Support Vector Machines and Boosting

Roger Grosse

1 Introduction

We now introduce two additional algorithms which build on the principles we've just covered. Support vector machines (SVMs) are another kind of linear classifier, and before the deep learning revolution, they were one of the best general-purpose machine learning algorithms. They were motivated by very different principles from logistic regression, but at the end of the day, we'll see that the two algorithms are very similar.

The second algorithm we cover today, AdaBoost, isn't a linear classifier per se. It's another kind of ensemble algorithm, which is interesting to contrast with boosting. But we're covering it now because the tools we've developed for understanding linear models turn out to be useful in understanding what AdaBoost is doing and how to generalize it.

2 Support Vector Machines

In this section, we'll focus on binary (rather than multi-class) classification. Recall that a binary linear classifier first computes a linear function $z = \mathbf{w}^\top \mathbf{x} + b$, and then thresholds the result at 0. The decision boundary of a binary linear classifier in D dimensions is therefore a $D-1$ dimensional hyperplane given by the linear equation $\mathbf{w}^\top \mathbf{x} + b = 0$. We'll change notation slightly, and assume the targets take values in $\{-1, 1\}$ rather than $\{0, 1\}$.

Focusing on the binary case is not just a simplifying assumption. Unlike logistic regression, SVMs don't generalize so naturally to the multiclass case. For ways to generalize it, look up "one vs. one" and "one vs. all".

A binary classification dataset is linearly separable if there is a linear decision boundary which correctly classifies all the training examples. If that's the case, then there are generally multiple (actually inifintely many) such decision boundaries (see Figure 1). How do we choose between them? The key idea is that some of the boundaries classify the training data with a larger **margin** than others. The margin refers to the closest Euclidean distance from a training example to the decision boundary. If the margin is large, then we'd expect the classifier to be more robust to the random variability caused by the sampling of the training data. This suggests that among the decision boundaries that correctly classify all the training examples, we should choose the one with the largest margin. This is known as a **max-margin**, or **large margin**, criterion.

To compute the margin, think back to linear algebra class, and refer to Figure 1. We have a training example $\mathbf{x}$, and we want to compute the distance to the hyperplane defined by $\mathbf{w}^\top \mathbf{x} + b = 0$. We can do this by picking an arbitrary point $\mathbf{x}_0$ on the hyperplane, and then computing $|\mathbf{v}^\top (\mathbf{x} - \mathbf{x}_0)|$ for some unit vector $\mathbf{v}$ orthogonal to the hyperplane. In our case, the decision boundary is orthogonal to the classifier weights, so all we

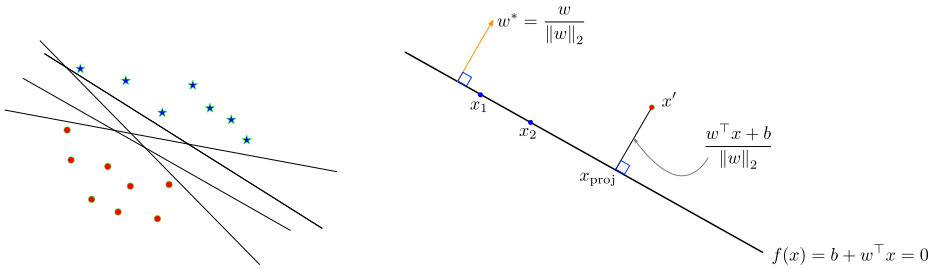

Figure 1: **Left:** There are many decision boundaries that separate the data, but some do so with a larger margin than others. **Right:** Computing the Euclidean distance from a training example to the decision boundary.

need to do is normalize the weights to be a unit vector: $\mathbf{w}_* = \mathbf{w}/\|\mathbf{w}\|_2$. Note also that $\mathbf{w}^\top \mathbf{x}_0 = -b$ by definition. Therefore,

$$\begin{aligned} \mathbf{w}_*^\top (\mathbf{x} - \mathbf{x}_0) &= \frac{\mathbf{w}^\top (\mathbf{x} - \mathbf{x}_0)}{\|\mathbf{w}\|_2} \\ &= \frac{\mathbf{w}^\top \mathbf{x} + b}{\|\mathbf{w}\|_2} \end{aligned} \tag{1}$$

We could take the absolute value of this to get the distance, but it's more convenient to refer to Eqn. 1 as the **signed distance**, which is positive on the positive side of the decision boundary and negative on the negative side.

We'd like each training example $\mathbf{x}^{(i)}$ to be on the correct side of the decision boundary with a margin of C. This can be written in terms of the following optimization problem:

Take a minute to think about why this equation corresponds to the margin constraint. Recall that $t^{(i)} \in \{-1, 1\}$.

$$\begin{aligned} &\max_{\mathbf{w}, b} C \\ &\text{s.t. } \frac{t^{(i)}(\mathbf{w}^\top \mathbf{x} + b)}{\|\mathbf{w}\|_2} \geq C \qquad i = 1, \ldots, N \end{aligned} \tag{2}$$

Unfortunately, this optimization problem is fairly awkward computationally. The problem is that $\|\mathbf{w}\|_2$ appears in the denominator, so the constraints are very nonlinear in $\mathbf{w}$, and the computation gets unstable when $\mathbf{w}$ is close to $\mathbf{0}$.

To fix this, observe that the decision boundary is invariant to rescaling $\mathbf{w}$ and b by the same scalar $\alpha > 0$ (i.e., you still get the same decision boundary). Therefore, we can impose an arbitrary scaling constraint on $\mathbf{w}$ without limiting the expressiveness of the model. In this case, we'll impose the constraint $C = 1/\|\mathbf{w}\|_2$. Plugging this into the margin constraints, we get:

$$\frac{t^{(i)}(\mathbf{w}^\top \mathbf{x}^{(i)} + b)}{\|\mathbf{w}\|_2} \geq \frac{1}{\|\mathbf{w}\|_2} \qquad \iff \qquad t^{(i)}(\mathbf{w}^\top \mathbf{x}^{(i)} + b) \geq 1 \tag{3}$$

The left-hand inequality is our original margin constraint; we now refer to this as the **geometric margin constraint** to emphasize that it directly constrains the Euclidean distance. The right-hand inequality is known as the **algebraic margin constraint**, and the quantity $t^{(i)}(\mathbf{w}^\top \mathbf{x}^{(i)} + b)$ is the

algebraic margin. The algebraic margin isn't a very meaningful quantity on its own, since it doesn't correspond to Euclidean distance. However, it's much more computationally convenient, since it's a linear function of $\mathbf{w}$ and b, and linear constraints are much more convenient from an optimization perspective. At the end of the day, we get the following optimization objective:

$$\begin{aligned} &\min \|\mathbf{w}\|_2^2 \\ &\text{s.t.} \quad t^{(i)}(\mathbf{w}^\top \mathbf{x}^{(i)} + b) \geq 1 \qquad i = 1, \ldots, N \end{aligned} \tag{4}$$

It is the combination of the algebraic margin constraint with the normalization condition $C = 1/\|\mathbf{w}\|_2$ which actually constrains our model.

We squared the L^2 norm for the optimization objective because squaring is monotonic, and squared L^2 norms are easier to work with.

