Contents

6 Machine Learning

6.1 Introduction

Machine learning algorithms (or just *learning* algorithms) are general purpose tools that solve problems from many domains without detailed domain-specific knowledge. They have proven to be very effective in a large number of contexts, including computer vision (such as detecting and recognizing faces in images), speech recognition, document classification, spam filtering, decision support, game playing, and many more.

A core problem that underlies many machine learning applications, and what we will focus on in this chapter, is the problem of *learning a good classifier from labeled data*. In this problem we have a domain of interest $\mathcal{X}$, called the *instance space*, such as email messages or patient records, and a classification task we would like to perform, such as classifying email messages into spam versus non-spam, or determining which patients will respond well to a given medical treatment. To do this, our algorithm is given labeled *training examples*, which are items from the domain paired with their correct classification. For example, this could be a collection of patients, each labeled by whether or not they responded well to the given medical treatment, or a collection of email messages, each labeled as spam or not. We will want our algorithm to use these to produce a classification rule that performs well over new data. We will focus in this chapter on *binary classification* (classifying items in $\mathcal{X}$ into two categories, as in the medical and spam-detection examples above) but one of course can consider multi-way classification as well.[1] The two categories are typically called the *positive* class and the *negative* class (from our perspective, it doesn't matter which category is called the positive class and which is called the negative class so long as we are consistent!).

The ideal function that maps each item in $\mathcal{X}$ to the correct category for that item is called the *target function* and we denote it as f^*. In other words, there is some function $f^* : \mathcal{X} \to \{-1, +1\}$ (where we are associating "+1" with positive examples and "−1" with negative examples) and our goal is to approximate f^* from labeled data. For this reason, the problem of learning a good classifier is often called the *function approximation* problem. In order to get started at this task, we will assume that our data items are represented using *d features* that are (hopefully) relevant to the classification we are trying to perform. These could be binary features such as "did this email message come from an email address the user has previously sent email to?" or real-valued features such as "what is the patient's LDL cholesterol level?" In particular, from now on we will associate $\mathcal{X}$ with R^d or $\{0, 1\}^d$. Additionally, our learning algorithm will be trying to approximate f^* by using a classifier from some class $\mathcal{H}$ called the *hypothesis class.* For example, $\mathcal{H}$ might be the set of all *linear separator* predictors: functions $h_{\mathbf{w}}$ (each associated

[1]One can reduce a multi-way classification task to a series of binary classification tasks, though in practice one may wish to solve the multi-way problem directly.

with its own vector $\mathbf{w} \in R^d$) such that $h_{\mathbf{w}}(\mathbf{x}) = 1$ if $\mathbf{w}\cdot\mathbf{x} > 0$ and $h_{\mathbf{w}}(\mathbf{x}) = -1$ if $\mathbf{w}\cdot\mathbf{x} \leq 0$.[2]

At this point, the reader may feel that our goal of learning a good classifier seems hopeless without some sort of assumptions. Indeed, such a reader would be correct. We will specifically consider two standard models:

Batch learning: In this model we assume that there is some (perhaps unknown) *probability distribution* $\mathcal{D}$ over the instance space $\mathcal{X}$. We assume we are given a training sample S consisting of *i.i.d.* draws from $\mathcal{D}$ and our goal is to perform well on new data items also drawn from $\mathcal{D}$. For instance, in the case of determining which patients will respond well to a given medical treatment, $\mathcal{D}$ would correspond to the overall population of people who are candidates for that treatment, and our assumption would be that our training data is a random sample from this population. A nice feature of this model is there is a well-defined notion of the *true error rate* of some proposed classifier h, namely $\Pr_{\mathbf{x}\sim\mathcal{D}}[h(\mathbf{x}) \neq f^*(\mathbf{x})]$, the chance of making a mistake on a new example from $\mathcal{D}$.

Online learning: In this model, we will no longer assume that data is drawn from some probability distribution. Instead, learning proceeds as a sequence of *trials*. In each trial, we are presented with some example $\mathbf{x} \in \mathcal{X}$, we are asked to predict its true label $f^*(\mathbf{x})$, and after that we are told if we made a mistake. For example, each morning you might look out the window and decide based on what you see whether or not to bring an umbrella to school; at the end of the day, you know if you made the right decision. In this model, our goal will be to bound the total number of *mistakes* we make over time. Note that we might or might not learn anything new from a new example—e.g., if we see the same $\mathbf{x}$ over and over again, we won't learn anything new, but this is fine since we (presumably) will not be making mistakes on it either. The key challenge is that when we *do* make a mistake, we will need to learn from it. Algorithms in this model need to very explicitly be able to learn from their mistakes.

In the following sections, we consider first batch learning and then online learning. We will see a formal connection between learning and the notion of Occam's razor, and discuss a range of algorithms including the Perceptron algorithm, Stochastic Gradient Descent, Kernel methods, and Boosting. We will examine notions of regularization and confidence bounds, and will see the important notion of VC-dimension for controlling overfitting.

6.2 Batch learning, Occam's razor, and Uniform convergence

We now consider the batch learning model mentioned above, and discuss how in this model we can develop algorithms and also make a formal connection to the philosophical

[2]Technically, this is the class of *homogeneous* linear separators because the separating hyperplane passes through the origin. More generally, one could also consider a nonzero threshold b. Note that by adding a "dummy feature" x_0 that is equal to 1 in every example, one can achieve the same classification with a homogeneous linear separator by setting $w_0 = -b$.

notion of Occam's razor.

Recall that in the batch model, we are given a sample S of labeled training data, e.g., 100 email message correctly labeled according to whether or not they are spam. These data points are assumed to be drawn from some probability distribution $\mathcal{D}$ over the instance space, and our goal is to use this data to produce a classifier $h \in \mathcal{H}$ that performs well on new examples from $\mathcal{D}$.[3] In particular, as mentioned above we can formally define the *true error rate* of some classifier h produced by the algorithm as $err_{\mathcal{D}}(h) = \Pr_{\mathbf{x}\sim\mathcal{D}}[h(\mathbf{x}) \neq f^*(\mathbf{x})]$. This is what we will want to be low. There is also a natural notion of the observed, or *empirical* error rate of some proposed h, namely the fraction of data points in S on which h makes a mistake. We will call this $err_S(h)$. A proposed rule h is said to be *overfitting the training data* if $err_S(h)$ is substantially less than $err_{\mathcal{D}}(h)$. As part of our analysis and algorithm development, we will need to find ways to ensure that our algorithms are not "fooled" into producing rules of high true error that only *appear* to be good because they are overfitting their data.

How can we design good learning algorithms and also prevent overfitting? We begin with a simple problem of learning disjuctions.

6.2.1 Learning disjunctions

Suppose that the instance space $\mathcal{X} = \{0,1\}^d$ and we believe that the target function f^* can be represented by a *disjunction*, or OR-function, over features, such as $f^*(\mathbf{x}) = x_1 \vee x_4 \vee x_7 \vee x_8$. For example, if we are trying to predict whether an email message is spam or not, and our features correspond to the presence or absence of different possible indicators of spam-ness, then this would correspond to the belief that there is some subset of these indicators such that every spam email has at least one of them and every non-spam email has none of them. Here is a simple algorithm we could use in this case. What we will do is (a) design an efficient algorithm that finds a disjunction consistent with our training sample S if one exists, and then (b) argue that so long as our training sample S is large enough, it is highly unlikely that such a rule will be overfitting by too much.

Simple Disjunction Learner: Given sample S, discard all features that are set to 1 in any negative example in S. Output the function h that is the disjunction of all features that remain.

Lemma 6.1 *The Simple Disjunction Learner produces a disjunction h that is consistent with the sample S (i.e., with $err_S(h) = 0$) whenever such a disjunction exists.*

Proof: Suppose there exists a disjunction f with $err_S(f) = 0$. Then for any $x_i \in f$, x_i will not be set to 1 in any negative example by definition of the OR-function. Therefore,

[3]In the case of email messages, this is probably not a great model because the notion of what a "typical" spam or non-spam email looks like may change over time, though it may be reasonable in the short term.

h will contain x_i as well, i.e., $\{x_i \in h\} \supseteq \{x_i \in f\}$. This means that h will be correct on all positive examples in S (since each one must have some $x_i \in f$ set to 1). Furthermore, h will be correct on all negative examples in S since by design all features set to 1 in any negative example were discarded. Therefore, h is correct on all examples in S. ∎

We now argue that so long as S is sufficiently large, the hypothesis h produced by the above algorithm is unlikely to have overfit by much. This is sometimes called a *sample complexity* bound or a *generalization guarantee*. We will show this by arguing that it is unlikely there will be *any* disjunction of high true error consistent with the training data.

Lemma 6.2 *Let $\epsilon, \delta > 0$. If a training sample S is drawn from $\mathcal{D}$ of size*

$$|S| \geq \frac{1}{\epsilon}[d\ln(2) + \ln(1/\delta)],$$

then with probability $\geq 1-\delta$, every disjunction h with $err_{\mathcal{D}}(h) \geq \epsilon$ has $err_S(h) > 0$ (equivalently, every disjunction h with $err_S(h) = 0$ has $err_{\mathcal{D}}(h) < \epsilon$).

Proof: Let $h_1, h_2, \ldots, h_N$ be the set of all disjunctions with $err_{\mathcal{D}}(h) \geq \epsilon$ (i.e., these are the "bad" disjunctions that we don't want to output). Consider now drawing the sample S and let A_i be the event that h_i is consistent with S. Since every example in S is drawn iid from distribution $\mathcal{D}$, for any given i we have:

$$\Pr[A_i] \;\leq\; (1-\epsilon)^{|S|},$$

by the fact that h_i has true error rate at least ϵ. In other words, if we fix some h_i and then draw our sample S, the chance that h_i makes no mistakes on S is at most the probability that a coin of bias ϵ comes up tails $|S|$ times in a row, which is $(1-\epsilon)^{|S|}$. Therefore, by the union bound and the fact that $N \leq 2^d$ (since there are only 2^d disjunctions in total) we have:

$$\Pr[\cup_i A_i] \;\leq\; 2^d(1-\epsilon)^{|S|}.$$

Finally, using the fact that $(1-\epsilon)^{1/\epsilon} \leq 1/e$, we have that the probability that any disjunction h with $err_{\mathcal{D}}(h) \geq \epsilon$ has $err_S(h) = 0$ is at most $2^d e^{-\epsilon|S|}$. Plugging in the sample size bound in the Lemma, this is at most $2^d e^{-d\ln(2)-\ln(1/\delta)} = \delta$ as desired. ∎

Together, Lemmas 6.1 and 6.2 give us the following theorem:

Theorem 6.3 *If the target function f^* is a disjunction, then with probability $\geq 1-\delta$, the Simple Disjunction Learner returns a classifier h with $err_{\mathcal{D}}(h) < \epsilon$ given a training sample S of size at least $\frac{1}{\epsilon}[d\ln(2) + \ln(1/\delta)]$.*

What we have just shown is sometimes called a "PAC-learning guarantee" since we have argued that the rule produced by our algorithm is *P*robably *A*pproximately *C*orrect.

