# CS449 Lecture Notes:
# Basic Machine Learning Theory

Christino Tamon
Clarkson University

October 30, 2023

## 1 Bayes

Consider the following guessing game on distributions:

> Let $D_0$ and $D_1$ be two distributions over $\mathbb{R}$ with distinct means. For a fixed $p \in (0,1)$, let $Y \sim \text{Bernoulli}(p, 1-p)$ and $X \sim D_Y$. The problem is to predict $Y \in \{0,1\}$ given $X \in \mathbb{R}$.

A predictor $f : \mathbb{R} \to \{0,1\}$ is called **optimal** if

$$\mathbb{P}_{(X,Y)}[f(X) \neq Y] \leq \mathbb{P}_{(X,Y)}[g(X) \neq Y]$$

for any predictor $g : \mathbb{R} \to \{0,1\}$.

Assume nothing is unknown about $D_0$, $D_1$ and $p$. If no assumption is made on how these parameters are chosen or generated, we run into a No-Free-Lunch theorem.

**Fact 1.1.** *For any predictor $f : [0,1] \to \{0,1\}$, there are distributions $D_0$ and $D_1$ with $p = \frac{1}{2}$, so that*

$$\mathbb{P}_{(X,Y)}[f(X) \neq Y] = \tfrac{1}{2}.$$

*Proof.* Let $f : [0,1] \to \{0,1\}$ be a given predictor. Assume both $A_0 = f^{-1}(0)$ and $A_1 = f^{-1}(1)$ are measurable subsets over $[0,1]$ (say, with respect to the Lebesgue measure $\mu$). We define $D_0$ and $D_1$ using their distribution functions $F_0$ and $F_1$. Moreover, assume

$$F_0(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } 0 \leq x \leq 1 \\ 1 & \text{if } x > 1 \end{cases}$$

which corresponds to the uniform distribution over $[0,1]$. Now, we partition $A_0 = A_{00} + A_{01}$ into two disjoint subsets so that $\mu(A_{00}) = \mu(A_{01})$. This follows, for example, by the Ham Sandwich theorem (see a book by Matousek). We create a similar partition for $A_1 = A_{10} + A_{11}$. We define $F_1(x) = 0$ if $x \in A_{00} \cup A_{10}$, and $F_1(x) = 1$, if $x \in A_{10} \cup A_{11}$. It can be verified that $\mathbb{P}[f(X) \neq Y] = 1/2$. $\square$

Next, we assume the distributions $D_0, D_1$ and $p$ are completely known. In this case, the Bayes predictor is optimal.

**Theorem 1.** *The predictor $\beta(x) = \mathbf{1}pD_1(x) \geq (1-p)D_0(x)$ is optimal.*

*Proof.* Let $\tau(x) = \frac{pD_1(x)}{pD_1(x)+(1-p)D_0(x)}$. Note that $\beta(x) = \mathbf{1}\tau(x) \geq 1/2$. Let $g : \mathbb{R} \to \{0,1\}$ be an arbitrary predictor. Fix a real number $x$. Then

$$\begin{aligned}
\mathbb{P}[g(X) = Y | X = x] &= \mathbb{P}[g(X) = 1, Y = 1 | X = x] + \mathbb{P}[g(X) = 0, Y = 0 | X = x] \\
&= \mathbf{1}g(x) = 1\mathbb{P}[Y = 1|X = x] + \mathbf{1}g(x) = 0\mathbb{P}[Y = 0|X = x] \\
&= \mathbf{1}g(x) = 1\tau(x) + \mathbf{1}g(x) = 0(1-\tau(x)).
\end{aligned}$$

Therefore,

$$\begin{aligned}
&\mathbb{P}[g(X) \neq Y|X = x] - \mathbb{P}[\beta(X) \neq Y|X = x] \\
&= \tau(x)(\mathbf{1}\beta(x) = 1 - \mathbf{1}g(x) = 1) + (1-\tau(x))(\mathbf{1}\beta(x) = 0 - \mathbf{1}g(x) = 0) \\
&= (2\tau(x) - 1)(\mathbf{1}\beta(x) = 1 - \mathbf{1}g(x) = 1) \\
&\geq 0.
\end{aligned}$$

Thus, $\mathbb{P}[g(X) \neq Y] \geq \mathbb{P}[\beta(X) \neq Y]$. □

## 2 Perceptron

A vector $w \in \mathbb{R}^n$ defines a linear classifier $h_w(x) = \text{sign}\langle w, x\rangle$ where

$$\text{sign}(z) = \begin{cases} +1 & \text{if } z \geq 0 \\ -1 & \text{if } z < 0 \end{cases}$$

**Perceptron algorithm:**

1. Input: a labeled sample $S_m = \{(x_i, y_i) \in \mathbb{R}^n \times \{-1, 1\} : \|x_i\| = 1, i \in [m]\}$.
2. Initialize normal vector $w \leftarrow 0$.
3. While there is $i$ so that $y_i\langle w, x_i\rangle < 0$ do:
   (a) Update $w \leftarrow w + y_i x_i$.
4. Output $w$.

The following mistake bound result for Perceptron is due to Block and Novikoff.

**Theorem 2.** *Assume $S_m = \{(x_i, y_i) \in \mathbb{R}^n \times \{-1, 1\} | \|x_i\| = 1, i = 1, \ldots, n\}$ is linearly separable by $w^\star$ where $\|w^\star\| = 1$. Let $\gamma = \min_{i=1,\ldots,n} |\langle w^\star, x_i\rangle|$. Then, the number of mistakes made by Perceptron algorithm is at most $1/\gamma^2$.*

*Proof.* Let $w_t$ be the weight vector maintained by the Perceptron algorithm after the $t$-th mistake is made. We analyze two quantities: $\langle w^\star, w_t\rangle$ and $\|w_t\|$. Suppose the $t$-th mistake was made on $(x_i, y_i)$. Therefore,

$$\begin{aligned}
\langle w^\star, w_{t+1}\rangle &= \langle w^\star, w_t + y_i x_i\rangle && (1) \\
&= \langle w^\star, w_t\rangle + y_i\langle w^\star, x_i\rangle && (2) \\
&\geq \langle w^\star, w_t\rangle + \gamma. && (3)
\end{aligned}$$