This optimization problem has the interesting property that only a subset of the training examples actually influence the optimal solution. In particular, if the margin constraint is not tight for a particular $\mathbf{x}^{(i)}$, then we can remove that training example, and the optimal solution doesn't change. The important training examples are the ones which *do* lie exactly on the margin, and these are called the **support vectors**. This is what gives this algorithm the name **support vector machine (SVM)**. Derivations like the one we just did are used beyond the classification setting, and the general class of methods is known as **max-margin**, or **large margin**.

It turns out that the optimal solution can be expressed as a linear combination of the support vectors. This is an important fact about SVMs which makes possible dual optimization as well as the kernel trick; see below.

For another important example of max-margin training, see the classic 2004 paper "Max-margin Markov networks", by Taskar et al.

2.1 Soft-Margin SVMs

You might have noticed a problem with the above formulation: what if the data aren't linearly separable? Then the optimization problem is infeasible, i.e. it's impossible to satisfy all the constraints. The solution is to replace the hard constraints with soft constraints, which one is allowed to violate, but at a penalty. This model is known as a **soft-margin SVM**, and the formulation from the preceding section is known as the **hard-margin SVM**.

Assuming the data are linearly separable isn't as ridiculous as it sounds, since the dimension is often larger than the number of training examples.

We represent the soft constraints by introducing some **slack variables** ξ_i which determine the size of the violation. We require that:

$$\frac{t^{(i)}(\mathbf{w}^\top \mathbf{x}^{(i)} + b)}{\|\mathbf{w}\|_2} \geq C(1 - \xi_i), \tag{5}$$

which is identical to the hard margin constraint (4) except for the factor of $1 - \xi_i$ on the right-hand side. Notice that if $\xi_i = 0$, then the hard margin constraint is satisfied, if $\xi_i = 1$, the training example can lie exactly on the decision boundary, and if $\xi_i > 1$, the example can be incorrectly classified. We'll penalize the sum of the ξ_i, but we also require that each $\xi_i \geq 0$ so that we don't get extra credit for classifying some particular training example with an even larger margin. All in all, our optimization problem is as follows:

$$\begin{aligned} &\min \|\mathbf{w}\|_2^2 + \gamma \sum_{i=1}^{N} \xi_i \\ &\text{s.t.} \quad t^{(i)}(\mathbf{w}^\top \mathbf{x}^{(i)} + b) \geq 1 - \xi_i \qquad i = 1, \ldots, N \\ &\quad \xi_i \geq 0 \qquad i = 1, \ldots, N \end{aligned} \tag{6}$$

The hyperparameter γ controls the tradeoff between having a large margin vs. consistently satisfying the margin constraint. Consider some extreme cases: if $\gamma = 0$, then violations aren't penalized, so one can simply minimize the objective by setting $\mathbf{w} = \mathbf{0}$ and setting the ξ_i large enough to ensure all the constraints are satisfied. Conversely, for large enough γ, it is painful enough to violate a single constraint that the algorithm is equivalent to a hard-margin SVM (assuming the data are linearly separable).

Would you expect large/small values of γ to lead to overfitting/underfitting?

2.2 Hinge Loss

So far, the motivation has been very different from the linear models we've discussed previously. When we discussed linear regression and logistic regression, we started with a loss function and then figured out how to optimize it. But if we play around a bit with the soft margin SVM, we can write it in a similar form.

Specifically, let's **eliminate** the slack variables. I.e., let's determine their optimal value given a particular weight vector, and then substitute that value back into the optimization objective. So fix $\mathbf{w}$ and b. Since each ξ_i appears independently as a term in the sum, we'd like to make each one as small as possible. There are two caes to consider:

- **Case 1:** $1 - t^{(i)}(\mathbf{w}^\top \mathbf{x}^{(i)} + b) \leq 0$. Then the smallest non-negative value that satisfies the constraint is $\xi_i = 0$.
- **Case 2:** $1 - t^{(i)}(\mathbf{w}^\top \mathbf{x}^{(i)} + b) \geq 0$. Then the smallest non-negative value that satisfies the constraint is $\xi_i = 1 - t^{(i)}(\mathbf{w}^\top \mathbf{x}^{(i)} + b)$.

We can summarize both results with a single formula,

$$\xi_i = \max(0,\ 1 - t^{(i)}(\mathbf{w}^\top \mathbf{x}^{(i)} + b)). \tag{7}$$

We can write this using the convenient shorthand $(y)_+ = \max(0, y)$. Plugging this back in to (6), we get the following (unconstrained) optimization problem:

$$\min_{\mathbf{w},b} \sum_{i=1}^{N} \left(1 - t^{(i)}(\mathbf{w}^\top \mathbf{x}^{(i)} + b)\right)_+ + \frac{1}{2\gamma}\|\mathbf{w}\|_2^2. \tag{8}$$

Here, we swapped the two terms and divided through by γ to make the cost function more closely resemble those of linear regression and logistic regression.

This cost function is basically the sum of the losses $\mathcal{L}_{\mathrm{H}}(y, t) = (1 - ty)_+$ over all the training examples, plus an L^2 regularization term. The loss function $\mathcal{L}_{\mathrm{H}}$ is known as **hinge loss** because visually, it has a hinge at $y = 1$. The slack parameter γ controls the strength of the L^2 regularizer, and behaves like $1/\lambda$, where λ is the L^2 penalty hyperparameter.

Hinge loss is plotted along with various other loss functions in Figure 2. Notice that the asymptotic behavior matches that of the logistic-cross-entropy loss, i.e. the slopes are nearly equal for very small or very large values of z. This means that at the end of the day, SVMs are basically very similar to logistic regression. In fact, any place you're using logistic-cross-entropy loss, you can try replacing it with hinge loss, or vice versa, and it could slightly help performance.

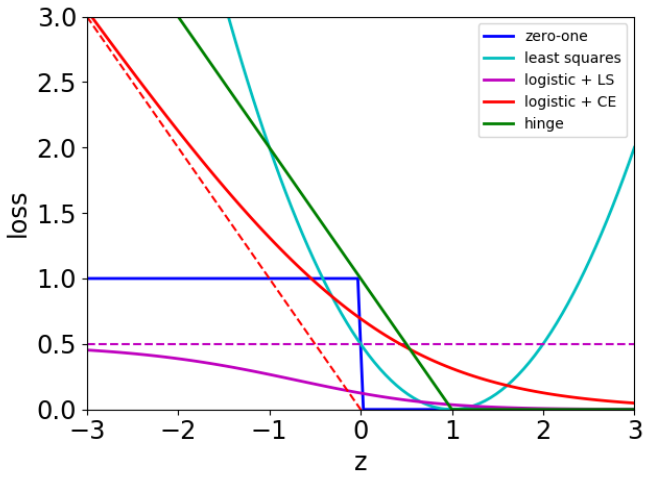

Figure 2: A comparison of binary classification loss functions for a positive training example.

2.3 Lagrange Duality (Optional)

The optimization problems (4) and (6) are both **quadratic programs**, i.e. they involve a convex quadratic cost function with linear constraints. One way to solve the soft-margin version (6) is by doing (stochastic) gradient descent on the hinge loss formulation (8). This is actually a reasonably effective strategy in practice, and is still one of the best ways to train linear SVMs on large datasets. But it's not great from an optimization perspective, for the same reason that gradient descent doesn't work for L^1-regularized linear regression: at the optimal solution, at least some of the training examples will satisfy the margin constraint exactly, but in general, gradient descent will overshoot the constraint and never satisfy it exactly.