6.2.2 Occam's razor

Occam's razor is the notion, stated by William of Occam around AD 1320, that in general one should prefer simpler explanations over more complicated ones.[4] Why should one do this, and can we make a formal claim about why this is a good idea? What if each of us disagrees about precisely which explanations are simpler than others? It turns out we will be able to build on the analysis in the proof of Lemma 6.2 to make a general and compelling mathematical statement of Occam's razor that addresses these issues.

First, the reader might notice that in the proof of Lemma 6.2, the only place we used the fact that h was a disjunction was in arguing that $N \leq 2^d$. We could just have easiliy used any hypothesis class $\mathcal{H}$, replacing the term "$d\ln(2)$" with $\ln(|\mathcal{H}|)$. In particular, we have:

Theorem 6.4 *Fix any hypothesis class $\mathcal{H}$ and let $\epsilon, \delta > 0$. If a training sample S is drawn from $\mathcal{D}$ of size*

$$|S| \geq \frac{1}{\epsilon}[\ln(|\mathcal{H}|) + \ln(1/\delta)],$$

then with probability $\geq 1-\delta$, every $h \in \mathcal{H}$ with $err_{\mathcal{D}}(h) \geq \epsilon$ has $err_S(h) > 0$. Equivalently, with probability $\geq 1-\delta$, every $h \in \mathcal{H}$ with $err_S(h) = 0$ has

$$err_{\mathcal{D}}(h) < \frac{\ln(|\mathcal{H}|) + \ln(1/\delta)}{|S|}.$$

Now, what do we mean by a rule being "simple"? Let's assume that each of us has some way of describing rules, using bits (since we are computer scientists). The methods, also called *description languages*, used by each of us may be different, but one fact we can say for certain is that in any given description language, there are at most 2^b rules that can be described using fewer than b bits (because $1 + 2 + 4 + \ldots + 2^{b-1} < 2^b$). Therefore, by setting $\mathcal{H}$ to be the set of all rules that can be described in fewer than b bits and plugging into Theorem 6.4, we have the following:

Theorem 6.5 (Occam's razor) *Fix any description language, and consider a training sample S drawn from distribution $\mathcal{D}$. With probability $\geq 1-\delta$, any rule consistent with S that can be described in this language using fewer than b bits will have $err_D(h) \leq \epsilon$ for*

$$|S| = \frac{1}{\epsilon}[b\ln(2) + \ln(1/\delta)], \quad \textit{or equivalently} \quad err_D(h) \leq \frac{b\ln(2) + \ln(1/\delta)}{|S|}.$$

For example, using the fact that $\ln(2) < 1$ and ignoring the low-order $\ln(1/\delta)$ term, this means that if the number of bits it takes to write down our rule is at most 10% of the number of data points in our sample, then we can be confident it will have error at most 10% with respect to $\mathcal{D}$. What is perhaps surprising about this theorem is that it means that we can each have different ways of describing rules and yet all use Occam's razor. Note that the theorem does *not* say that complicated rules are necessarily bad, only that simple rules are unlikely to fool us since there are just not that many simple rules.

[4]The statement more explicitly was that "Entities should not be multiplied unnecessarily."

6.2.3 Application: learning decision trees

One popular practical method for machine learning is to learn a *decision tree*; see Figure **??**. While finding the smallest decision tree that fits a given training sample S is NP-hard, there are a number of heuristics that are used in practice. One popular heuristic, called ID3, selects the feature to put inside any given node v by choosing the feature of largest *information gain*, a measure of how much it is directly improving prediction.[5] This then continues until all leaves are pure—they have only positive or only negative examples. Suppose we run ID3 on a training set S and it outputs a tree with n nodes. Such a tree can be described using $O(n \log d)$ bits: $\log_2(d)$ bits to give the index of the feature in the root, $O(1)$ bits to indicate for each child if it is a leaf and if so what label it should have, and then $O(n_L \log d)$ and $O(n_R \log d)$ bits respectively to describe the left and right subtrees, where n_L is the number of nodes in the left subtree and n_R is the number of nodes in the right subtree. So, by Theorem 6.5, we can be confident the true error is low if we can produce a consistent tree with much fewer than $\epsilon|S|/\log(d)$ nodes.

6.2.4 Agnostic Learning and Uniform Convergence

Our analysis so far has addressed the case that we are able to find a hypothesis with zero error on the training set S. But what if the best $h \in \mathcal{H}$ we can find has, say, 5% error on S? Can we still be confident that its true error under $\mathcal{D}$ is low, say at most 10%? For this, we want an analog of Theorem 6.4 that says that with high probability, *every* $h \in \mathcal{H}$ has $err_S(h)$ within $\pm\epsilon$ of $err_D(h)$. Such a statement is called *uniform convergence* because we are asking that empirical errors converge to their true errors uniformly over all functions in $\mathcal{H}$. These are also often called "agnostic learning" bounds because they are most relevant when we do not necessarily believe that f^* lies in $\mathcal{H}$, we simply want to perform as well as we can using the hypothesis class $\mathcal{H}$ at hand.[6]

To prove uniform convergence bounds, we will need to use a tail inequality for sums of independent Bernoulli random variables (i.e., coin tosses). The following is particularly convenient and is a variation on the Chernoff bounds in Section **??** of the appendix.

Theorem 6.6 (Hoeffding bounds) *Let $X_1, \ldots, X_n$ be independent $\{0,1\}$-valued random variables with $p = \Pr[X_i = 1]$, and let $s = \sum_i X_i$ (equivalently, flip n coins of bias p*

[5]Formally, using S_v to denote the set of examples in S that reach node v, and supposing that feature x_i partitions S_v into S_v^0 and S_v^1 (the examples in S_v with $x_i = 0$ and $x_i = 1$, respectively), the information gain of x_i is defined as: $Ent(S_v) - [\frac{|S_v^0|}{|S_v|}Ent(S_v^0) + \frac{|S_v^1|}{|S_v|}Ent(S_v^1)]$. Here, $Ent(S')$ is the binary entropy of the label proportions in set S'; that is, if a p fraction of the examples in S' are positive, then $Ent(S') = p\log_2(1/p) + (1-p)\log_2(1/(1-p))$, defining $0\log_2(0) = 0$.

[6]It could also be that the features we are measuring are simply insufficient to make a perfect prediction. E.g., in general from observable features it may be impossible to perfectly predict whether a patient will respond well to a given medical treatment.

and let s be the total number of heads). Then for any $0 \leq \alpha \leq 1$,

$$\begin{aligned}\Pr[s/n > p+\alpha] &\leq e^{-2n\alpha^2} \\ \Pr[s/n < p-\alpha] &\leq e^{-2n\alpha^2}.\end{aligned}$$

Using Theorem 6.6 we immediately have the following uniform convergence analog of Theorem 6.4.

Theorem 6.7 (Uniform convergence) *Fix any hypothesis class $\mathcal{H}$ and let $\epsilon, \delta > 0$. With probability $\geq 1-\delta$, if a training sample S of size*

$$|S| \geq \frac{1}{2\epsilon^2}[\ln(|\mathcal{H}|) + \ln(2/\delta)]$$

is drawn from $\mathcal{D}$, then $|err_S(h) - err_D(h)| \leq \epsilon$ for all $h \in \mathcal{H}$.

Proof: Fix some $h \in \mathcal{H}$ and let X_i be the indicator random variable for the event that h makes a mistake on the ith example in S. These are independent $\{0,1\}$ random variables with $\Pr[X_i = 1] = err_D(h)$, and the fraction of the X_i's equal to 1 is exactly the empirical error of h. Therefore, Hoeffding bounds guarantee that $\Pr[err_S(h) - err_D(h)| \geq \epsilon] \leq 2e^{-2|S|\epsilon^2}$. Applying the union bound to all $h \in \mathcal{H}$ we have

$$\Pr[\exists h \in \mathcal{H} : |err_S(h) - err_D(h)| \leq \epsilon] \quad \leq \quad 2|\mathcal{H}|e^{-2|S|\epsilon^2}$$

Plugging in the value of $|S|$ from the theorem statement we find that the right-hand-side above is at most δ as desired. ■

Theorem 6.7 justifies the approach of optimizing over our training sample S even if we are not able to find a rule of zero empirical error. If our training set S is sufficiently large, we are guaranteed that with high probability, good performance on S will translate to good performance under $\mathcal{D}$.

6.2.5 Regularization

If we take Theorem 6.7 and rewrite the guarantee to solve for ϵ in terms of $|S|$ and $|\mathcal{H}|$, we get that with probability $\geq 1-\delta$, all $h \in \mathcal{H}$ satisfy

$$err_D(h) \quad \leq \quad err_S(h) + \sqrt{\frac{\ln(|\mathcal{H}|) + \ln(2/\delta)}{2|S|}}.$$

The second term on the right-hand-side above is a bound on the maximum amount of overfitting that occurs over all functions in $\mathcal{H}$, with high probability. Here, the "$\ln(|\mathcal{H}|)$" term can be viewed as a measure of complexity of this class of functions. Now, suppose we want to optimize over functions of multiple levels of complexity. To address this, consider fixing some description language and let $\mathcal{H}_i$ denote those functions that can be described in i bits in this language, so $|\mathcal{H}_i| \leq 2^i$. Let $\delta_i = \delta/2^i$. We know that with probability at least $1-\delta_i$, all $h \in \mathcal{H}_i$ satisfy $err_D(h) \leq err_S(h) + \sqrt{\frac{\ln(|\mathcal{H}_i|)+\ln(2/\delta_i)}{2|S|}}$. Now, applying the union bound over all i, using the fact that $\delta_1 + \delta_2 + \delta_3 + \ldots = \delta$, and also the fact that $\ln(|\mathcal{H}_i|) + \ln(2/\delta_i) \leq i\ln(4) + \ln(2/\delta)$, we get the following corollary.

Corollary 6.8 *Fix any description language, and consider a training sample S drawn from distribution $\mathcal{D}$. With probability $\geq 1-\delta$, all hypotheses h satisfy*

$$err_D(h) \quad \leq \quad err_S(h) + \sqrt{\frac{\text{size}(h)\ln(4) + \ln(2/\delta)}{2|S|}}$$

where $\text{size}(h)$ *denotes the number of bits needed to describe h in the given language.*

Corollary 6.8 has an interesting implication. It tells us that rather than searching for a rule of low training error, we instead may want to search for a rule with a low right-hand-side in the displayed formula. If we can find one for which this quantity is small, we can be confident true error will be low as well. This is the idea of *regularization*. We add onto the training error a term called a *regularizer* that penalizes complex functions, and then search for a rule with a low sum of training error plus regularizer. Later, we will see in Support Vector Machines the use of regularizers such that if we replace training error with an upper bound called "hinge loss," we can efficiently optimize the sum via convex optimization.

6.3 The Perceptron Algorithm and Stochastic Gradient Descent

We now describe a classic algorithm for learning linear separators called the Perceptron algorithm, and a widely-used generalization of this algorithm known as Stochastic Gradient Descent.

6.3.1 The Perceptron Algorithm

The *Perceptron algorithm* is a classic algorithm for learning a linear separator [Blo62, Nov62, MP69]. We will describe it here as an algorithm in the batch learning model, though it is also naturally viewed as an online algorithm, and we will examine its properties in the online model in Section 6.4. The Perceptron algorithm learns a homogeneous linear separator, i.e., a linear separator that passes through the origin, and we will assume that all data points have been pre-scaled to lie in the unit ball. Note that scaling the length of the examples does not affect which side of the separator they are on.