On the other hand,

$$\begin{aligned}\|w_{t+1}\|^2 &= \langle w_{t+1}, w_{t+1}\rangle & (4)\\ &= \langle w_t + y_i x_i, w_t + y_i x_i\rangle & (5)\\ &= \langle w_t, w_t\rangle + 2y_i\langle w_t, x_i\rangle + y_i^2\langle x_i, x_i\rangle & (6)\\ &\leq \|w_t\|^2 + 1. & (7)\end{aligned}$$

Thus, $\langle w^\star, w_T\rangle \geq \gamma T$ and $\|w_T\|^2 \leq T$. By Cauchy-Schwarz inequality, $|\langle u, v\rangle| \leq \|u\|\,\|v\|$, we have

$$\gamma T \leq \langle w^\star, w_T\rangle \leq \|w_T\| \leq \sqrt{T}. \tag{8}$$

□

## 2.1 Kernel trick

The following observation is due to Vapnik. The Perceptron algorithm returns a weight vector that is of the form

$$w = \sum_j m_j y_j x_j$$

for some integer vector $m$ (whose dimension equals to the number of training points). In fact, $m_i$ counts the number of mistakes made (by the Perceptron algorithm) on the $i$th point. To see this, recall that the weight update used by the Perceptron algorithm is

$$w \leftarrow w + y_i x_i$$

whenever a mistake is made on $(x_i, y_i)$. Thus, $w$ is a linear combination of the vectors $y_i x_i$ where the weight of $y_i x_i$ corresponds to the number of mistakes made on the $i$th point.

The separating hyperplane condition is:

$$\sum_j m_j y_i y_j \langle x_i, x_j\rangle \geq 0, \qquad \text{for all } i.$$

Here, we may generalize the inner product $\langle x_i, x_j\rangle$ to $k(x_i, x_j)$ for a kernel map.

*Example*: Consider a kernel of the form $k(x,z) = (\langle x, z\rangle + c)^2$ where $x, z \in \mathbb{R}^2$. Note that

$$\begin{aligned}(\sum_i x_i z_i + c)^2 &= (\sum_i x_i z_i + c)(\sum_j x_j z_j + c)\\ &= \sum_{i,j}(x_i x_j)(z_i z_j) + \sum_i(\sqrt{2c}x_i)(\sqrt{2c}z_i) + c\cdot c\\ &= \sum_i (x_i^2)(z_i^2) + \sum_{i,j}(\sqrt{2}x_i x_j)(\sqrt{2}z_i z_j) + \sum_i(\sqrt{2c}x_i)(\sqrt{2c}z_i) + c\cdot c\end{aligned}$$

Then, $k(x,z)$ computes an inner product between the following 2 vectors in $\mathbb{R}^6$:

$$\begin{pmatrix} x_1^2 \\ x_2^2 \\ \sqrt{2}x_1x_2 \\ \sqrt{2c}x_1 \\ \sqrt{2c}x_2 \\ c \end{pmatrix}, \qquad \begin{pmatrix} z_1^2 \\ z_2^2 \\ \sqrt{2}z_1z_2 \\ \sqrt{2c}z_1 \\ \sqrt{2c}z_2 \\ c \end{pmatrix}.$$

A crucial point here is that the computation of $k(x, z)$ involves only vectors in $\mathbb{R}^2$ (that is, we never *explicitly* compute with the longer vectors). If we define

$$\phi(x) = \begin{pmatrix} x_1^2 & x_2^2 & \sqrt{2}x_1x_2 & \sqrt{2c}x_1 & \sqrt{2c}x_2 & c \end{pmatrix}$$

then

$$(\langle x, z\rangle + c)^2 = k(x, z) = \langle \phi(x), \phi(z)\rangle.$$

## 3 Closure

Let $R_1 = \{f_\alpha | \alpha \in \mathbb{R}\}$ be a set of half-infinite intervals over the reals where $f_\alpha(x) = \text{sign}(x - \alpha)$. Note $f_\alpha(x) = 1$ if $x \geq \alpha$, and $-1$ otherwise.

**Closure algorithm for $R_1$:**

1. Input: a labeled sample $S_m = \{(x_i, y_i) \in \mathbb{R} \times \{-1, 1\} : \|x_i\| = 1, i \in [m]\}$.
2. Let $\alpha = \min_i\{x_i | y_i = 1\}$.
3. Output $\alpha$.

We say a learning algorithm is **consistent** if it output a hypothesis which has zero error on the sample. Assume there is $\alpha^\star$ for which $y_i = f_{\alpha^\star}(x_i)$ for $i = 1, \ldots, m$.

**Claim 3.1.** *The Closure algorithm for $R_1$ is consistent.*

*Proof.* Note $f_\alpha(x_i) = y_i$ for all $i = 1, \ldots, m$. $\square$

**PAC learning model.** Assume the examples $x_i \in \mathbb{R}$ are independent and identically distributed according to an unknown probability distribution $D$. Suppose there is a target $\alpha^\star$ so that $y_i = f_{\alpha^\star}(x_i)$. Let $\epsilon, \delta \in (0, 1)$. We say a learning algorithm $A$ is $(\epsilon, \delta)$-**PAC** (Probably Approximately Correct) if given a sample $S_m$ labeled according to $f_{\alpha^\star}$ and drawn according to $D$, the output $f_\alpha$ of $A$ satisfies

$$\mathbb{P}[D(f_\alpha \triangle f_{\alpha^\star}) \leq \epsilon] \geq 1 - \delta$$

where the probability is taken with respect to the sample that is drawn iid according to $D$ and to the internal randomization of $A$ (in producing the output $h$). Here, $f \triangle g = \{x : f(x) \neq g(x)\}$ denotes the symmetric difference of the functions $f$ and $g$.