If we care about actually converging to the optimum, we can do this by applying one of the most fundamental ideas in convex optimization, namely **Lagrange duality**. The general form for a convex optimization problem is as follows (where we denote the optimization variables with the vector $\boldsymbol{\theta}$):

In general, convex optimization problems can also have linear inequality constraints, but we ignore those here since they're not needed for SVMs.

$$\begin{aligned} &\min\ f(\boldsymbol{\theta}) \\ &\text{s.t. } g_i(\boldsymbol{\theta}) \leq 0 \qquad i = 1, \ldots, N, \end{aligned} \tag{9}$$

where f is a convex function (the optimization objective) and the g_i are convex functions defining the constraints. For a quadratic program (which the SVM is an instance of), f is a convex quadratic, and the g_i are all linear functions of $\boldsymbol{\theta}$.

What are $\boldsymbol{\theta}$, f, and g_i for the hard and soft margin SVM objectives? The answer is below, but try to figure it out yourself first.

Constraints can be rewritten as functions which take the value 0 if the constraint is satisfied and ∞ if the constraint is not satisfied. I.e., we can rewrite the optimization problem as the "unconstrained" optimization problem

$$\min f(\boldsymbol{\theta}) + \sum_i \tilde{g}_i(\boldsymbol{\theta}), \tag{10}$$

where

$$\tilde{g}_i(\boldsymbol{\theta}) = \begin{cases} 0 & \text{if } g_i \leq 0 \\ \infty & \text{if } g_i > 0. \end{cases} \tag{11}$$

We now make the trivial observation that

$$\tilde{g}_i(\boldsymbol{\theta}) = \max_{\alpha_i \geq 0} \alpha_i g_i(\boldsymbol{\theta}). \tag{12}$$

Hence, the original optimization problem can be written as the following **minmax** objective:

$$\min_{\boldsymbol{\theta}} \max_{\boldsymbol{\alpha} \geq \mathbf{0}} \underbrace{f(\boldsymbol{\theta}) + \sum_i \alpha_i g_i(\boldsymbol{\theta})}_{\triangleq \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\alpha})}. \tag{13}$$

Here, $\boldsymbol{\alpha}$ is the vector containing the α_i, and $\boldsymbol{\alpha} \geq \mathbf{0}$ is a convenient notation for all the entries being nonnegative. The function $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\alpha})$ is known as the **Lagrangian**, and the variables α_i are known as the **Lagrange multipliers**.

We always have the following inequality for interchanging the min and max:

$$\min_{\boldsymbol{\theta}} \max_{\boldsymbol{\alpha} \geq \mathbf{0}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\alpha}) \geq \max_{\boldsymbol{\alpha} \geq \mathbf{0}} \min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\alpha}) \tag{14}$$

Intuition: for zero-sum games, such as scissors-paper-rock, you'd rather be the one to move second.

However, if $\mathcal{L}$ is convex as a function of $\boldsymbol{\theta}$, and a bunch more technical conditions are satisfied (as they are in the case of the SVM objectives), then the inequality actually becomes an equality:

$$\min_{\boldsymbol{\theta}} \max_{\boldsymbol{\alpha} \geq \mathbf{0}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\alpha}) = \max_{\boldsymbol{\alpha} \geq \mathbf{0}} \min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\alpha}) \tag{15}$$

If it's possible to analytically determine $\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\alpha})$, then we can eliminate $\boldsymbol{\theta}$ (just as we eliminated the slack variables earlier) and reformulate the problem as an optimization over $\boldsymbol{\alpha}$. The resulting optimization problem is what's known as the **Lagrange dual**. In this context, the original optimization problem is known as the **primal**. To solve the primal problem, we simply maximize the dual objective with respect to $\boldsymbol{\alpha}$, and then plug in our analytical solution for $\boldsymbol{\theta}$.

It can be shown that for convex optimization problems, the solutions to the primal and dual optimization problems are exactly those pairs $(\boldsymbol{\theta}, \boldsymbol{\alpha})$ which satisfy the following conditions, called the **Karush-Kuhn-Tucker (KKT) Conditions**:

$$\begin{aligned}
\textbf{Stationarity:} \quad & \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} = \mathbf{0} \\
\textbf{Primal Feasibility:} \quad & g_i(\boldsymbol{\theta}) \leq 0 && \text{for } i = 1, \ldots, N \\
\textbf{Dual Feasibility:} \quad & \alpha_i \geq 0 && \text{for } i = 1, \ldots, N \\
\textbf{Complementary Slackness:} \quad & \alpha_i g_i(\boldsymbol{\theta}) = 0 && \text{for } i = 1, \ldots, N
\end{aligned} \tag{16}$$

The stationarity condition simply indicates that the Lagrangian is minimized with respect to $\boldsymbol{\theta}$. The two feasibility conditions simply indicate that the primal and dual variables satisfy their respective constraints. But the complementary slackness condition is very interesting, because it gives a key interpretation of the Lagrange multipliers. I.e., if $\alpha_i > 0$, then $g_i(\boldsymbol{\theta}) = 0$,

i.e. the inequality constraint is tight. Hence, we can determine which of the inequality constraints affect the optimal solution by checking the Lagrange multipliers.

Now let's apply this to the hard-margin SVM (4). The optimization variables are $\boldsymbol{\theta} = (\mathbf{w}^\top \ b)^\top$, and the functions are given by:

$$\begin{aligned} f(\mathbf{w}, b) &= \|\mathbf{w}\|_2^2 \\ g_i(\mathbf{w}, b) &= 1 - t^{(i)}(\mathbf{w}^\top \mathbf{x}^{(i)} + b) \end{aligned} \tag{17}$$

The Lagrangian is:

$$\mathcal{L}(\mathbf{w}, b, \boldsymbol{\alpha}) = \|\mathbf{w}\|_2^2 + \sum_i \alpha_i - \sum_i \alpha_i t^{(i)}(\mathbf{w}^\top \mathbf{x}^{(i)} + b) \tag{18}$$

Notice that this is a convex quadratic function of $\mathbf{w}$ and b. We can minimize this cost function exactly by setting the partial derivatives to zero, just like we did for linear regression. This gives us the following solution:

$$\begin{aligned} \mathbf{w}_\star &= \sum_i \alpha_i t^{(i)} \mathbf{x}^{(i)} \\ b_\star &= -\frac{1}{2}\left(\max_{i:t^{(i)}=-1} \mathbf{w}_\star^\top \mathbf{x}^{(i)} + \min_{i:t^{(i)}=1} \mathbf{w}_\star^\top \mathbf{x}^{(i)}\right) \end{aligned} \tag{19}$$

Remember our claim that only the support vectors affect the optimal solution? We've just made this rigorous. The support vectors are exactly those points for which $\alpha_i > 0$ (see the above discussion of the KKT conditions). The optimal weights $\mathbf{w}_\star$ are a linear combination of the input vectors, and the terms for which $\alpha_i = 0$ are all zero.

