix multiplications and a matrix inversion. As a sanity check, you can make sure that the dimensions match. The matrix $\mathbf{X}^\top\mathbf{X}$ has dimension $D\times D$, and therefore so does the inverse term. The inverse is $D\times D$ and $\mathbf{X}^\top$ is $D\times N$, so that product is $D\times N$. Multiplying through by the $N\times 1$ vector $\boldsymbol{y}$ yields a $D\times 1$ vector, which is precisely what we want for the weights.

? For those who are keen on linear algebra, you might be worried that the matrix you must invert might not be invertible. Is this actually a problem?

Note that this gives an *exact solution*, modulo numerical innacuracies with computing matrix inverses. In contrast, gradient descent will give you progressively better solutions and will "eventually" converge to the optimum at a rate of $1/k$. This means that if you want an answer that's within an accuracy of $\epsilon = 10^{-4}$, you will need something on the order of one thousand steps.

The question is whether getting this exact solution is always more efficient. To run gradient descent for one step will take $\mathcal{O}(ND)$ time, with a relatively small constant. You will have to run K iterations, yielding an overall runtime of $\mathcal{O}(KND)$. On the other hand, the closed form solution requires constructing $\mathbf{X}^\top\mathbf{X}$, which takes $\mathcal{O}(D^2N)$ time. The inversion take $\mathcal{O}(D^3)$ time using standard matrix inversion routines. The final multiplications take $\mathcal{O}(ND)$ time. Thus, the overall runtime is on the order $\mathcal{O}(D^3 + D^2N)$. In most standard cases (though this is becoming less true over time), $N > D$, so this is dominated by $\mathcal{O}(D^2N)$.

Thus, the overall question is whether you will need to run more than D-many iterations of gradient descent. If so, then the matrix inversion will be (roughly) faster. Otherwise, gradient descent will be (roughly) faster. For low- and medium-dimensional problems (say,

$D \leq 100$), it is probably faster to do the closed form solution via matrix inversion. For high dimensional problems ($D \geq 10,000$), it is probably faster to do gradient descent. For things in the middle, it's hard to say for sure.

7.7 Support Vector Machines

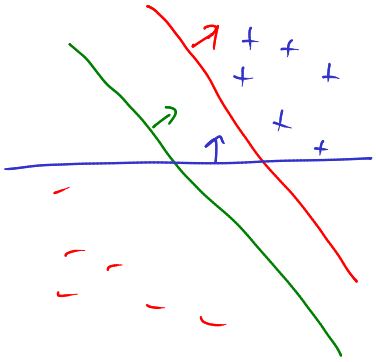

Figure 7.10: picture of data points with three hyperplanes, RGB with G the best

At the beginning of this chapter, you may have looked at the convex surrogate loss functions and asked yourself: where did these come from?! They are all derived from different underlying principles, which essentially correspond to different inductive biases.

Let's start by thinking back to the original goal of linear classifiers: to find a hyperplane that separates the positive training examples from the negative ones. Figure 7.10 shows some data and three potential hyperplanes: red, green and blue. Which one do you like best?

Most likely you chose the green hyperplane. And most likely you chose it because it was furthest away from the closest training points. In other words, it had a large **margin**. The desire for hyperplanes with large margins is a perfect example of an inductive bias. The data does not tell us which of the three hyperplanes is best: we have to choose one using some other source of information.

Following this line of thinking leads us to the **support vector machine** (SVM). This is simply a way of setting up an optimization problem that attempts to find a separating hyperplane with as large a margin as possible. It is written as a **constrained optimization problem**:

$$\begin{aligned} \min_{\boldsymbol{w},b} \quad & \frac{1}{\gamma(\boldsymbol{w},b)} & (7.37)\\ \text{subj. to} \quad & y_n\left(\boldsymbol{w}\cdot\boldsymbol{x}_n + b\right) \geq 1 \qquad (\forall n) \end{aligned}$$

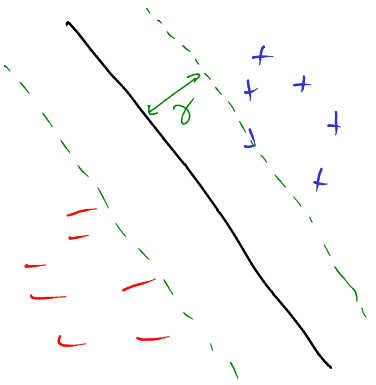

Figure 7.11: hyperplane with margins on sides

In this optimization, you are trying to find parameters that maximize the margin, denoted γ, (i.e., minimize the reciprocal of the margin) subject to the constraint that *all* training examples are correctly classified.

The "odd" thing about this optimization problem is that we require the classification of each point to be greater than *one* rather than simply greater than *zero*. However, the problem doesn't fundamentally change if you replace the "1" with any other positive constant (see Exercise **??**). As shown in Figure 7.11, the constant one can be interpreted visually as ensuring that there is a non-trivial margin between the positive points and negative points.