**Theorem 3.** *For any $\epsilon, \delta \in (0, 1)$, the Closure algorithm for $R_1$ is an $(\epsilon, \delta)$-PAC algorithm provided $m = \Omega(1/\epsilon \ln(1/\delta))$.*

*Proof.* Let $\alpha$ be the smallest positive example returned by the closure algorithm. Let $\beta$ be defined as

$$\beta = \sup\{x : D(x, \alpha) > \epsilon\}.$$

The probability that $f_\alpha$ has error at least $\epsilon$ equals to the probability that all $m$ sample points are drawn from outside $(\beta, \alpha)$. The latter probability is at most $(1 - \epsilon)^m \leq \delta$, by the choice of $m$. $\square$

## 4 Occam's Razor

**Theorem 4.** *For $\epsilon, \delta \in (0,1)$, a consistent learning algorithm for a finite concept class $C$ is a $(\epsilon, \delta)$-PAC algorithm if the sample size is at least $\Omega(\frac{1}{\epsilon} \ln \frac{|C|}{\delta})$.*

*Proof.* Let $S_m$ be random sample drawn according to $D$. Assume that $f$ is the true target concept that is used to label $S_m$. Let $H_\epsilon = \{h : D(h \triangle f) > \epsilon\} \subset C$ be a set of functions that are not $\epsilon$-close to $f$. Let $h$ be the output hypothesis the learning algorithm $A$ that is consistent with $S_m$. Note $h$ is a random variable which depends on $S_m$ and the internal randomization of $A$. The failure probability of $A$ is bounded from above by

$$\mathbb{P}[(\exists h \in H_\epsilon)(h \text{ consistent with } S_m)] \leq |C|(1-\epsilon)^m.$$

So, this probability is at most $\delta$ if $m \geq \frac{1}{\epsilon} \ln \frac{|C|}{\delta}$. $\square$

A learning algorithm is called an **Occam** algorithm if it is consistent and, given a sample of size $m$, for a concept over an $n$-dimensional space, it outputs a hypothesis of size $O(n^a m^b)$ where $a > 0$ and $0 \leq b < 1$. By hypothesis size of $h$, we mean the binary encoding length of $h$.

**Theorem 5.** *An Occam algorithm is a PAC algorithm.*

*Proof.* For some constant $c$, let $N \leq cn^a m^b$ be the size of the hypothesis output by the Occam algorithm. By the Occam property, the cardinality of the hypothesis class is at most $2^N$. From the proof of Theorem 4, if the sample size $m$ satisfies $m \geq \frac{1}{\epsilon} \log \frac{1}{\delta}$ and

$$m \geq \frac{1}{\epsilon} \ln N = \frac{cn^a m^b}{\epsilon} \quad \text{or} \quad m \geq \left(\frac{cn^a}{\epsilon}\right)^{1/(1-b)}.$$

then the algorithm is a $(\epsilon, \delta)$-PAC algorithm. So, we take the maximum of both bounds. $\square$

## 5 VC dimension

To extend Theorem 4, we consider the VC dimension of a concept class. Let $C \subseteq \{f | f : X \to \{0,1\}\}$ be a collection of Boolean concepts over a space $X$.

A subset $S \subseteq X$ is **shattered** by $C$ if for any subset $A \subseteq S$, there is a concept $f \in C$ so that $f(x) = 1$ for all $x \in A$ and $f(x) = 0$ for all $x \in S \setminus A$. The **Vapnik-Chervonenkis dimension** of $C$ is the size of the largest shattered subset.

We define a bit more notation. Let $\Pi_C(S) = \{f \cap S | f \in C\}$ be the concept class $C$ projected on a sample $S$. Let $\Delta(f) = \{f \triangle h | h \in C\}$ be the set of error regions between $f$ and each concept in $C$. Let $\Delta_\epsilon(f) = \{r \in \Delta(f) | D(r) > \epsilon\}$ be the set of error regions whose probability weight under $D$ is at least $\epsilon$. We say $S$ is a $\epsilon$-net for $f$ if $S \cap r \neq \emptyset$ for all $r \in \Delta_\epsilon(f)$. Note if $S$ is a $\epsilon$-net for the target concept $f$, then any $h$ that is consistent with $f$ on $S$ has error smaller than $\epsilon$.

**Theorem 6.** *Let $C$ be a concept class with VC dimension $d < \infty$. Then, the consistent learning algorithm is a $(\epsilon, \delta)$-PAC algorithm if*

$$m = O\left(\frac{1}{\epsilon} \ln \frac{1}{\delta} + \frac{d}{\epsilon} \ln \frac{1}{\epsilon}\right).$$

*Proof.* We draw two samples $S_1$ and $S_2$ each of size $m$. Let $A$ be the event that $S_1$ fails to be a $\epsilon$-net for $\Delta(f)$. Let $B$ be the event that $A$ holds and $|r \cap S_2| \geq \epsilon m/2$ for some $r \in \Delta_\epsilon(f)$ where $r \cap S_1 = \emptyset$.

We claim that $\mathbb{P}[B|A] \geq 1/2$. This can be proved using concentration bounds (Hoeffding bounds). Using this, we have

$$\mathbb{P}[B] = \mathbb{P}[B|A]\mathbb{P}[A] + \mathbb{P}[B|\overline{A}]\mathbb{P}[\overline{A}] \geq \frac{1}{2}\mathbb{P}[A]$$

since $\mathbb{P}[B|\overline{A}] = 0$. Thus, $\mathbb{P}[A] \leq 2\mathbb{P}[B]$.