When we substitute our formula for $\mathbf{w}_\star$ back into the Lagrangian, we get the following optimization objective:

$$\begin{aligned} \min_{\boldsymbol{\alpha}} \ & \sum_i \alpha_i - \sum_i \sum_{i'} t^{(i)} t^{(i')} \alpha_i \alpha_{i'} \mathbf{x}^{(i)\top} \mathbf{x}^{(i')} \\ \text{s.t. } & \alpha_i \geq 0 \qquad \text{for } i = 1, \ldots, N \end{aligned} \tag{20}$$

Notice that this optimization problem is a quadratic program, just like the original SVM objective. The cost function is a convex quadratic in $\boldsymbol{\alpha}$, and the constraints are all linear inequalities. So what have we gained? The main thing we've gained is that the constraints are much simpler than before. In the primal formulation, the feasible set is a complicated linear polytope, and whenver one updates the weights, one needs to examine all the training examples to make sure the constraints are all satisfied. Whereas in the dual formulation, the feasible set is simply the **nonnegative orthant**, i.e. the set of vectors with nonnegative entries.

One example of an algorithm the dual formulation makes easier is **projected gradient descent**. This is an iterative procedure where in each iteration, we take the gradient descent step, and then project into the feasible set (i.e. find the nearest point in the set). In the primal formulation, this projection operation is itself a nontrivial optimization problem: we need to find the point in a polytope which minimizes the Euclidean distance to some

other point. But in the dual formulation, projection is very easy: we simply clip any negative values to zero. Here is the projected gradient ascent update (ascent because we're maximizing rather than minimizing):

$$\boldsymbol{\alpha} \leftarrow \left(\boldsymbol{\alpha} + \eta \frac{\partial \mathcal{J}}{\partial \boldsymbol{\alpha}}\right)_+. \tag{21}$$

But the dual formulation has a second property that's very convenient, namely **sparsity**. It could be that the number of support vectors is much smaller than the total number of training examples. Hence, one can design optimization algorithms to focus only on those values α_i which are currently positive, or likely to become positive. One such algorithm is **Sequential Minimal Optimization (SMO)**, which repeatedly minimizes the dual objective with respect to pairs of variables. You can read about this method in the classic 1998 paper by John Platt, "Sequential Minimal Optimization: A fast algorithm for training support vector machines".

The above discussion all focuses on the hard-margin SVM. We can do a similar derivation for the soft-margin SVM, and we wind up with a dual formulation that's only slightly different from the hard-margin one:

$$\begin{aligned} \min_{\boldsymbol{\alpha}} \; & \sum_i \alpha_i - \sum_i \sum_{i'} t^{(i)} t^{(i')} \alpha_i \alpha_{i'} \mathbf{x}^{(i)\top} \mathbf{x}^{(i)} \\ \text{s.t. } & 0 \le \alpha_i \le \gamma \qquad \text{for } i = 1, \ldots, N \end{aligned} \tag{22}$$

The only difference is that the α_i are now bounded above by γ, the slack penalty. Essentially, softening the constraints limits the extent to which any particular constraint (training example) can affect the optimal weights (recall our formula for $\mathbf{w}_\star$).

Think about happens when $\gamma = 0$ or $\gamma \to \infty$. Are these behaviors consistent with what we noted in Section 2.1?

2.4 The Kernel Trick (Optional)

In Section 2.2, we reformulated the SVM objective in a way that shows it's very similar to logistic regression. What's so great about SVMs, then, if logistic regression is already so simple and reliable? The answer is that they play very nicely with the Kernel Trick, a powerful idea that lets us convert linear models into highly nonlinear ones.

Let's start with the example of polynomial regression. Recall that for univariate inputs, we could implement degree-K polynomial regression using the following feature mapping:

$$\boldsymbol{\psi}_K(x) = \begin{pmatrix} 1 \\ x \\ \vdots \\ x^K \end{pmatrix} \tag{23}$$

Similarly, in D dimensions, we can perform polynomial regression by defining a feature vector consisting of all monomials of degree K or less. For

$K = 2$ (i.e., quadratic polynomials), we have:

This representation is somewhat redundant because it has equivalent terms such as $x_1 x_2$ and $x_2 x_1$. We'll ignore this, since collecting terms is complicated and only saves us a constant factor (for a given K).

$$\boldsymbol{\psi}_2(x_1, \ldots, x_D) = \begin{pmatrix} 1 \\ x_1 \\ \vdots \\ x_D \\ x_1^2 \\ x_1 x_2 \\ \vdots \\ x_D x_{D-1} \\ x_D^2 \end{pmatrix} \tag{24}$$

Unfortunately, the number of such monomials is $\mathcal{O}(D^K)$, so the size of the representation is *exponential* in the dimension.

The key insight is that, even though these vectors are exponentially large, it's possible to compute dot products between them in linear time. In the quadratic case ($K = 2$), we have:

We're denoting two different inputs as $\mathbf{x}$ and $\mathbf{y}$ rather than $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$ to prevent a proliferation of sub/superscripts.

$$\begin{aligned} \boldsymbol{\psi}_2(\mathbf{x})^\top \boldsymbol{\psi}_2(\mathbf{y}) &= 1 + \sum_i x_i y_i + \sum_i \sum_j x_i x_j y_i y_j \\ &= (1 + \sum_i x_i y_i)^2 \\ &= (1 + \mathbf{x}^\top \mathbf{y})^2. \end{aligned} \tag{25}$$

More generally, it's possible to show that

$$\boldsymbol{\psi}_K(\mathbf{x})^\top \boldsymbol{\psi}_K(\mathbf{y}) = (1 + \mathbf{x}^\top \mathbf{y})^K. \tag{26}$$

Hence, even though the feature vectors $\boldsymbol{\psi}_K(\mathbf{x})$ and $\boldsymbol{\psi}_K(\mathbf{y})$ have $\mathcal{O}(D^K)$ entries, we can compute dot products between them in $\mathcal{O}(D)$ time — we've gone from exponential to linear!

Polynomials are not the only case where dot products can be computed efficiently in high-dimensional spaces. There are lots more examples of functions $k(\mathbf{x}, \mathbf{y})$ which implicitly compute dot products between high-dimensional (or even infinite-dimensional!) feature vectors; such functions are known as **kernels**. If we can express a learning algorithm purely in terms of dot products, then we can **kernelize** it by expressing it in terms of kernels; this is known as the **kernel trick**.

Consider, for instance, the formula (19) for the optimal weights for the hard-margin SVM. These weights are a linear combination of the training examples, so given a new input $\mathbf{x}$, we can efficiently compute the dot product using the kernel:

$$\begin{aligned} \mathbf{w}_\star^\top \boldsymbol{\psi}(\mathbf{x}) &= \sum_i \alpha_i t^{(i)} \boldsymbol{\psi}(\mathbf{x}^{(i)})^\top \boldsymbol{\psi}(\mathbf{x}) \\ &= \sum_i \alpha_i t^{(i)} k(\mathbf{x}^{(i)}, \mathbf{x}). \end{aligned} \tag{27}$$

The dual SVM objective itself can be rewritten in terms of the kernel:

$$\begin{aligned} &\min_{\boldsymbol{\alpha}} \sum_i \alpha_i - \sum_i \sum_{i'} t^{(i)} t^{(i')} \alpha_i \alpha_{i'} k(\mathbf{x}^{(i)}, \mathbf{x}^{(i')}) \\ &\text{s.t. } \alpha_i \geq 0 \qquad \text{for } i = 1, \ldots, N \end{aligned} \tag{28}$$

Hence, as long as the kernel can be computed efficiently, we can train the SVM in an extremely high-dimensional feature space without ever having to explicitly construct the feature vectors.