The Perceptron Algorithm:

```
Given: training sample $S$ of points in $R^d$ with labels $f^*(\mathbf{x}) \in \{-1,1\}$. Assume $|\mathbf{x}| \leq 1$ for
    all $\mathbf{x} \in S$.

Goal: produce a weight vector $\mathbf{w}$ such that $h_{\mathbf{w}}(\mathbf{x}) = \text{sgn}(\mathbf{w}\cdot\mathbf{x})$ is correct for all $\mathbf{x} \in S$.

1. Start with the all-zeroes weight vector $\mathbf{w} = \mathbf{0}$.

2. For each $\mathbf{x} \in S$, if $\text{sgn}(\mathbf{w}\cdot x) \neq f^*(\mathbf{x})$, update: $\mathbf{w} \leftarrow \mathbf{w} + f^*(\mathbf{x})\mathbf{x}$.

3. Repeat Step 2 until $\text{sgn}(\mathbf{w}\cdot x) = f^*(\mathbf{x})$ for all $\mathbf{x} \in S$.
```

Assume that our data set S is indeed linearly separable: namely, there exists a weight

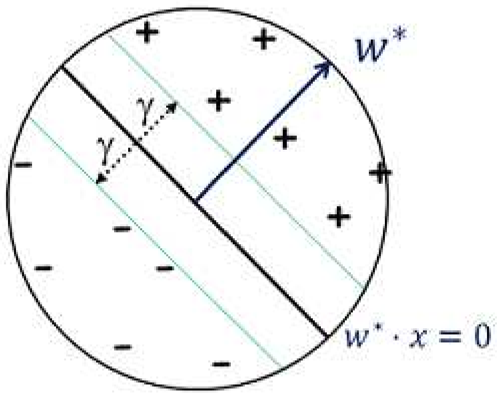

Figure 6.1: Positive and negative examples separated by margin γ. The vector w^* is orthogonal to the hyperplane $w^* \cdot x = 0$.

vector $\mathbf{w}^*$ such that $f^*(\mathbf{x}) = \text{sgn}(\mathbf{w}^* \cdot \mathbf{x})$ for all $\mathbf{x} \in S$. Without loss of generality, assume $\mathbf{w}^*$ is a unit-length vector; this does not affect the sign of the dot products. Define the *margin* of separation

$$\gamma = \min_{\mathbf{x} \in S} |\mathbf{w}^* \cdot \mathbf{x}|.$$

This is the minimum distance between any data point and the hyperplane $\mathbf{w}^* \cdot \mathbf{x} = 0$ (see Figure 6.1). We will show that the Perceptron algorithm finds a consistent weight vector $\mathbf{w}$ after at most $1/\gamma^2$ updates. In particular, this will be true when $\mathbf{w}^*$ is defined to be the maximum margin separator: the separator of maximum γ.

Theorem 6.9 *The Perceptron algorithm finds a consistent weight vector after at most $1/\gamma^2$ updates.*

Proof: We will examine two quantities: $\mathbf{w} \cdot \mathbf{w}^*$ and $|\mathbf{w}|^2$.

The first claim is that after each update, $\mathbf{w} \cdot \mathbf{w}^*$ increases by at least γ. That is because, by definition of γ, if we update by adding a positive example $\mathbf{x}$, then we have $(\mathbf{w} + \mathbf{x}) \cdot \mathbf{w}^* = \mathbf{w} \cdot \mathbf{w}^* + \mathbf{x} \cdot \mathbf{w}^* \geq \mathbf{w} \cdot \mathbf{w}^* + \gamma$. Similarly, if we update by subtracting a negative example, then we have $(\mathbf{w} - \mathbf{x}) \cdot \mathbf{w}^* = \mathbf{w} \cdot \mathbf{w}^* - \mathbf{x} \cdot \mathbf{w}^* \geq \mathbf{w} \cdot \mathbf{w}^* + \gamma$.

The second claim is that after each update, $|\mathbf{w}|^2$ increases by at most 1. This uses the fact that we only update on examples for which we made a mistake. Specifically, if we update by adding a positive example $\mathbf{x}$, then we have

$$(\mathbf{w} + \mathbf{x}) \cdot (\mathbf{w} + \mathbf{x}) = |\mathbf{w}|^2 + 2\mathbf{w} \cdot \mathbf{x} + |\mathbf{x}|^2 \leq |\mathbf{w}_t|^2 + 1.$$

The last inequality above uses the fact that (a) $\mathbf{w} \cdot \mathbf{x} \leq 0$ since we only update when $\text{sgn}(\mathbf{w} \cdot \mathbf{x}) \neq f^*(\mathbf{x})$, and (b) $|\mathbf{x}| \leq 1$. The exact same thing holds (flipping signs) when we update on a negative $\mathbf{x}$, namely $(\mathbf{w} - \mathbf{x}) \cdot (\mathbf{w} - \mathbf{x}) = |\mathbf{w}|^2 - 2\mathbf{w} \cdot \mathbf{x} + |\mathbf{x}|^2 \leq |\mathbf{w}_t|^2 + 1$.

Putting the above two claims together, after T updates we have $\mathbf{w} \cdot \mathbf{w}^* \geq \gamma T$ and $|\mathbf{w}|^2 \leq T$. Using the fact that $|\mathbf{w}| \geq \mathbf{w} \cdot \mathbf{w}^*$ (since $\mathbf{w}^*$ is a unit-length vector), we have:

$$\sqrt{T} \geq |\mathbf{w}| \geq \mathbf{w} \cdot \mathbf{w}^* \geq \gamma T.$$

This implies that $T \leq 1/\gamma^2$ as desired. ■

What can we say about performance on new data? Using Theorem 6.5, one statement we can make is that if we run the Perceptron algorithm on a machine that stores each weight as a 64-bit floating-point number, then if S has size at least $\frac{1}{\epsilon}[64d\ln(2)+\ln(1/\delta)]$, we can be confident that any rule $h_{\mathbf{w}}$ produced with $err_S(h_{\mathbf{w}}) = 0$ will have $err_{\mathcal{D}}(h_{\mathbf{w}}) \leq \epsilon$. In Section 6.4.4 (Online to Batch Conversion) we will see a different generalization guarantee we can give in terms of γ rather than d, and in Section 6.9 (VC-dimension) we will see a bound in terms of d that does not depend on the bit-precision of our machine.

6.3.2 Stochastic Gradient Descent

We now describe a very practical and widely-used algorithm in machine learning, called *stochastic gradient descent* (SGD). The Perceptron algorithm we examined in Section 6.3.1 can be viewed as a special case of this algorithm, as can methods for deep learning.

For stochastic gradient descent, we assume our hypotheses can be described using a vector of weights. That is, $\mathcal{H} = \{h_{\mathbf{w}}\}$, where $h_{\mathbf{w}}$ is parametrized by a weight vector $\mathbf{w} \in R^m$, and typically $m \geq d$. The function $h_{\mathbf{w}}$ is of the form $h_{\mathbf{w}}(\mathbf{x}) = \text{sgn}(f_{\mathbf{w}}(\mathbf{x}))$, where $f_{\mathbf{w}} : R^d \to R$ is a real-valued function in an auxiliary class $\mathcal{F}$. So, $f_{\mathbf{w}}$ is actually doing all the work, and $h_{\mathbf{w}}$ is just converting its real-valued prediction to "positive" or "negative". To apply stochastic gradient descent, we also need a *loss function* $L(f_{\mathbf{w}}(\mathbf{x}), f^*(\mathbf{x}))$ that describes the real-valued penalty we will associate with function $f_{\mathbf{w}}$ for its prediction on an example $\mathbf{x}$ whose true label is $f^*(\mathbf{x})$. The algorithm is then the following:

Stochastic Gradient Descent:

Given: starting point $\mathbf{w} = \mathbf{w}_{init}$ and learning rates $\lambda_1, \lambda_2, \lambda_3, \ldots$
(e.g., $\mathbf{w}_{init} = \mathbf{0}$ and $\lambda_t = 1$ for all t, or $\lambda_t = 1/\sqrt{t}$).

Consider a sequence of random examples[7] $(\mathbf{x}_1, f^*(\mathbf{x}_1)), (\mathbf{x}_2, f^*(\mathbf{x}_2)), \ldots$.

1. Given example $(\mathbf{x}_t, f^*(\mathbf{x}_t))$, compute the gradient $\nabla L(f_{\mathbf{w}}(\mathbf{x}_t), f^*(\mathbf{x}_t))$ of the loss of $f_{\mathbf{w}}(\mathbf{x}_t)$ with respect to the weights $\mathbf{w}$. This is a vector in R^m whose ith component is $\frac{\partial L(f_{\mathbf{w}}(\mathbf{x}_t), f^*(\mathbf{x}_t))}{\partial w_i}$.
2. Update: $\mathbf{w} \leftarrow \mathbf{w} - \lambda_t \nabla L(f_{\mathbf{w}}(\mathbf{x}_t), f^*(\mathbf{x}_t))$.

[7]We will think of each example as a new random draw from $\mathcal{D}$, though in practice we will be cycling through the training set S. Bottou [?] recommends randomly shuffling S at the start just in case (in practice) it was not actually an i.i.d random sample in the first place, along with other useful practical recommendations.

Let's now try to understand the algorithm better by seeing a few examples of instantiating the class of functions $\mathcal{F}$ and loss function L.

First, consider $m = d$ and $f_{\mathbf{w}}(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x}$, so $\mathcal{F}$ is the class of linear predictors. Let's also assume we have pre-scaled each example to lie inside the unit ball. Consider the loss function $L(f_{\mathbf{w}}(\mathbf{x}), f^*(\mathbf{x})) = \max(0, -f^*(\mathbf{x})f_{\mathbf{w}}(\mathbf{x}))$, and recall that $f^*(\mathbf{x}) \in \{-1, 1\}$. In other words, if $f_{\mathbf{w}}(\mathbf{x})$ has the correct sign, then we have a loss of 0, otherwise we have a loss equal to the magnitude of $f_{\mathbf{w}}(\mathbf{x})$. In this case, if $f_{\mathbf{w}}(\mathbf{x})$ has the correct sign and is non-zero, then the gradient will be zero since an infinitesimal change in any of the weights will not change the sign. So, when $h_{\mathbf{w}}(\mathbf{x})$ is correct, the algorithm will leave $\mathbf{w}$ alone. On the other hand, if $f_{\mathbf{w}}(\mathbf{x})$ has the wrong sign, then $\frac{\partial L}{\partial w_i} = -f^*(\mathbf{x})\frac{\partial \mathbf{w}\cdot\mathbf{x}}{\partial w_i} = -f^*(\mathbf{x})x_i$. So, using $\lambda_t = 1$, the algorithm will update $w \leftarrow w + f^*(\mathbf{x})\mathbf{x}$. Note that this is exactly the Perceptron algorithm. (Technically we must address the case that $f_{\mathbf{w}}(\mathbf{x}) = 0$; in this case, we should view $f_{\mathbf{w}}$ as having the wrong sign just barely.)

