Easy Learning Theory for Highly Scalable Algorithms

Claudio Gentile
DICOM
Università dell’Insubria, Italy
claudio.gentile@uninsubria.it

June 19, 2005

In this tutorial, we present a survey on recent advances in on-line prediction algorithms and their connections to statistical methods in Machine Learning.

An on-line learning algorithm is an incremental algorithm that sweeps through a sequence of examples only once. As such, on-line algorithms are both highly adaptive and highly scalable. They seem to be an unavoidable choice when dealing with very large datasets and/or when the datasets are rapidly changing with time.

In the worst-case setting of on-line learning, prediction performance is proven via a pointwise analysis, i.e., an analysis that avoids statistical assumptions on the way data are generated. Recent developments in the on-line learning literature show a natural way of turning on-line algorithms working in the worst-case setting into batch algorithms working under more standard stochastic (e.g., i.i.d.) assumptions on the data-generation process. The underlying statistical theory is fairly "easy", while the resulting algorithms are "highly scalable" (hence the title of this tutorial).

The error bounds one can derive from this theory are both data-dependent and algorithm-dependent. Besides, and most importantly, these bounds are trivial to compute and are likely to be sharp (at least as sharp as the existing literature on, say, Support Vector Machines).

There are two immediate practical benefits from this theory: 1) It delivers efficient and adaptive learning algorithms achieving the state-of-the-art error bounds obtained by computationally intensive methods; 2) it allows sharp and rigorous error reporting.

In this tutorial, we first review some of the most relevant (and recent) on-line algorithms in the worst-case setting (Part 1), then we discuss how to adapt these algorithms to a more classical statistical learning framework (Part 2).

Some of the results presented are well-known (and have been widely publicized), others are not and, we believe, deserve more attention by machine learning practitioners.
This tutorial is aimed at introducing the reader to statistical learning techniques derived from on-line learning algorithms working in the worst-case setting.

We say that an algorithm learning from examples works in an \textit{on-line} fashion when the examples in the sequence of examples are presented to the algorithm one at a time. At each time step (we will be calling a time step a \textit{trial}) the algorithm maintains an internal state and, after the current example has been processed, the algorithm changes its internal state to adapt to the new information the example has provided. The \textit{worst-case} setting of on-line learning is one in which no assumptions whatsoever are made on the way the stream of examples is generated. As a consequence, the performance bounds we can derive for these algorithms are \textit{pointwise} bounds, i.e., they do depend on (specific properties of) the sequence of examples the algorithm is executed on.

This tutorial is divided into two parts. The content of the first part is as follows. We first introduce a simple worst-case on-line learning setting, along with one of the simplest (and most powerful) learning algorithms in this setting, the so-called \textit{halving algorithm}. We generalize this setting to the framework of learning with \textit{expert advice}. The algorithms we will be mentioning within this framework might be seen as simple instances of a classical Bayes voting scheme. We then further generalize to the problem of learning linear-threshold functions.

All the algorithms we describe can be analyzed in the worst-case setting of on-line learning. The traditional analysis performed in this setting aims at providing bounds on the number of \textit{prediction mistakes} the algorithms make on the specific sequence of examples they are predicting on.

A worst-case analysis is a \textit{regret} analysis when the number of mistakes of the algorithm is compared to the number of mistakes of the best predictor in a comparison class of predictors. In this context, learning linear-threshold functions corresponds to assuming the comparison class be just a class of linear-threshold predictors.

One can extend the above framework along several directions. This tutorial briefly considers two of them:

1) We allow the best predictor in the comparison class change with time (hence we can view the learning algorithm as a target tracker);

2) We try to perform the same inference as above, while saving as many labels as possible.

For the sake of simplicity, all of the above will be treated in the case of binary classification problems (i.e., the case when the labels to be predicted are either +1 or -1), though many of the results we review in this tutorial can easily be extended to more general regression frameworks.
Slide 3

In the second part of this tutorial we show how these performance guarantees (the mistake bounds) can be directly turned into data and algorithm-dependent\(^1\) bounds on the generalization error in the standard statistical pattern recognition setting (examples drawn i.i.d. according to an arbitrary and unknown probability distribution). It is worth emphasizing that the resulting statistical bounds:

- Are tight; they are as tight as the generalization bounds one can derive by the most up to date data-dependent techniques (such as Rademacher complexity);
- Are easy to prove; they just rely on a couple of standard martingale inequalities;
- Are trivial to compute from the data;
- Refer to on-line algorithms, i.e., to highly scalable and highly adaptive learning procedures.

Throughout this tutorial we will motivate and expand on the above four points.

\(^1\)One can bound the generalization ability of learning algorithms through a simple expectation analysis ("what is on average (over a training set of size \(T\)) the probability of a mistake on the \((T+1)st\) example?"). This expectation analysis, however, provides less information than a data-dependent one, and is omitted from this tutorial.
Slide 4

We start by considering a simple problem of weather forecasting. We want to devise an algorithm to predict tomorrow’s weather (say, ”+1” = ”sunny” and ”-1” = ”rainy”) on a day-by-day basis. Assume this algorithm has at its disposal a set of $n$ experts, denoted by $E_1$, $E_2$, ..., $E_n$, which give us some advice about tomorrow’s weather in the form of binary predictions. For instance, on Day 1 the first expert predicted ”+1”, the second expert predicted ”+1”, the third expert predicted ”-1”, ..., the $n$-th expert predicted ”-1”. After observing all such predictions the algorithm is supposed to provide its own binary prediction (“-1”, in this case). Then tomorrow comes and the true label (“+1” in this case) is revealed to the algorithm. We say that the algorithm has made a prediction mistake if its prediction does not agree with the true label. We equivalently say that the algorithm incurred (zero-one) loss 1. The same story repeats on Day 2: the algorithm observes the advice from the experts as a binary ($\pm 1$) $n$-dimensional vector, delivers its prediction, observes the true label and incurs 1 unit of loss whenever prediction and true label disagree. In order to introduce some notation, we say that in the generic Day $t$ the algorithm gets a vector $z_t = (z_{t,1}, ..., z_{t,n}) \in \{-1, +1\}^n$ from the experts, makes prediction $\hat{y}_t \in \{-1, +1\}$, then gets the true label $y_t \in \{-1, +1\}$, and incurs loss $\frac{1}{2}|y_t - \hat{y}_t| \in \{0, 1\}$. This is our on-line learning protocol. To measure the algorithm’s performance, we just count the total number of mistakes the algorithm makes on the sequence of days Day 1, Day 2, ..., Day $T$, i.e., $\sum_{t=1}^{T} \frac{1}{2}|y_t - \hat{y}_t|$.

In the slides, we tend to use the following color conventions: algorithms’ predictions are coloured blue, labels are red, in-slide citations are green. A list of references is reported at the end of this write-up.
The first on-line algorithm we consider is the so-called Halving Algorithm (HA). This algorithm maintains and updates the set of consistent experts, namely the set of experts whose predictions agreed with all the labels seen so far. At the beginning, this set is the set of all $n$ experts. In the generic trial $t$ the algorithm observes the predictions made by the pool of consistent experts. Some of them predict +1, the remaining ones (within the pool) predict −1. The algorithm just goes with the majority. The experts which happen to be wrong on that trial, i.e., the experts $E_i$ belonging to the current pool and such that $z_{t,i} \neq y_t$, are erased from the pool to be used in the next trial. Thus the pool progressively shrinks. What’s important here is to relate the shrinkage rate of the pool of consistent experts to the number of trials when the algorithm turns out to be wrong, i.e., to the number of $t$ such that $\hat{y}_t \neq y_t$. Since the algorithm predicts with the majority of experts, the condition $\hat{y}_t \neq y_t$ clearly implies that at least half of the experts in the pool did wrong. Since the algorithm drops wrong experts from the pool, we have that on mistaken trials the number of consistent experts is at least halved. As a simple consequence, if there is at least one consistent expert in the initial pool of $n$ experts, then the number of mistakes made by HA is at most $\log_2 n$. This is because the consistent experts are not eliminated from the pool, and when only consistent experts remain HA predicts correctly.

The reader familiar with Bayes voting schemes and version space arguments will recognize HA as a simple example of a version space algorithm, where the version space is here the current pool of experts.
Slide 6

Let us consider HA in action. We have 8 experts. In trial 1 four experts predict +1, the remaining four predict −1. Assume ties are broken in favor of +1. Hence HA predicts +1, but the true label \( y_1 \) equals −1. The algorithm incurs one unit of loss and drops from the pool the four wrong experts \( E_1, E_2, E_5 \) and \( E_6 \). In trial 2 three experts predict +1, one expert predicts −1, the algorithm predicts +1, which turns out to be correct. Hence the algorithm incurs no loss and drops the wrong expert \( E_3 \) from the pool. In trial 3 expert \( E_4 \) predicts +1, while the remaining two experts predict −1. HA goes with the majority, making another mistake. The only remaining expert is \( E_4 \). The total number of mistakes made by the algorithm on this sequence is \( 2 \leq \log_2 8 = 3 \).

It is worth observing a few things:

- HA and its logarithmic performance bound is strongly dependent on the existence of a consistent expert. If there is no expert in the initial pool which is consistent with all labels in the sequence then HA just fails (all experts are eventually dropped from the pool). On the other hand, if more than one expert happens to be consistent with the sequence of true labels we can significantly refine the above logarithmic bound. Let \( k \geq 1 \) be the number of consistent experts. Then we have that

\[
\text{Number of mistakes of HA} \leq \log_2 (n/k).
\]

For a proof, just observe that if HA has made \( m \) mistakes, then it must be the case that the current pool has at most \( n/2^m \) experts. Since the consistent experts are never dropped, we have \( n/2^m \geq k \). Then we solve for \( m \).

- In the second trial HA has erased experts even if its prediction was correct. In practice, dropping wrong experts in non-mistaken trials tends to speed up the shrinkage process of the pool of experts. In general, however, it is not easy to take into account the behavior of the algorithm on these trials. Thus, for the very sake of proving a logarithmic mistake bound, whether or not HA drops experts in trials \( t \) such that \( \hat{y}_t = y_t \) is actually \textit{immaterial}.

- It should be clear that what we called "experts" can be pretty much everything. An expert is just a "device" that outputs a prediction. In fact, a prediction algorithm (such as HA) can in turn play the role of an "expert" exploited by some meta-prediction algorithm running at some higher level.
Slide 7

What if no expert is consistent with the label sequence?

Let us introduce some more notation. Denote by $S$ a (finite) sequence of examples $S = (z_1, y_1), (z_2, y_2), \ldots, (z_T, y_T)$. Notice that, as is customary, here an example $(z_i, y_i)$ is made up of what the algorithm observes before making a prediction (i.e., $z_i$, the vector of expert advices) and the value to be predicted (i.e., $y_i$).

For a generic prediction algorithm $A$, we denote by $L_A(S)$ the total loss (number of mistakes) the algorithm $A$ incurs on sequence $S$. Also, $L_i(S)$ is the total loss (number of mistakes) incurred by the $i$-th expert $E_i$ on the same sequence.

We are then looking for cumulative loss bounds of the form

$$L_A(S) \leq a \min_{i=1}^{n} L_i(S) + b \log n,$$

holding for any data sequence $S$, where $a$ and $b$ are suitable constants (independent of $S$). Such bounds we call relative loss bounds, i.e., bounds on the cumulative loss relative to the cumulative loss of the best expert in hindsight for the sequence $S$.

Notice that assuming the existence of a consistent expert (the condition under which HA works) is equivalent to $\min_{i=1}^{n} L_i(S) = 0$. 
One simple and powerful way of obtaining relative loss bounds is to associate with each expert a (nonnegative) weight. The weights quantify the reliability of the experts. At the beginning all experts have the same weight, meaning that the experts are equally important/reliable (no prior information). Then, as the time passes, we tend to rely only on experts $E_i$ which have made few prediction mistakes (i.e., those experts whose cumulative loss so far is small).

The Weighted Majority algorithm (WMA) might be seen as a "smoothed" version of HA. WMA adopts the following multiplicative weighting scheme. Let $w_{i,t}$ be the weight vector associated to expert $E_i$ at the beginning of trial $t$ (i.e., before the algorithm outputs its $t$-th prediction). The starting weights $w_{i,1}$ are all equal to 1. In the generic trial $t$ the algorithm observes $z_i \in \{-1, +1\}^n$, then compares the sum of the weights of all experts that predicted +1 to the sum of the weights of all experts that predicted −1. As for HA, WMA goes with the majority. After observing the true label $y_t \in \{-1, +1\}$, the weights of the experts which turn out to be wrong are slashed by a factor $\beta \in [0,1)$, thereby reducing their importance on future predictions.

Notice that WMA with $\beta = 0$ is actually HA, since an expert whose weight has reached 0 will no longer be involved in the algorithm’s voting mechanism (this is why we claim that WMA is a smoothed version of HA).

Again, the reader familiar with Bayes algorithms will recognize WMA as a simple example of a Bayes voting scheme, where the initial weights $w_{1,i}$ are an unnormalized prior distribution over the experts, and the actual weights $w_{t,i}$ are an unnormalized posterior.
Slide 9

Let’s now consider how one can prove relative loss bounds for WMA. We need some short-hand notation. Let $L_{i,t}$ be the cumulative loss of expert $E_i$ up to time $t - 1$, so that $L_{i,T+1} = L_i(S)$. Then the weight $w_{i,t}$ associated with $E_i$ at the beginning of trial $t$ equals $\beta^{L_{i,t}}$. Let also $W_t$ be the total weight at time $t$, i.e., $W_t = \sum_{i=1}^n w_{i,t}$.

At time $t$ we split the experts into the “majority of experts” and the “minority of experts”. The former are the experts whose prediction coincides with WMA’s. The latter are the experts whose prediction differs from WMA’s. Clearly, the total weight of the majority of experts is at least one half the total weight $W_t$, whereas the total weight of the minority of experts is at most one half the total weight $W_t$. Let’s study the behavior of $W_t$ over mistaken trials. If WMA’s prediction is wrong at time $t$, then the weights of the majority of experts gets multiplied by $\beta$, while the remaining weights are left unchanged. Hence, if WMA’s prediction is wrong at time $t$ we can write

\[
W_{t+1} \leq 1 \text{ Minority} + \beta \text{ Majority} \\
\leq \frac{1 + \beta}{2} \text{ (Majority+Minority)} \\
= \frac{1 + \beta}{2} W_t,
\]

where the second inequality follows from $\text{Majority} \geq \text{Majority}$. Thus on a mistaken trial the total weight decreases by a factor at least $\frac{1+\beta}{2} < 1$. On the other hand, if WMA’s prediction is correct at time $t$, we have $W_{t+1} \leq W_t$ anyway.
Slide 10

Thus by induction we conclude that if WMA has made $L_{WMA}(S)$ mistakes on sequence $S$, the total final weight $W_{T+1}$ satisfies

$$W_{T+1} \leq \left( \frac{1 + \beta}{2} \right)^{L_{WMA}(S)} W_1 = \left( \frac{1 + \beta}{2} \right)^{L_{WMA}(S)} n,$$

(since the initial weights $w_{1,i} = 1$).

On the other hand, the behavior of $W_{T+1}$ is also ruled by the cumulative loss of the experts on sequence $S$. We can write

$$W_{T+1} = \sum_{j=1}^{n} w_{T+1,j} = \sum_{j=1}^{n} \beta^{L_j(S)} \geq \beta^{L_i(S)},$$

the last inequality holding for any expert $E_i$.