Next, we analyze $\mathbb{P}[B]$. Let $S = S_1 \cup S_2$ be a sample of size $2m$ from which we derive $S_1$ and $S_2$. The event $B$ states that there is $r \in \Pi_{\Delta_\epsilon(f)}(S)$ so that $|r| \geq \epsilon m/2$ and $r \cap S_1 = \emptyset$. For the analysis, consider the following experiment. Suppose there are $2m$ balls for which $\ell$ balls are red and $m - \ell$ balls are blue, where $\ell \geq \epsilon m/2$. On a random partition of $S$ into $S_1$ and $S_2$, the probability that all red balls land in $S_2$ is given by

$$\frac{\binom{m}{\ell}}{\binom{2m}{\ell}} \leq \frac{1}{2^\ell} \leq \frac{1}{2^{\epsilon m/2}}$$

But, this argument is for a fixed region $r \in \Pi_{\Delta_\epsilon(f)}(S)$. By a combinatorial result (Sauer's lemma), the total number of regions in $\Pi_{\Delta_\epsilon(f)}(S)$ is finite if $d$ is finite:

$$|\Pi_{\Delta_\epsilon(f)}(S)| \leq |\Pi_{\Delta(f)}(S)| \leq |\Pi_C(S)| \leq \left(\frac{2em}{d}\right)^d.$$

Putting all these together, we have

$$\mathbb{P}[A] \leq 2\mathbb{P}[B] \leq 2\left(\frac{2em}{d}\right)^d \frac{1}{2^{\epsilon m/2}} \leq \delta$$

since $m = O(\ln(1/\delta)/\epsilon + d\ln(1/\epsilon)/\epsilon)$. □

We prove the important combinatorial lemma used in the proof above. Let $\Phi_d(m)$ be defined as

$$\Phi_d(m) = \Phi_d(m-1) + \Phi_{d-1}(m-1), \quad \text{for } m, d \geq 1$$

and $\Phi_0(m) = \Phi_d(0) = 1$.

**Lemma 6.1.** *(Sauer) If $C$ has VC dimension $d$ then $\Pi_C(m) \leq \Phi_d(m)$.*

*Proof.* If $d = 0$ or $m = 0$, the claim holds. Suppose $C$ is a concept class with VC dimension $d$ and let $S$ be a sample of size $m1$, where $m, d \geq 1$. Fix an element $x_0 \in S$. We partition the subsets in $\Pi_C(S)$ into two disjoint collections. To avoid double-counting, define

$$C' = \{f \in \Pi_C(S) : x_0 \notin f, f \cup \{x_0\} \in \Pi_C(S)\}.$$

Each $f \in C'$ is a *shadow* of another subset in $C$ which includes $x_0$. Then,

$$|\Pi_C(S)| = |\Pi_C(S \setminus \{x_0\})| + |C'|.$$

Note $C' = \Pi_{C'}(S \setminus \{x_0\})$. Also, the VC dimension of $C'$ is at most $d-1$; otherwise, if $S' \subseteq S \setminus \{x_0\}$ is shattered by $C'$, then $S' \cup \{x_0\}$ is shattered by $C$. Therefore,

$$|\Pi_C(S)| \leq \Phi_d(m-1) + \Phi_{d-1}(m-1) = \Phi_d(m).$$

□

**Claim 5.1.** $\Phi_d(m) = \sum_{i=0}^{d} \binom{m}{i}$.

*Proof.* By induction on $d$ and $m$, we have

$$\begin{aligned}
\Phi_d(m) &= \Phi_d(m-1) + \Phi_{d-1}(m-1) && (9)\\
&= \sum_{i=0}^{d}\binom{m-1}{i} + \sum_{i=0}^{d-1}\binom{m-1}{i} && (10)\\
&= 1 + \sum_{i=1}^{d}\binom{m-1}{i} + \sum_{i=0}^{d-1}\binom{m-1}{i} && (11)\\
&= 1 + \sum_{i=0}^{d-1}\binom{m-1}{i+1} + \sum_{i=0}^{d-1}\binom{m-1}{i} && (12)\\
&= 1 + \sum_{i=0}^{d-1}\binom{m}{i+1} && (13)\\
&= \sum_{i=0}^{d}\binom{m}{i}. && (14)
\end{aligned}$$

□

**Claim 5.2.** *If $m > d$, $\sum_{i=0}^{d} \binom{m}{i} \le \left(\frac{em}{d}\right)^d$.*

*Proof.* Since $0 \le d/m < 1$, we have

$$\left(\frac{d}{m}\right)^d \sum_{i=0}^{d}\binom{m}{i} \le \sum_{i=0}^{d}\left(\frac{d}{m}\right)^i\binom{m}{i} \le \sum_{i=0}^{m}\left(\frac{d}{m}\right)^i\binom{m}{i} = \left(1+\frac{d}{m}\right)^m \le e^d.$$

□

## 6 Neural networks

Let $\sigma(z)$ be a smooth (continuous and differentiable) activation function. For example, $\sigma_1(y) = 1/(1+e^{-y})$ (sigmoid) and $\sigma_2(y) = \tanh(z)$ (hyperbolic tangent) are natural choices.

We consider a sigmoided perceptron $y := f_w(x) = \sigma(\langle w, x\rangle)$. Given a point $(x, y^\star)$ where $y^\star = f_{w^\star}(x)$ for some target $w^\star$, the loss (square error) of $w$ is given by

$$\ell(w, w^\star) = \frac{1}{2}(y - y^\star)^2.$$

Given a set of labeled points $\{(x_i, y_i^\star) : i \in [m]\}$, the main goal is to minimize the empirical loss given by

$$\hat{\ell}(w, w^\star) = \frac{1}{2m}\sum_{i=1}^{m}(y_i - y_i^\star)^2.$$

We use this to estimate the expected true loss given by $\ell(w, w^\star) = \mathbb{E}_D[\frac{1}{2}(f_w(x) - f_{w^\star}(x))^2]$ assuming the points $x_i$ are drawn independently and identically according to a probability distribution $D$ over the example space.

A standard method to minimize the loss is to find $w$ for which the gradient of $\ell$ vanishes, that is, $\nabla \ell(w) = 0$. For this, we use the *gradient descent* update

$$w \leftarrow w - \eta \nabla \ell(w)$$

for some learning parameter $\eta > 0$. Thus, for the sigmoided perceptron $y = \sigma(z)$ with $z = \langle w, x \rangle$, note that $\nabla \ell(w) = (y - y^\star)\sigma'(z)$.