2.4.1 What Can we Kernelize?

How broadly applicable is the kernel trick? Basically, we need to be able to express the algorithm only in terms of dot products between feature vectors, and the weight vector needs to be a linear combination of the training feature vectors. The **Representer Theorem** gives a very broad range of situations where this works. Let's prove a simple special case; the more general statement and its proof have a similar flavor.

This captures cases like linear regression, logistic regression, and the hinge loss formulation of SVMs.

Suppose we have a linear model $y = \mathbf{w}^\top \boldsymbol{\psi}(\mathbf{x})$, and are trying to minimize the empirical loss with an L^2 regularization term:

$$\mathcal{J}(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}(y^{(i)}, t^{(i)}) \; + \; \frac{\lambda}{2}\|\mathbf{w}\|_2^2. \tag{29}$$

It's a basic fact of linear algebra that given a subspace $\mathcal{S}$, a vector $\mathbf{v}$ can be decomposed as $\mathbf{v} = \mathbf{v}_\mathcal{S} + \mathbf{v}_\perp$, where $\mathbf{v}_\mathcal{S} \in \mathcal{S}$ and $\mathbf{v}_\perp \in \mathcal{S}^\perp$ (the space of vectors orthogonal to $\mathcal{S}$). Let's use this to decompose the weights as $\mathbf{w} = \mathbf{w}_\mathcal{S} + \mathbf{w}_\perp$, where $\mathcal{S}$ is the span of the feature vectors $\{\boldsymbol{\psi}(\mathbf{x}^{(i)})\}_{i=1}^N$, and $\mathbf{w}_\perp$ is perpendicular to all of them. Observe that $y = \mathbf{w}^\top\boldsymbol{\psi}(\mathbf{x}) = \mathbf{w}_\mathcal{S}^\top\boldsymbol{\psi}(\mathbf{x})$ because $\mathbf{w}_\perp^\top\boldsymbol{\psi}(\mathbf{x}) = 0$. Furthermore, $\|\mathbf{w}\|_2^2 = \|\mathbf{w}_\mathcal{S}\|_2^2 + \|\mathbf{w}_\perp\|_2^2$ because $\mathbf{w}_\mathcal{S}$ is orthogonal to $\mathbf{w}_\perp$. Combining these two facts, we find that if $\mathbf{w}_\perp \neq \mathbf{0}$, then we can strictly reduce the cost by setting $\mathbf{w}_\perp = \mathbf{0}$. (This doesn't affect the loss term, but strictly reduces the regularization term.) Hence, the optimal weights $\mathbf{w}$ must have $\mathbf{w}_\perp = \mathbf{0}$, i.e. they must lie in the subspace spanned by the training feature vectors. I.e., the optimal weights can always be represented as

$$\mathbf{w} = \sum_{i=1}^{N} \eta_i \boldsymbol{\psi}(\mathbf{x}^{(i)}) \tag{30}$$

When *can't* we apply the kernel trick? The key question is whether the algorithm is **rotation invariant**, i.e. whether you get an equivalent solution if you rotate the feature vectors. More precisely, suppose we transform the feature vectors as $\tilde{\boldsymbol{\psi}}(\mathbf{x}) = \mathbf{Q}\boldsymbol{\psi}(\mathbf{x})$ for some orthogonal matrix $\mathbf{Q}$. Then you can apply the same rotation to the weights: $\tilde{\mathbf{w}} = \mathbf{Q}\mathbf{w}$. This preserves the dot products, since

$$\tilde{\mathbf{w}}^\top\tilde{\boldsymbol{\psi}}(\mathbf{x}) = \mathbf{w}^\top\mathbf{Q}^\top\mathbf{Q}\boldsymbol{\psi}(\mathbf{x}) = \mathbf{w}^\top\boldsymbol{\psi}(\mathbf{x}). \tag{31}$$

Hence, if an algorithm depends only on dot products, then it must be rotation invariant. Conversely, if it's not rotation invariant, then it depends on more than dot products, i.e. it's not kernelizable. A canonical example of a non-rotation-invariant algorithm is L^1-regularized linear regression. The L^1 norm is not rotation invariant; furthermore, we clearly can't make the objective rotation invariant by fiddling with the terms, since the optimal solution is sparse, and sparsity depends on a particular choice of coordinate system. Hence, L^1-regularized regression is not kernelizable.

2.4.2 Computational Complexity

To understand when kernels are a good idea, let's examine the computational complexity of kernelized and non-kernelized algorithms.

- If one optimizes an objective like (31) using (stochastic) gradient descent by constructing the feature vectors explicitly, then each pass over the data requires $\mathcal{O}(NF)$ time to compute all the dot products, where F is the feature dimension. (If the feature vectors need to be computed, this could entail additional complexity.)
- Whereas in the kernelized representation, it requires $\mathcal{O}(N^2D)$ time to precompute all of the kernels (once, at the beginning of training). Then each pass over the data requires at least $\mathcal{O}(N^2)$ time, assuming it accesses all the kernel values at least once.

Neither one of these approaches strictly dominates the other. While the specifics will depend on the algorithm, roughly speaking, kernelization helps if $F \gg N$ (e.g. for high-degree polynomials), while it hurts if $F \ll N$ (e.g., linear kernels in low dimensions). For large datasets (e.g. more than tens of thousands of data points), it's prohibitive to precompute all $\mathcal{O}(N^2)$ kernel values, so kernelization is impractical unless there's additional structure we can exploit.

This is where kernel SVMs really shine. It could be that the optimal SVM solution is **sparse**, in the sense that only a small number $K \ll N$ of data points are support vectors, i.e. only K Lagrange multipliers are nonzero. In that case, $\mathbf{w}_\star$ is a linear combination of only K feature vectors, and one can compute $\mathbf{w}_\star^\top \boldsymbol{\psi}(\mathbf{x})$ in $\mathcal{O}(KD)$ time, rather than $\mathcal{O}(ND)$. Algorithms like SMO are good at exploiting this sparse structure, since they only need to compute kernels involving α_i which are nonzero or likely to become nonzero. This makes it possible to solve SVMs exactly for large datasets (e.g. millions of training examples). Even if the optimal solution is not exactly sparse, one can often do pretty well by finding a sparse approximation.

Note that this sense of sparsity is different from, and in fact incompatible with, the sense of sparsity from L^1 regularization. Here we mean sparsity of $\boldsymbol{\alpha}$; there we meant sparsity of $\mathbf{w}$.