Putting together

$$\left( \frac{1 + \beta}{2} \right)^{L_{WMA}(S)} n \geq \beta^{L_i(S)}.$$  

We could interpret this inequality by saying that the total weight $W_i$ decreases with time as a function of the cumulative loss of the algorithm, but it cannot decrease too fast. Solving the above for $L_{WMA}(S)$ gives the by now classical relative loss bound for WMA:

$$L_{WMA}(S) \leq \frac{1/\beta}{\log \frac{2}{1+\beta}} \min_i L_i(S) + \frac{1}{\ln \frac{2}{1+\beta}} \log n.$$  

In particular, setting $\beta = 1/e$ ($e$ is the base of natural logarithms) yields a bound which is easier to read:

$$L_{WMA}(S) \leq 2.63 \min_i L_i(S) + 2.63 \log n.$$  

The above relative loss bound holds for any sequence $S$, and any set of experts $E_1, ..., E_n$. This bound is telling us that the number of mistakes made by WMA on $S$ is comparable to the number of mistakes made by the best expert (as measured by the number of prediction mistakes) on the same sequence $S$. Thus, if there is an expert making only few mistakes on $S$ then WMA will make few mistakes, too.
Slide 11

It is worth catching a glimpse of more general frameworks than binary classification. Consider, for instance the case when the true labels $y_t$ lie in the real interval $[-1, +1]$ and the experts $E_i$ are allowed to output predictions $z_{t,i} \in [-1, +1]$. In this slightly more general situation it is natural to let the prediction algorithm produce values $\hat{y}_t$ lying in the same interval, and measure the discrepancy between prediction and true label by a loss function which is able to deal with real numbers. One such loss (the one we consider here) is the absolute loss $\frac{1}{2}|\hat{y}_t - y_t| \in [0, 1]$. The extension of WMA coping with this situation maintains negative exponential weights of the form $w_{t,i} = \beta^{L_{t,i}}$. The algorithm uses normalized weights $v_{t,i}$ and predicts at time $t$ with the inner product $\hat{y}_t = v_t^T z_t = \sum_{i=1}^n v_{t,i} z_{t,i} \in [-1, +1]$.

As usual, we want to compare the cumulative (absolute) loss $L_A(S) = \sum_{t=1}^T \frac{1}{2}|y_t - \hat{y}_t|$ of algorithm $A$ to the (absolute) loss of the best expert for the sequence $S$. 
Slide 12

By a technique similar to the one we sketched in Slides 9 and 10, one can prove a bound of the form

$$L_A(S) \leq (1 + \eta) \min_i L_i(S) + \frac{1 + \eta}{\eta} \log n,$$

where $\eta = \log 1/\beta$. Compared to the one in Slide 10, this relative loss bound tends to gain a factor of two. This is roughly due to the fact that the algorithm is not forced to produce predictions at the two extremes of the interval $[-1, +1]$ (as was the case for binary classification). In a worst-case scenario like the one we are considering here, hedging bets tends to be a convenient strategy.

When parameter $\beta$ (usually called learning rate) is constant with time, then the resulting constant in front of the loss of the best expert is larger\(^2\) than one. In order to obtain regret bounds on the absolute loss, i.e., bounds on the difference

$$L_A(S) - \min_i L_i(S),$$

it is better using an adaptive (time-changing) learning rate. This specific topic, however, goes beyond the scope of this tutorial. What we have written so far only scratches the surface of the large body of literature on learning with expert advice. The interested reader is referred to the following papers (and references therein): [L88, LW94, Vo90, FMG92, FS97, FSSW97, CB+97, HKW98, Vo98, KW99, ACBG02].

---

\(^2\)This actually depends on the fact that we are considering the absolute loss as our measure of prediction accuracy. See [Vo90].
The Weighted Majority algorithm (WMA) is just a Bayes voting scheme that makes it easy to combine prediction algorithms (experts) so that the resulting combined algorithm performs (almost) as good as the best expert in hindsight.

The performance bounds we can prove for WMA, such as the relative loss bounds in Slides 10 and 12, have always two components: the first component is the contribution from the best expert, the second component is the cost of aggregating experts. Notice that the cost of aggregation has a mild \((\text{logarithmic})\) dependence on the number of experts.

We now want to extend the learning framework a little bit. The problem of learning with expert advice (the way we presented it) might be seen as the problem of learning relative to the best (fixed) component of the vector of observations \(z_i \in \{-1, +1\}^n\) the algorithm receives before making predictions. Within this restricted setup, the algorithm's performance is compared to the performance of the best fixed component of vectors \(z_i\).

In fact, nothing prevents us from being more demanding. We would like to generalize our framework to "learning relative to the best thresholded linear combination of components of \(z_i\)". As it turns out, this is just a complicated way of saying "learning linear-threshold functions in an on-line fashion".
Let’s consider an even more general point of view.

In the generic trial $t$, algorithm $A$ receives an instance vector $x_t \in \mathbb{R}^n$, outputs its binary prediction $\hat{y}_t \in \{-1,+1\}$ and, upon receiving the true label $y_t \in \{-1,+1\}$ associated with $x_t$, incurs 0-1 loss $L(y_t, \hat{y}_t) = \frac{1}{2} |y_t - \hat{y}_t|$.

The algorithms’ prediction performance on a sequence of examples $S = (x_1, y_1), ..., (x_T, y_T) \in \mathbb{R}^n \times \{-1,+1\}$ is measured again by the cumulative loss (number of mistakes) $L_A(S) = L(y_1, \hat{y}_1) + ... + L(y_T, \hat{y}_T)$.

Given a comparison class of predictors $\{u\}$, we would like $A$ be almost as good as the best predictor in $\{u\}$ on sequence $S$. For this purpose, we would like to prove bounds on the following kind of regret:

$$L_A(S) - \inf_u \text{Loss}_u(S),$$

where Loss is some loss function (possibly equal to $L$), measuring the cumulative prediction performance of the predictors in $\{u\}$.

It is easy to see that the above setup generalizes our previous setup about learning with expert advice. Let the class $\{u\}$ be a class of linear predictors (so that the prediction of $u$ on instance $x$ is $u(x) = u^T x$). If $\{u\}$ is made up of the $n$ unit vectors $u_1 = (1, 0, 0, ..., 0)$, $u_2 = (0, 1, 0, ..., 0), ..., u_n = (0, 0, ..., 0, 1)$ and the instance vectors $x_t$ are binary ($x_t \in \{-1,+1\}^n$), then we could interpret $u_i^T x_t = x_{t,i}$ as the $t$-th binary prediction of the $i$-th expert (what we denoted by $z_{t,i}$ in earlier slides). Setting $\text{Loss}_{u_i}(S) = \sum_{t=1}^T \frac{1}{2} |y_t - x_{t,i}|$ gives exactly the binary expert framework.
Figure 1: Two-dimensional version space for HA. Vector \( \mathbf{u} \) is a linear separator for sequence \( S \), while the red cone where \( \mathbf{u} \) lies denotes the set of consistent experts. Observe that \( \mathbf{u} \) is in the middle of this cone. The angle \( 2\mu \) of this cone is related via \( \theta \) to the margin \( \gamma \) of \( \mathbf{u} \) over \( S \). Here \( \theta + \mu = \pi/2 \). The two arrows tagged by \( y\mathbf{x}/||\mathbf{x}|| \) are the edges of the circular sector (whose angle is \( 2\theta \)) where the vectors \( y_t\mathbf{x}_t \) are allowed to lie (to let all experts in the red-marked cone be consistent).

Slides 15–16

We think it’s instructive to consider a further application of the Halving Algorithm (HA).

Consider a sequence of examples \( S = (\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_T, y_T) \), where the instance vectors \( \mathbf{x}_t \) lie on the plane \( \mathbb{R}^2 \), and the labels \( y_t \) are binary. We assume \( S \) is linearly separable with margin \( \gamma > 0 \), i.e., we assume the existence of a unit-norm vector \( \mathbf{u} \in \mathbb{R}^2 \) such that \(^3\) \( y_t\mathbf{u}^T\mathbf{x}_t \geq \gamma \), for any \( t \). Finally, we let \( R > 0 \) be the radius of the smallest origin-centered circle containing the vectors \( \mathbf{x}_t \).

HA has at its disposal a set of \( n \) experts which are linear-threshold functions evenly scattered over the unit circle. The \( i \)-th expert is associated with a unit-norm vector \( \mathbf{u}_i \). In the generic trial \( t \) this expert outputs the binary prediction \( z_{t,i} = \text{SGN}(\mathbf{u}_i^T\mathbf{x}_t) \).

The following protocol combines learning with expert advice with the learning model we introduced in Slide 14.

In trial 1 all experts are fed with \( \mathbf{x}_1 \), and output their predictions \( z_{1,i} = \text{SGN}(\mathbf{u}_i^T\mathbf{x}_1) \), which we collect into the prediction vector \( \mathbf{z}_1 = \langle z_{1,1}, z_{1,2}, \ldots, z_{1,n} \rangle \in \{-1, +1\}^n \). HA goes with the majority. In the bottom-right figure of Slide 15 we have 16 experts predicting \(+1\) and 16 experts predicting \(-1\). For definiteness, assume ties are broken in favor of \(-1\). Then HA predicts \(-1\), the true label \( y_1 = +1 \) is revealed and HA drops all mistaken experts. These experts correspond to vectors \( \mathbf{u}_i \) such that the angle between \( y_1\mathbf{x}_1 \) and \( \mathbf{u}_i \) is larger than \( \pi/2 \).

In trial 2 vector \( \mathbf{x}_2 \) is fed to the remaining 16 experts. Out of the 16 experts, 7 predict \(+1\), 9 predict \(-1\). Then HA predicts \(-1\). The true label \( y_2 \) is again \(+1\) and HA makes another mistake. The algorithm eliminates the 9 mistaken experts and goes to the next trial.

\(^3\)This condition tells us that the absolute value of the inner product \( \mathbf{u}^T\mathbf{x}_t \) is at \( \gamma \), while the sign of \( \mathbf{u}^T\mathbf{x}_t \) equals \( y_t \). Geometrically, \( y_t = \text{SGN}(\mathbf{u}^T\mathbf{x}_t) \) means that the angle between \( \mathbf{u} \) and \( y_t\mathbf{x}_t \) is at most \( \pi/2 \). See Figure 1.

15
A reasonable choice of Loss is one in which \( \text{Loss}_u(S) = 0 \) whenever \( u \) is consistent with \( S \) (this is the case, for example, when Loss is the standard 0-1 loss). Thus, we can apply the argument we made in Slide 6. As we pointed out there, if the number of experts consistent with the sequence of labels \( y_1, y_2, \ldots, y_T \) is at least \( k \), then the number \( m_{HA} \) of prediction mistakes made by HA can be bounded from above by \( m_{HA} \leq \log_2(n/k) \). In our case, we are thinking of \( n \) as being very large, implying that the ratio of the number of consistent experts to the total number of experts (i.e., \( k/n \)) tends to be proportional to the angle of the circular sector determined by the consistent experts (bottom-right figure in Slide 16).

Now consider Figure 1. If \( n \) is very large we can consider the experts as being uniformly distributed over the unit circle. Thus we can write \( \frac{k}{n} \approx \frac{\theta}{2\pi} \). On the other hand, by standard trigonometry and the margin assumption we also have \( \cos \theta = \frac{y^\top u}{||x||} \geq \gamma/R \). Combining with \( \theta + \mu = \pi/2 \) results in

\[
\frac{k}{n} \approx \frac{1}{2} - \frac{\theta}{\pi} = \frac{1}{2} - \frac{1}{\pi} \arccos \left( \frac{y^\top u}{||x||} \right) \geq \frac{1}{2} - \frac{1}{\pi} \arccos \left( \frac{\gamma}{R} \right).
\]

Now, since \( \frac{1}{2} - \frac{1}{\pi} \arccos(x) \geq x/\pi \), for \( x \in [0, 1] \), we conclude that

\[
m_{HA} \leq \log_2(n/k) \leq \log_2(R\pi/\gamma),
\]

provided \( n \) is large.
The previous argument can be generalized to a \( d \)-dimensional instance space. Set \( S = (x_1, y_1), \ldots, (x_T, y_T) \), with \( x_t \in \mathbb{R}^d \), such that \( ||x_t|| \leq R \). Assume \( S \) is linearly separated with margin \( \gamma > 0 \) by a unit-norm \( d \)-dimensional vector \( u \), and suppose the experts are associated with points uniformly spread over the surface of the \( d \)-dimensional unit sphere.

The argument is roughly the following. Since \( u \) linearly separates \( S \), if we pick a unit-norm vector \( u' \) close enough to \( u \), we still obtain a linear separator for \( S \). More precisely, it is not hard to show that the condition

\[
y_t u^T x_t \geq \gamma \text{ and } ||u - u'|| < \gamma / R
\]

implies \( y_t (u')^T x_t > 0 \). Hence, if we denote by consistent(\( S \)) the set of experts consistent with \( S \), this argument shows that there exists a \((d - 1)\)-dimensional ball \( B \) of radius \( \gamma / 2R \) which is included in consistent(\( S \)). The \((d - 1)\)-dimensional area of this ball is at least

\[
(\gamma / 2R)^{d-1} \mu(\text{surface of the } d\text{-dim. unit sphere}).
\]

Since

\[
k/n \approx \mu(B) / \mu(\text{surface of } d\text{-dim. unit sphere}),
\]

we conclude that, when \( n \) is large,

\[
m_{HA} \leq \log_2(n/k) = O(d \log(R / \gamma)).
\]

The above mistake bound holds only in the case of linearly separable data. The bound has an attractive logarithmic dependence on the margin \( \gamma \), but also an inconvenient linear dependence on the dimension \( d \) of the space. Furthermore, the practical implementation of this algorithm is not at all easy, since we have to keep track of the pool of consistent experts as (spherical) caps over the surface of the \( d \)-dimensional unit sphere, which might lead to an impractical algorithm. See [GBNT04] (and reference therein) for recent advances on this specific topic.

**Remark** [GB04]

As a matter of fact, the linear dependence on \( d \) can be removed via *random projection*. We take the instance vectors \( x_t \in \mathbb{R}^d \) and randomly project them onto a lower dimensional space \( \mathbb{R}^p \), with \( p < d \). Since this projection preserves (with high probability) Euclidean distances among vectors, the new projected training set is likely to be linearly separable (with margin, say, \( \gamma / 2 \), instead of \( \gamma \)). Then we can run HA on the projected space \( \mathbb{R}^p \) and get a mistake bound having a linear dependence on \( p \). If \( p \) is carefully chosen, this procedure overcomes the above linear dependence problem. See, e.g., [AV99] for details.

---

\(^4\)In order to avoid confusion, in this slide we denote by \( d \) the dimension of the instance space, whereas \( n \) still denotes the number of experts. Here \( n \) is meant to be very large, possibly infinite.
Slide 18

We now consider an alternative (and well-known) method for learning linear-threshold functions. It is the so-called Perceptron algorithm [Ro62, Bl62, No62]. We call it first-order Perceptron algorithm to distinguish it from the second-order algorithm we will encounter in subsequent slides.

As we will see, the algorithm learns relative to the best thresholded linear combination of experts, but instead of comparing to the cumulative 0-1 loss of the best linear-threshold function, it compares to a convex upper bound on this cumulative loss. This substitution makes the algorithm a bit less powerful in terms of prediction performance, but also far more efficient in terms of running time.