Now, consider a network (or directed acyclic graph) $N$ which consists of sigmoided perceptron units. For unit $i$, we denote its output as $y_i = \sigma(z_i)$ where $z_i = \langle w_i, x \rangle$. Here, the input $x$ consists of the outputs of all units in preceding layer. We label the weight of the incoming wire from unit $j$ into unit $i$ as $w_{ij}$. We need to compute $\partial \ell / \partial w_{i,j}$ for all weights $w_{ij}$ appearing in $N$. This allows us to compute the overall gradient $\nabla \ell(W)$ where $W = (w_{ij})$ which is used in the gradient descent update $W \leftarrow W - \eta \nabla \ell(W)$ as described above.

To compute $\partial \ell / \partial w_{ij}$ for all weights, we use the **backpropagation** algorithm. The idea is to first compute

$$\delta_i := \frac{\partial \ell}{\partial z_i}$$

for all $i$. Using this, we have

$$\frac{\partial \ell}{\partial w_{ij}} = \frac{\partial \ell}{z_i} \frac{\partial z_i}{\partial w_{ij}} = \delta_i y_j.$$

Now, to compute $\delta_i = \partial \ell / \partial z_i$, note that

$$\delta_i = \sum_k \frac{\partial \ell}{\partial z_k} \frac{\partial z_k}{\partial z_i} = \sum_k \delta_k \frac{\partial z_k}{\partial y_i} \frac{\partial y_i}{\partial z_i} = \sum_k \delta_k w_{ki} \sigma'(z_i).$$

Here, the summation is over all units $k$ which uses the output $y_i$ (of unit $i$) as an input. Therefore, we backpropagate the $\delta$ values using the *recursion*

$$\delta_i = \sum_k \delta_k w_{ki} \sigma'(z_i).$$

## 7 Weighted Majority

**Online learning model**. The **online** model is an interactive game between the learner $A$ and the oracle which proceeds in rounds (or epochs). At round $t$, the oracle sends a challenge prediction $x_t$ to the learner. The learner replies by sending the oracle a predicted value $\hat{y}_t$. The oracle reports with the value $y_t$ and the learner obtains a loss of $\ell_t = \ell(\hat{y}_t, y_t)$, for some fixed loss function $\ell$. The goal of the learner is to minimize the total loss after $T$ rounds:

$$\mathcal{L}_T(A) = \sum_{t=1}^{T} \ell_t.$$

The learner is assumed to have access to a set of $n$ experts $E_1, \ldots, E_n$. Each expert $E_i$ makes its own prediction $\hat{y}_{t,i} \in \{-1, 1\}$ on $x_t$ for each $t \in [T]$. We define the total loss of expert $E_i$ as

$$\mathcal{L}_T(E_i) = \sum_{t=1}^{T} \ell(\hat{y}_{t,i}, y_t).$$

We say $A$ is a $(\alpha, \beta)$-competitive online learner if

$$\mathcal{L}_T(A) \le \alpha \min_i \mathcal{L}_T(E_i) + \beta.$$

**Weighted Majority (WM):**

1. Input: experts $E_1, \ldots, E_n$ with $\{-1, 1\}$ predictions.
2. Initialize $w_i = 1$ for $i = 1, \ldots, n$.
3. Given input $x$, predict with $\hat{y}$ where

$$\hat{y} = \begin{cases} +1 & \text{if } \sum_{i:E_i(x)=1} w_i \ge \sum_{i:E_i(x)=-1} w_i \\ -1 & \text{otherwise} \end{cases}$$

4. If $\hat{y} \ne y$:
   (a) Update $w_i \leftarrow \frac{1}{2} w_i$ for each $i$ where $E_i(x) \ne y$.

**Theorem 7.** *Let $m$ be the number of mistakes made by the best expert and let $M$ be the number of mistakes made by Weighted Majority. Then*

$$M \le \frac{1}{\log_2(4/3)}(m + \log_2 n).$$

*Proof.* Let $W_t$ denote the total sum of the weights of the experts after the $t$-th mistake made by WM, where $W_0 = n$. Note that $W_{t+1} \le \frac{3}{4} W_t$. To see this, suppose $\alpha \ge 1/2$ is the fraction of the total weight $W_t$ of the experts which made the mistake. Then,

$$W_{t+1} = \left(\frac{1}{2}\alpha + (1-\alpha)\right) W_t = \left(1 - \frac{1}{2}\alpha\right) W_t \le \frac{3}{4} W_t.$$

Therefore,

$$\frac{1}{2^m} \le W_{M+1} \le n \left(\frac{3}{4}\right)^M$$

□

## 7.1 Drifting Weighted Majority

Here, we allow the online game to be partitioned into disjoint intervals where the best expert in each interval may vary. The algorithm is similar to WM but with the following revised update rule:

(a) Update $w_i \leftarrow \frac{1}{2} w_i$ only if $w_i \ge \frac{1}{10n} W$ where $W = \sum_i w_i$.

**Theorem 8.** *Let $m$ be the number of mistakes made by the best sequence of drifting experts and let $M$ be the number of mistakes made by Drifting Weighted Majority. Then*

$$M \le \frac{1}{\log_2(5/4)}(m + \zeta \log_2(20n))$$

*where $\zeta$ denotes the number of times the best expert changes (or total number of intervals which contains a single best expert).*

*Proof.* Fix some interval and let $W_0$ be the total weights of the experts at the beginning of this interval. If a mistake is made, then at least $W/2$ is the fraction of weights will be halved but at most $W/10$ can not be halved. So, at least $W/2 - W/10 = 2W/5$ can be halved. Thus, the weight update yields $W' \leq 4W/5$. Also, note that the weight of the best expert in this interval is at most $W_0/20n$. Then, we have

$$\frac{1}{20n} W_0 \frac{1}{2^m} \leq W \leq W_0 \left(\frac{4}{5}\right)^M.$$

Since this argument holds in all intervals, we obtain the claim. □

### 7.2 Randomized Weighted Majority

We modify Weighted Majority (WM) by predicting with a single expert chosen with probability $w_j/W$ and penalizing each mistaken expert by $w_j \leftarrow \beta w_j$ for some $0 < \beta < 1$.