As a small modification to the above example, consider the same class of linear predictors $\mathcal{F}$ but now slightly modify the loss function to $L(f_{\mathbf{w}}(\mathbf{x}), f^*(\mathbf{x})) = \max(0, 1 - f^*(\mathbf{x})f_{\mathbf{w}}(\mathbf{x}))$. This loss function, called *hinge loss*, now requires $f_{\mathbf{w}}(\mathbf{x})$ to have the correct sign *and* magnitude at least 1 in order to be zero. Hinge loss has the useful property that it is an upper bound on error rate: for any sample S, $err_S(h_{\mathbf{w}}) \le \sum_{\mathbf{x}\in S} L(f_{\mathbf{w}}(\mathbf{x}), f^*(\mathbf{x}))$. If we instantiate SGD with this $(\mathcal{F}, L)$ we get a method called the *margin perceptron* algorithm.

More generally, we could have a much more complex class $\mathcal{F}$. For example, consider a layered circuit of "soft threshold" gates: each node in the circuit computes a linear function of its inputs and then passes this value through an "activation function" such as $a(x) = 1/(1-e^{-x})$. This circuit could have multiple layers with the output of layer i being used as the input to layer $i+1$. The vector $\mathbf{w}$ would be the concatenation of all the weight vectors in the network, so we might have $m \gg d$. This is the idea of *deep neural networks.*

While it is difficult to give general guarantees on when SGD will succeed in finding a hypothesis of low error on its training set S, we can use Theorems 6.5 and 6.7 to argue that if it does so and if S is sufficiently large, we can be confident that its true error will be low as well. Specifically, suppose that SGD is run on a machine where each weight is stored as a 64-bit floating point number. This means that its hypothesis can each be described using $64m$ bits. So, applying Theorem 6.5, if S has size at least $\frac{1}{\epsilon}[64m\ln(2) + \ln(1/\delta)]$, it is unlikely any hypothesis in $\mathcal{H}$ of true error greater than ϵ will be consistent with the sample, and so if it finds a hypothesis consistent with S, we can be confident its true error is at most ϵ. Or, using Theorem 6.7, if $|S| \ge \frac{1}{2\epsilon^2}[64m\ln(2) + \ln(2/\delta)]$ then we can be confident that the final hypothesis h produced by SGD satisfies $err_{\mathcal{D}}(h) \le err_S(h) + \epsilon$.

6.4 Online learning

We now switch to the online learning model. Recall that in this model, learning proceeds in a sequence of trials: in each trial we are given an example x, asked to predict its label $f^*(x)$, and then are told whether we were correct or not. Rather than assuming that data necessarily arrives from some probability distribution and aiming to find a function of low "true error", we will simply aim to minimize the number of mistakes made over time. In particular, we will aim for theoretical guarantees that apply to any sequence of examples. We begin by revisiting the disjunction-learning problem discussed in Section 6.2.1 but now in the online model, examine the Perceptron algorithm discussed in Section 6.3.1 from this perspective, and then give a general online-to-batch conversion procedure that converts any algorithm with a good mistake bound in the online model to one with strong guarantees in the batch (distributional) model.

6.4.1 Online learning of disjunctions

Recall the setting of Section 6.2.1: our instance space $\mathcal{X} = \{0,1\}^d$ and we are told the target function f^* can be represented as an OR-function over features. For this problem, we have the following simple algorithm that guarantees never to make more than d mistakes.

Simple Online Disjunction Learner:

Initialize $h = x_1 \vee x_2 \vee \ldots \vee x_d$.

1. Given example x, predict $h(x)$.
2. If $h(x)$ predicted positive but $f^*(x)$ is negative, remove from h any variable set to 1 in x.

Theorem 6.10 *The Simple Online Disjunction Learner makes at most d mistakes whenever the target function f^* is a disjunction.*

Proof: We first claim that the algorithm maintains the invariant that $\{x_i \in h\} \supseteq \{x_i \in f^*\}$. The reason is that (a) this holds at the start by the initialization step, and (b) variables are only removed from h when they are set to 1 in negative examples, which means they cannot be in f^*. Next, observe that the invariant guarantees that the algorithm will never make a mistake on a positive example, since if x satisfies f^*, it must satisfy h as well. Finally, each mistake on a negative example removes at least one variable from h. Since we begin with d variables in h and the total number of variable can never be negative, this means the total number of mistakes is at most d. ∎

We complement Theorem 6.10 with a matching lower bound. For this lower bound, assume that a disjunction of size 0 is allowed and corresponds to the "all false" function.

Theorem 6.11 *For any deterministic algorithm A there exists a sequence of examples σ and disjunction f^* such that A makes d mistakes on sequence σ labeled by f^*.*

Proof: Let σ be the sequence $e_1, e_2, \ldots, e_d$ where e_j is the example that is zero everywhere except for a 1 in the jth position. Imagine running A on sequence σ and telling A it made a mistake on every example; that is, if A predicts positive on e_j we set $f^*(e_j) = -1$ and if A predicts negative on e_j we set $f^*(e_j) = +1$. This target corresponds to the disjunction of all x_j such that A predicted negative on e_j, so it is a legal disjunction. Since A is deterministic, the fact that we constructed f^* by running A is not a problem: it would make the same mistakes if re-run from scratch on the same sequence and same target. Therefore, A makes d mistakes on this σ and f^*. ∎

6.4.2 Halving and Occam's razor

If we are not concerned with running time, a simple algorithm that guarantees to make at most $\log_2(|\mathcal{H}|)$ mistakes for a target belonging to any given class $\mathcal{H}$ is called the *halving algorithm*. This algorithm simply maintains the *version space* $\mathcal{H}' \subseteq \mathcal{H}$ consisting of all $h \in \mathcal{H}$ consistent with the labels on every example seen so far, and predicts based on majority vote. Each mistake is guaranteed to reduce the size of $\mathcal{H}'$ by at least half (hence the name).

Suppose that we are interested in a class such as decision trees where some hypotheses are more complicated than others. Specifically, assume we have some description language that allows us to describe each hypothesis using bits, and let us assume this description language is *prefix free*, meaning that no hypothesis has a description that is a prefix of the description of any other hypothesis. E.g., if we think of binary a tree where a 0 means "branch left" and a 1 means "branch right" then each hypothesis can be placed as a leaf of this tree, at depth equal to the number of bits in its description. Now, suppose we give each hypothesis h a weight $w(h) = 1/2^{\text{size}(h)}$, where size$(h)$ is the number of bits in its description. Then we have $\sum_h w(h) \leq 1$.[8] This means that if we modify the halving algorithm so that it predicts based on a *weighted* majority vote over its version space, then since the total sum of weights is initially at most 1 and can never be less than $w(f^*)$, the total number of mistakes is at most size(f^*). This is called the *generalized halving algorithm*. Thus we have the following analog to Theorem 6.5.

Theorem 6.12 *Given any prefix-free description language, generalized halving makes at most* size(f^*) *mistakes on any sequence of examples consistent with f^*.*

[8]This can be seen by imagining placing 1 ounce of gold dust at the root of the tree, then splitting this dust equally between its children, and continuing the process down to the leaves; a leaf at depth d will receive exactly $1/2^d$ ounces of the gold dust.

6.4.3 The Perceptron Algorithm

We can restate the Perceptron algorithm discussed earlier as a learning algorithm in the online model. Assume that all examples have been pre-scaled to lie inside the unit ball.

The Perceptron Algorithm:

1. Start with the all-zeroes weight vector $\mathbf{w} = \mathbf{0}$.
2. Given example $\mathbf{x}$, predict $\operatorname{sgn}(\mathbf{w} \cdot x)$.
3. If the prediction on $\mathbf{x}$ was incorrect, update: $\mathbf{w} \leftarrow \mathbf{w} + f^*(\mathbf{x})\mathbf{x}$.

We can restate Theorem 6.9 in the online model as follows. Say that a sequence of examples is consistent with a separator of margin γ if there exists a unit-length vector $\mathbf{w}^*$ such that $f^*(\mathbf{x})(\mathbf{w}^* \cdot \mathbf{x}) \geq \gamma$ for all examples $\mathbf{x}$ in the sequence.

Theorem 6.13 *The Perceptron algorithm makes at most $1/\gamma^2$ updates on any sequence of examples consistent with a separator of margin γ.*

6.4.4 Online to Batch Conversion

Suppose we have an online algorithm with a good mistake bound. Can we use it to get a guarantee in the distributional (batch) learning setting? Intuitively, the answer should be yes since the online setting is only harder. Indeed, this intuition is correct. We present here two natural approaches for such online to batch conversion.

Conversion procedure 1: Random Stopping. Suppose we have an online algorithm $\mathcal{A}$ with mistake-bound M. Say we run the algorithm in a single pass on a sample S of size M/ϵ. Let X_i be the indicator random variable for the event that $\mathcal{A}$ makes a mistake on the ith example. Since $\sum_{i=1}^{|S|} X_i \leq M$ for *any* set S, we certainly have that $\mathbf{E}[\sum_{i=1}^{|S|} X_i] \leq M$ where the expectation is taken over the random draw of S from $\mathcal{D}^{|S|}$. By linearity of expectation, and dividing both sides by $|S| = M/\epsilon$ we therefore have:

$$\frac{1}{|S|}\sum_{i=1}^{|S|} \mathbf{E}[X_i] \quad \leq \quad \epsilon. \tag{6.1}$$

Let h_i denote the hypothesis used by algorithm $\mathcal{A}$ to predict on the ith example. Since the ith example was randomly drawn from $\mathcal{D}$, we have $\mathbf{E}[err_{\mathcal{D}}(h_i)] = \mathbf{E}[X_i]$. This means that if we choose i at random from 1 to $|S|$ (i.e., stop the algorithm at a random time), the expected error of the resulting prediction rule, taken over the randomness in the draw of S and the choice of i, is at most ϵ as given by equation (6.1). Thus we have:

Theorem 6.14 (Online to Batch via Random Stopping) *If an online algorithm $\mathcal{A}$ with mistake-bound M is run on a sample S of size M/ϵ and stopped at a random time between 1 and $|S|$, the expected error of the hypothesis h produced satisfies $\mathbf{E}[err_{\mathcal{D}}(h)] \leq \epsilon$.*

Conversion procedure 2: Large Gap. A second natural approach to using an online learning algorithm $\mathcal{A}$ in the distributional setting is to just run it on random examples until a sufficiently large gap between consecutive mistakes is observed. For simplicity, assume for now that algorithm $\mathcal{A}$ is "conservative," meaning that it does not change its hypothesis when it predicts correctly (e.g., the Perceptron and disjunction algorithms are all conservative). Consider running $\mathcal{A}$ on a series of random examples from $\mathcal{D}$, and let e_i denote the error rate of the hypothesis used between the ith and $(i+1)$st mistake. If $e_i > \epsilon$ then the chance that this hypothesis predicts correctly for $\frac{1}{\epsilon}\log(\frac{1}{\delta_i})$ examples in a row is at most

$$(1-\epsilon)^{\frac{1}{\epsilon}\log(\frac{1}{\delta_i})} \leq \delta_i.$$

Let $\delta_i = \delta/(i+2)^2$ so we have $\sum_{i=0}^{\infty}\delta_i = (\frac{\pi^2}{6}-1)\delta \leq \delta$. Applying the union bound over all i, we can halt with confidence whenever $\mathcal{A}$ predicts correctly for so many examples in a row:

Theorem 6.15 (Online to Batch via Large Gap) *Let $\mathcal{A}$ be a conservative online learning algorithm, and suppose we run $\mathcal{A}$ until it has predicted correctly for $\frac{1}{\epsilon}\log(\frac{1}{\delta_i})$ examples in a row after its ith mistake. Then with probability at least $1-\delta$, this procedure will either run forever or halt with a hypothesis of error at most ϵ. Moreover, if $\mathcal{A}$ has a mistake bound of M, then this procedure will halt after $O(\frac{M}{\epsilon}\log(\frac{M}{\delta}))$ examples.*

Proof: The first claim follows from the discussion above. The second claim follows from the fact that $\sum_{i=0}^{M}\frac{1}{\epsilon}\log(1/\delta_i) \leq \sum_{i=0}^{M}\frac{1}{\epsilon}\log((M+2)^2/\delta) = O((\frac{M}{\epsilon}\log(\frac{M}{\delta}))$. ∎

We can remove the assumption that $\mathcal{A}$ is conservative by referring not to the error of the final hypothesis but instead to the average error of hypotheses used since its last mistake, and using the fact that $(1-\epsilon_1)(1-\epsilon_2) \leq (1-\frac{\epsilon_1+\epsilon_2}{2})^2$. In other words, algorithm $\mathcal{A}$ cannot increase its chance of satisfying our stopping criteria with average error greater than ϵ by varying the error rates of the hypotheses it is using. This has a nice implication for Stochastic Gradient Descent, which in general is not conservative. If we run SGD until the gap between consecutive mistakes is sufficiently large, we can be confident that the average error of hypotheses produced in that gap is low.