Unlike HA, the Perceptron algorithm maintains an $n$-dimensional weight vector $\mathbf{w}_t$. The algorithm works as follows. It starts off with some initial vector $\mathbf{w}_1 \in \mathbb{R}^n$ (typically $\mathbf{w}_1 = 0$). In the generic trial $t$, the algorithm receives instance $\mathbf{x}_t \in \mathbb{R}^n$ and predicts the associated label $y_t \in \{-1,+1\}$ by $\hat{y}_t = \text{SGN}(\mathbf{w}_t^\top \mathbf{x}_t) \in \{-1,+1\}$. If $\hat{y}_t \neq y_t$ (i.e., if $y_t \mathbf{w}_t^\top \mathbf{x}_t \leq 0$) then a mistake occurs and the algorithm updates its weight vector according to the additive rule

$$\mathbf{w}_{t+1} = \mathbf{w}_t + y_t \mathbf{x}_t.$$  

Geometrically, the weight update rule moves the new weight vector towards the direction of $y_t \mathbf{x}_t$. Observe that

$$y_t \mathbf{w}_{t+1}^\top \mathbf{x}_t = y_t \mathbf{w}_t^\top \mathbf{x}_t + ||\mathbf{x}_t||^2 \geq y_t \mathbf{w}_t^\top \mathbf{x}_t.$$  

Thus we might view the update rule as an attempt to force the algorithm’s margin $y \mathbf{w}^\top \mathbf{x}$ be positive on all examples $(\mathbf{x}, y)$ in the training sequence.
What kind of mistake bound can we prove for the Perceptron algorithm?

Consider an arbitrary sequence of examples \( S = (\mathbf{x}_1, y_1), ..., (\mathbf{x}_T, y_T) \in \mathbb{R}^n \times \{-1, +1\} \). The sequence need not be linearly separable (as was the case for the application of the Halving Algorithm described in Slides 15-17). Since the performance of the Perceptron algorithm is naturally expressed in the context of linearly separable sequences, we need some measure that tells us to what extent the sequence \( S \) deviates from linear separability. Many of these measures can be defined. The one we use here is fairly popular in the literature of linear-threshold learning, and has been called in many ways: deviation, hinge loss, soft margin. Given a margin value \( \gamma > 0 \) and a unit-norm weight vector \( \mathbf{u} \) (which is meant to represent a hyperplane passing through the origin), we define the deviation of \( \mathbf{u} \) over example \((\mathbf{x}, y) \in \mathbb{R}^n \times \{-1, +1\}\) as

\[
D_\gamma((\mathbf{x}, y); \mathbf{u}) = \max \{0, 1 - y \mathbf{u}^\top \mathbf{x}/\gamma\}.
\]

Since \( \mathbf{u} \) is implicitly representing the linear-threshold predictor

\[
\mathbf{u} : \mathbf{x} \rightarrow \text{SGN}(\mathbf{u}^\top \mathbf{x}),
\]

the quantity \( D_\gamma((\mathbf{x}, y); \mathbf{u}) \) is a (convex) upper bound on the 0-1 loss function

\[
\frac{1}{2}|y - \text{SGN}(\mathbf{u}^\top \mathbf{x})| = \begin{cases} 1 & \text{if } y \mathbf{u}^\top \mathbf{x} \leq 0; \\ 0 & \text{otherwise}. \end{cases}
\]

See the figure in Slide 19. We also define the deviation of \( \mathbf{u} \) over a sequence \( S \) as

\[
D_\gamma(S; \mathbf{u}) = \sum_{t=1}^{T} D_\gamma((\mathbf{x}_t, y_t); \mathbf{u}) = \sum_{t=1}^{T} \max \{0, 1 - y_t \mathbf{u}^\top \mathbf{x}_t/\gamma\}.
\]

Observe that if \( S \) is separated by \( \mathbf{u} \) with margin at least \( \gamma \) then \( D_\gamma(S; \mathbf{u}) = 0 \).

We now prove the following mistake bound:

\[
m_{\text{perc}} \leq \inf_{\gamma > 0, \norm{\mathbf{u}}_2 = 1} \left( D_\gamma(\mathbf{u}; S) + \frac{\sqrt{\sum_{t \in \mathcal{M}} \norm{\mathbf{x}_t}_2^2}}{\gamma} \right),
\]

where \( \mathcal{M} \subseteq \{1, 2, ..., T\} \) is the set of trials where the Perceptron algorithm made a mistake. As we already mentioned, \( D_\gamma(\mathbf{u}; S) \) is our convex upper bound on the 0-1 loss of \( \mathbf{u} \) on \( S \). This upper bound can be somewhat sharpened by restricting to trials \( t \in \mathcal{M} \). Thus our measure of prediction performance of the comparison class \( \{\mathbf{u}\} \) over \( S \) is actually

\[
D_\gamma(S; \mathbf{u}) = \sum_{t \in \mathcal{M}} D_\gamma((\mathbf{x}_t, y_t); \mathbf{u}) = \sum_{t \in \mathcal{M}} \max \{0, 1 - y_t \mathbf{u}^\top \mathbf{x}_t/\gamma\}.
\]

The proof of this statement is fairly simple. Consider a generic mistaken trial \( t \in \mathcal{M} \), and let \( \mathbf{u} \) be an arbitrary unit-norm vector. By the update rule we have

\[
\mathbf{u}^\top \mathbf{w}_{t+1} - \mathbf{u}^\top \mathbf{w}_t = y_t \mathbf{u}^\top \mathbf{x}_t \geq \mathbf{u}^\top \mathbf{w}_t + \gamma - \gamma D_\gamma((\mathbf{x}_t, y_t); \mathbf{u}),
\]
where the inequality follows from the very definition of $D_\gamma((x_i, y_i); u)$ and holds for any $\gamma > 0$. Summing over $t \in \mathcal{M}$ and assuming $w_1 = 0$ we can write

$$u^\top w_{T+1} \geq \gamma m_{\text{Perc.}} - \gamma \sum_{t \in \mathcal{M}} D_\gamma((x_t, y_t); u).$$

On the other hand, by the Cauchy-Schwartz inequality $u^\top w_{T+1} \leq ||u|| ||w_{T+1}|| = ||w_{T+1}||$. Now, observe that the norm of the weight vector $w_t$ cannot grow too fast over mistaken trials. In fact,

$$||w_{t+1}||^2 - ||w_t||^2 = 2y_t w_t^\top x_t + ||x_t||^2 \leq ||x_t||^2$$

(since $y_t w_t^\top x_t \leq 0$).

Therefore, if we sum again over $t \in \mathcal{M}$ we get

$$||w_{T+1}||^2 \leq \sum_{t \in \mathcal{M}} ||x_t||^2.$$

Putting together we have obtained

$$\sqrt{\sum_{t \in \mathcal{M}} ||x_t||^2} \geq \gamma m_{\text{Perc.}} - \gamma \sum_{t \in \mathcal{M}} D_\gamma((x_t, y_t); u).$$

Solving for $m_{\text{Perc.}}$ gives the stated bound.
The above bound on $m_{\text{Perc}}$, is stated in an implicit form. This has the purpose of emphasizing the loss of the comparison class. When this loss is zero, we can turn the bound into a more familiar formula. Let $S$ be linearly separable with margin $\gamma > 0$, i.e., assume there exists $u$, such that $y_i u^T x_i \geq \gamma$, $\forall t$, $||u||_2 = 1$. If we further assume $||x_i|| \leq R$ the above bound implies the familiar Perceptron’s mistake bound

$$m_{\text{Perc}} \leq \frac{R^2}{\gamma^2}.$$  

This bound is a pointwise bound on the number of mistakes, in the sense that it relates the prediction performance of the Perceptron algorithm over a sequence of examples $S$ to some specific properties of $S$. The ratio $R/\gamma$ is indeed a geometric property of $S$. Notice that, according to Slides 15-17, the behavior of HA seems to depend on the same ratio when learning linear threshold functions. Therefore, if we restrict to linearly separable problems, the two algorithms we have seen so far for learning linear-threshold functions seem to be mainly affected by examples $(x_i, y_i)$ having maximal length $||x_i||$ and minimal margin (dotted circles in the figure of Slide 20). For the Perceptron algorithm this is easily explained by a simple example. The figure of Slide 20 illustrates the behavior of the Perceptron algorithm on extreme (though linearly separable) cases. Here $u$ denotes the hidden target vector, $w_t$ is the weight vector maintained by the algorithm at the beginning of trial $t$ and $x_t$ is the instance vector observed in that trial. We are assuming that all instances have bounded (euclidean) length $R$, and that the examples are linearly separable with margin $\gamma > 0$ (so that no vector in the sequence of examples can lie within the two dotted lines running parallel to the decision boundary of $u$). Since the angle between $u$ and $x_t$ is (slightly) larger than 90 degrees the label $y_t$ is assigned the value $-1$. On the other hand, $w_t^T x_t > 0$ holds; hence the algorithm makes a mistake. Now, $x_t$ lies exactly on one of the two dotted lines, and has maximal length $R$, but has also a small projection along the direction of $u$, meaning that the direction marked by $x_t$ is (almost) irrelevant to $u$. However, the simple additive rule of the Perceptron algorithm makes the new weight vector $w_{t+1}$ even farther from $u$ than the old one!
We now turn to a relatively new on-line algorithm for learning linear-threshold functions. It is the so-called Second-order Perceptron algorithm [CBCG05].

In its basic form (the one we consider here), the second-order Perceptron algorithm takes an input parameter \( a > 0 \). To compute its prediction in trial \( t \) the algorithm uses an \( n \times n \) matrix \( S_t \) and an \( n \)-dimensional weight vector \( \mathbf{w}_t \). Initially, the algorithm sets \( S_1 = 0 \) and \( \mathbf{w}_1 = 0 \). Upon receiving the \( t \)-th instance \( \mathbf{x}_t \in \mathbb{R}^n \), the algorithm predicts the label \( y_t \) of \( \mathbf{x}_t \) with \( \hat{y}_t = \text{sgn} \left( \mathbf{w}_t^T \left( aI + S_t \right)^{-1} \mathbf{x}_t \right) \), with \( I \) being the \( n \times n \) identity matrix (the addition of \( aI \) guarantees that the above inverse always exists). If \( \hat{y}_t \neq y_t \) then a mistake occurs and the algorithm updates both \( \mathbf{w}_t \) and \( S_t \). In order to simplify the subsequent exposition, we assume the second-order algorithm only "sees" normalized instance vectors \( \tilde{\mathbf{x}}_t = \mathbf{x}_t / ||\mathbf{x}_t|| \). Vector \( \mathbf{w}_t \) is updated using the (first-order) Perceptron rule \( \mathbf{w}_{t+1} = \mathbf{w}_t + y_t \tilde{\mathbf{x}}_t \), whereas matrix \( S_t \) is updated using \( S_{t+1} = S_t + \tilde{\mathbf{x}}_t \tilde{\mathbf{x}}_t^\top \), i.e., by adding the rank-one matrix \( \tilde{\mathbf{x}}_t \tilde{\mathbf{x}}_t^\top \) to \( S_t \). The new matrix \( S_{t+1} \) and the new vector \( \mathbf{w}_{t+1} \) will be used in the next prediction. If \( \hat{y}_t = y_t \) no update takes place.

This algorithm is called second-order Perceptron because it keeps track of second-order information about the data (the data correlation matrices \( S_t \)), and therefore is similar to second-order descent methods in convex optimization, such as the Newton method. The positive parameter \( a \) interpolates between first-order and second-order structure of the algorithm. Observe that when \( a \to \infty \) the second-order Perceptron algorithm turns to the first-order one.

The second-order Perceptron algorithm might be viewed as an adaptation to on-line binary classification of the well-known ridge regression method [HK70]. Check [Vo01, AzW01, FoW02, FoW03] for on-line variants of ridge regression. But, unlike ridge regression algorithms, the second-order Perceptron algorithm only keeps track of past trials where the algorithm made a mistake. This has the big computational advantage of making the algorithm produce a sparse hypothesis. See Slides 25-27.
Let $S = (\hat{x}_1, y_1), ..., (\hat{x}_T, y_T) \in \mathbb{R}^n \times \{-1, +1\}$ be linearly separated by a unit-norm vector $\mathbf{u}$, with margin $\gamma \leq y_i \mathbf{u}^\top \hat{x}_i$, $|\hat{x}_i| = 1$, for all $t$. Then one can show that the second-order Perceptron algorithm achieves the following bound on the number of mistakes [CBCG05, Ge04]:

$$m_{SO\text{Perc.}} \leq a + \sum_{i=1}^n \log(1 + \lambda_i/a),$$

where $\lambda_i$ are the (nonzero) eigenvalues of the Gram matrix $[\hat{x}_j^\top \hat{x}_k]_{j,k \in \mathcal{M}}$ made up of inner products among (normalized) vectors $\hat{x}_i$ corresponding to mistaken trials (recall that $\mathcal{M} \subseteq \{1, ..., T\}$ denotes the set of mistaken trials). A more complicated bound (involving the deviation $D_r(.)$) holds in the nonseparable case, and is omitted from this tutorial.

The above bound suggests that the behavior of the second-order algorithm is affected by the margin (the smaller the margin the harder the problem), but also by the scattering of the data, as expressed by the aforementioned eigenvalues. The figure in Slide 22 shows a scattering of data which the second-order Perceptron algorithm can take advantage of. Here all instance vectors lie on a flat ellipsoid enclosed in an origin-centered ball of radius 1. The examples are linearly separable with margin $\gamma > 0$ via a hyperplane whose normal vector $\mathbf{u}$ is aligned with the small axis of the ellipse. Thus for any instance vector $\hat{x}_i$ the projection $|\mathbf{u}^\top \hat{x}_i|$ onto $\mathbf{u}$ is “small” (though not smaller than $\gamma$). This does not make any difference for the standard Perceptron algorithm, whose worst-case behavior is essentially ruled by the instances lying in the two dotted circles (recall the figure in Slide 20).

It can be shown that, up to the choice of $a$, the above bound is always sharper than the corresponding bound $m_{\text{Perc.}} \leq 1/\gamma^2$ for the first-order algorithm.

From a technical point of view, the three bounds we have seen so far about linear-threshold learning (HA, first-order Perceptron, second-order Perceptron) are all pointwise bounds. These bounds deliver a quantitative measure of the algorithms’ prediction performance on a sequence of examples $S$, depending on specific geometrical properties of $S$. 


We now show that the first-order and the second-order Perceptron algorithms can be equivalently formulated in dual variables. This formulation allows us to run the algorithms efficiently in any given reproducing kernel Hilbert space. As a consequence, we are able to derive a kernel version of the bounds in Slides 19, 20, 22.

We recall that a kernel function (see, e.g., [CST00, SS02, Va98]) is a nonnegative function \( K : \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R} \) satisfying
\[
\sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j K(x_i, x_j) \geq 0,
\]
for all \( \alpha_1, \ldots, \alpha_m \in \mathbb{R}, x_1, \ldots, x_m \in \mathbb{R}^n \), and \( m \in \mathbb{N} \) (such functions are also called positive definite). Given a kernel \( K \), we can define the linear space
\[
H_K = \left\{ f(\cdot) = \sum_{i=1}^{m} \alpha_i K(x_i, \cdot) : \alpha_i \in \mathbb{R}, x_i \in \mathbb{R}^n, i = 1, \ldots, m, m \in \mathbb{N} \right\}
\]
with norm defined by
\[
\|f\|_K = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j K(x_i, x_j)}.
\]
If this space is completed, by adding all limit points of sequences \( f_1, f_2, \ldots \in H_K \) that are convergent in the norm \( \|f\|_K \), the resulting space is called reproducing kernel Hilbert space (induced by the kernel \( K \)). Classical examples of kernel functions include the so-called polynomial kernel \( K(x, y) = (1 + x^T y)^d \), where \( d \) is a positive integer, and the Gaussian kernel \( K(x, y) = \exp(-||x - y||^2/2\sigma^2) \), \( \sigma > 0 \). Kernels can actually be defined over several discrete structures, such as strings and trees. See, e.g., [CST04] for an introductory exposition on this subject.