**Theorem 9.** *Let $M$ be the expected number of mistakes made by RWM and let $m$ be the number of mistakes made by the best expert. Then,*

$$M \leq \frac{1}{1-\beta}\left(\ln\left(\frac{1}{\beta}\right) m + \ln n\right).$$

*Proof.* Let $F_j$ be the fraction of total weight on wrong answer for $j$th mistake. Let $M = \sum_j F_j$ is the expected number of mistakes. The weight change on the $j$th mistake is

$$W \leftarrow W(1 - (1-\beta)F_j).$$

So, after $t$ mistakes, we have

$$\beta^m \leq W = n \prod_{j=1}^{t} (1 - (1-\beta)F_j) \leq n \exp\left(-(1-\beta)\sum_{j=1}^{t} F_j\right).$$

This yields the claim. □

## 8 Boosting

The idea of boosting is to convert a weak learning algorithm into a strong learning algorithm. A weak learning algorithm outputs a hypothesis $h$ whose accuracy is only slightly better than random guessing. That is, $\mathbb{P}[h(x) = f(x)] = 1/2 + \gamma$, for some small $\gamma > 0$. Boosting allows us to improve the accuracy to $1 - \epsilon$, for $\epsilon > 0$.

A crucial assumption placed on boosting is the following. The weak learning algorithm will output a weak hypothesis with respect to *any* distribution on the training sample. So, we require

$$\mathbb{P}_D[h(x) = f(x)] = \frac{1}{2} + \gamma$$

to hold under an arbitrary distribution $D$ defined over the sample. This is a strong assumption.

We describe a boosting algorithm due to Schapire and Singer where the boosting algorithm can tolerate a varying level of accuracy $\gamma$ during training (without the need of knowing it ahead of time). The older boosting algorithms require a priori bound on $\gamma$ to set the number of boosting rounds.

**Algorithm**: `AdaBoost.SS`
**Input**: sample $S = \{(x_i, y_i) : x_i \in X, y_i \in \{\pm 1\}, i = 1, \ldots, m\}$.

```
1. Initialize $D_1(i) = 1/m$ for all $i = 1, \ldots, m$.
2. For $t = 1, \ldots, T$ do:
    (a) $h_t \leftarrow \texttt{WeakLearn}(S, D_t)$.
    (b) Compute $\alpha_t \in \mathbb{R}_+$.
    (c) Update $D_{t+1}(i) = D_t(i) e^{-\alpha_t y_i h_t(x_i)}/Z_t$, for all $i$, where $Z_t = \sum_{i=1}^m D_t(i) e^{-\alpha_t y_i h_t(x_i)}$ is a normalization term.
```

**Output**: $H(x) = \text{sign}(\sum_{t=1}^T \alpha_t h_t(x))$.

**Theorem 10.** $\mathbb{P}_{i \sim U}[H(x_i) \neq y_i] \leq \prod_{t=1}^T Z_t$.

*Proof.* Let $f(x) = \sum_t \alpha_t h_t(x)$. By induction, $D_{T+1}(i) = e^{-y_i f(x_i)}/(m \prod_t Z_t)$. Note $H(x_i) \neq y_i$ implies $y_i f(x_i) \leq 0$ which in turn implies $e^{-y_i f(x_i)} \geq 1$. So, $\mathbf{1}_{\{H(x_i) \neq y_i\}} \leq e^{-y_i f(x_i)}$. Therefore,

$$\frac{1}{m} \sum_i \mathbf{1}_{\{H(x_i) \neq y_i\}} \leq \frac{1}{m} \sum_i e^{-y_i f(x_i)} = \prod_t Z_t.$$

$\square$

To motivate the choice of $\alpha_t$, assume $h : X \to [-1, 1]$. Fix the index of iteration $t$ and drop it. Let $u_i = y_i h(x_i)$. Then

$$Z = \sum_i D(i) e^{-\alpha u_i} \leq \sum_i D(i) \left( \frac{1 + u_i}{2} e^{-\alpha} + \frac{1 - u_i}{2} e^{\alpha} \right) \tag{15}$$

since the exponential function is convex. To see this, the equation of the line that passes through $(-1, e^{\alpha})$ and $(1, e^{-\alpha})$ is given by $(y - e^{\alpha})/(x+1) = (e^{-\alpha} - e^{\alpha})/2$ which implies $y = \frac{1+x}{2} e^{-\alpha} + \frac{1-x}{2} e^{\alpha}$.

Now, we choose $\alpha$ to minimize the upper bound in (15) by differentiation with respect to $\alpha$:

$$Z'(\alpha) = \sum_i D(i) \left( \frac{1 - u_i}{2} e^{\alpha} - \frac{1 + u_i}{2} e^{-\alpha} \right) = 0.$$

This yields

$$\alpha^\star = \tfrac{1}{2} \ln \left( \frac{1 + r}{1 - r} \right)$$

where $r = \sum_i D(i) u_i$. If $\epsilon = \frac{1}{2} - \gamma$, then

$$r = \mathbb{P}_{D(i)}[y_i = h(x_i)] - \mathbb{P}_{D(i)}[y_i \neq h(x_i)] = 2\gamma.$$

So,

$$\alpha^\star = \tfrac{1}{2} \ln \left( \frac{1 + 2\gamma}{1 - 2\gamma} \right) = \tfrac{1}{2} \ln \left( \frac{1 - \epsilon}{\epsilon} \right).$$

By using this minimizing $\alpha^\star$, we get

$$Z(\alpha^\star) \le \sqrt{1-r^2} \le e^{-r^2/2}.$$

Thus, the error of the boosting final hypothesis is bounded as follows:

$$\mathbb{P}_{i\sim U}[H(x_i) \neq y_i] \le \exp\left(-\frac{1}{2}\sum_t r_t^2\right).$$

(Freund-Schapire) Consider a special case where $h_t : X \to \{-1,1\}$ is a Boolean classifier If $\mathbb{P}_{i\sim D_t}[h_t(x_i) \neq y_i] = \frac{1}{2} - \gamma_t$, for some $\gamma_t > 0$, then $r_t = 2\gamma_t$.