2.4.3 Constructing Kernels

Polynomial feature maps are not the only useful example of kernels. In fact, it can be shown that any function k satisfying some basic properties is a valid kernel, i.e. it computes the dot product in some (possibly infinite dimensional) feature space. Those properties are:

- $k(\mathbf{x}, \mathbf{x}') = k(\mathbf{x}', \mathbf{x})$ for any $(\mathbf{x}, \mathbf{x}')$.
- For any finite set of points $(\mathbf{x}_1, \ldots, \mathbf{x}_D)$, the Gram matrix is positive semidefinite.

The **Gram matrix** is the D-dimensional matrix whose (i, j) entry is $k(\mathbf{x}_i, \mathbf{x}_j)$. A matrix $\mathbf{K}$ is **positive semidefinite (PSD)** if $\mathbf{v}^\top \mathbf{K} \mathbf{v} \geq 0$ for any vector $\mathbf{v}$. Verifying the symmetry property is typically straightforward; verifying the PSD property for particular kernels is cumbersome, but fortunately we don't need to do it very often.

It can be shown that the following function, called the **squared-exp kernel**, or **radial basis function (RBF) kernel)**, is a kernel:

$$k_{\text{SE}}(\mathbf{x}, \mathbf{x}'; \ell) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{2\ell^2}\right) \tag{32}$$

This kernel is large if two points are close together (in terms of Euclidean distance) and close to zero if they are far apart. The value ℓ is a hyperparameter called the **lengthscale** which determines how far apart two inputs can be while still having a large kernel value. (Like other hyperparameters, it's typically fixed during training, and we can choose the value using cross-validation.)

Suppose we fit a model (e.g. linear regression, SVM) using an RBF kernel. Then the function computed has the following form, following (30):

$$\begin{aligned} y = f(\mathbf{x}) &\triangleq \mathbf{w}^\top \boldsymbol{\psi}(\mathbf{x}) \\ &= \sum_{i=1}^{N} \eta_i \boldsymbol{\psi}(\mathbf{x}^{(i)})^\top \psi(\mathbf{x}) \\ &= \sum_{i=1}^{N} \eta_i k(\mathbf{x}^{(i)}, \mathbf{x}). \end{aligned} \tag{33}$$

Viewed as a function of $\mathbf{x}$, each term in the sum $\eta_i k(\mathbf{x}^{(i)}, \cdot)$ looks like a bell-shaped bump centered at $\mathbf{x}^{(i)}$, scaled by η_i. By taking linear combinations of lots of these bumps, the kernelized regression model is good at representing smooth functions.

There is a very rich space of kernels to choose from, because kernels can be built from other kernels using the **composition rules**:

- The sum of two kernels, $(k_1 + k_2)(\mathbf{x}, \mathbf{x}') = k_1(\mathbf{x}, \mathbf{x}') + k_2(\mathbf{x}, \mathbf{x}')$, is a kernel.
- The product of two kernels, $(k_1 k_2)(\mathbf{x}, \mathbf{x}') = k_1(\mathbf{x}, \mathbf{x}')\, k_2(\mathbf{x}, \mathbf{x}')$, is a kernel.

As an example of composite kernels, consider what happens if our inputs are 2-dimensional, and we define RBF kernels on the two input dimensions individually:

$$k_1(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{(x_1 - x_1')^2}{2\ell^2}\right) \qquad k_2(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{(x_2 - x_2')^2}{2\ell^2}\right) \tag{34}$$

Starting from (30), we can express the prediction function for the additive kernel $k_1 + k_2$ as:

$$\begin{aligned} f(\mathbf{x}) &= \sum_{i=1}^{N} \eta_i (k_1 + k_2)(\mathbf{x}^{(i)}, \mathbf{x}) \\ &= \sum_{i=1}^{N} \eta_i k_1(\mathbf{x}^{(i)}, \mathbf{x}) + \eta_i k_2(\mathbf{x}^{(i)}, \mathbf{x}) \\ &= \underbrace{\sum_{i=1}^{N} \eta_i k_1(\mathbf{x}^{(i)}, \mathbf{x})}_{\triangleq f_1(\mathbf{x})} + \underbrace{\sum_{i=1}^{N} \eta_i k_2(\mathbf{x}^{(i)}, \mathbf{x})}_{\triangleq f_2(\mathbf{x})} \end{aligned} \tag{35}$$

Hence, $f(\mathbf{x})$ is decomposed as a sum of a smooth function $f_1(\mathbf{x})$ which only depends on x_1 and another smooth function $f_2(\mathbf{x})$ which only depends on x_2. This general structure, where the predictions are made using a sum of (possibly nonlinear) functions associated with each dimension, is known as an **additive model**. This generalizes to more than 2 dimensions in the obvious way.

On the other hand, you can check that the product kernel $k_1 k_2$ is equivalent to an RBF kernel over both dimensions. Hence, sum and product kernels produce interestingly different behavior.

For an intuition of how the composition rules behave, think about the kernel between two inputs as representing their similarity. The sum kernel $k_1 + k_2$ says that $\mathbf{x}$ and $\mathbf{x}'$ are similar if they are similar under k_1 OR they are similar under k_2. The product kernel $k_1\, k_2$ says that $\mathbf{x}$ and $\mathbf{x}'$ are similar if they are similar under k_1 AND they are similar under k_2. Since you can define a lot of interesting boolean functions using AND and OR, you should be able to construct some pretty interesting kernels by taking sums and products of simpler kernels. A system called the **Automatic Statistician** exploited this insight by automatically searching over a large, open-ended space of kernel structures in order to best explain time series datasets. It would then give the user an automatically generated natural language report summarizing the structure in the dataset.[1]

From 2000 to 2010 or so, kernel SVMs were regarded as the best general-purpose classification algorithm. (I.e., you could do better than kernel SVMs by exploiting problem-specific structure, but if you just wanted to apply a learning algorithm to your data without thinking about it, SVMs were hard to beat.) The reason for this is that depending on the kernel, you could achieve very different behavior. A kernel SVM with a linear kernel is equivalent to a linear SVM, and therefore behaves similarly to logistic regression. A kernel SVM with an RBF kernel can learn complex, nonlinear decision boundaries, much like K-nearest-neighbors. Other kernels gave still different behavior. Hence, a single software package (LibSVM, now part of scikit-learn) was able to capture a wide range of model complexities, and one could choose between them simply by choosing a kernel (e.g. on a validation set).

Kernels aren't limited to vectors in $\mathbb{R}^D$. It's also possible to define kernels on discrete objects such as strings or graphs. This allows kernel SVMs to be extended to domains where it's not even obvious how to define a linear model to begin with.

3 Boosting

Now let's move onto the second class of algorithms for this lecture, namely boosting. Boosting is not actually a linear model per se, i.e. the decision boundaries aren't hyperplanes. It is actually another kind of ensemble method. But we discuss it today because it turns out we can interpret it in terms of minimizing a loss function, which lets us contrast it with other classification models we've covered. For simplicity, we'll focus on the setting

[1] See `https://www.automaticstatistician.com/index/` and the associated papers.

of binary classification; however, it's possible to generalize boosting to other situations.

Recall that ensemble methods combine the predictions of lots of individual models into an aggregated prediction; usually this is done by taking a (possibly weighted) vote of the individual predictors. We've seen one example of an ensemble method: bagging, where we train a bunch of models independently on datasets randomly sampled from the main dataset. We saw that the purpose of bagging is to reduce the variance of the predictions, but that it doesn't reduce the bias. Unlike with bagging, the goal of boosting is to make the prediction algorithm more powerful, i.e. to reduce its bias.