6.5 Margins, Hinge-loss, Support-Vector Machines, and Perceptron revisited

So far we have considered the problem of learning a linear separator when there is a perfect separator to be found. Often, however, data will only be "mostly" linearly separable. Unfortunately, finding the separator that makes the fewest mistakes on a given dataset S is NP-complete. However, one task we *can* solve efficiently is to find a separator of minimum *hinge loss*. Specifically, for each example $\mathbf{x}_i$ we define a slack variable $\xi_i \geq 0$

and then solve the linear program:

$$\begin{aligned} \text{minimize} \quad & \sum_i \xi_i \\ \text{subject to} \quad & \mathbf{w} \cdot \mathbf{x}_i \geq 1 - \xi_i \text{ for all positive examples } \mathbf{x}_i \\ & \mathbf{w} \cdot \mathbf{x}_i \leq -1 + \xi_i \text{ for all negative examples } \mathbf{x}_i \\ & \xi_i \geq 0 \text{ for all } i \end{aligned}$$

The sum of slack variables is called the total hinge loss (see also Section 6.3.2). In practice, one problem with this linear program is that given a choice of a perfect separator with a very tiny margin, or a separator with a small amount of hinge loss but a larger margin, one may prefer the latter. The reason is that separators with large margin turn out to enjoy stronger overfitting guarantees. In particular, we may wish to use $1/\gamma^2$ as a regularization term where γ is the margin of the separator. Recall that when we defined margin, we normalized by the length of the prediction vector; if we do that in the linear program above, we see that the margin is $1/|\mathbf{w}|$, so $1/\gamma^2 = |\mathbf{w}|^2$. The method known as Support Vector Machines (SVMs) does exactly this regularization, optimizing for a combination of hinge loss and margin. Specifically, SVMs solve the convex optimization problem (here, C is a constant that is determined empirically):

$$\begin{aligned} \text{minimize} \quad & |\mathbf{w}|^2 + C \sum_i \xi_i \\ \text{subject to} \quad & \mathbf{w} \cdot \mathbf{x}_i \geq 1 - \xi_i \text{ for all positive examples } \mathbf{x}_i \\ & \mathbf{w} \cdot \mathbf{x}_i \leq -1 + \xi_i \text{ for all negative examples } \mathbf{x}_i \\ & \xi_i \geq 0 \text{ for all } i. \end{aligned}$$

Alternatively, rather than perform this convex optimization, it turns out that the simple Perceptron algorithm actually enjoys strong guarantees in terms of hinge loss and margin, *automatically* doing nearly as well as the optimal tradeoff between the two quantities:

Theorem 6.16 *On any sequence of examples in the unit ball, the number of mistakes of the Perceptron algorithm satisfies:*

$$\#\ \textit{mistakes} \leq \min_{\mathbf{w}^*} \left[|\mathbf{w}^*|^2 + 2(\textit{total hinge loss of } \mathbf{w}^*) \right].$$

Proof: Fix some $\mathbf{w}^*$ and consider the Perceptron algorithm run on a sequence of examples $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \ldots$. Define ξ_i to be the minimum non-negative value such that $f^*(\mathbf{x}_i)(\mathbf{w}^* \cdot \mathbf{x}_i) \geq 1 - \xi_i$, i.e., the hinge loss of $\mathbf{w}^*$ on $\mathbf{x}_i$. Now, if the Perceptron algorithm makes a mistake on some $\mathbf{x}_i$, we add $f^*(\mathbf{x}_i)\mathbf{x}_i$ to $\mathbf{w}$. This increases $\mathbf{w}^* \cdot \mathbf{w}$ by at least $1 - \xi_i$. So, if the Perceptron algorithm makes M mistakes we have $\mathbf{w} \cdot \mathbf{w}^* \geq M - L$ where L is the total hinge loss of $\mathbf{w}^*$. On the other hand, each mistake increases $|\mathbf{w}|^2$ by at most 1 since if we make a mistake on a example $\mathbf{x}_i$ we have

$$|\mathbf{w} + f^*(\mathbf{x}_i)\mathbf{x}_i|^2 = |\mathbf{w}|^2 + 2(\mathbf{w} \cdot f^*(\mathbf{x}_i)\mathbf{x}_i) + |\mathbf{x}_i|^2 \leq |\mathbf{w}^2| + 1$$

where the last inequality comes from the fact that $\mathbf{w}$ made a mistake so $\mathbf{w} \cdot f^*(\mathbf{x}_i)\mathbf{x}_i \leq 0$, and $\mathbf{x}_i$ lies in the unit ball so $|\mathbf{x}_i|^2 \leq 1$. Now, using the fact that $\mathbf{w}^* \cdot \mathbf{w} \leq |\mathbf{w}^*||\mathbf{w}|$ we get

$$\begin{aligned}
M - L &\leq |\mathbf{w}^*|\sqrt{M} \\
M^2 - 2ML + L^2 &\leq |\mathbf{w}^*|^2 M \qquad \text{(square both sides)} \\
M - 2L + L^2/M &\leq |\mathbf{w}^*|^2 \qquad \text{(divide by } M\text{)} \\
M &\leq |\mathbf{w}^*|^2 + 2L. \quad \text{(ignore negative term } -L^2/M\text{)}
\end{aligned}$$

■

6.6 Nonlinear Separators and Kernel Functions

What if our data doesn't even have a "pretty good" linear separator? For example, perhaps the positive examples lie inside the unit ball and the negative examples lie outside the unit ball. Or perhaps there is some other smooth but nonlinear surface that separates the positive and negative examples. An approach to addressing problems of this nature is to use a tool called Kernel functions, also called the *kernel trick*.

One thing we might like to do is map our data to a higher dimensional space, e.g., look at all products of pairs or triples of features, in the hope that data will be linearly separable in this $O(d^2)$ or $O(d^3)$-dimensional space. If we are lucky, data will be separable by a large margin in this new space so we don't have to pay a lot in terms of mistakes if we run, say, the Perceptron algorithm. But this is going to be a huge amount of work computationally if we have to explicitly compute all these products to map our data into this higher-dimensional space. However, it turns out that many learning algorithms only access data through performing dot-products—we will see how to interpret the Perceptron algorithm in this way shortly. So, perhaps we can perform our mapping *implicitly* by just providing a simple way to compute the associated dot-product. This is the idea behind kernel functions.

Definition 6.1 *A* kernel function *is a function $K(\mathbf{x}, \mathbf{x}')$ such that for some $\phi : \mathcal{X} \to R^m$ (m could even be infinite) we have*

$$K(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x}) \cdot \phi(\mathbf{x}').$$

Some examples of kernel functions are:

- $K(\mathbf{x}, \mathbf{x}') = (1 + \mathbf{x} \cdot \mathbf{x}')^k$ for integer $k \geq 1$. This is called the polynomial kernel.
- $K(\mathbf{x}, \mathbf{x}') = (1 + x_1 x_1')(1 + x_2 x_2')...(1 + x_d x_d')$. This is called the all-products kernel.
- String kernels, which count how many different substrings of some given length p two strings $\mathbf{x}$ and $\mathbf{x}'$ have in common.

Let's see why the above functions indeed satisfy the kernel definition. The easiest is the string kernel. This corresponds to a ϕ that maps examples into an implicit feature space with one coordinate for each possible string of length p: $\phi(x)$ has a 1 in coordinate j if x has string j as a substring, and a 0 if not.

For the all-products kernel, let's first look at the case $d = 2$. Here we get $K(\mathbf{x}, \mathbf{x}') = 1 + x_1x_1' + x_2x_2' + x_1x_2x_1'x_2'$. Thus, this corresponds to a mapping $\phi(\mathbf{x}) = (1, x_1, x_2, x_1x_2)$. For $d = 3$ we get the previous kernel times $(1 + x_3x_3')$ which corresponds to the mapping $\phi(\mathbf{x}) = (1, x_1, x_2, x_1x_2, x_3, x_1x_3, x_2x_3, x_1x_2x_3)$. More generally, by induction, we can see this kernel corresponds to a dot-product in a space with one coordinate for every subset $A \subseteq \{1, \ldots, d\}$ where coordinate A of $\phi(\mathbf{x})$ equals $\prod_{i \in A} x_i$.

For the polynomial kernel, it is helpful to establish some composition properties of kernel functions. The fact that it is a legal kernel follows immediately from the following theorem (plus the fact that the constant function $K(\mathbf{x}, \mathbf{x}') = 1$ is a legal kernel).

Theorem 6.17 *Suppose K and K' are kernel functions. Then*

1. *For any constant $c \geq 0$, cK is a legal kernel.*
2. *The sum $K + K'$, is a legal kernel.*
3. *The product, KK', is a legal kernel.*

Proof: Let ϕ, ϕ' denote mappings such that $K(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x}) \cdot \phi(\mathbf{x}')$ and $K'(\mathbf{x}, \mathbf{x}') = \phi'(\mathbf{x}) \cdot \phi'(\mathbf{x}')$. We can now define ϕ'' for the new kernel in each case as follows.