### 8.1 Decision Trees

Assume $X = \bigcup_{i=1}^N X_i$ is the disjoint partitioning of the domain $X$. Suppose we require that each hypothesis $h$ must satisfy $h(x) = h(y)$ whenever $x, y$ belong to the same partition. Given the partitioning and the distribution $D$, what prediction values $y$ should be assigned for each block of the partition?

Let $c_j = h(X_j)$. For $j \in [N]$ and $b \in \{-1,+1\}$, let

$$W_b^j = \sum_i D(i)\mathbf{1}_{\{x_i \in X_j, y_i = b\}} = \mathbb{P}[x_i \in X_j, y_i = b].$$

Then,

$$Z = \sum_j \sum_{i:x_i\in X_j} D(i)e^{-y_i c_j} = \sum_j (W_{+1}^j e^{-c_j} + W_{-1}^j e^{c_j})$$

is minimized if

$$c_j^\star = \frac{1}{2}\ln\left(\frac{W_{+1}^j}{W_{-1}^j}\right).$$

This implies

$$Z = 2\sum_j \sqrt{W_{+1}^j W_{-1}^j} \tag{16}$$

which provides a splitting criterion for building a decision tree while the optimal values $c_j^\star$ give prediction values for each leaf. This observation is due to Kearns and Mansour.

## 9 Automata

We describe Angluin's $L^\star$ algorithm for learning a DFA (see Kearns and Vazirani [3]).

Recall that a *Deterministic Finite Automaton* (DFA) is given by the tuple $M = (\Sigma, S, \delta, s_0, F)$ where $\Sigma$ is a finite alphabet, $S$ is a finite set of states, $\delta : S \times \Sigma \to S$ is the transition function, $s_0 \in S$ is the start state, and $F \subseteq S$ is the set of accept states.

To each DFA $M$, we may associate a Boolean function $f_M : \Sigma^\star \to \{0,1\}$ where

$$f_(\alpha) = \begin{cases} 1 & \text{if } \delta(s_0,\alpha) \in F \\ 0 & \text{if } \delta(s_0,\alpha) \notin F \end{cases}$$

Each state $s$ of the DFA $M$ can be associated with an *access string* $\alpha$ where $\delta(s_0, \alpha) = s$. For two distinct states $s, s'$ of the DFA $M$, a string $\beta$ is called a *distinguishing string* for $s$ and $s'$ if $f_M(s \cdot \beta) \neq f_M(s' \cdot \beta)$. Here, we have identified the states with their access strings.

For the remainder of this section, we assume $\Sigma = \{0, 1\}$.

The $L^\star$ algorithm maintains a classification tree $T$ to represent a DFA $M$. Some properties of this tree $T$ are as follows:

- The internal nodes are labeled with *distinguishing* strings.
- The leaf nodes are labeled with *access* strings.
  So each leaf node represents a state in the DFA.
- The label of the *least common ancestor* node of two leaf nodes is a distinguishing string of the states represented by the two leaves.
- The empty string $\epsilon$ is always the label of the root node.
  This implies that the leaves on the left subtree of the root represent non-accepting states while the leaves on the right subtree represent accepting states.
- The empty string $\epsilon$ is always a label of one of the leaf nodes (this represents the start state).

$L^\star$ **learning algorithm**:

1. $b \leftarrow MQ(\epsilon)$ (ask Membership queries on the empty string).
   `/* This establishes whether` $s_0 \in F$ `or not */`

2. Let $M_0$ be a single state DFA which rejects all strings if $b = 0$ or accepts all strings if $b = 1$.
   `/* This constructs the simplest consistent one-state DFA */`

3. Ask Equivalence query $EQ(M_0)$.
   If oracle returns $YES$ then halt and output $M_0$, otherwise let $x_0$ be the counterexample.
   `/* If not done, the hidden DFA has at least 2 states */`

4. Let $T$ be a binary tree with a single root labeled $\epsilon$ with two children (leaf nodes).
   If $b = 0$, the left child is labeled $\epsilon$ and the right child is labeled $x_0$; otherwise, if $b = 1$, the right child is labeled $\epsilon$ and the left child is labeled $x_0$.
   `/* Construct T which encodes a consistent two-state DFA */`

5. While not done:

   (a) $\hat{M} \leftarrow$ `MakeDFA`$(T)$.
   `/* Transform T to a DFA */`

   (b) Ask $EQ(\hat{M})$. If oracle returns $YES$, halt and return $\hat{M}$; otherwise, let $x$ be the counterexample.
   `/* Check if done, otherwise get new counterexample (hence new state) */`

   (c) $T \leftarrow$ `UpdateTree`$(x, T, \hat{M})$.
   `/* Revise tree to reflect new counterexample */`

MakeDFA(tree $T$):
This algorithm constructs a DFA $M$ from a classification tree $T$.

```
1. For each access string (or leaf node) $s \in T$:
   Place $s$ in the set of states $S$.
   /* Each leaf of T is a DFA state */

2. For each state $s \in M$, for each $b \in \{0, 1\}$:
   If $s' =$ Sift$(s \cdot b, T)$ then $\delta(s, b) = s'$.
   /* Use Sift to find state transitions */

3. The start state $s_0$ is associated with the leaf labeled $\epsilon$.

4. The states whose access string belong to the right subtree of the root are placed into the set
   $F$ of accept states.

5. Return the DFA $M = (\{0, 1\}, S, \delta, s_0, F)$.
```

Sift(string $s$, tree $T$):
This algorithm traverses a classification tree $T$ from the root down to the leaf nodes using a given input string $s$. It returns the access string which labeled the resulting leaf node.

```
1. Let $c$ be the root of $T$

2. While $c$ is not a leaf node:

   (a) Let $d$ be the distiguishing string that is the label of $c$.
   (b) If $MQ(s \cdot d) = 1$ (accept), set $c$ to be the right child of $c$,
   (c) Else if $MQ(s \cdot d) = 0$ (reject), set $c$ to be the left child of $c$.

3. Return the access string which labels the leaf node $c$.
```