The difficulty with the optimization problem in Eq (7.37) is what happens with data that is not linearly separable. In that case, there *is no* set of parameters $\boldsymbol{w}, b$ that can simultaneously satisfy all the

constraints. In optimization terms, you would say that the **feasible region** is *empty*. (The feasible region is simply the set of all parameters that satify the constraints.) For this reason, this is refered to as the **hard-margin SVM**, because enforcing the margin is a hard constraint. The question is: how to modify this optimization problem so that it can handle inseparable data.

The key idea is the use of **slack parameters**. The intuition behind slack parameters is the following. Suppose we find a set of parameters $\boldsymbol{w}, b$ that do a really good job on 9999 data points. The points are perfectly classifed and you achieve a large margin. But there's one pesky data point left that cannot be put on the proper side of the margin: perhaps it is noisy. (See Figure 7.12.) You want to be able to pretend that you can "move" that point across the hyperplane on to the proper side. You will have to pay a little bit to do so, but as long as you aren't moving a *lot* of points around, it should be a good idea to do this. In this picture, the amount that you move the point is denoted ξ (xi).

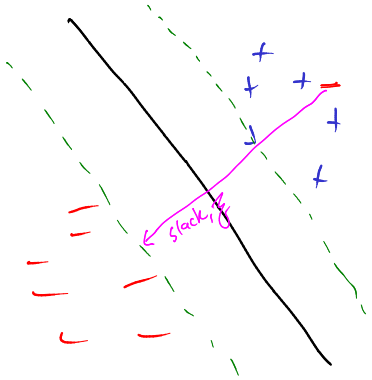

Figure 7.12: one bad point with slack

By introducing one slack parameter for each training example, and penalizing yourself for having to use slack, you can create an objective function like the following, **soft-margin SVM**:

$$\min_{\boldsymbol{w},b,\boldsymbol{\xi}} \quad \underbrace{\frac{1}{\gamma(\boldsymbol{w},b)}}_{\text{large margin}} + \underbrace{C\sum_n \xi_n}_{\text{small slack}} \tag{7.38}$$

$$\text{subj. to} \quad y_n\left(\boldsymbol{w}\cdot\boldsymbol{x}_n + b\right) \geq 1 - \xi_n \qquad (\forall n)$$

$$\xi_n \geq 0 \qquad (\forall n)$$

The goal of this objective function is to ensure that all points are correctly classified (the first constraint). But if a point n cannot be correctly classified, then you can set the slack ξ_n to something greater than zero to "move" it in the correct direction. However, for all non-zero slacks, you have to pay in the objective function proportional to the amount of slack. The hyperparameter $C > 0$ controls overfitting versus underfitting. The second constraint simply says that you must not have negative slack.

? What values of C will lead to overfitting? What values will lead to underfitting?

One major advantage of the soft-margin SVM over the original hard-margin SVM is that the feasible region is *never empty*. That is, there is always going to be some solution, regardless of whether your training data is linearly separable or not.

It's one thing to write down an optimization problem. It's another thing to try to solve it. There are a very large number of ways to optimize SVMs, essentially because they are such a popular learning model. Here, we will talk just about one, very simple way. More complex methods will be discussed later in this book once you have a bit more background.

? Suppose I give you a data set. Without even looking at the data, construct for me a feasible solution to the soft-margin SVM. What is the value of the objective for this solution?

To make progress, you need to be able to measure the size of the margin. Suppose someone gives you parameters $\boldsymbol{w}, b$ that optimize the hard-margin SVM. We wish to measure the size of the margin. The first observation is that the hyperplane will lie *exactly* halfway between the nearest positive point and nearest negative point. If not, the margin could be made bigger by simply sliding it one way or the other by adjusting the bias b.

By this observation, there is some positive example that that lies exactly 1 unit from the hyperplane. Call it $\boldsymbol{x}^+$, so that $\boldsymbol{w} \cdot \boldsymbol{x}^+ + b = 1$. Similarly, there is some negative example, $\boldsymbol{x}^-$, that lies exactly on the other side of the margin: for which $\boldsymbol{w} \cdot \boldsymbol{x}^- + b = -1$. These two points, $\boldsymbol{x}^+$ and $\boldsymbol{x}^-$ give us a way to measure the size of the margin. As shown in Figure 7.11, we can measure the size of the margin by looking at the difference between the lengths of projections of $\boldsymbol{x}^+$ and $\boldsymbol{x}^-$ onto the hyperplane. Since projection requires a normalized vector, we can measure the distances as:

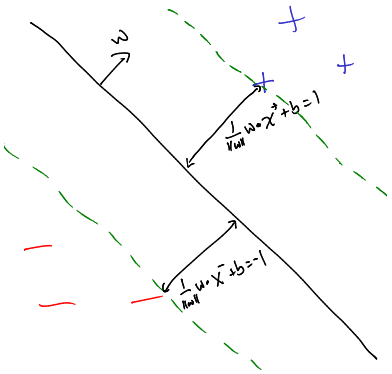

Figure 7.13: copy of figure from p5 of cs544 svm tutorial

$$d^+ = \frac{1}{||\boldsymbol{w}||} \boldsymbol{w} \cdot \boldsymbol{x}^+ + b - 1 \tag{7.39}$$

$$d^- = -\frac{1}{||\boldsymbol{w}||} \boldsymbol{w} \cdot \boldsymbol{x}^- - b + 1 \tag{7.40}$$

We can then compute the margin by algebra:

$$\gamma = \frac{1}{2}\left[d^+ - d^-\right] \tag{7.41}$$

$$= \frac{1}{2}\left[\frac{1}{||\boldsymbol{w}||} \boldsymbol{w} \cdot \boldsymbol{x}^+ + b - 1 - \frac{1}{||\boldsymbol{w}||} \boldsymbol{w} \cdot \boldsymbol{x}^- - b + 1\right] \tag{7.42}$$

$$= \frac{1}{2}\left[\frac{1}{||\boldsymbol{w}||} \boldsymbol{w} \cdot \boldsymbol{x}^+ - \frac{1}{||\boldsymbol{w}||} \boldsymbol{w} \cdot \boldsymbol{x}^-\right] \tag{7.43}$$

$$= \frac{1}{2}\left[\frac{1}{||\boldsymbol{w}||}(+1) - \frac{1}{||\boldsymbol{w}||}(-1)\right] \tag{7.44}$$

$$= \frac{1}{||\boldsymbol{w}||} \tag{7.45}$$

This is a remarkable conclusion: the size of the margin is inversely proportional to the norm of the weight vector. Thus, **maximizing the margin is equivalent to minimizing** $||\boldsymbol{w}||$**!** This serves as an additional justification of the 2-norm regularizer: having small weights means having large margins!

However, our goal wasn't to justify the regularizer: it was to understand hinge loss. So let us go back to the soft-margin SVM and plug in our new knowledge about margins:

$$\min_{\boldsymbol{w}, b, \boldsymbol{\xi}} \underbrace{\frac{1}{2}||\boldsymbol{w}||^2}_{\text{large margin}} + \underbrace{C \sum_n \xi_n}_{\text{small slack}} \tag{7.46}$$

$$\begin{aligned} \text{subj. to} \quad & y_n \left(\boldsymbol{w} \cdot \boldsymbol{x}_n + b \right) \geq 1 - \xi_n && (\forall n) \\ & \xi_n \geq 0 && (\forall n) \end{aligned}$$

Now, let's play a thought experiment. Suppose someone handed you a solution to this optimization problem that consisted of weights ($\boldsymbol{w}$) and a bias (b), but they forgot to give you the slacks. Could you recover the slacks from the information you have?

In fact, the answer is yes! For simplicity, let's consider positive examples. Suppose that you look at some positive example $\boldsymbol{x}_n$. You need to figure out what the slack, ξ_n, would have been. There are two cases. Either $\boldsymbol{w} \cdot \boldsymbol{x}_n + b$ is at least 1 or it is not. If it's large enough, then you want to set $\xi_n = 0$. Why? It cannot be less than zero by the second constraint. Moreover, if you set it greater than zero, you will "pay" unnecessarily in the objective. So in this case, $\xi_n = 0$. Next, suppose that $\boldsymbol{w} \cdot \boldsymbol{x}_n + b = 0.2$, so it is not big enough. In order to satisfy the first constraint, you'll need to set $\xi_n \geq 0.8$. But because of the objective, you'll not want to set it any larger than necessary, so you'll set $\xi_n = 0.8$ exactly.

Following this argument through for both positive and negative points, if someone gives you solutions for $\boldsymbol{w}, b$, you can automatically compute the optimal ξ variables as:

$$\xi_n = \begin{cases} 0 & \text{if } y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b) \geq 1 \\ 1 - y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b) & \text{otherwise} \end{cases} \tag{7.47}$$

In other words, the optimal value for a slack variable is *exactly* the hinge loss on the corresponding example! Thus, we can write the SVM objective as an *unconstrained* optimization problem:

$$\min_{\boldsymbol{w},b} \underbrace{\frac{1}{2} ||\boldsymbol{w}||^2}_{\text{large margin}} + \underbrace{C \sum_n \ell^{(\text{hin})}(y_n, \boldsymbol{w} \cdot \boldsymbol{x}_n + b)}_{\text{small slack}} \tag{7.48}$$

Multiplying this objective through by λ / C, we obtain exactly the regularized objective from Eq (7.8) with hinge loss as the loss function and the 2-norm as the regularizer!

7.8 Further Reading

TODO further reading