In practice, any algorithm depending on the instance vectors only through inner products \( x_i^T x_j \) can be turned into a more general kernel version just by replacing throughout standard inner products \( x_i^T x_j \) by kernel inner products \( K(x_i, x_j) \).

The (first-order) Perceptron algorithm is trivially expressed in terms of inner products of instance vectors \( x_i \). In fact, at any time \( t \) the current weight vector \( w_t \) computed by the algorithm is a linear combination of past instances corresponding to mistaken trials:
\[
w_t = \sum_{j \in \mathcal{M}_t} y_j x_j,
\]
where \( \mathcal{M}_t \subseteq \{1, \ldots, t-1\} \) is the set of mistaken trials up to time \( t-1 \). Since the algorithm’s prediction at time \( t \) is
\[
y_t = \text{SGN}(w_t^T x) = \text{SGN}\left(\sum_{j \in \mathcal{M}_t} y_j x_j^T x_t\right),
\]
we conclude that the current hypothesis computed by the Perceptron algorithm is a linear combination of inner products. Thus, in order to run the algorithm with kernels, the only thing we have to do is to maintain a pool of current “support vectors” \( \mathcal{M}_t \) aiming at
representing (in dual form) the current hypothesis

$$\hat{y}_i(x) = \text{SGN} \left( \sum_{j \in M_i} y_j K(x_j, x) \right)$$

produced by the Perceptron algorithm.

Experiments with the kernel Perceptron algorithm on real-world problems (such as Optical Character Recognition) have been reported in, e.g., [FS98, Ge01].
Slide 24

As far as mistake bounds are concerned, we can directly exploit the argument made in Slides 19-20.

Given an arbitrary sequence of examples $S = (x_1, y_1), ..., (x_T, y_T) \in \mathbb{R}^n \times \{-1, +1\}$, the following mistake bound holds for the kernel Perceptron algorithm:

$$m_{K_{\text{Perc.}}} \leq \inf_{\gamma > 0, f \in H_K, \|f\|_K = 1} \left( D_\gamma(f; S) + \sqrt{\sum_{t \in M} K(x_t, x_t)} \right),$$

where

$$D_\gamma(f; S) = \sum_{t \in M} \max\{0, 1 - y_t f(x_t)/\gamma\},$$

$M$ is the set of mistaken trials, and the minimization is over the free parameter $\gamma > 0$ and all possible functions $f \in H_K$ such that $\|f\|_K = 1$.

Compared to the standard formulation in Slides 19-20, the inner product $u^\top x_t$ is replaced here by $f(x_t)$, for some $f \in H_K$, such that $\|f\|_K = 1$. This is because we can represent the comparison class $\{u\}$ as the set of all possible (normalized) linear combinations $\sum_{t=1}^T \alpha_t x_t$, $\alpha_t \in \mathbb{R}$, of instance vectors in the sequence $S$.

Also $\|x_t\|^2 = x_t^\top x_t$ is replaced by its kernel version $K(x_t, x_t)$.

Again, if $S$ is separable we can prove a simpler bound. We say that $S$ is separable with margin $\gamma > 0$ in $H_K$ if there exists $f \in H_K$ such that $y_t f(x_t) \geq \gamma$, for all $t$, $\|f\|_K = 1$. Then the above bound implies the kernel Perceptron’s mistake bound

$$m_{K_{\text{Perc.}}} \leq \frac{R^2}{\gamma^2},$$

where $K(x_t, x_t) \leq R^2$ for all $t$.

Again, these are pointwise bounds depending on specific geometric properties of the sequence of examples $S$. Clearly enough, these properties are also determined by the choice of the kernel function $K$.

---

5 Basically, there is no loss of generality in restricting to linear combinations of vectors occurring in $S$. 26
Like the standard first-order algorithm, the second-order Perceptron algorithm can be run with kernels. Again, it suffices to maintain a pool of “support vectors” corresponding to mistaken trials, and show that the prediction rule computed by the second-order algorithm depends on data only through inner products between those vectors. Let $\mathcal{M}_t \subseteq \{1, \ldots, t-1\}$ be the set of mistaken trials up to time $t-1$, and $\mathbf{y}_t$ be the column vector of elements $y_i$, for $i \in \mathcal{M}_t$. Also, let $[\tilde{G}(\mathbf{x}_i, \mathbf{x}_j)]_{i,j \in \mathcal{M}_t}$ be the normalized$^6$ Gram matrix built from vectors $\mathbf{x}_i$, for $i \in \mathcal{M}_t$, i.e.,

$$\tilde{G}(\mathbf{x}_i, \mathbf{x}_j) = \frac{\mathbf{x}_i^\top \mathbf{x}_j}{||\mathbf{x}_i|| \cdot ||\mathbf{x}_j||},$$

and $\mathbf{v}_t$ be the column vector whose components are $\tilde{G}(\mathbf{x}_i, \mathbf{x}_i)$, for $i \in \mathcal{M}_t$. Then, exploiting a simple duality relationship about matrix inversion, one can show that the algorithm’s prediction at time $t$ is given by

$$\hat{y}_t = \text{SGN} \left( \mathbf{y}_t^\top \left( \alpha \mathbf{I} + [\tilde{G}(\mathbf{x}_i, \mathbf{x}_j)]_{i,j \in \mathcal{M}_t} \right)^{-1} \mathbf{v}_t \right).$$

Thus, in order to run the algorithm with kernels we just need to replace throughout $\tilde{G}(\mathbf{x}_i, \mathbf{x}_j)$ by

$$\hat{K}(\mathbf{x}_i, \mathbf{x}_j) = \frac{K(\mathbf{x}_i, \mathbf{x}_j)}{\sqrt{K(\mathbf{x}_i, \mathbf{x}_i) K(\mathbf{x}_j, \mathbf{x}_j)}}.$$

---

$^6$Recall that in Slide 21 we assumed the second-order Perceptron algorithm only observes normalized instances.
As for Slide 24, we can turn a primal formulation bound into a dual formulation bound. Again, for simplicity of exposition, we restrict to the separable case.

Given a sequence of examples $S = (x_1, y_1), ..., (x_T, y_T) \in \mathbb{R}^n \times \{-1, +1\}$, and a kernel function $K$, we say that $S$ is separable in $H_K$ with margin $\gamma > 0$ if there exists a function $f(\cdot) = \sum_{i=1}^T \alpha_i \hat{K}(x_i, \cdot)$, $\alpha_i \in \mathbb{R}$ such that $y_t f(x_t) \geq \gamma$, for any $t$, and $\|f\|_K = 1$. Under these hypotheses, the following mistake bound holds for the kernel second-order Perceptron algorithm:

$$m_{KSOPer} \leq \frac{a + \sum_i \log(1 + \lambda_i/a)}{\gamma},$$

where $\lambda_i$ are the nonzero eigenvalues of the (normalized) kernel Gram matrix $[\hat{K}(x_i, x_j)]_{i,j \in \mathcal{M}}$, and $\mathcal{M}$ is the set of mistaken trials.

Note that the number of nonzero eigenvalues of the kernel Gram matrix is, in general, equal to $|\mathcal{M}|$, since the very nature of kernel functions makes the dimension of the space $H_K$ be very large, possibly infinite (hence the kernel Gram matrix is likely to be full rank).

Again, the above is a pointwise bound depending on spectral properties of the training sequence, as induced by the chosen kernel function.
It is worth making a few remarks on the practical implementation of the second-order Perceptron algorithm. These remarks allow us to investigate some key computational aspects about this algorithm.

If we formulate the algorithm as in Slide 21 (primal formulation), we see that the most relevant computational effort has to be devoted to calculating the inverse matrix \((aI + S_t)^{-1}\). Since the algorithm is on-line, this calculation is done in an incremental way. Thus, in the generic trial \(t\), the computation of \((aI + S_t)^{-1}\) is based on the (already available) inverse matrix \((aI + S_{t-1})^{-1}\). Using a well-known rank-one adjustment formula (the so-called Sherman-Morrison formula, e.g., [HJ85]), it is easy to see that the extra time per trial is actually \(O(n^2)\), being \(n\) the dimension of the instance space.

On the other hand, if we formulate the algorithm as in Slide 25 (dual formulation), we see that when computing the inverse matrix \(\left(aI + [\hat{K}(x_i, x_j)]_{i,j \in \mathcal{M}_t}\right)^{-1}\) we can rely on the previous inverse \(\left(aI + [\hat{K}(x_i, x_j)]_{i,j \in \mathcal{M}_{t-1}}\right)^{-1}\). Using a known adjustment formula for partitioned matrices (e.g., [HJ85]) we conclude that the number of extra inner products (or kernel evaluations) to be computed in trial \(t\) is \(O(|\mathcal{M}_t|^2)\), i.e., quadratic in the number of mistakes (number of “support vectors”) made so far. The interested reader is referred to [CBCG05] for details.

The sparsity of the solution produced by the second-order Perceptron algorithm is reflected by the magnitude of the eigenvalues of the (kernel) Gram matrix \([\hat{K}(x_i, x_j)]_{i,j \in \mathcal{M}}\) (the smaller \(|\mathcal{M}|\) the smaller the eigenvalues, i.e., the higher the prediction performance), but is also affecting in a fairly direct way the overall running time of the algorithm (the smaller \(|\mathcal{M}|\) the smaller the running time).

It is worth observing that, unlike other sparsification methods applied to kernel Gram matrices (e.g., [FiSh01, BJ02, EMM03, RYP03]), which operate a sort of post-processing of the hypothesis computed by the algorithm (independent of the labels), here the sparsification process is driven by the algorithm itself, and can be viewed as a by-product of its inference mechanism.
Figure 2: Rotation invariance: blue crosses represent positive (+1) examples, red dashes are negative (−1) examples. The black separating line is roughly the maximal margin separator for those examples (as computed by, say, an SVM). If all instance vectors in the upper figure are rotated by some angle (resulting in the data scattering displayed in the lower figure), the maximal margin separator gets rotated the same way.

Slide 28

The algorithms we have seen so far for learning linear-threshold functions (HA, first-order Perceptron and second-order Perceptron) enjoy a neat geometrical property which we call \textit{rotation invariance}. A rotation invariant algorithm is one depending on the data instances only through inner products (or angles among instance vectors). The rotation invariance property, illustrated in Figure 2, can be explained as follows.

Given a training sequence \( S = (\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_T, y_T) \), we run a rotation invariance algorithm \( A \) on \( S \), obtaining (at the end of the \( T \)-th trial) the final hypothesis \( h_{A,S}(\mathbf{x}) = \text{SGN}(\mathbf{w}^\top \mathbf{x}) \in \{-1, +1\} \). Now, let \( R \) be a rotation matrix\(^7\) and \( S' = (R\mathbf{x}_1, y_1), \ldots, (R\mathbf{x}_T, y_T) \) be the \( R \)-rotated sequence derived from \( S \). If we run \( A \) on \( S' \), the final hypothesis we get is just \( h_{A,S'}(\mathbf{x}) = \text{SGN}((R\mathbf{w})^\top \mathbf{x}) \). See Figure 2. Since \((R\mathbf{w})^\top (R\mathbf{x}_t) = \mathbf{w}^\top \mathbf{x}_t\) we have that \( h_{A,S'}(R\mathbf{x}_t) = h_{A,S}(\mathbf{x}_t) \) for any \( t \).

Broadly speaking, a rotation invariant algorithm relies on linear algebra (the math involved is about linear transformations, eigenvalues of matrices and so forth). Every algorithm depending on the data only through inner products is rotation invariant: Support Vector Machines and Principal Component Analysis are further notable examples of rotation invariance.

\(^7\)Recall that an \( n \times n \) rotation matrix \( R \) satisfies \( R^\top R = I \).
variant methods. As such, these kinds of algorithms can be “kernelized” fairly easily: it suffices to replace every inner product by a kernel inner product.\footnote{The reader is urged to verify that even the version of HA presented in Slides 15–16 is a rotation invariant algorithm.} Intuitively speaking, a rotation invariant algorithm does not have any special preference when searching for a separating hyperplane: all target vectors are equally likely. Thus, as an intuitive corollary, a rotation invariant algorithm cannot have any built-in feature selection mechanism when learning linear-threshold functions, since axes-parallel directions are not preferred over any other.
The purpose of this and the next few slides is to introduce the reader to non-rotation invariant algorithms. These algorithms are sometimes called nonadditive, as opposed to the additive algorithms. This terminology is borrowed from [KWA97], where a nice contrasting discussion about additive and nonadditive algorithms in the context of on-line learning is carried out.

Broady speaking, an additive algorithm is an incremental algorithm with an additive update rule (such as first- and second-order Perceptron weight update rules). Incremental additive algorithms are typical examples of rotation invariant algorithms.

A nonadditive algorithm keeps the current hypothesis in the form of a weight vector, but it performs a nonadditive weight update. A nonadditive algorithm does not rely on linear algebra, is not rotation invariant, and is likely to be hard\textsuperscript{9} to kernelize. Besides, most of the nonadditive algorithms proposed in the on-line learning literature have a bias for “sparse” separating hyperplanes, i.e., for hyperplanes having many null components. This makes them preferable when the standard label prediction task has to be combined with the feature selection task.

Among the nonadditive algorithms proposed in the on-line literature are to be mentioned the Winnow algorithm [L88, L91, AW98], the Zero-threshold Winnow algorithm [L89a, GLS01, CBL03], the Binary Exponentiated algorithm [By97], the $p$-norm algorithms (with $p > 2$) [GLS01, GL99, Ge01], the maximum entropy discrimination algorithm analyzed in [LoWu04].

Among the additive algorithms, beyond first and second-order Perceptron, are ROMMA [LL02] and Alma\textsubscript{2} [Ge01].

In this tutorial, our paradigmatic example of nonadditive algorithm is the so-called $p$-norm (Perceptron) algorithm. The $p$-norm algorithm is a simple generalization of the (first-order) Perceptron algorithm, from which it inherits flexibility and robustness.

\textsuperscript{9}By this we just mean that the kernelized version of a nonadditive algorithm might still be formulated, but it need not result in an efficient algorithm.
A $p$-norm Perceptron operates like a standard Perceptron (Slide 18), but for the fact that just before prediction the weight vector $w_t$ is mapped through a nonlinear transformation $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$. This transformation is a simple componentwise polynomial mapping:

$$f(w) = \begin{bmatrix} SGN(w_1)|w_1|^{p-1} \\ SGN(w_2)|w_2|^{p-1} \\ \vdots \\ SGN(w_n)|w_n|^{p-1} \end{bmatrix},$$

where $w = (w_1, ..., w_n)$. The norm value $p$ is a real-valued parameter of the algorithm, and is assumed to be at least 2. This parameter rules the actual behavior of the algorithm in terms of additivity and feature selection ability. Note that $p = 2$ makes $f$ be the identity mapping (thus the 2-norm Perceptron is actually the standard Perceptron). The larger is $p$ the closer a $p$-norm algorithm gets to known multiplicative algorithms\textsuperscript{10} such as Winnow [L88]. The most notable characteristic multiplicative algorithms is that their cumulative loss grows only logarithmically with the number of irrelevant components of the target weight vector. This "logarithmic behavior" can be obtained with the $p$-norm algorithms by setting the norm value $p$ to be an appropriate constant multiple of $\log n$. The $p$ parameter is then used to smoothly interpolate between the two meaningful extreme values $p = 2$ and $p = O(\log n)$.