Boosting is different from bagging in that it's **adaptive**: each model is trained in a way that accounts for the errors made by previous models. More specifically, we construct a weighted training set, where examples we've done poorly on are weighted more heavily. In the classification setting, the **weighted error rate** can be written in the following way:

$$\frac{\sum_{i=1}^{N} w_i \mathbb{I}\{h_1(\mathbf{x}^{(i)}) \neq t^{(i)}\}}{\sum_{i=1}^{N} w_i}, \tag{36}$$

where the w_i are (nonnegative) weights assigned to the training examples and $\mathbb{I}\{\cdots\}$ is the indicator function for some condition, i.e. it returns 1 if that condition is true, and 0 otherwise. Observe that this is just the formula for the weighted average of 0–1 loss.

To introduce boosting, we'll first introduce the idea of a **weak classifier**. This is a classifier that is able to classify any (weighted) dataset with slightly better than chance accuracy; specifically, it achieves a (weighted) error rate of no more than $\frac{1}{2} - \gamma$ for some positive value γ. Note that this requirement is trivial to achieve for $\gamma = 0$, since if any weak classifier has an error rate above $\frac{1}{2}$, we can simply flip the predictions to get an error rate below $\frac{1}{2}$. Hence, we need $\gamma > 0$ in order for this requirement to be nontrivial. Boosting is based on the following question: given a weak classification algorithm, can we "boost" it into a strong classifier, i.e. one which gets near-perfect accuracy on the training set?

The canonical example of a weak classifier is a **decision stump**. As the name suggests, this is a decision tree of depth 1. I.e., we choose a single attribute and threshold its value. Fitting a decision stump to a weighted training set is simple: we just iterate over all attributes and thresholds, and choose the one that minimizes the weighted classification error.

We'll cover a boosting algorithm called **AdaBoost**, which was the first practically effective one. This is an iterative procedure as follows:

> Initialize all the weights uniformly, i.e. $w_i = 1/N$ for all i.
>
> For $t = 1, \ldots, T$:
>
> > Fit the weak classifier h_t to the current weighted training set.
> >
> > Compute its weighted error rate using (36).
> >
> > Compute
> >
> > $$\alpha_t = \frac{1}{2} \log \left(\frac{1 - \text{err}}{\text{err}} \right) \tag{37}$$
> >
> > Update the weights as follows:
> >
> > $$w_i \leftarrow w_i \exp \left(2\alpha_t \mathbb{I}\{h_t(\mathbf{x}^{(i)}) \neq t^{(i)}\} \right) \tag{38}$$
>
> Compute the final predictions as:
>
> $$H(\mathbf{x}) = \text{sign} \left(\sum_{t=1}^{T} \alpha_t h_t(\mathbf{x}) \right). \tag{39}$$

Observe that the value α_t is nonnegative because the best hypothesis must have a weighted error rate below $\frac{1}{2}$ (see above). Therefore, the final classifier $H(\mathbf{x})$ can be seen as a weighted majority vote of the individual weak classifiers. The formula for α_t may seem somewhat mysterious, but we'll see one interpretation for it later, and you'll derive another one for homework. The weight update (38) essentially upweights by a factor of $\exp(2\alpha_t)$ all the training examples which were classified incorrectly by the current hypothesis h_t. (Since only the normalized weights matter, this is equivalent to downweighting the correctly classified examples.)

It is possible to prove the following theorem which shows that we eventually wind up with a strong classifier:

> Assume that at each iteration of AdaBoost the WeakLearn returns a hypothesis with error $\text{err}_t \leq \frac{1}{2} - \gamma$ for all $t = 1, \ldots, T$ with $\gamma > 0$. The training error of the output hypothesis $H(\mathbf{x}) = \text{sign} \left(\sum_{t=1}^{T} \alpha_t h_t(\mathbf{x}) \right)$ is at most
>
> $$L_N(H) = \frac{1}{N} \sum_{i=1}^{N} \mathbb{I}\{H(\mathbf{x}^{(i)}) \neq t^{(i)})\} \leq \exp\left(-2\gamma^2 T\right).$$

This implies that the training error decreases exponentially as a function of the number of iterations. However, the rate of decrease is given by γ^2, so it might learn very slowly if the weak classifiers are only slightly better than chance. (And if the weak classifiers have only chance accuracy, i.e. $\gamma = 0$, then the bound is vacuous, as we would expect.)

Note that this bound is only about the training error, and doesn't say anything about generalization. Running more iterations of AdaBoost increases the complexity of the classifier, and hence could lead to overfitting. As with other learning algorithms, you might want to decide when to stop by monitoring the error on a validation set.

3.1 Interpretation as Stagewise Training

Our presentation of AdaBoost is basically a procedure, and doesn't give much insight into what it's actually doing. For most of the other algorithms we've discussed, we started with a loss function and figured out how to optimize it. This provided a form of modularity, since if one's unhappy with the performance, one can replace the loss function, while if the training is just too slow, then one can replace the algorithm. Can we do the same for boosting?

We'll interpret AdaBoost as fitting an **additive model**, which means its predictions are made using a sum

$$H_m(x) = \sum_{i=1}^{m} \alpha_i h_i(\mathbf{x}), \tag{40}$$

where the h_i correspond to the individual hypotheses (weak learners), and in the context of additive modeling, are also called **bases**. Note that additive models are generally more powerful than linear models, since the h_i themselves are nonlinear functions of $\mathbf{x}$. We interpret AdaBoost as an instance of **stagewise training**, which is a greedy approach to additive modeling which works as follows:

1. Initialize $H_0(x) = 0$
2. For $m = 1$ to T:
   - Compute the m-th hypothesis and its coefficient

$$(h_m, \alpha_m) \leftarrow \operatorname*{argmin}_{h \in \mathcal{H}, \alpha} \sum_{i=1}^{N} \mathcal{L}\left(H_{m-1}(\mathbf{x}^{(i)}) + \alpha h(\mathbf{x}^{(i)}), t^{(i)}\right)$$

   - Add it to the additive model

$$H_m = H_{m-1} + \alpha_m h_m$$

In other words, in each iteration we greedily choose a new basis and weight to minimize some loss function. But we don't go back and revisit our earlier choices.