1. Use $\phi''(\mathbf{x}) = \sqrt{c}\phi(\mathbf{x})$. So $\phi''(\mathbf{x}) \cdot \phi''(\mathbf{x}') = \sqrt{c}\phi(\mathbf{x}) \cdot \sqrt{c}\phi(\mathbf{x}') = cK(\mathbf{x}, \mathbf{x}')$.
2. Use $\phi''(\mathbf{x}) = \phi(\mathbf{x}) \circ \phi'(\mathbf{x})$ where "$\circ$" is concatenation. Then $\phi''(\mathbf{x}) \cdot \phi''(\mathbf{x}') = \phi(\mathbf{x}) \cdot \phi(\mathbf{x}') + \phi'(\mathbf{x}) \cdot \phi'(\mathbf{x}') = K(\mathbf{x}, \mathbf{x}') + K'(\mathbf{x}, \mathbf{x}')$.
3. Use $\phi''(\mathbf{x})$ as the outer-product of $\phi(\mathbf{x})$ and $\phi'(\mathbf{x})$. In particular, $\phi''(\mathbf{x})_{ij} = \phi(\mathbf{x})_i\phi'(\mathbf{x})_j$. Now we get:

$$\begin{aligned}
\phi''(\mathbf{x}) \cdot \phi''(\mathbf{x}') &= \sum_{ij} \phi(\mathbf{x})_i\phi'(\mathbf{x})_j\phi(\mathbf{x}')_i\phi'(\mathbf{x}')_j \\
&= (\sum_i \phi(\mathbf{x})_i\phi(\mathbf{x}')_i)(\sum_j \phi'(x)_j\phi'(\mathbf{x}')_j) \\
&= K(\mathbf{x}, \mathbf{x}')K'(\mathbf{x}, \mathbf{x}').
\end{aligned}$$

■

What makes kernel functions important for learning is that many of the algorithms for learning linear separators only access their data via taking dot-products. So, if we use the kernel function any time that a dot product is requested, the algorithm will act *as if* we had explicitly mapped all examples $\mathbf{x}$ using $\phi(\mathbf{x})$. Let's consider the Perceptron algorithm. Let $\mathcal{M}$ denote the set of examples on which a mistake was made by the algorithm so far. Then, the current weight vector $\mathbf{w}$ is exactly:

$$\mathbf{w} = \sum_{\mathbf{x}_i \in \mathcal{M}} f^*(\mathbf{x}_i)\mathbf{x}_i.$$

The algorithm makes predictions by computing $\mathbf{w} \cdot \mathbf{x}$. So, to run it in the "$\phi$ space" we just plug in the above formula for $\mathbf{w}$ and replace each dot-product with a kernel function. That is, we predict on example $\mathbf{x}$ using:

$$\operatorname{sgn}\left(\sum_{\mathbf{x}_i \in \mathcal{M}} f^*(\mathbf{x}_i) K(\mathbf{x}_i, \mathbf{x}) \right).$$

Support Vector Machines can similarly be "kernelized", that is, run using a kernel function instead of dot-product, by writing them in their dual form.

6.7 Strong and Weak Learning - Boosting

A strong learner for a class $\mathcal{H}$ is an algorithm that if $f^* \in \mathcal{H}$ is able with high probability to achieve any desired error rate ϵ, using a number of samples that may depend polynomially on $1/\epsilon$. For example, we presented a strong learner for the class of disjunctions. A weak learner for a class $\mathcal{H}$ is an algorithm that just needs to do a little bit better than random guessing. It is only required to get error rate $(\frac{1}{2} - \gamma)$ for some constant $\gamma > 0$. We show here that if we have a weak-learner for a class $\mathcal{H}$, and this algorithm achieves the weak-learning guarantee for any distribution of data $\mathcal{D}$, we can "boost" it to a strong learner, using the technique of Boosting.

For simplicitly, lets assume our given weak learning algorithm A outputs hypotheses that can be described using at most b bits. Our boosted algorithm will produce hypotheses that will be majority votes over T hypotheses from A, for T defined below. This means the hypotheses of our boosted algorithm can be described using $O(bT)$ bits. So, by Theorem 6.5, it will suffice to draw a sample S of n labeled examples $\mathbf{x}_1, \ldots, \mathbf{x}_n$ at the beginning of the process for $n \gg \frac{1}{\epsilon}(bT + \log(1/\delta))$ and show that the rule produced by our procedure is correct on the sample.

Our assumption is that A is a weak-learner over any distribution on data. This in particular includes distributions that correspond to different ways of weighting the points in the sample S. This is in fact all we will need. Specifically, we can define our assumption on A as follows.

Definition 6.2 (γ-Weak learner on sample) *A weak learner over a training sample $S = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ is an algorithm that given the examples, their labels, and a nonnegative real weight w_i on each example $\mathbf{x}_i$ as input, produces a classifier that correctly labels a subset of examples with total weight at least $(\frac{1}{2} + \gamma) \sum_{i=1}^{n} w_i$.*

Figure 6.2: Learner produced by boosting algorithm

We will show that a strong learner can be built by making $T = O(\log n)$ calls to a weak learner by a method called boosting. Boosting makes use of the intuitive notion that if an example was misclassified, one needs to pay more attention to it.

Boosting algorithm

> Make the first call to the weak learner with all w_i set equal to 1.
>
> At time $t+1$ multiply the weight of each example that was misclassified the previous time by $\alpha = \frac{\frac{1}{2}+\gamma}{\frac{1}{2}-\gamma}$. Leave the other weights as they are. Make a call to the weak learner.
>
> After T steps, stop and output the following classifier:
>
> > Label each of the examples $\{\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_n}\}$ by the label given to it by a majority of calls to the weak learner. Assume T is odd, so there is no tie for the majority.

Suppose m is the number of examples the final classifier gets wrong. Each of these m examples was misclassified at least $T/2$ times so each has weight at least $\alpha^{T/2}$. This says the total weight is at least $m\alpha^{T/2}$. On the other hand, at time $t+1$, only the weights of examples misclassified at time t were increased. By the property of weak learning, the total weight of misclassified examples is at most $(\frac{1}{2}-\gamma)$ of the total weight at time t. Let weight(t) be the total weight at time t. Then

$$\begin{aligned}\text{weight}(t+1) &\le \left(\alpha\left(\frac{1}{2}-\gamma\right)+\left(\frac{1}{2}+\gamma\right)\right)\times\text{weight}(t)\\ &= (1+2\gamma)\times\text{weight}(t).\end{aligned}$$

Thus, since weight$(0) = n$,

$$m\alpha^{T/2} \le \text{Total weight at end} \le n(1+2\gamma)^T.$$

Plugging in $\alpha = \frac{1/2+\gamma}{1/2-\gamma} = \frac{1+2\gamma}{1-2\gamma}$ and rearranging terms we get:

$$m \;\le\; n[(1-2\gamma)(1+2\gamma)]^{T/2} \;=\; n[1-4\gamma^2]^{T/2}.$$

Finally, using the fact that $1-x \le e^{-x}$ we have:

$$m \;\le\; ne^{-2T\gamma^2}.$$

For $T > \frac{\ln n}{2\gamma^2}$ we have $m < 1$, so the number of misclassified items m must be 0. ∎

Now that we have completed the proof of the boosting result, here are two interesting observations:

Connection to Hoeffding bounds: The boosting result applies even if our weak learning algorithm is "adversarial", giving us the least helpful classifier possible subject to satisfying Definition 6.2. (For instance, this is why we don't want to set α to be *too* large in the Boosting algorithm: because then the weak learner could just give us the negation of the classifier it gave us last time.) Suppose, though, that the weak learning algorithm was "nice" and each time just gave a classifier that for each example $\mathbf{x}_i$, flipped a coin and produced the correct answer with probability $\frac{1}{2} + \gamma$ and the wrong answer with probability $\frac{1}{2} - \gamma$ (so it is a γ-weak learner in expectation). In that case, if we repeatedly called it T times, for any fixed $\mathbf{x}_i$, Hoeffding bounds imply the chance the majority vote of those classifiers is incorrect on $\mathbf{x}_i$ is at most $e^{-2T\gamma^2}$. So, the expected total number of mistakes m is at most $ne^{-2T\gamma^2}$. What is interesting is that this is the exact bound we get from boosting (without the expectation) for an adversarial weak-learner!

A minimax view: Consider a 2-player zero-sum game with one row for each example $\mathbf{x}_i$ and one column for each hypothesis h_j that the weak-learning algorithm might possibly output. Say that if the row player chooses row i and the column player chooses column j, then the column player gets a payoff of 1 if $h_j(\mathbf{x}_i)$ is correct, else it gets a payoff of 0 if $h_j(\mathbf{x}_i)$ is incorrect. The γ-weak learning assumption implies that for any randomized strategy for the row player (any "mixed strategy" in the language of game theory), there exists a response h_j that gives the column player an expected payoff at least $\frac{1}{2} + \gamma$. The von Neumann minimax theorem states that this implies that there exists a probability distribution on the columns (a mixed strategy for the column player) such that for any $\mathbf{x}_i$, at least a $\frac{1}{2} + \gamma$ probability mass of the columns under this distribution is correct on $\mathbf{x}_i$. We can think of boosting as a fast way of finding a very simple probability distribution on the columns (just an average over $O(\log n)$ columns, possibly with repetitions) that is nearly as good (for any $\mathbf{x}_i$, more than half are correct) that moreover works even if our only access to the columns is by running the weak learner and observing what it outputs.

6.8 Combining (Sleeping) Expert Advice

Imagine you have access to a large collection of rules-of-thumb that specify what to predict in different situations. For example, in classifying news articles, you might have one that says "if the article has the word 'football' then classify it as sports" and another that says "if the article contains a dollar figure, then classify it as business". These rules might at times contradict each other (e.g., a news article that has both the word 'football' and a dollar figure) and it may be that none is perfectly accurate and indeed some are much better than others. We present here an algorithm for combining a large number of such rules with the guarantee that if any of them indeed are good, the algorithm will perform nearly as well as each good rule on the examples on which that rule applies.

Formally, assume we have n rules $h_1, \ldots, h_n$, and let S_i denote the subset of examples in which rule h_i fires, i.e., the examples satisfying its if-condition. We consider the online learning model, and let $mistakes(A, S)$ denote the number of mistakes of some algorithm A on a sequence of examples S. Then the guarantee of our algorithm A is that:

$$\text{For all } i, E[mistakes(A, S_i)] \leq (1+\epsilon) \cdot mistakes(h_i, S_i) + O\left(\frac{\log n}{\epsilon}\right)$$

where ϵ is a parameter of algorithm A and the expectation is over internal randomness in the randomized algorithm A.

As a special case, if $h_1, \ldots, h_n$ always fire, and indeed are the members of a hypothesis class $\mathcal{H}$, then A performs nearly as well as the best function in $\mathcal{H}$. This can be viewed as a noise-tolerant version of the Halving Algorithm of Section 6.4.2 for the case that no function in $\mathcal{H}$ is perfect. The case of rules that all fire on every example is often called the problem of *combining expert advice*, and the more general case of rules that sometimes fire is called the *sleeping experts* problem (viewing a rule not firing as it being asleep).

Combining Sleeping Experts Algorithm:

1. Initialize each expert h_i with a weight $w_i = 1$. Let $\epsilon \in (0, 1)$.
2. Given example x, let H_x denote the experts h_i that make a prediction, and let $W_x = \sum_{h_j \in H_x} w_j$. Choose $h_i \in H_x$ with probability $p_{ix} = w_i / W_x$ and predict $h_i(x)$.
3. For each $h_i \in H_x$, update its weight as follows:
    - Let $m_{ix} = 1$ if $h_i(x)$ was incorrect, else let $m_{ix} = 0$.
    - Let $r_{ix} = \left(\sum_{h_j \in H_x} p_{jx} m_{jx}\right) / (1+\epsilon) - m_{ix}$.
    - Update $w_i \leftarrow w_i (1+\epsilon)^{r_{ix}}$.
      Note that $\sum_{h_j \in H_x} p_{jx} m_{jx}$ represents the algorithm's probability of making a mistake on example x. So, h_i is rewarded for predicting correctly ($m_{ix} = 0$) especially when the algorithm had a high probability of making a mistake, and h_i is penalized for predicting incorrectly ($m_{ix} = 1$) especially when the algorithm had a low probability of making a mistake.