A simple "analytical" intuition about why a $p$-norm algorithm tends to perform feature selection derives from the observation that when $p > 2$ this algorithm tends to produce weight vectors whose components are widely different in magnitude. Set for instance $p = 4$. Then the transformation $f(\cdot)$ takes the simple form

$$f(w) = (w_1^3, w_2^3, ..., w_n^3)^\top.$$

Now, assume at some point the $w$-additive rule "$w_{t+1} = w_t + y_t x_t$" has produced a weight vector whose first component has magnitude significantly larger than 1, and whose second component has magnitude smaller than 1. The mapping $f$ greatly emphasizes this difference in magnitude: the first component of $f(w)$ will be much larger than 1, while the second component will be close to zero. Since the algorithm’s prediction rule "$\hat{y}_t = SGN(f(w)^\top x_t)$" just amounts to compare the inner product $f(w)^\top x_t$ to a zero threshold, what really matters to the algorithm is the relative magnitude of the components of $f(w)$. The final weight vector computed by a 4-norm Perceptron algorithm on a sequence of $T$ examples is $f(w_{T+1})$. Since this weight vector tends to have many very small components, we obtain a linear-threshold prediction where many features might be disregarded.

\textsuperscript{10}A multiplicative on-line algorithm is an algorithm with a multiplicative update rule.
Slide 31

The mistake bound one can prove for the $p$-norm Perceptron algorithm is a generalization of the one we have proven in Slides 19 and 20. This bound involves a pair of dual norms $p$ (the norm parameter of the algorithm) and $q = \frac{p}{p-1} \in [1, 2]$. The norm $p$ will be used to measure the length of instance vectors $\mathbf{x}_i$, while $q$ will be used for the length of target vectors $\mathbf{u}$.

Let $S = (\mathbf{x}_1, y_1), ..., (\mathbf{x}_T, y_T) \in \mathbb{R}^n \times \{-1, +1\}$ be an arbitrary sequence of examples. Then, recalling the notation in Slide 19, one can show that the number of mistakes made by the $p$-norm Perceptron algorithm on $S$ is bounded by [GLS01, GL99, Ge03]

\[
m_{p\text{Perce}} \leq \inf_{\gamma > 0, \|\mathbf{u}\|_q = 1} \left( D_\gamma(\mathbf{u}; S) + \frac{\sqrt{1} \sum_{i \in M} (p - 1)\|\mathbf{x}_i\|_p^2}{\gamma} \right).
\]

The reason why these kinds of algorithms are advantageous over a rotation invariant algorithm when learning sparse target vectors is better seen in the linearly separable case (next slide).
Slide 32

As for Slide 20, we can prove a bound in explicit form in the case when the loss of the comparison class is zero. Since we are dealing with dual norms, we need to consider a consistent notion of margin. We say that a sequence of examples \( S = (x_1, y_1), \ldots, (x_T, y_T) \) is linearly separable with \( q \)-norm margin \( \gamma > 0 \) when there exists a weight vector \( \mathbf{u} \) with \( \| \mathbf{u} \|_q = 1 \) such that for any \( t \) we have \( y_t \mathbf{u}^T x_t \geq \gamma \). If we also assume \( \| x_t \|_p \leq R \), we see that the bound of the previous slide implies

\[
m_{\text{perc}} \leq \left( p - 1 \right) \frac{R^2}{\gamma^2}.
\]

Therefore, this bound depends on the \( p \)-norm length of the instance vectors in \( S \) and the \( q \)-norm margin of \( S \).

To see why the above bound is sharper than the standard Perceptron bound given in Slide 20, consider the following simple (and extreme) situation. Suppose \( \mathbf{u} \) has just one nonzero component (out of \( n \)) of value 1, and \( x_t \) belongs to the Boolean hypercube \( \{-1, +1\}^n \). Notice that this is the learning scenario we sketched at the end of Slide 14, which is just the binary expert framework. In terms of feature selection, we are in a (rather extreme) situation where the target vector (viewed as a linear-threshold function) has only one relevant component. If the training sequence is linearly separable, the margin \( \gamma \) (either the standard 2-norm margin or the \( q \)-norm margin, with \( q < 2 \)) has to be at least 1. Let us compare the standard Perceptron mistake bound with the one achieved by a \( p \)-norm algorithm with \( p = 2 \log n \). We have

\[
m_{\text{perc}} \leq \frac{\| x_t \|^2}{\gamma^2} \leq n,
\]

and

\[
m_{\text{perc}} \leq \left( p - 1 \right) \frac{\| x_t \|^2}{\gamma^2} \leq (p - 1)n^{2/p} = e(2\log n - 1) = O(\log n),
\]

where \( e \) is the base of natural logarithms. Thus when the target vector \( \mathbf{u} \) is very sparse (many null components) and \( p = O(\log n) \), we recover the “logarithmic” behavior exhibited by the Halving Algorithm and the Weighted Majority algorithm described in Slides 6–14. Notice that on the same task the standard Perceptron\(^{11}\) has a linear dependence on the number of irrelevant components. This exponential separation (“logarithmic” vs. “linear”) between the two kinds of algorithms (“additive” vs. “nonadditive/multiplicative”) is the main reason why non-rotation invariant algorithms are interesting.

The \( p \)-norm Perceptron bound is again a pointwise mistake bound, relating the prediction ability of the algorithm to specific properties of the sequence to predict on. We have just seen that these properties are not the same as those a rotation invariant algorithm relies on.

\(^{11}\) We are viewing here the standard Perceptron algorithm as a prototypical member of the rotation invariant family.
Slides 33-35

These three slides are aimed at showing how to run a $p$-norm Perceptron algorithm with kernels. The slides are tagged “wild” since they refer to unpublished and (possibly) more advanced material, which we will not comment on.
Consider a generic \textit{batch} classification algorithm \( A \) operating on a training set \( S = (\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_T, y_T) \in \mathbb{R}^n \times \{-1, +1\} \), such as Naive Bayes or Support Vector Machines.

We can clearly associate with \( A \) a mistake bound by running \( A \) in an on-line fashion: In the generic trial \( t \) we train \( A \) on the training set \( S_{t-1} = (\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_{t-1}, y_{t-1}) \). This gives us a (binary) classifier \( h_t(\mathbf{x}) = A_{S_{t-1}}(\mathbf{x}) \). We say that this classifier has made a prediction mistake in trial \( t \) if \( h_t(\mathbf{x}_t) \neq y_t \). In trial \( t+1 \) we re-train \( A \) (from scratch) on the augmented training set \( S_t \), and keep going.

At the end of trial \( T \) we count the number of mistakes that \( A \) has made on \( S \). This will provide us with another pointwise relationship between the prediction ability of the algorithm and some specific characteristics of the training sequence.
The methods we have seen so far work in the case when a single target (expert/vector) performs well on the whole sequence of examples $S$. In this and the next few slides we would like to model nonstationary targets/problems (i.e., targets that change with time).

Consider, for simplicity, the expert framework. The sequence of examples $S$ is partitioned into $k + 1$ arbitrary segments. Each segment has an associated expert. We call a partition a sequence of segments with the associated sequence of experts. For instance, in the figure reported in Slide 37 we have 5 experts and 6 segments ($k = 5$). We also define the loss of the partition over a sequence of examples $S$ as the sum of the total losses of the experts associated with each segment of the partition. The best partition of size $k$ over $S$ is the partition with $k + 1$ segments having the smallest loss.

The way we model nonstationary targets is to modify our original goal about relative loss bounds. Instead of devising a prediction algorithm which is good relative to the best (fixed) expert in hindsight over the sequence $S$, we would like our algorithm do almost as good as the best partition of size $k$. This modified goal takes into account the fact that different experts might predict better on different segments of the sequence $S$, thereby modeling real-world scenarios where the underlying “structure” of the problem changes during learning. When we have $T$ trials, $k + 1$ segments and $n$ experts, there are $(T-1) \choose k$ different partitions. In principle, we could get a good bound for this problem just by using the standard Weighted Majority Algorithm (WMA): we expand the $n$ experts into $(T-1) \choose k$ ”partition experts”. Each partition expert represent a single partition over the trial sequence, and predicts on each trial as the expert associated with the segment containing that trial. For instance, in the figure of Slide 37, the depicted partition expert predicts with $E_1$ up to trial $t_1$, with $E_4$ from trial $t_1 + 1$ to trial $t_2$, and so on. Thus, using $(T-1) \choose k$ $n(n-1)^k$ experts aggregated through WMA we would immediately obtain a relative loss bound of the form

$$L_{WMA}(S) \leq a \min_{P(k)} L_P(S) + b(k \log n + k \log(T/k)), \quad (1)$$

where the minimum is over all partitions $P(k)$ of size $k$, and $a, b$ are positive constants. However, the above algorithm is inefficient, since it has to maintain as many weights as the the number of partitions, which is exponential in the size $k$.

Moving to the more general problem of learning linear-threshold functions, if we are given a loss function $Loss$, a distance function $dist$ and a sequence of examples $S$, our goal is to devise a prediction algorithm $A$ for which we can bound from above

$$L_A(S) = \min_{u_1, \ldots, u_T} \left( Loss_{u_1, \ldots, u_T}(S) + \sum_{t=1}^{T} dist(u_t, u_{t+1}) \right), \quad (2)$$

where we interpret $Loss_{u_1, \ldots, u_T}(S) + \sum_{t=1}^{T} dist(u_t, u_{t+1})$ as the cumulative loss of the shifting target $u_t$ over $S$. 

38
Slide 38

As usual, if the $u_i$ are chosen among the $n$ unit vectors, $x_i$ lie in $[-1, +1]^n$, the loss function is

$$Loss_{u_1, u_2, \ldots, u_T}(S) = \frac{1}{2} \sum_{t=1}^{T} |u_i^T x_t - y_t|,$$

and

$$dist(u_t, u_{t+1}) = ||u_t - u_{t+1}||,$$

for some norm $|| \cdot ||$, we essentially recover the expert framework of the previous slide. Notice that

$$\sum_{t=1}^{T} dist(u_t, u_{t+1})$$

counts the number of target shifts in the partition (number of segments minus one), while

$$\frac{1}{2} \sum_{t=1}^{T} |u_i^T x_t - y_t|$$

is the loss of the partition determined by $u_1, u_2, \ldots, u_T$.

In the more general case of linear-threshold functions we let $Loss_{u_1, u_2, \ldots, u_T}(S)$ be the cumulative hinge loss at margin $\gamma$

$$Loss_{u_1, u_2, \ldots, u_T}(S) = \sum_{t=1}^{T} \max\{0, 1 - y_i u_i^T x_t / \gamma\},$$

while

$$dist(u_t, u_{t+1}) = ||u_t - u_{t+1}||$$

measures the distance between consecutive vectors $u_t$ and $u_{t+1}$, for some norm $|| \cdot ||$. 

39
As far as algorithms are concerned, we start by the approach taken in [LW94, HW98, AW98] (see also [Vo97]). We modify WMA by adding an additional update step. Since this new model allows the best expert shift over a series of trials, we cannot simply use exponential weights of the form $\beta^{L_i,t}$. Indeed, before an expert is best in a given segment, its loss in previous segments might be arbitrarily large, hence its weight might be arbitrarily close to zero at the beginning of that segment. This suggests to modify the multiplicative update rule by letting small weights to be recovered quickly.

Thus, after the multiplicative update, each expert "shares" a portion of its weight with the other experts, in the sense that a (fixed) fraction of each experts’ weight is added to the weight of each other expert. This additional "share update" guarantees that the ratio of the weight of any expert to the total weight of all experts can be bounded from below. In other words, the share update guarantees that the relative importance of the experts (as measured by the corresponding weights) is suitably balanced.
When learning linear threshold functions, we basically do a similar thing for similar reasons: we keep the magnitude of the current weight vector components suitably balanced by projecting onto a convex set (such as the unit ball).

To fix ideas, consider the standard (first-order) Perceptron algorithm. Let $t$ be a mistaken trial. After the usual additive update $\mathbf{w}'_t = \mathbf{w}_t + \eta y_t \mathbf{x}_t$, we check whether the resulting vector $\mathbf{w}'_t$ lies in the unit ball $\{\mathbf{w} \in \mathbb{R}^n : ||\mathbf{w}||_2 \leq 1\}$. If so, we leave it unchanged and use $\mathbf{w}'_t$ in the next trial prediction. Otherwise, we project $\mathbf{w}'_t$ onto the unit ball through a simple normalization: $\mathbf{w}_{t+1} = \mathbf{w}'_t/||\mathbf{w}'_t||$, yielding the new weight vector which will be used in trial $t + 1$.

**Remark:** The introduction of a learning rate $\eta$ here is due to technical reasons. In fact, the projection step introduces scaling factors that force the weight vector $\mathbf{w}_t$ be updated appropriately.

We refer the reader to [AW98, HW01, KSW04], for a more thorough algorithmic treatment.
The mistake bounds one can prove for the two algorithms are have mentioned are the following.

The fixed share algorithm working in the expert framework essentially achieves the kind of relative loss bound given in (1). Observe that the excess loss $k \log(T/K) + k \log n$ is basically the logarithm of the number of partition experts or, equivalently, the number of bits needed to encode these experts. In fact, we need roughly $k \log(T/K)$ bits to encode the $k$ trials when shifts occur (out of a total of $T$ trials), and roughly $k \log n$ bits to encode the experts active at each segment. Notice that we do this via an algorithm working in linear time in the number $n$ of experts.

The projection-based Perceptron algorithm achieves a mistake bound which is essentially a generalization of the standard Perceptron bound. Let us restrict, for the sake of simplicity to the case when the sequence of examples $S$ can be linearly separated with margin at least $\gamma$ by each vector of the comparison sequence $u_1, u_2, \ldots, u_T$. Then, letting the learning rate $\eta$ scale with $\gamma$ as $\eta = \frac{\gamma}{2R^2}$ one can easily prove the following shifting target bound on the number of mistakes:

$$m_{pr-basedPerc} \leq \frac{4R^2}{\gamma^2} \left( \frac{1}{2} + \sum_{t=1}^{T} ||u_{t+1} - u_t|| \right).$$

Once again, the above is a pointwise bound since it fully depends on the data. Notice that the vectors $u_1, \ldots, u_T$ occurring in the bound are data-dependent quantities, since they are actually chosen so as to minimize $\sum_{t=1}^{T} ||u_{t+1} - u_t||$ among those vectors that achieve margin at least $\gamma$ on $S$.

It is worth mentioning that the above bound can be extended to cover:

- The non-separable case
- The $p$-norm case
- The kernel case
- The second-order case

and combinations of the above.

To learn more about shifting targets we refer the reader to more recent work, such as [BW02, KSW04].
We now consider learning binary classification tasks with partially labelled data via selective sampling. A selective sampling algorithm (e.g., [ACL90, FSST97, CCG03] and references therein) is an on-line learning algorithm that receives a sequence of unlabelled instances, and decides whether or not to query the label of the current instance based on instances and labels observed so far. The idea is to let the algorithm decide which labels are most useful to its inference mechanism, so that redundant examples can be discarded on the fly and labels can be saved. Therefore we call this kind of inference a label-efficient prediction.

The overall goal of label-efficient prediction is to fit real-world scenarios where labels are scarce or expensive. For instance, in a web-searching task, collecting web pages is a fairly automated process, but assigning them a label (a set of topics) often requires time-consuming and costly human expertise. In these cases, it is clearly important to devise learning algorithms having the ability to exploit the label information as much as possible. Furthermore, when we consider kernel-based algorithms, saving labels directly implies saving support vectors in the currently built hypothesis, which, in turn, implies saving running time in both training and test phases, thanks to the "sparsity" of the classifier.