The loss function we'll try to minimize is **exponential loss**, which is a bit different from the ones we've seen so far:

$$\mathcal{L}_{\mathrm{E}}(y, t) = \exp(-ty). \tag{41}$$

Now let's minimize it with respect to h and α. The first step is to factor out the part that depends on our previous choices, leaving us with a cost

function that only depends on the current h and α:

$$\begin{aligned}(h_m, \alpha_m) \leftarrow \operatorname*{argmin}_{h\in\mathcal{H},\alpha} \sum_{i=1}^{N} \exp\left(-\left[H_{m-1}(\mathbf{x}^{(i)}) + \alpha h(\mathbf{x}^{(i)})\right] t^{(i)}\right) \\ = \sum_{i=1}^{N} \exp\left(-H_{m-1}(\mathbf{x}^{(i)})t^{(i)} - \alpha h(\mathbf{x}^{(i)})t^{(i)}\right) \\ = \sum_{i=1}^{N} \exp\left(-H_{m-1}(\mathbf{x}^{(i)})t^{(i)}\right) \exp\left(-\alpha h(\mathbf{x}^{(i)})t^{(i)}\right) \\ = \sum_{i=1}^{N} w_i^{(m)} \exp\left(-\alpha h(\mathbf{x}^{(i)})t^{(i)}\right).\end{aligned} \tag{42}$$

In the last step, we simply *defined* $w_i^{(m)} \triangleq \exp\left(-H_{m-1}(\mathbf{x}^{(i)})t^{(i)}\right)$, i.e., we haven't yet related $w_i^{(m)}$ to the weights computed by AdaBoost. However, our suggestive notation is intentional, as these values will turn out to be exactly the AdaBoost weights. Hence, we want to solve the following minimization problem:

$$(h_m, \alpha_m) \leftarrow \operatorname*{argmin}_{h\in\mathcal{H},\alpha} \sum_{i=1}^{N} w_i^{(m)} \exp\left(-\alpha h(\mathbf{x}^{(i)})t^{(i)}\right).$$

First, suppose α is given, and try to find the optimal h. Observe that:

- If $h(\mathbf{x}^{(i)}) = t^{(i)}$, we have $\exp\left(-\alpha h(\mathbf{x}^{(i)})t^{(i)}\right) = \exp(-\alpha)$.
- If $h(\mathbf{x}^{(i)}) \neq t^{(i)}$, we have $\exp\left(-\alpha h(\mathbf{x}^{(i)})t^{(i)}\right) = \exp(+\alpha)$.

Hence, we get a loss of $\exp(\alpha)$ for every mistake and $\exp(-\alpha)$ for every correct answer. Since $\alpha > 0$, this basically means we'd like to minimize our weighted classification error. The following derivation works this out in more detail:

$$\begin{aligned}\sum_{i=1}^{N} w_i^{(m)} \exp\left(-\alpha h(\mathbf{x}^{(i)})t^{(i)}\right) =& e^{-\alpha} \sum_{i=1}^{N} w_i^{(m)} \mathbb{I}\{h(\mathbf{x}^{(i)}) = t_i\} + e^{\alpha} \sum_{i=1}^{N} w_i^{(m)} \mathbb{I}\{h(\mathbf{x}^{(i)}) \neq t_i\} \\ =& (e^{\alpha} - e^{-\alpha}) \sum_{i=1}^{N} w_i^{(m)} \mathbb{I}\{h(\mathbf{x}^{(i)}) \neq t_i\} + \\ & e^{-\alpha} \sum_{i=1}^{N} w_i^{(m)} \left[\mathbb{I}\{h(\mathbf{x}^{(i)}) \neq t_i\} + \mathbb{I}\{h(\mathbf{x}^{(i)}) = t_i\}\right] \\ =& (e^{\alpha} - e^{-\alpha}) \sum_{i=1}^{N} w_i^{(m)} \mathbb{I}\{h(\mathbf{x}^{(i)} \neq t_i\} + \\ & e^{-\alpha} \sum_{i=1}^{N} w_i^{(m)} \left[\mathbb{I}\{h(\mathbf{x}^{(i)} \neq t_i\} + \mathbb{I}\{h(\mathbf{x}^{(i)}) = t_i\}\right] \\ =& (e^{\alpha} - e^{-\alpha}) \sum_{i=1}^{N} w_i^{(m)} \mathbb{I}\{h(\mathbf{x}^{(i)}) \neq t_i\} + e^{-\alpha} \sum_{i=1}^{N} w_i^{(m)}.\end{aligned}$$

The second term is independent of h, and the first term is proportional to the weighted error rate. Hence, we minimize the loss by choosing h to minimize the weighted error rate.

Now for α:

$$\begin{aligned}
&\min_{\alpha} \min_{h \in \mathcal{H}} \sum_{i=1}^{N} w_i^{(m)} \exp\left(-\alpha h(\mathbf{x}^{(i)}) t^{(i)}\right) = \\
&\min_{\alpha} \left\{ (e^{\alpha} - e^{-\alpha}) \sum_{i=1}^{N} w_i^{(m)} \mathbb{I}\{h_m(\mathbf{x}^{(i)}) \neq t_i\} + e^{-\alpha} \sum_{i=1}^{N} w_i^{(m)} \right\} \\
=&\min_{\alpha} \left\{ (e^{\alpha} - e^{-\alpha}) \mathrm{err}_m \left(\sum_{i=1}^{N} w_i^{(m)} \right) + e^{-\alpha} \left(\sum_{i=1}^{N} w_i^{(m)} \right) \right\}
\end{aligned}$$

Taking the derivative with respect to α and setting it to zero, we get that

$$e^{2\alpha} = \frac{1 - \mathrm{err}_m}{\mathrm{err}_m} \Rightarrow \alpha = \frac{1}{2} \log\left(\frac{1 - \mathrm{err}_m}{\mathrm{err}_m}\right).$$

We've shown that α and h are each chosen the same way as in AdaBoost. It remains to show that the weights $w_i^{(m)}$ match the ones produced by AdaBoost:

$$\begin{aligned}
w_i^{(m+1)} &= \exp\left(-H_m(\mathbf{x}^{(i)}) t^{(i)}\right) \\
&= \exp\left(-\left[H_{m-1}(\mathbf{x}^{(i)}) + \alpha_m h_m(\mathbf{x}^{(i)})\right] t^{(i)}\right) \\
&= \exp\left(-H_{m-1}(\mathbf{x}^{(i)}) t^{(i)}\right) \exp\left(-\alpha_m h_m(\mathbf{x}^{(i)}) t^{(i)}\right) \\
&= w_i^{(m)} \exp\left(-\alpha_m h_m(\mathbf{x}^{(i)}) t^{(i)}\right) \\
&= w_i^{(m)} \exp\left(-\alpha_m \left(2\mathbb{I}\{h_m(\mathbf{x}^{(i)}) = t^{(i)}\} - 1\right)\right) \\
&= \exp(\alpha_m) w_i^{(m)} \exp\left(-2\alpha_m \mathbb{I}\{h_m(\mathbf{x}^{(i)}) = t^{(i)}\}\right).
\end{aligned}$$

This is the same as the AdaBoost formula up to the scale factor of $\exp(\alpha_m)$, which doesn't matter since the algorithm is invariant to rescaling the weights by a positive factor.

Hence, we've shown that AdaBoost greedily minimizes exponential loss $\mathcal{L}_{\mathrm{E}}(y, t) = \exp(-ty)$. What does this say about the algorithm? Think about how exponential loss compares to other loss functions. As the prediction is more confidently correct, the loss goes to zero, similarly to logistic-cross-entropy. But as the predictions get more wrong, the loss grows exponentially. This means the algorithm is really unhappy to make a confident wrong prediction, and will spend a lot of effort to prevent this from happening. Unfortunately, this means it can be sensitive to outliers or mislabeled data.

Interpreting boosting in terms of a loss function allows us to generalize the basic idea to other loss functions. All sorts of algorithms have been proposed along these lines. One particular software package called XGBoost implements boosting for general loss functions, and is currently by far the most successful black-box method at winning Kaggle competitions.