   For each $h_i \notin H_x$, leave w_i alone.

Theorem 6.18 *For any set of n if-then rules (sleeping experts) $h_1, \ldots, h_n$, and for any sequence of examples S, the Combining Sleeping Experts Algorithm A satisfies:*

$$\textit{For all } i, E[mistakes(A, S_i)] \leq (1+\epsilon) \cdot mistakes(h_i, S_i) + O\left(\frac{\log n}{\epsilon}\right)$$

where $S_i = \{x \in S : h_i \in H_x\}$.

Proof: Consider some sleeping expert h_i. The weight of h_i after the sequence of examples S is exactly:

$$\begin{aligned} w_i &= (1+\epsilon)^{\sum_{x \in S_i}\left[\left(\sum_{h_j \in H_x} p_{jx} m_{jx}\right)/(1+\epsilon) - m_{ix}\right]} \\ &= (1+\epsilon)^{E[mistakes(A,S_i)]/(1+\epsilon) - mistakes(h_i,S_i)}. \end{aligned}$$

Let $W = \sum_j w_j$. Clearly $w_i \leq W$. Therefore, taking logs, we have:

$$E[mistakes(A,S_i)]/(1+\epsilon) - mistakes(h_i,S_i) \;\leq\; \log_{1+\epsilon} W.$$

So, using the fact that $\log_{1+\epsilon} W = O(\frac{\log W}{\epsilon})$,

$$E[mistakes(A,S_i)] \;\leq\; (1+\epsilon)\cdot mistakes(h_i,S_i) + O\left(\frac{\log W}{\epsilon}\right).$$

Initially, $W = n$. To prove the theorem, it thus is enough to just prove that W never increases. To do so, we need to show that for each x, $\sum_{h_i \in H_x} w_i(1+\epsilon)^{r_{ix}} \leq \sum_{h_i \in H_x} w_i$, or equivalently (dividing both sides by $\sum_{h_j \in H_x} w_j$) that $\sum_i p_{ix}(1+\epsilon)^{r_{ix}} \leq 1$, where for convenience we define $p_{ix} = 0$ for $h_i \notin H_x$.

For this we will use the inequalities that for $\beta, z \in [0,1]$ we have $\beta^z \leq 1 - (1-\beta)z$ and $\beta^{-z} \leq 1 + (1-\beta)z/\beta$. Specifically, we will use $\beta = (1+\epsilon)^{-1}$. We now have:

$$\begin{aligned} \sum_i p_{ix}(1+\epsilon)^{r_{ix}} &= \sum_i p_{ix}\beta^{m_{ix} - (\sum_j p_{jx} m_{jx})\beta} \\ &\leq \sum_i p_{ix}\Big(1 - (1-\beta)m_{ix}\Big)\left(1 + (1-\beta)\left(\sum_j p_{jx} m_{jx}\right)\right) \\ &\leq \left(\sum_i p_{ix}\right) - (1-\beta)\sum_i p_{ix} m_{ix} + (1-\beta)\sum_i p_{ix}\sum_j p_{jx} m_{jx} \\ &= 1 - (1-\beta)\sum_i p_{ix} m_{ix} + (1-\beta)\sum_j p_{jx} m_{jx} \\ &= 1, \end{aligned}$$

where the second-to-last line follows from using $\sum_i p_{ix} = 1$ in two places. So W never increases and the bound follows as desired. ■

6.9 VC-Dimension

In Section 6.2 we presented several theorems showing that so long as the training set S is sufficiently large compared to $\log(|\mathcal{H}|)$, we can be confident that functions $h \in \mathcal{H}$ that perform well on S will also perform well on $\mathcal{D}$. In essence, these results used $\log(|\mathcal{H}|)$ as a measure of complexity of class $\mathcal{H}$. VC-dimension is a different, tighter measure of complexity for a class of functions and, as we will see, also is sufficient to yield confidence bounds. For any class $\mathcal{H}$, $\text{VCdim}(\mathcal{H}) \leq \log_2(|\mathcal{H}|)$ and often it is quite a bit smaller. For example, there are infintely many linear separators in R^2 and yet their VC-dimension is only 3. We begin with an auxiliary definition:

Definition 6.3 *Given a set S of examples and class of functions $\mathcal{H}$, let $\mathcal{H}[S]$ denote the set of all labelings of the points in S that are consistent with some function in $\mathcal{H}$. We say that S is* **shattered** *if $|\mathcal{H}[S]| = 2^{|S|}$.*

For example, any set S of three non-collinear points in the plane is shattered by the set of linear separators, since any of the 8 possible ways of labeling them with labels in $\{-1, 1\}$ can be realized by a function in this class.

Definition 6.4 *For integer m and class $\mathcal{H}$, let $\mathcal{H}[m] = \max_{|S|=m} |\mathcal{H}[S]|$; this is called the* **growth function** *of $\mathcal{H}$. The* **VC-dimension** *of $\mathcal{H}$ is the largest m such that $\mathcal{H}[m] = 2^m$. That is, VC-dimension is the size of the largest shattered set.*

For example, consider the class of linear separators in the plane. As noted above, any set of three non-collinear points is shattered by the class of linear separators, and yet it is not hard to see that no set of 4 points can be shattered. In particular, for any set of 4 points in the plane, out of all the 16 possible binary labelings, at most 14 will be realizable using linear separators. So, the VC-dimension of linear separators in the plane is 3. The growth function of this class satisfies $\mathcal{H}[m] = O(m^2)$, since for any labeling consistent with a linear separator, you can describe a consistent separator using two boundary data points along with $O(1)$ additional bits giving the labels of those two points and stating which side is which. More generally, the VC-dimension of linear separators in R^d is $d+1$ (see Section 6.9.4). As another example, the class of intervals $[a, b]$ on the real line has VC-dimension 2 since any set of two points is shattered, and yet for three points $x_1 < x_2 < x_3$ it is not possible to label x_1 and x_3 positive but x_2 negative. The growth function of this class also satisfies $\mathcal{H}[m] = O(m^2)$. Notice that $\text{VCdim}(\mathcal{H}) \leq \log_2(|\mathcal{H}|)$ since $\mathcal{H}$ must have at least 2^d functions in order to shatter d points.

We now will prove three important theorems relating VC-dimension to learnability. The first two can be viewed as analogs of Theorem 6.4 and Theorem 6.7 respectively, replacing the size of the class $\mathcal{H}$ with its growth function. The third relates the growth function of a class to its VC-dimension.

Theorem 6.19 (VC Occam bound) *For any class $\mathcal{H}$ and distribution $\mathcal{D}$, if a training sample S is drawn from $\mathcal{D}$ of size*

$$m \;\geq\; \frac{2}{\epsilon}[\log_2(2\mathcal{H}[2m]) + \log_2(1/\delta)]$$

then with probability $\geq 1-\delta$, every $h \in \mathcal{H}$ with $err_{\mathcal{D}}(h) \geq \epsilon$ has $err_S(h) > 0$ (equivalently, every $h \in \mathcal{H}$ with $err_S(h) = 0$ has $err_{\mathcal{D}}(h) < \epsilon$).

Theorem 6.20 (VC uniform convergence) *For any class $\mathcal{H}$ and distribution $\mathcal{D}$, if a training sample S is drawn from $\mathcal{D}$ of size*

$$m \geq \frac{8}{\epsilon^2}[\ln(2\mathcal{H}[2m]) + \ln(1/\delta)]$$

then with probability $\geq 1-\delta$, every $h \in \mathcal{H}$ will have $|err_S(h) - err_{\mathcal{D}}(h)| \leq \epsilon$.

Theorem 6.21 (Sauer's lemma) $\mathcal{H}[m] \leq \sum_{i=0}^{\mathrm{VCdim}(\mathcal{H})} \binom{m}{i} \leq m^{\mathrm{VCdim}(\mathcal{H})}$.

Notice that Sauer's lemma is not necessarily tight: e.g., in the case of linear separators in the plane we have $\mathcal{H}[m] = O(m^2)$ and yet $\mathrm{VCdim}(\mathcal{H}) = 3$. Putting Theorems 6.19 and 6.21 together, with a little algebra we get the following corollary (a similar corollary results by combining Theorems 6.20 and 6.21):

Corollary 6.22 *For any class $\mathcal{H}$ and distribution $\mathcal{D}$, a training sample S of size*

$$O\left(\frac{1}{\epsilon}[\mathrm{VCdim}(\mathcal{H})\log(1/\epsilon) + \log(1/\delta)]\right)$$

is sufficient to ensure that with probability $\geq 1-\delta$, every $h \in \mathcal{H}$ with $err_{\mathcal{D}}(h) \geq \epsilon$ has $err_S(h) > 0$ (equivalently, every $h \in \mathcal{H}$ with $err_S(h) = 0$ has $err_{\mathcal{D}}(h) < \epsilon$).

Before proving the above theorems, let's first use Sauer's lemma to show some interesting facts about VC-dimension. Specifically, given a class $\mathcal{H}$ and integer k, define $\mathrm{MAJ}_k(\mathcal{H})$ to be the set of functions achievable by taking a majority vote over k functions in $\mathcal{H}$.

Corollary 6.23 *If $\mathcal{H}$ has VC-dimension d then $MAJ_k(\mathcal{H})$ has VC-dimension $O(kd\log(kd))$.*

Proof: Let m be the VC-dimension of $\mathrm{MAJ}_k(\mathcal{H})$, so by definition, there must exist a set S of m points shattered by $\mathrm{MAJ}_k(\mathcal{H})$. We know by Sauer's lemma that there are at most m^d ways of partitioning the points in S using functions in $\mathcal{H}$. Since each function in $\mathrm{MAJ}_k(\mathcal{H})$ is determined by k functions in $\mathcal{H}$, this means there are at most $(m^d)^k = m^{kd}$ ways of partitioning the points using functions in $\mathrm{MAJ}_k(\mathcal{H})$. Since S is shattered, we therefore must have $2^m \leq m^{kd}$, or equivalently $m \leq kd\log_2(m)$. We can solve this as follows. First, assuming $m \geq 16$ we have $\log_2(m) \leq \sqrt{m}$ so $kd\log_2(m) \leq kd\sqrt{m}$ which implies that $m \leq (kd)^2$. To get the better bound, we can then just plug back in to our original inequality: since $m \leq (kd)^2$, it must be that $\log_2(m) \leq 2\log_2(kd)$, so our original inequality implies $m \leq 2kd\log_2(kd)$. ■

6.9.1 Proof of Theorem 6.19 (VC Occam bound)

Consider drawing a set S of m examples from $\mathcal{D}$ and let A denote the event that there exists $h \in \mathcal{H}$ with $err_{\mathcal{D}}(h) > \epsilon$ but $err_S(h) = 0$. Our goal is to prove that $\Pr[A] \leq \delta$.

Now, consider drawing *two* sets S, S' of m examples each from $\mathcal{D}$. Define A as above and let B denote the event that there exists $h \in \mathcal{H}$ with $err_S(h) = 0$ but $err_{S'}(h) \geq \epsilon/2$.

Lemma 6.24 $\Pr[A] \le 2\Pr[B]$.