Many algorithms have been proposed in the literature to cope with the broad task of learning with partially labelled data, working under both probabilistic and worst-case assumptions, for either on-line or batch settings. These range from active learning algorithms [CCS00, TK00], to the query-by-committee algorithm [FSST97], to the adversarial “apple tasting” and label-efficient algorithms investigated in [HLL00] and [HP97, CLS04], respectively.

In this tutorial, we follow the approach in [CGZ04]. We turn Perceptron-like algorithms working under worst-case assumptions on the data sequence into label-efficient algorithms whose goal is to achieve the best bound on the number of prediction mistakes with as few queried labels as possible.

We need to revise the on-line learning protocol as follows. In the generic trial $t$ the algorithm receives instance $\mathbf{x}_t \in \mathbb{R}^n$, outputs a prediction $\hat{y}_t \in \{-1, +1\}$ about the unknown label $y_t \in \{-1, +1\}$ associated with $\mathbf{x}_t$, and decides whether or not to query the label $y_t$. No matter what the algorithm decides, we say that the algorithm has made a prediction mistake if $\hat{y}_t \neq y_t$. We again measure the prediction performance of the algorithm by the total number of mistakes it makes on a sequence $S$, as compared to the number of mistakes made on the same sequence by the best off-line predictor taken from a given comparison class.

As we said, in this framework we have two conflicting goals. On the one hand, we would like to make as few prediction mistakes as possible. On the other hand, we would like to use as few labels as possible. The first goal certainly benefits from querying many labels. Achieving the second goal, however, tends to produce performance degradation.

It is reasonable to expect that these two goals can be traded off against each other. Since this tutorial is about pointwise bounds and conversions to standard statistical settings, we will follow the route traced by [CLS04, CGZ04]. Our algorithms exploit a (margin-based) randomized rule to decide which labels to query: The higher the margin achieved
on instance $x_t$ the less likely the associated label $y_t$ is requested (the idea is to interpret the algorithm’s margin as a measure of classification confidence). This allows us to prove (in expectation) essentially the same mistake bounds as those achieved by standard (full information) algorithms, but using fewer labels.
As a simple example, we present a label-efficient version of the classical Perceptron algorithm. The algorithm has a real parameter $b > 0$ which might be viewed as a noise parameter, ruling the extent to which a linear threshold model fits the data at hand. The algorithm maintains a vector $\mathbf{w} \in \mathbb{R}^n$ (whose initial value is zero). In each trial $t$ the algorithm observes an instance vector $\mathbf{x}_t \in \mathbb{R}^n$ and predicts the binary label $y_t$ through the sign of the margin $r_t = \mathbf{w}_t^\top \mathbf{x}_t$. Then the algorithm decides whether to ask for the label $y_t$ through a simple randomized rule: a coin with bias $b/(b + |r_t|)$ is flipped; if the coin turns up heads ($Z_t = 1$ in Slide 43) then the label $y_t$ is revealed. Moreover, on a prediction mistake ($\hat{y}_t \neq y_t$) the algorithm updates vector $\mathbf{w}_t$ according to the usual Perceptron additive rule. On the other hand, if either the coin turns up tails or $\hat{y}_t = y_t$ no update takes place.
Slide 44

As we said, the bounds one can prove for the label-efficient Perceptron algorithm are essentially the same as those for its deterministic counterpart. Below we give the bound for the linearly separable case. Again, this theory can be easily generalized to cope with:

- The non-separable case
- the $p$-norm case
- the kernel case
- the second-order case
- the shifting target case,

and combinations thereof.

If $S = (x_1, y_1), \ldots, (x_T, y_T)$ is a sequence of training examples which are linearly separated with margin $\gamma > 0$ by a unit-norm hyperplane, then the expected number of labels requested by the algorithm is equal to

$$\sum_{t=1}^{T} E \left[ \frac{b}{b + |r_t|} \right]$$

(notice that this equality trivially follows from the algorithm’s query strategy). Moreover, if $||x_i|| \leq R$, setting $b = R^2/2$ yields the following bound on the expected number of mistakes:

$$E[m_{Label\_ef\_Perc}] \leq \frac{R^2}{\gamma^2}.$$  

Notice that this bound is the same as the standard Perceptron’s.

In both cases the expectation is w.r.t. the sequence of independent coin flips (the $Z_t$ variables), while the sequence of examples $S$ does remain ”worst-case”.
Slide 45-46

In order to give some evidence of the theoretical behavior of our label-efficient algorithms, we present some experiments we have made [CGZ04] on two medium-size text categorization tasks. These experiments confirmed our theoretical results, and show the effectiveness of our margin-based label selection rule.

The empirical evaluation was carried out on two datasets of free-text documents. The first dataset is made up of the first (in chronological order) 40,000 newswire stories from Reuters Corpus Volume 1 (RCV1) [RCV1]. The resulting set of examples was classified over 101 categories. The second dataset is a specific subtree of the OHSUMED corpus of medical abstracts [OH]: the subtree rooted in “Quality of Health Care” (MeSH code N05,712). From this subtree we randomly selected a subset of 40,000 abstracts. The resulting number of categories was 94. A standard preprocessing was performed on the two datasets.

Two kinds of experiments were made on each dataset. In the first experiment we compared label-efficient versions of the first-order and the second-order Perceptron algorithms (for different values of b), with the standard second-order Perceptron algorithm (requesting all labels). Such a comparison was devoted to studying the extent to which a reduced number of label requests might lead to performance degradation. In the second experiment, we compared variable vs. constant label-request rate. That is, we fixed a few values for parameter b, run the label-efficient version of the second-order algorithm, and computed the fraction of labels requested over the training set. Call this fraction \( \hat{p} = \hat{p}(b) \). We then run a second-order label-efficient algorithm with (constant) label request probability equal to \( \hat{p} \) (independent of \( t \)). The aim of this experiment was to investigate the effectiveness of a margin-based selective sampling criterion, as opposed to a random one.

The plots in slides 45 and 46 summarize the results we obtained on RCV1 (the results on OHSUMED turned out to be similar). For the purpose of this graphical representation, we selected the 50 most frequent categories from RCV1, those with frequency larger than 1%. The standard second-order algorithm is denoted by 2ND-ORDER-ALL-LABELS, the label-efficient algorithms are denoted by 1ST-ORDER and 2ND-ORDER, whereas the second-order algorithm with constant label request is denoted by 2ND-ORDER-FIXED-BIAS.\(^\text{12}\) The figures plot the instantaneous F-measure and the instantaneous label-request rate on the RCV1 dataset. We solved a binary classification problem for each class and then (macro)averaged the results. All curves tend to flatten after about 24,000 examples (out of 40,000). Figure (a) plots the instantaneous macroaveraged F-measure of 2ND-ORDER (for three values of \( b \)) and their corresponding label-request curves. For the very sake of comparison, we also include the F-measure of 2ND-ORDER-ALL-LABELS. Figure (b) gives a comparison among 2ND-ORDER, 1ST-ORDER and 2ND-ORDER-FIXED-BIAS. Figure (c) gives the same comparison on a specific category. Figure (d) gives the F-measure of 2ND-ORDER vs. F-measure of 2ND-ORDER-FIXED-BIAS for 5 values of parameter \( b \), after 40,000 examples.

As evinced by Figure (a), there is a range of values for parameter \( b \) that makes 2ND-ORDER achieve almost the same performance as 2ND-ORDER-ALL-LABELS, but with a sub-

\(^{12}\)We omitted to report on the first-order algorithms 1ST-ORDER-ALL-LABELS and 1ST-ORDER-FIXED-BIAS, since they are always outperformed by their corresponding second-order algorithms.
stantial reduction in the total number of queried labels.\textsuperscript{13} In Figure (b) we report the results of running 2\textsc{nd-}ORDER, 1\textsc{st-}ORDER and 2\textsc{nd-}ORDER-FIXED-\textsc{bias} after choosing values for $b$ that make the average F-measure achieved by 2\textsc{nd-}ORDER just slightly larger than those achieved by the other two algorithms. We then compared the resulting label request rates and found 2\textsc{nd-}ORDER largely best among the three algorithms (its instantaneous label rate after 40,000 examples is less than 19\%). We made similar experiments for specific categories in RCV1. On the most frequent ones (such as category 70 – Figure (c)) this behavior gets emphasized.

Finally, in Figure (d) we report a direct macroaveraged F-measure comparison between 2\textsc{nd-}ORDER and 2\textsc{nd-}ORDER-FIXED-BIAS for 5 values of $b$. On the x-axis are the resulting 5 values of the constant bias $\hat{p}(b)$. As expected, 2\textsc{nd-}ORDER outperforms 2\textsc{nd-}ORDER-FIXED-BIAS, though the difference between the two tends to shrink as $b$ (or, equivalently, $\hat{p}(b)$) gets larger.

\textsuperscript{13}Notice that the figures are plotting \textit{instantaneous} label rates, hence the overall fraction of queried labels is obtained by integration.
Slide 47

End of Part 1
We now turn to the statistical Pattern Recognition problem, and relate its algorithmic aspects to the on-line algorithms we have seen so far.

We view the (binary) Pattern Recognition problem as a problem of learning from examples. A learning algorithm for pattern recognition (or learner for short) takes in input a training set, i.e., a multiset of examples \((x_i, y_i) \in \mathcal{X} \times \{-1, +1\}\), where \(\mathcal{X}\) is some instance domain such as \(\mathcal{X} = \mathbb{R}^n\), and outputs a function \(h : \mathcal{X} \rightarrow \{-1, +1\}\), assigning a binary label to each element of \(\mathcal{X}\). We call hypothesis any function returned by a learner. Learning algorithms generate hypotheses from specific families of functions, such as linear-threshold functions, decision trees, or multivariate polynomials. We call hypothesis space (denoted by \(\mathcal{H}\)) the family from which a learning algorithm picks its hypotheses.

It is sometimes useful to allow a learning algorithm to output hypotheses of the form \(h : \mathcal{X} \rightarrow \mathcal{D}\), where \(\mathcal{D}\) is a decision space not necessarily equal to \(\{-1, +1\}\). The goodness of hypothesis \(h\) on example \((x, y)\) is measured by the quantity \(\text{loss}(y, h(x))\), where \(\text{loss} : \{-1, +1\} \times \mathcal{D} \rightarrow \mathbb{R}\) is a nonnegative and bounded loss function. For instance, in the binary classification case we are dealing with, we may take \(\mathcal{D} = [-1, 1]\), and \(\text{loss}(y, h(x)) = \frac{1}{2}|h(x) - y|\) (the absolute loss function). This is the usual 0-1 loss when \(h(x) \in \{-1, +1\}\).

From now on we consider learning algorithms within the framework of statistical pattern recognition (see, e.g., [DGL96]). In this framework all the examples \((x, y)\) are generated by independent draws from a fixed and unknown probability distribution on \(\mathcal{X} \times \{-1, +1\}\). This assumption allows us to view the training set as a statistical sample, and thus to investigate the learning problem as a problem of statistical inference. In what follows, probabilities \(\mathbb{P}\) and expectations \(\mathbb{E}\) will be understood with respect to the fixed and unknown underlying distribution according to which all examples are drawn.

In general, we would like a hypothesis \(h\) to predict well on examples drawn from the same source that generated the training set for \(h\). In statistical pattern recognition, the success of a hypothesis \(h\) is measured by the risk of \(h\), denoted by \(\text{risk}(h)\). This is the expected loss of \(h\) on an example \((X, Y)\) drawn from the underlying distribution,

\[
\text{risk}(h) = \mathbb{E}_{(Y,X)} \text{loss}(Y, h(X)).
\]

We identify a generic learner \(A\) with the random hypothesis \(\hat{H} = \hat{H}(S)\) returned by \(A\) when the training set \(S\) is fed in input. Our goal is to keep the risk of \(\hat{H}\) small on most sample realizations, that is,

\[
\mathbb{P} \left( \text{risk}(\hat{H}) \leq \inf_{h \in \mathcal{H}} \text{risk}(h) + \varepsilon \right) \geq 1 - \delta
\]

for small enough \(\varepsilon > 0\) and \(0 < \delta < 1\). (Here the probability is taken with respect to the distribution of the training sample \(S\).)
To achieve this goal, we can use the method of uniform convergence, whose study was pioneered by Vapnik and Chervonenkis [VC71] (see also [Vap89, Vap99]). Let \( \text{risk}_{\text{emp}}(h) \) be the empirical risk of \( h \) on a sample \( S = (X_1, Y_1), \ldots, (X_T, Y_T) \),

\[
\text{risk}_{\text{emp}}(h) = \frac{1}{n} \sum_{t=1}^{T} \text{loss}(Y_t, h(X_t)).
\]

Uniform convergence means that, for all probability distributions, the empirical risk of \( h \) is, with high probability, close to its true risk uniformly over all \( h \in \mathcal{H} \). In the case of a class \( \mathcal{H} \) of \( \{-1, +1\} \)-valued functions, a sufficient (and also necessary) condition for uniform convergence is that \( \mathcal{H} \) has finite VC dimension \( d \) — similar conditions apply to multi-valued or real-valued function classes and bounded losses. If this condition holds then, for each \( 0 < \delta < 1 \) and sample size \( T \),

\[
\mathbb{P} \left( \sup_{h \in \mathcal{H}} |\text{risk}_{\text{emp}}(h) - \text{risk}(h)| \geq c \sqrt{\frac{d + \ln(1/\delta)}{T}} \right) \leq \delta
\]

(3)

holds (for a proof of this result see, e.g., [Lo99]).

Uniform convergence implies that \( \mathcal{H} \) can be learned by the empirical risk minimizer, i.e., by the algorithm returning the hypothesis

\[
\hat{H} = \text{arginf}_{h \in \mathcal{H}} \text{risk}_{\text{emp}}(h).
\]

Once we have a uniform convergence result like (3), the risk analysis for \( \hat{H} \) is immediate. Let \( h^* = \text{arginf}_{h \in \mathcal{H}} \text{risk}(h) \). Then, with probability at least \( 1 - \delta \),

\[
\text{risk}((\hat{H})) \leq \text{risk}_{\text{emp}}(\hat{H}) + c \sqrt{\frac{d + \ln(2/\delta)}{T}} \leq \text{risk}_{\text{emp}}(h^*) + c \sqrt{\frac{d + \ln(2/\delta)}{T}} \leq \text{risk}(h^*) + 2c \sqrt{\frac{d + \ln(2/\delta)}{T}},
\]

where we applied (3) in the first and the last step.
A different approach to uniform convergence, pioneered in [Ba98], replaces the square-root term in (3) with the random quantity

$$C_T(S) + c \sqrt{\frac{\ln(1/\delta)}{T}},$$

where $C_T(S)$ is a sample statistic depending on $\mathcal{H}$ and $c$ is a universal constant. For example, $C_T(S)$ can be the empirical VC-entropy [BLM00, WSTSS99], the Rademacher complexity [BM02], or the maximum discrepancy [BBL01] of the class $\mathcal{H}$. In general, this approach is advantageous when the mean of $C_T(S)$ is significantly smaller than $\sqrt{d/T}$ and when large deviations of $C_T(S)$ are unlikely. In these cases such “data-dependent” uniform convergence bounds are stronger than those based on the VC dimension since, with high probability, we have

$$C_T(S) \approx \mathbb{E}C_T(S) \ll \sqrt{d/T}.$$  

A data-dependent bound relates the prediction performance of a learner to the data sequence at hand via specific functions of this sequence. As such, data-dependent bounds are practically useful (e.g., for rigorous error reporting) whenever the empirical quantity $C_T(S)$ is easy to compute from $S$. However, this need not be the case for the random quantities proposed in the literature about uniform convergence.
In some cases, the statistic $C_T(S)$ directly depends on the empirical behavior of the hypothesis $h \in \mathcal{H}$ under consideration, yielding bounds of the form

$$
P \left( \forall h \in \mathcal{H} : \text{risk}(h) \leq \text{risk}_{\text{emp}}(h) + C_T(h, S) + c \sqrt{\frac{\ln(1/\delta)}{T}} \right) \geq 1 - \delta . \quad (4)$$

Prominent examples of this kind are the bounds for linear-threshold classifiers, where $C_T(S)$ depends on the margin of $h$ on $S$ [AKLL02, KP02, LSM01, SFBL98], and the bounds for Bayesian mixtures, where $C_T(S)$ depends on the Kullback-Leibler divergence between the data-dependent mixture coefficients and the a priori coefficients [MZ03]. Note that bounds of the form (4) leave open the algorithmic problem of finding the hypothesis $h \in \mathcal{H}$ optimizing the tradeoff between the terms $\text{risk}_{\text{emp}}(h)$ and $C_T(h, S)$.