UpdateTree(string $x$, tree $T$, DFA $\hat{M}$):
This algorithm updates a classification tree $T$ based on a counterexample $x$ and the current DFA $\hat{M}$.

```
1. For each prefix $x[i] = x_1 \ldots x_i$ of $x$:
   Let $s_i \leftarrow$ Sift$(x[i], T)$ and $\hat{s}_i \leftarrow \hat{M}(x[i])$.

2. Let $j$ be the smallest index $i$ for which $s_i \neq \hat{s}_i$.

3. Replace the node labeled $s_{j-1}$ in $T$ with an internal node labeled $x_j \cdot d$ where $d$ is the
   distinguishing string of $s_j$ and $\hat{s}_j$.
   Let the two children of this new internal node be labeled with $s_{j-1}$ and a new access string
   $x[j-1]$. Let $s_{j-1}$ be the left child if $MQ(s_{j-1} \cdot x_j \cdot d) = 0$; otherwise let it be the right child.

4. Return the revised tree $T$.
```

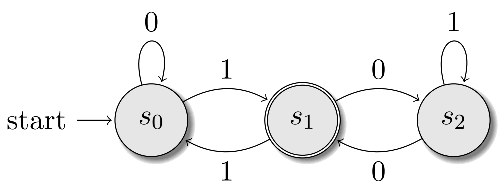


Figure 1: A 3-state Deterministic Finite Automata $M$.

## 9.1 Example

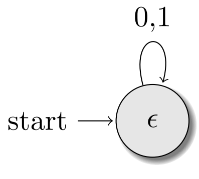
Alice (the Oracle) hides a deterministic finite automata (DFA) $M = (\Sigma, S, \delta, s_0, F)$ shown in Figure 1, where $S = \{s_0, s_1, s_2\}$ is the set of states, $\Sigma = \{0, 1\}$ is the finite alphabet, $s_0$ is the start state, and $F = \{s_1\}$ is the set of accept states. The transition function $\delta$ is deducible from Figure 1.

Bob (the Learner) uses the $L^\star$ algorithm to learn $M$ using *equivalence* and *membership* queries. Alice (the Oracle) always returns the **shortest** counterexample for each equivalence query.

**Trace of $L^\star$**

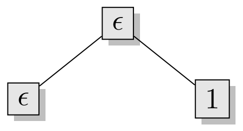
1. Ask $MQ(\epsilon) \to 0$.

2. Let $M_0$ be a DFA that rejects all strings given by

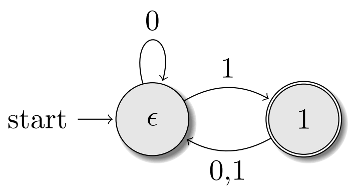
   

   Ask $EQ(M_0) \to 1$.

3. Build tree $T_1$ where

4. Build DFA $M_1 =$ `make-DFA`$(T_1)$.

   The construction of $M_1$ from $T_1$ requires computing

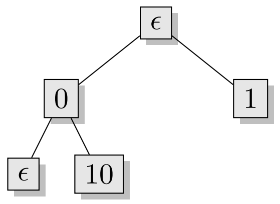
   | |
   |---|
   | `sift`$(T_1, \epsilon \cdot 0) = \epsilon$ |
   | `sift`$(T_1, \epsilon \cdot 1) = 1$ |
   | `sift`$(T_1, 1 \cdot 0) = \epsilon$ |
   | `sift`$(T_1, 1 \cdot 1) = \epsilon$ |

5. Ask $EQ(M_1) \to 100 = w_1$.

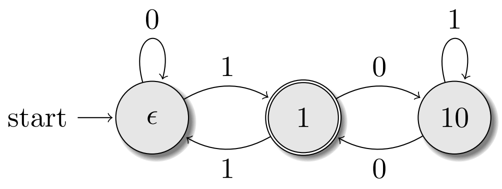
6. Update tree to $T_2$.

| prefix of $\beta = 100$ | $\texttt{sift}(T_1, \beta)$ | $M_1(\beta)$ |
|---|---|---|
| $\epsilon$ | $s_0 = \epsilon$ | $\hat{s}_0 = \epsilon$ |
| 1 | $s_1 = \epsilon$ | $\hat{s}_1 = \epsilon$ |
| 10 | $s_2 = \epsilon$ | $\hat{s}_2 = \epsilon$ |
| 100 | $s_3 = 1$ | $\hat{s}_3 = \epsilon$ |

Replace tree node labeled $s_2 = \epsilon$ (result of sifting prefix 10 on $T_1$) with an internal node with two leaf nodes: one leaf node is labeled with $s_2 = \epsilon$ and the other one with the corresponding prefix $\beta[2] = 10$. The label of the new internal node is $\beta_3\delta = 0$ since $\delta = \epsilon$ is the distinguishing string of $s_3 = 1$ and $\hat{s}_3 = \epsilon$.



7. Build DFA $M_2 = \texttt{make-DFA}(T_2)$.



The construction of $M_2$ from $T_2$ requires computing

| |
|---|
| $\texttt{sift}(T_2, \epsilon \cdot 0) = \epsilon$ |
| $\texttt{sift}(T_2, \epsilon \cdot 1) = 1$ |
| $\texttt{sift}(T_2, 1 \cdot 0) = 10$ |
| $\texttt{sift}(T_2, 1 \cdot 1) = \epsilon$ |
| $\texttt{sift}(T_2, 10 \cdot 0) = 1$ |
| $\texttt{sift}(T_2, 10 \cdot 1) = 10$ |

8. Ask $EQ(M_2) \to \texttt{YES}$ (since $M$ is equivalent to $M_2$).

## References

[1] N. Cristianini, J. Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods.* Cambridge University Press, 2000.

[2] M. Mohri, A. Rostamizadeh, A. Talwalkar. *Foundations of Machine Learning*, 2nd edition. The MIT Press, 2018.

[3] M. Kearns, U. Vazirani. *An Introduction to Computational Learning Theory.* The MIT Press, 1994.

[4] R. Schapire, Y. Freund. *Boosting.* The MIT Press, 2012.