Proof: Clearly, $\Pr[B] \ge \Pr[A \cap B] = \Pr[A]\Pr[B|A]$. Consider drawing set S and suppose event A occurs. Let h be some hypothesis in $\mathcal{H}$ with $err_{\mathcal{D}}(h) > \epsilon$ but $err_S(h) = 0$. Now, draw set S'. $\mathbf{E}[err_{S'}(h)] = err_{\mathcal{D}}(h) > \epsilon$. So, by Chernoff bounds, since $m > 8/\epsilon$, $\Pr[err_{S'}(h) \ge \epsilon/2] \ge 1/2$. So, we have $\Pr[B|A] \ge 1/2$ and thus $\Pr[B] \ge \Pr[A]/2$ as desired. ∎

So, it suffices to prove that $\Pr[B] \le \delta/2$. Now, let us consider a third experiment. Suppose we randomly draw a set S'' of $2m$ points from $\mathcal{D}$ and then randomly partition S'' into two sets S and S' of m points each. Let B^* denote the event that there exists $h \in \mathcal{H}$ with $err_S(h) = 0$ but $err_{S'}(h) \ge \epsilon/2$ under this experiment. $\Pr[B^*] = \Pr[B]$ since drawing $2m$ points i.i.d. from $\mathcal{D}$ and randomly partitioning them into two sets of size m produces the same distribution on (S, S') as does drawing S and S' directly. The advantage of this new experiment, however, is that we can now argue that $\Pr[B^*]$ is low by attempting to argue that for *any* set S'' of size $2m$, $\Pr[B^*|S'']$ is low, with probability now taken over just the random partition of S'' into S and S'. The key point now is that since S'' is fixed, there are only $|\mathcal{H}[S'']|$ events to worry about. Specifically, it suffices to prove that for any fixed $h \in \mathcal{H}[S'']$, the probability over the partition of S'' that h makes 0 mistakes on S but more than $\epsilon m/2$ mistakes on S' is at most $\delta/(2\mathcal{H}[2m])$. We can then just apply the union bound over all $h \in \mathcal{H}[S'']$.

To make the calculations easier, let's consider the following specific method for partitioning S'' into S and S'. First, randomly put the points in S'' into pairs: (a_1, b_1), (a_2, b_2), $\ldots$, (a_m, b_m). Then, for each index i, flip a fair coin: if heads put a_i into S and b_i into S', else if tails put a_i into S' and b_i into S. Now, fix some $h \in \mathcal{H}[S'']$ and let us consider the probability over these m fair coin flips that h makes 0 mistakes on S but more than $\epsilon m/2$ mistakes on S'. First of all, if for any index i, h makes a mistake on both a_i and b_i then the probability is zero (because it cannot possibly make 0 mistakes on S). Second, if there are fewer than $\epsilon m/2$ indices i such that h makes a mistake on either a_i or b_i then again the probability is zero (because it cannot possibly make more than $\epsilon m/2$ mistakes on S'). So, we may assume there are $r \ge \epsilon m/2$ indices i such that h makes a mistake on exactly one of a_i or b_i. In this case, the chance that *all* of those mistakes land in S' is exactly $1/2^r$. This quantity is at most $1/2^{\epsilon m/2} \le \delta/(2\mathcal{H}[2m])$ as desired for m as given in the theorem statement. ∎

6.9.2 Proof of Theorem 6.20 (VC uniform convergence)

This proof is identical to the proof of Theorem 6.19 except B^* is now the event that there exists $h \in \mathcal{H}[S'']$ such that $|err_S(h) - err_{S'}(h)| \ge \epsilon/2$. We again consider the experiment where we randomly put the points in S'' into pairs (a_i, b_i) and then flip a fair coin for each index i, if heads placing a_i into S and b_i into S', else placing a_i into S' and b_i into S. Let us consider the difference between the number of mistakes h makes on S and the number of mistakes h makes on S' and observe how this difference changes as we flip coins for

$i = 1, 2, \ldots, m$. Initially, the difference is 0. If h makes a mistake on both or neither of (a_i, b_i) then the difference does not change. Else, if h makes a mistake on exactly one of a_i or b_i, then with probability 1/2 the difference increases by 1 and with probability 1/2 the difference decreases by 1. Say there are $r \leq m$ such pairs. Then we are asking the question: if we take a random walk of $r \leq m$ steps, what is the probability that we end up more than $\epsilon m/2$ steps away from the origin? This is equivalent to asking: if we flip $r \leq m$ fair coins, what is the probability the number of heads differs from its expectation by more than $\epsilon m/4$. By Hoeffding bounds, this is at most $2e^{-\epsilon^2 m/8}$. This quantity is at most $\delta/(2\mathcal{H}[2m])$ as desired for m as given in the theorem statement. ■

6.9.3 Proof of Theorem 6.21 (Sauer's lemma)

Let $d = \text{VCdim}(\mathcal{H})$. Our goal is to prove for any set S of m points that $|\mathcal{H}[S]| \leq \binom{m}{\leq d}$, where we are defining $\binom{m}{\leq d} = \sum_{i=0}^{d} \binom{m}{i}$; this is the number of distinct ways of choosing d or fewer elements out of m. We will do so by induction on m. As a base case, our theorem is trivially true if $m \leq d$.

As a first step in the proof, notice that:

$$\binom{m}{\leq d} = \binom{m-1}{\leq d} + \binom{m-1}{\leq d-1} \tag{6.2}$$

because we can partition the ways of choosing d or fewer items into those that do not include the first item (leaving $\leq d$ to be chosen from the remainder) and those that do include the first item (leaving $\leq d-1$ to be chosen from the remainder).

Now, consider any set S of m points and pick some arbitrary point $x \in S$. By induction, we may assume that $|\mathcal{H}[S \setminus \{x\}]| \leq \binom{m-1}{\leq d}$. So, by equation (6.2) all we need to show is that $|\mathcal{H}[S]| - |\mathcal{H}[S \setminus \{x\}]| \leq \binom{m-1}{\leq d-1}$. Thus, our problem has reduced to analyzing how many *more* labelings there are of S than there are of $S \setminus \{x\}$ using functions in $\mathcal{H}$.

If $\mathcal{H}[S]$ is larger than $\mathcal{H}[S \setminus \{x\}]$, it is because of pairs of labelings in $\mathcal{H}[S]$ that differ only on point x and therefore collapse to the same labeling when x is removed. For labeling $h \in \mathcal{H}[S]$ (this is a Boolean function defined only on S), define $\mathsf{twin}(h)$ to be the same as h except giving the opposite value to x; this may or may not belong to $\mathcal{H}[S]$. Let $\mathcal{T} = \{h \in \mathcal{H}[S] : h(x) = 1 \text{ and } \mathsf{twin}(h) \in \mathcal{H}[S]\}$. Notice $|\mathcal{H}[S]| - |\mathcal{H}[S \setminus \{x\}]| = |\mathcal{T}|$.

Now, what is the VC-dimension of $\mathcal{T}$? If $d' = \text{VCdim}(\mathcal{T})$, this means there is some set R of d' points in $S \setminus \{x\}$ that are shattered by $\mathcal{T}$. By definition of $\mathcal{T}$, all $2^{d'}$ labelings of R can be extended both ways to x using labelings in $\mathcal{H}[S]$; i.e., $R \cup \{x\}$ is shattered by $\mathcal{H}$. This means, $d' + 1 \leq d$. Since $\text{VCdim}(\mathcal{T}) \leq d - 1$, by induction we have $|\mathcal{T}| \leq \binom{m-1}{\leq d-1}$ as desired.

6.9.4 The VC-dimension of linear separators

We saw earlier that the class of linear separators in R^2 has a VC-dimension of 3. Here, we prove that the class of linear separators in R^d has a VC-dimension of $d+1$.

The easy direction is that there exists a set of size $d+1$ that can be shattered. Select the d unit-coordinate vectors plus the origin to be the $d+1$ points. Suppose A is any subset of these $d+1$ points not including the origin. Take a 0-1 vector $\mathbf{w}$ which has 1's precisely in the coordinates corresponding to vectors in A. Then the linear separator $\mathbf{w}\cdot\mathbf{x} \geq 1/2$ labels A as positive and its complement as negative, and similarly $\mathbf{w}\cdot\mathbf{x} \leq 1/2$ labels A negative and its complement as positive.

We now show that no set of $d+2$ points in d-dimensions can be shattered by this class. We will do this by proving that any set of $d+2$ points can be partitioned into two disjoint subsets A and B of points whose convex hulls intersect. This establishes the claim since any linear separator with A one one side must have its entire convex hull on that side,[9] so it is not possible to have a linear separator with A on one side and B on the other.

Let *convex*(S) denote the convex hull of point set S.

Theorem 6.25 (**Radon**)*: Any set $S \subseteq R^d$ with $|S| \geq d+2$, can be partitioned into two disjoint subsets S_1 and S_2 such that convex$(S_1) \cap$ convex$(S_2) \neq \phi$.*

Proof: Without loss of generality, assume $|S| = d+2$. Form a $d \times (d+2)$ matrix with one column for each point of S. Call the matrix A. Add an extra row of all 1's to construct a $(d+1) \times (d+2)$ matrix B. Clearly, since the rank of this matrix is at most $d+1$, the columns are linearly dependent. Say $\mathbf{x} = (x_1, x_2, \ldots, x_{d+2})$ is a nonzero vector with $B\mathbf{x} = 0$. Reorder the columns so that $x_1, x_2, \ldots, x_s \geq 0$ and $x_{s+1}, x_{s+2}, \ldots, x_{d+2} < 0$. Normalize $\mathbf{x}$ so $\sum\limits_{i=1}^{s} |x_i| = 1$. Let $\mathbf{b_i}$ (respectively $\mathbf{a_i}$) be the i^{th} column of B (respectively A). Then, $\sum\limits_{i=1}^{s} |x_i|\mathbf{b_i} = \sum\limits_{i=s+1}^{d+2} |x_i|\mathbf{b_i}$ from which it follows that $\sum\limits_{i=1}^{s} |x_i|\mathbf{a_i} = \sum\limits_{i=s+1}^{d+2} |x_i|\mathbf{a_i}$ and $\sum\limits_{i=1}^{s} |x_i| = \sum\limits_{i=s+1}^{d+2} |x_i|$. Since $\sum\limits_{i=1}^{s} |x_i| = 1$ and $\sum\limits_{i=s+1}^{d+2} |x_i| = 1$ each side of $\sum\limits_{i=1}^{s} |x_i|\mathbf{a_i} = \sum\limits_{i=s+1}^{d+2} |x_i|\mathbf{a_i}$ is a convex combination of columns of A which proves the theorem. Thus, S can be partitioned into two sets, the first consisting of the first s points after the rearrangement and the second consisting of points $s+1$ through $d+2$. Their convex hulls intersect as required. ■

As noted above, Radon's theorem immediately implies that linear separators in R^d cannot shatter any set of $d+2$ points.

[9] If any two points $\mathbf{x}_1$, $\mathbf{x}_2$ lie on the same side of a separator, so must any convex combination: if $\mathbf{w}\cdot\mathbf{x}_1 \geq b$ and $\mathbf{w}\cdot\mathbf{x}_2 \geq b$ then $\mathbf{w}\cdot(a\mathbf{x}_1 + (1-a)\mathbf{x}_2) \geq b$.