The techniques based on uniform convergence, which we have quickly mentioned, lead to probabilistic statements which hold simultaneously for all hypotheses in the class. A different approach used to derive data-dependent risk bounds yields statements which only hold for the hypotheses generated by learning algorithms satisfying certain properties. Examples along these lines are the notions of self-bounding learners [BL03, Fr98], algorithmic stability [BE02], and algorithmic luckiness [HeWi02].

We call these bounds data and algorithm-dependent bounds.

The goal of the remaining part of this tutorial is to describe a simple and general way of obtaining such bounds. In fact, there are natural reductions from on-line algorithms working in the worst-case setting to batch algorithms working under i.i.d. assumptions on the data. As we will see, the underlying theory is fairly easy to understand. The statistical tools involved are discrete-time martingales (which are introduced here), along with simple concentration properties.

What's important is that these reductions actually preserve the scalability of the (on-line) algorithms we start from. Therefore we come up with a simple statistical theory producing scalable algorithms. Besides, this theory gives rise to sharp error bounds which are also trivial to compute from the data.
Slide 52

In this and the next slide we make a little detour towards martingales and related concepts. The beginning of the below discussion is basically taken from [Fe71].

Consider a standard coin-tossing game in which the gambler chooses his/her stakes according to some rules involving his/her success in previous trials. Let the sequence of outcomes be denoted by $X_1, X_2, ..., X_t, ...$, where $X_t \in \{-1, +1\}$ are i.i.d. with

$$\mathbb{P}(X_t = +1) = \mathbb{P}(X_t = -1) = 1/2.$$ 

The gambler adopts at each trial some strategy depending on the past observations. Let the successive gains of the gambler be denoted by $L_1, L_2, ..., L_t, ...$. It should be clear that these variables are no longer independent. We say that this game is fair if

$$\mathbb{E}[L_{t+1} \mid L_1, L_2, ..., L_t] = 0 \quad (5)$$

holds for any $t$. The idea of a fair game is that the knowledge of the past should not enable improvements on the gambler’s fortune. Intuitively, this means that a fair game should remain fair under any gambling system.

The conditioning in (5) is not particularly fortunate since one has frequently to replace the conditioning variables $L_1, ..., L_t$ by some function of them. A greater defect is revealed by our coin-tossing example. The underlying process is represented by a sequence of random variables $X_t$ and the gambler’s gain at the end of the $(t+1)$-st trial is some function of $X_1, ..., X_t, X_{t+1}$, and possibly other variables. The observable past is represented by $X_1, ..., X_t$, which may provide more information than the past gains. For example, if the gambler skips trials number 1, 3, 5, ... the knowledge of his/her gains up to time $2t$ is at best equivalent to the knowledge of $X_2, X_4, ..., X_{2t}$. Here the additional knowledge of $X_1, X_3, ...$ could in principle imply an advantage, and fairness in this case must be based on conditioning on $X_1, X_2, ..., X_t$. Thus, conditioning with respect to various sets of random variables$^{15}$ may be necessary. We say that a sequence of random variables $L_1, ..., L_t, ...$ is a martingale difference sequence w.r.t. a sequence of random variables $X_1, ..., X_t, ...$ if, for any $t$, $L_t$ is a function of $X_1, ..., X_t$ and

$$\mathbb{E}[L_{t+1} \mid X_1, X_2, ..., X_t] = 0 \quad (6)$$

holds. Notice that the sequence of zero-mean i.i.d. random variable $X_1, ..., X_t, ...$ is a special case of a martingale difference sequence.

The sequence of partial sums $S_1, S_2, ..., S_t, ...$, where $S_t = L_1 + L_2 + ... + L_t$, is called a martingale (w.r.t. the sequence $X_1, ..., X_t, ...$).

$^{14}$Later on we will mainly reason about losses, instead of gains. Thus we prefer to use $L_t$ instead of, say, $G_t$ to denote the result of the gambler’s strategy at time $t$.

$^{15}$In mathematical terms, this problem is basically solved by conditioning with respect to a sequence of $\sigma$-algebras including the $\sigma$-algebras generated by $X_1, X_2, ..., X_t$. Details can be found in any standard textbook on probability theory, e.g., [CT97].
Slide 53

Martingales share many properties of sums of zero-mean independent random variables. This is because a sequence of random variables satisfying (6) behaves like a sequence of (zero-mean) independent variables in terms of concentration properties (such as the law of large numbers).

Recall the standard exponential form of the law of large numbers given by Chernoff-Hoeffding’s inequality:

Let \( X_1, X_2, \ldots, X_T \) be a sequence of zero-mean i.i.d. random variables such that \( X_t \in [a, b] \) for all \( t \). Then for any \( \delta \in [0, 1] \) we have

\[
P \left( \left| \frac{X_1 + X_2 + \ldots + X_T}{T} \right| \leq (b - a) \sqrt{\frac{\log 2/\delta}{2T}} \right) \geq 1 - \delta.
\]

It turns out that the very same concentration statement actually holds for a martingale difference sequence (this is called Hoeffding-Azuma’s inequality):

Let \( L_1 + L_2 + \ldots + L_T \) be a martingale difference sequence w.r.t. another sequence \( X_1, X_2, \ldots, X_T \), where \( L_t \in [a, b] \) for all \( t \). Then for any \( \delta \in [0, 1] \) we have

\[
P \left( \left| \frac{L_1 + L_2 + \ldots + L_T}{T} \right| \leq (b - a) \sqrt{\frac{\log 2/\delta}{2T}} \right) \geq 1 - \delta.
\]

From the point of view of fair games, the above inequality states that if a gambler plays a fair game on the sequence of outcomes \( X_1, \ldots, X_T \), then, no matter what his or her strategy is, the cumulative gain (or loss) is likely to concentrate around zero, with a convergence rate\(^{16}\) of \( 1/\sqrt{T} \).

\(^{16}\)This rate can actually be improved if we take into account variance information. See Slide 69.
Let’s get back to on-line algorithms. As we have seen, an on-line algorithm $A$ works in a series of trials. In the generic trial $t$ the algorithm observes instance $x_t$, predicts the (binary) value $\hat{y}_t$, receives (binary) feedback $y_t$ and incurs a unit amount of loss if and only if $\hat{y}_t \neq y_t$ (in this case we say that $A$ has made a mistake). Then the algorithm updates its internal state and faces trial $t+1$.

Given a sequence $S = (x_1, y_1), (x_2, y_2), \ldots, (x_T, y_T)$, we count the total number of mistakes $A$ makes. We have mainly focused on linear-threshold functions (and their kernel extensions). We have seen many on-line algorithms, ranging from the Halving Algorithm, to the (first and second-order) Perceptron algorithms, to $p$-norm algorithms, to shifting algorithms, to label-efficient algorithms. In a sense, these algorithms have different goals (and different good and bad points), since they are able to capture different properties of the learning sequence.

The bounds we can prove for them are pointwise bounds of the form

$$\text{Total \# mistakes}_A(S) \leq \text{some}\_\text{function}(S),$$

where ”some\_function” extracts some specific characteristics from $S$, depending on the algorithm $A$ we are using. For instance, some\_function$(S)$ has to do with the margin $\gamma$ and the radius $R$, the eigenvalues of the Gram matrix, the dual-norm margin, the shifting target sequence, and so on.

No probability assertions are involved. We call them pointwise bounds since they hold for any sequence of examples.
Slide 55

By the way on-line algorithms work, we can view the learning process as producing a sequence of hypotheses $H_0, H_1, ..., H_t, ...$, where $H_0$ is some initial hypothesis and, at the end of trial $t$, $H_t$ is a function of the subsequence of examples $(x_1, y_1), ..., (x_t, y_t)$ observed so far. Now, if we assume $S = (x_1, y_1), ..., (x_T, y_T)$ is actually a sequence of i.i.d. random variables, then $H_0, H_1, ..., H_T$ will be a sequence of random variables as well, with $H_t$ depending only on the prefix $(x_1, y_1), ..., (x_t, y_t)$. We call the sequence $H_0, H_1, ..., H_T$ the ensemble of hypotheses produced by the algorithm during its run over sequence $S$.

Our statistical risk analysis hinges on the intuitive fact that if the on-line algorithm is ”doing a good job” on $S$, then there must be at least one ”good” hypothesis in the ensemble (namely, one with small risk). Our goal is to extract from the ensemble such a hypothesis.

The essence of the above observation, cast in a statistical learning context, dates back to Littlestone [L89b], who proposed to assess the prediction capability of each hypothesis in the ensemble by testing each one of them on a separate test. The hypothesis extracted is simply the one performing best on the test set.

There are, however, more effective ways of doing so, especially if we want to obtain data-dependent generalization bounds.
So, we have a sequence of hypotheses \( H_0, H_1, \ldots, H_T \). Which one should we choose? An obvious choice is to pick \( H_T \), i.e., the hypothesis that depends on the whole training set. However, without making specific assumptions on the way \( A \) operates, we cannot say about \( H_T \) much more than what could be said via standard uniform convergence arguments.

Thus we take a different route [CBCG04].

Suppose the decision space \( \mathcal{D} \) of \( A \) is a convex set and that the loss function \( \text{loss} \) is convex in its second argument. Define the average hypothesis

\[
\overline{H}(x) = \frac{1}{T} \sum_{t=1}^{T} H_{t-1}(x).
\]

The assumption on \( \mathcal{D} \) ensures that \( \overline{H} \) is indeed a map from \( \mathcal{X} \) to \( \mathcal{D} \). Since we are focusing on binary classification problems, this makes sense only when we are running an on-line algorithm for which we can prove a cumulative bound on a convex upper bound on the standard 0-1 loss. For instance, when dealing with linear-threshold predictors, the 0-1 loss \( \frac{1}{2} |\text{SGN}(\mathbf{w}^T \mathbf{x}) - y| \) is upper bounded by the convex function \( \max\{0, 1 - y\mathbf{w}^T \mathbf{x} / \gamma\} \), for any \( \gamma > 0 \). As we will see below, under the above assumptions, the risk of the average hypothesis is “small” whenever the underlying algorithm has a “small” cumulative loss.

When such a convex upper bound on the 0-1 loss is not available, we compute the empirical risk of each hypothesis \( H_t \) on the sequence of remaining examples. Then, to compensate for the fact that the hypotheses have been tested on portions of the sample of different length, a different penalization term is added to the empirical risk of each \( H_t \).

We define the penalized empirical risk of hypothesis \( H_t \) by

\[
\text{risk}_{\text{emp}}(H_t, t + 1) + \sqrt{\frac{1}{T - t} \ln \frac{T}{\delta}},
\]

where the first term

\[
\text{risk}_{\text{emp}}(H_t, t + 1) = \frac{1}{n - t} \sum_{i=t+1}^{n} \text{loss}(H_t(X_i), Y_i)
\]

is the empirical risk of \( H_t \) on the remaining sample \( (X_{i+1}, Y_{i+1}), \ldots, (X_T, Y_T) \), and the second term is a penalization term.

We define the best penalized hypothesis \( \hat{H} \) as the hypothesis minimizing the penalized risk estimate over all hypotheses in the ensemble, i.e.,

\[
\hat{H} = \arg\min_{t=0, \ldots, T-1} \left( \text{risk}_{\text{emp}}(H_t, t + 1) + \sqrt{\frac{1}{T - t} \ln \frac{T}{\delta}} \right).
\]
The proof technique resorts to the analytical tools we have already seen: martingales and related concentration results. We know that the sequence of hypotheses $H_0, H_1, H_2, \ldots$ are a sequence of random variables depending on the training sample, such that $H_0$ is constant, $H_1$ is a function of the first example $(X_1, Y_1)$, $H_2$ is a function of the first two examples $(X_1, Y_1), (X_2, Y_2)$, and so on.

Given this state of affairs, we can build a martingale difference sequence $L_1, L_2, \ldots$, where $L_t$ is basically the loss of the algorithm in the $t$-th trial minus its expectation. More precisely, we set

$$L_t = \text{loss}(Y_t, H_{t-1}(X_t)) - \text{risk}(H_{t-1}),$$

where $\text{risk}(H_{t-1})$ is the (conditional) risk of $H_{t-1}$, i.e.,

$$\text{risk}(H_{t-1}) = \mathbb{E}[\text{loss}(Y, H_{t-1}(X)) \mid (X_1, Y_1), \ldots, (X_{t-1}, Y_{t-1})].$$

Observe that since $H_{t-1}$ is a function of the random variables

$$(X_1, Y_1), \ldots, (X_{t-1}, Y_{t-1})$$

so is $\text{risk}(H_{t-1})$. On the other hand, $\text{loss}(Y_t, H_{t-1}(X_t))$ is a random variable, function of

$$(X_1, Y_1), \ldots, (X_{t-1}, Y_{t-1}), (X_t, Y_t).$$

As a result, $L_t$ turns out to be a random variable, function of

$$(X_1, Y_1), \ldots, (X_{t-1}, Y_{t-1}), (X_t, Y_t),$$

too. This holds for all $t$. 

59
Slide 58

Now, by the very definition of \( L_t \) we have

\[
\mathbb{E}[L_t \mid (X_1, Y_1), \ldots, (X_{t-1}, Y_{t-1})] \\
= \mathbb{E}[\text{loss}(Y, H_{t-1}(X)) \mid (X_1, Y_1), \ldots, (X_{t-1}, Y_{t-1})] \\
- \mathbb{E}[\text{risk}(H_{t-1}) \mid (X_1, Y_1), \ldots, (X_{t-1}, Y_{t-1})] \\
= \text{risk}(H_{t-1}) - \text{risk}(H_t) \\
= 0.
\]

Finally, since \( \text{loss} \) is the 0-1 loss function, \( L_t \) is forced to lie in \([-1, 1]\). Hence, for any fixed horizon \( T \), the sequence \( L_1, L_2, \ldots, L_T \) is a bounded martingale difference sequence w.r.t. \( (X_1, Y_1), \ldots, (X_T, Y_T) \).

From Hoeffding-Azuma’s inequality we conclude that the empirical average \( \frac{L_1 + L_2 + \ldots + L_T}{T} \) concentrates around 0. More precisely:

\[
\mathbb{P} \left( \left| \frac{L_1 + L_2 + \ldots + L_T}{T} \right| \leq 2 \sqrt{\frac{\log 2/\delta}{2T}} \right) \geq 1 - \delta,
\]

which, recalling the definition of \( L_t \), implies

\[
\mathbb{P} \left( \frac{1}{T} \sum_{t=1}^{T} \text{risk}(H_{t-1}) - \frac{1}{T} \sum_{t=1}^{T} \text{loss}(Y_t, H_{t-1}(X_t)) \leq 2 \sqrt{\frac{\log 2/\delta}{2T}} \right) \geq 1 - \delta. \tag{8}
\]
Slide 59

Notice that \( \sum_{t=1}^{T} \text{loss}(Y_t, H_{t-1}(X_t)) \) is actually the cumulative loss incurred by the algorithm on the sequence of examples \((X_1, Y_1), \ldots, (X_T, Y_T)\). Define the data-dependent statistic

\[
M_T = \frac{1}{T} \sum_{t=1}^{T} \text{loss}(Y_t, H_{t-1}(X_t)).
\]

\(M_T\) is the algorithm’s loss averaged over the trials.

In words, (8) states that the average conditional risk \( \frac{1}{T} \sum_{t=1}^{T} \text{risk}(H_{t-1}) \) of the ensemble concentrates around \(M_T\). Thus, if we pick a hypothesis at random from the ensemble, this hypothesis has, on average, a risk close to the average cumulative loss \(M_T\).

Anyway, under the above convexity assumptions, Jensen’s inequality and the linearity of expectation allows us to write the following assertion about the average hypothesis \(\overline{H}\):

\[
\text{risk}(\overline{H}) = \mathbb{E} \left[ \text{loss} \left( Y, \frac{1}{T} \sum_{t=1}^{T} H_{t-1}(X) \right) \mid (X_1, Y_1), \ldots, (X_{t-1}, Y_{t-1}) \right]
\leq \frac{1}{T} \sum_{t=1}^{T} \mathbb{E} \left[ \text{loss}(Y, H_{t-1}(X)) \mid (X_1, Y_1), \ldots, (X_{t-1}, Y_{t-1}) \right]
= \frac{1}{T} \sum_{t=1}^{T} \text{risk}(H_{t-1}) .
\]

Thus convexity implies that the risk of the average hypothesis of the ensemble is at most equal to the average conditional risk \( \frac{1}{T} \sum_{t=1}^{T} \text{risk}(H_{t-1}) \) of the ensemble.

In the more general case of nonconvex losses, a similar (though slightly weaker) assertion can be proven for the best penalized hypothesis \(\hat{H}\). With probability at least \(1 - \delta\) we have

\[
\text{risk}(\hat{H}) \leq \frac{1}{T} \sum_{t=1}^{T} \text{risk}(H_{t-1}) + 4 \sqrt{\frac{\log T}{T}}. \quad (10)
\]

Both (9) and (10) can be immediately combined with (8) – see next slide.
Slide 60

The easy consequences are the following two upper bounds.

Assume we have an on-line algorithm $A$ that sweeps through a sequence of examples $S = (X_1, Y_1), ..., (X_T, Y_T)$ drawn i.i.d. according to an unknown distribution over $\mathcal{X} \times \{-1, +1\}$.

Let $A$ work with a loss function $loss$ which is a convex upper bound on the 0-1 loss (as explained before) with bounded range $L$. Let $M_T$ be equal to the cumulative loss of $A$ on $S$ divided by the sequence length $T$. Then, combining (9) with (8) yields

$$
P\left( \text{risk}(\mathcal{H}) \leq M_T + L \sqrt{\frac{2 \log 2/\delta}{T}} \right) \geq 1 - \delta.
$$

On the other hand, if $A$ works directly with the 0-1 loss function, and we let $M_T$ be equal to the cumulative 0-1 loss of $A$ on $S$ divided by $T$, combining (10) with (8) implies

$$
P\left( \text{risk}(\hat{\mathcal{H}}) \leq M_T + 6 \sqrt{\frac{\log T/\delta}{T}} \right) \geq 1 - \delta.
$$

Both bounds are data-dependent bounds since they relate the risk of the computed hypothesis (either $\mathcal{H}$ or $\hat{\mathcal{H}}$) to the empirical quantity $M_T$, which is easily determined from the data as the algorithm runs. Moreover, both bounds are algorithm-dependent bounds since, unlike uniform convergence statements, they involve the generalization performance of specific hypotheses built from the data.
Slide 61

At this point, piecing together with our pointwise bounds is just an easy “plug and play”.

Recall the bound on the Halving Algorithm (HA) for learning linear-threshold functions in the separable case (Slide 17). We know for sure that an upper bound on the statistic $M_T$ computed by HA is\footnote{Recall that $d$ denotes the dimension of the instance space.}

\[ M_T \leq \frac{1}{T} O (d \log(R/\gamma)). \]

We run HA on $S$ producing the ensemble $H_0, \ldots, H_T$. Then we define $\hat{H}$ as in (7). Plugging back into

\[ \mathbb{P} \left( \text{risk}(\hat{H}) \leq M_T + 6 \sqrt{\frac{\log T/\delta}{T}} \right) \geq 1 - \delta \]

yields

\[ \mathbb{P} \left( \text{risk}(\hat{H}) \leq \frac{1}{T} O (d \log(R/\gamma)) + 6 \sqrt{\frac{\log T/\delta}{T}} \right) \geq 1 - \delta. \]

This bound turns out to be very similar to a bound proven by [HG02] following a “PAC-Bayesian” approach, involving rather complicated version space calculations.
Recall the bounds for the kernel Perceptron algorithm in both the separable and the nonseparable cases (Slide 24):

\[ M_T \leq \frac{1}{T} \max_{i \in \mathcal{M}} K(x_i, x_i) \gamma^2 \]

and

\[ M_T \leq \inf_{\gamma > 0, \ f \in \mathcal{H}_K, \ ||f||=1} \frac{1}{T} \left( D_\gamma(f; S) + \frac{\sqrt{\sum_{x \in \mathcal{M}} K(x, x)}}{\gamma} \right) \]

Plugging back into

\[ \Pr \left( \text{risk} (\hat{H}) \leq M_T + 6 \sqrt{\frac{1}{T} \ln \frac{T}{\delta}} \right) \geq 1 - \delta \]

yields results similar to those obtained on Support Vector Machines by [BM02], exploiting uniform convergence tools (Rademacher and Gaussian complexities of function classes).

Stressing the similarity between our bounds and the one in [BM02] is an important remark. Training on-line a (kernel) Perceptron is much faster than training a (kernel) SVM. This, however, should not lead to the (hasty) conclusion that we can in any practical situation replace an SVM by a Perceptron algorithm that cycles through the data just once. We do not expect such a Perceptron to have a better prediction performance than an SVM... Rather, we can perhaps conclude that the theory of generalization bounds based on uniform convergence is largely overestimating the generalization ability of SVM and related algorithms, making the bounds hardly useful in practical error reporting.
Yet another “plug and play”. Recalling the bound on the Second-order Perceptron algorithm with kernels reported in Slide 26, we can write

\[ M_T \leq \frac{1}{T} a + \frac{\sum_i \log(1 + \lambda_i/a)}{\gamma}. \]

Combining with

\[ \Pr \left( \text{risk}(\hat{H}) \leq M_T + 6 \sqrt{\frac{1}{T} \ln \frac{T}{\delta}} \right) \geq 1 - \delta \]

gives

\[ \Pr \left( \text{risk}(\hat{H}) \leq \frac{1}{T} a + \frac{\sum_i \log(1 + \lambda_i/a)}{\gamma} + 6 \sqrt{\frac{1}{T} \ln \frac{T}{\delta}} \right) \geq 1 - \delta. \]

The resulting data-dependent generalization bound is similar in spirit, though not readily comparable, to the bounds given in [WSTSS99] for SVM. In fact, the results in [WSTSS99] are derived, via involved covering numbers arguments, in terms of the Gram matrix of the \textit{whole} sequence of examples. More precisely, these results are expressed in terms of all the “large” eigenvalues of the Gram matrix, taken in decreasing order of magnitude up to the “effective number of dimensions”. In contrast to that, the above bound is in terms of all the eigenvalues of the Gram sub-matrix made up of instances where the Second-order Perceptron algorithm happened to have made a mistake.

---

\[ ^{18} \text{The “effective number of dimensions” depends, for instance, on the margin of the data.} \]
Slide 64

Similar “plug and play” with the Projection-based Perceptron algorithm.

\[ M_T \leq \frac{4 R^2}{T} \left( \frac{1}{\gamma^2} + \sum_{t=1}^{T} ||u_{t+1} - u_{t}|| \right) \]

to be substituted into

\[ \Pr \left( \text{risk}(\hat{H}) \leq M_T + 6 \sqrt{\frac{1}{T} \ln \frac{T}{\delta}} \right) \geq 1 - \delta. \]

Recall that \( u_1, ..., u_T \) are functions of the data sequence, thus they are actually random variables.
Slide 65

Using the pointwise bounds for the label-efficient Perceptron algorithm needs more care. In fact, the algorithm is a randomized one and we bounded (Slide 44) the expected number of mistakes and the expected number of labels requested during a run. Both expectations are w.r.t. the internal randomization of the algorithm (the independent coin flips). Let us denote this expectation by $E_\tau$.

Now we have an i.i.d. sequence of random variables $(X_1, Y_1), \ldots, (X_T, Y_T)$ (the training sample). The bound on the number of queried labels holds for any realization of the training sequence. Thus we can still write

$$E_\tau[\# \text{ of labels}] = \sum_{t=1}^T E_\tau \left[ \frac{b}{b + |r_t|} \right],$$

though it should be emphasized that both sides of the last equation are now random variables (they are functions of the training sample).

The risk of the best penalized hypothesis $\hat{H}$ resulting from running the label-efficient Perceptron algorithm on the sample is again an $E_\tau$-average risk. Hence with probability larger than $1 - \delta$ (over the random draw of the training sample) we obtain

$$E_\tau[\text{risk}(\hat{H})] \leq \frac{1}{T} R^2 \frac{1}{\gamma^2} + 6 \sqrt{\frac{1}{T} \ln \frac{T}{\delta}}.$$
We can, of course, apply what we have seen so far to any batch learning algorithm. We run this algorithm in an on-line fashion, count the number of prediction mistakes, and store the sequence of hypotheses produced during this run. Then we determine the best penalized hypothesis out of this sequence.

The generalization ability (as measured by the statistical risk) of this hypothesis is with high probability not much larger than the number of mistakes the algorithm has made during its on-line run.

Notice that in the above statement there is no direct mention of such notions as the “complexity of the function class” where the hypotheses are from. We expect, however, this complexity be somehow reflected by how many wrong predictions the algorithm makes on the training sequence.
The statistical bounds we have obtained so far:

- Are data and algorithm-dependent in style. In fact, the bounds refer to specific algorithms and relate the generalization error of these algorithms to empirical quantities, which are trivial to compute from the data. These bounds explicitly avoid the uniform convergence machinery and are therefore closer in spirit to the more recent literature on "algorithmic stability" [BE02] and "luckiness" framework [HeWi02].

- Are easy to derive, the proofs following from simple concentration properties of martingales.

- Refer to on-line algorithms, i.e., to highly scalable and highly adaptive learning procedures.

- Are tight: they are as tight as the generalization bounds one can derive by the most up to date data-dependent techniques (such as Rademacher complexity);

- Are widely applicable, since we can in principle run any algorithm in an on-line fashion.
Slide 68

There are at least two ways of refining the “on-line to batch” reductions we have seen so far. The first one is a simple improvement over the basic assertion we have exploited in Slides 59 and 60 (average conditional risk \( \frac{1}{T} \sum_{t=1}^{T} \text{risk}(H_{t-1}) \) concentrates around average cumulative loss \( M_T \)).

Define \( M_{t,T} \) as the average cumulative loss on the suffix \( (X_{t+1}, Y_{t+1}), \ldots, (X_T, Y_T) \):

\[
M_{t,T} = \frac{1}{T-t} \sum_{i=t+1}^{T} \text{loss}(Y_i, H_{i-1}(X_i)).
\]

Then it is fairly easy to show that the best penalized hypothesis \( \hat{H} \) satisfies

\[
\Pr \left( \text{risk}(\hat{H}) \leq \min_{t=0, \ldots, T-1} \left( M_{t,T} + 6 \sqrt{\frac{1}{T-t} \ln \frac{T}{\delta}} \right) \right) \geq 1 - \delta.
\]

The bound in Slide 60 is then obtained by upper bounding the minimum over \( t \) with \( t = 0 \). Again, this data-dependent bound is straightforward to compute from the data.
Slide 69

A second refinement comes from using slightly more subtle tools from martingale theory, i.e., Bernstein-type inequalities for martingales [F75, DZ01].

In particular, one can prove the following risk bound\footnote{The big-ooh hides reasonably small constants.}

\[
    \Pr \left( \text{risk} (\hat{H}) \leq M_T + O \left( \frac{1}{T} \ln \frac{T}{\delta} + \sqrt{\frac{M_T}{T}} \ln \frac{T}{\delta} \right) \right) \geq 1 - \delta, \tag{11}
\]

so long as we define the penalization term in \( \hat{H} \) in a way that matches the confidence term \( O \left( \frac{1}{T} \ln \frac{T}{\delta} + \sqrt{\frac{M_T}{T}} \ln \frac{T}{\delta} \right) \):

\[
    \hat{H} = \arg \min_{t=0 \ldots T-1} \left( \text{risk}_{\text{emp}} (H_t, t + 1) + \frac{1}{T-t} \ln \frac{T}{\delta} + \sqrt{\frac{\text{risk}_{\text{emp}} (H_t, t + 1)}{T-t}} \ln \frac{T}{\delta} \right),
\]

where

\[
    \text{risk}_{\text{emp}} (H_t, t + 1) = \frac{1}{T-t} \sum_{i=t+1}^{T} \text{loss} (Y_i, H_t (X_i)).
\]

If the convexity assumptions of Slide 60 hold, then the risk bound contained in (11) also holds for the average hypothesis \( \overline{H} \).

Observe that the convergence rate in (11) is always sharper than those in Slide 60, since \( M_T \leq 1 \) anyway. Besides, if the cumulative loss \( T M_T \) is small (say, \( M_T = c/T \), where \( c \) is constant with \( T \)), then the above best penalized hypothesis achieves a \( O ((\ln T)/T) \) risk bound. In this case, it is actually possible to achieve the sharper rate \( 1/T \). Details are omitted from this tutorial. The interested reader is referred to [CBG04, Z05].

Finally, combining "Refinement 1" with "Refinement 2" results in an even sharper bound.
Slide 70

In this tutorial, we have presented a survey on recent results in on-line learning along with their connections to statistical Pattern Recognition.

In the first part, we have presented a collection of algorithms and results within the worst-case setting of on-line learning, where prediction performance is expressed in terms of pointwise bounds. In the second part, we have described a simple way of turning a pointwise analysis into a statistical risk analysis. The underlying reductions preserve the scalability of the original on-line algorithms involved. In practice, we just need to run the algorithms and count the number of mistakes.

The generalization bounds we derived from this simple theory are both data-dependent and algorithm-dependent. The whole theory is immediately extended to more general regression frameworks.

A broader goal of this tutorial was to bridge Learning Theory and Machine Learning communities. We hope this exposition will contribute to increasing the cooperation between the two.
Disclaimer: This is by no means a complete bibliography on the subject of this tutorial

References


[OH] The OHSUMED test collection. URL: medir.ohsu.edu/pub/ohsumed/.

[RCV1] Reuters corpus volume 1. URL: about.reuters.com/researchandstandards/corpus/.


