Note 1: On-Line Learning Models

1 Introduction

The course deals with formal models of machine learning. Our main aim will be to study well-defined models of learnability and characterize the complexity of these problems. That is, if at all possible, we will try to design efficient algorithms for such learning problems and provide a guarantee that they perform the learning task correctly. At the same time, we will try to establish lower bounds for the complexity of the problems thus exploring the limits of efficient learnability. We will look at several models of learning, that is, several ways of defining the learning task so that “correctly” is interpreted differently in each of these. Naturally the various learning models will be related. Most of the material of the course is covered in the text [6]; this note collects some results for the “on-line” family of learning models not covered there.

2 On-Line Sequence Prediction

We use a simple a game of sequence prediction to illustrate various aspects of on-line learning models. Consider a game where you are shown a sequence of numbers, say, 3, 5, 7, and asked to guess the next number in the sequence. Seeing the number above you may guess that the sequence represents the odd numbers and the next number is 9. However, you are told that you made a mistake and the next number is in fact 11. Can you guess the next number in the sequence? If we repeatedly continue this game with the same sequence, one might hope that at some point you will discover the rule underlying the sequence and predict correctly for all next elements in the sequence. (The sequence above is meant to be the sequence of prime numbers greater than 3.) We can generalize this example so that a learner must perform this task for any sequence in a given class of sequences. This is formalized below.

Let $F$ be a class of total and computable functions from integers to integers. Any function $f: \mathbb{N} \rightarrow \mathbb{N}$ in $F$ defines a sequence $f(1), f(2), f(3), \ldots$ of its values. A sequence prediction game can be played between an adversary and a learner in stages as follows. First the adversary picks a function $f \in F$. Then, in the $i$'th stage (for $i = 1, 2, 3, \ldots$) the learner guesses the value of $f(i)$. It is then told the correct value, and if different from its guess we say that the learner made a mistake.

**Definition 2.1** An algorithm learns to predict sequences in $F$ in the limit if for every $f \in F$ there exits a finite $M$ such that the algorithm does not make mistakes after the $M$'th stage.

Note that learnability is a property of classes of sequences not a single sequence (since for a single sequence there is a “specialized algorithm” that knows it in advance and hence “learns” it). As the example above demonstrates, in principle a learner may converge (that is be in a state after which it does not make mistakes) but not be certain that it had converged. This does not cause a problem in the definition since such a learner is considered successful.
Let us consider what classes may be learnable in this sense. Consider first the class IP of functions \( f(n) \) expressible as polynomials in \( n \) with integer coefficients. For example, \( f(n) = n^2 + 3n + 1 \) is one such function that corresponds to the sequence 5, 11, 19, \ldots, to be presented to the learner. Is this class learnable? The answer is yes; in fact we will present two different algorithms that can learn to predict IP.

The first algorithm is very crude but it represents a general principle that can be used in other situations as well. The algorithm enumerates for \( d = 0, 1, 2, \ldots \), the finite set of polynomials of degree \( d \) with coefficients in \([-d, \ldots, -1, 0, 1, \ldots, d]\). This can be done by enumerating tuples \((a_0, \ldots, a_d)\) where each \( a_i \) is in the specified range. Now given an initial sequence \( f(1); f(2); \ldots; f(i) \) the algorithm finds the first polynomial \( p(n) \) in the enumeration above that computes correct values on this sequence (this can be done since one can evaluate polynomials efficiently) and its guess is \( p(i+1) \). To see that this algorithm succeeds, assume that the adversary picked \( q(n) = b_0 + b_1n + \ldots + b_kn^k \). Then once \( d \) exceeds both \( k \) and \( \max |b_i| \) then \( q(n) \) belongs to the enumeration. Therefore, once all polynomials that precede \( q(n) \) in the enumeration have been found incorrect (if this does not happen then the algorithm succeeds as well), \( q(n) \) is chosen and clearly it will always be correct. We therefore have:

**Theorem 2.1** The enumeration based algorithm learns to predict sequences in IP in the limit.

For a polynomial of degree \( k \) (and smaller coefficients) the algorithm will need to check \( k^k \) polynomials before finding the right one. Note that this algorithm can be adapted to learn any class of functions that can be enumerated and whose functions can be evaluated on any input; we leave the details of the following theorem as an exercise:

**Theorem 2.2** Let \( C \) be a class of functions that can be enumerated and whose functions can be evaluated on any input. Then the enumeration based algorithm learns to predict sequences in \( C \) in the limit.

The second algorithm does much better by using a technique that is specialized for IP, namely, interpolating the polynomial from the data. Given the initial sequence \( f(1), f(2), \ldots, f(i) \) the algorithm finds the polynomial of smallest degree that agrees with these data points. For a fixed degree, this can be done by solving a system of linear equations, one generated from every example. For example, \( f(2) \) generates \( f(2) = b_0 + 2b_1 + \ldots + 2^kb_k \). To get the smallest degree simply try all possible values in a bottom up manner. After the \( k \)th stage the algorithm will find the right polynomial. For details on interpolation see Chapter 32 of [3].

**Theorem 2.3** The interpolation based algorithm learns to predict sequences in IP in the limit.

This example illustrates some of the issues we will deal with in the course. The definition above required that after some time the learner can predict all future values of the sequence. In other models we will similarly require the learner to form a hypothesis that can be used in the future. It is sometimes tempting to define learning as just finding a hypothesis consistent with the examples (as our algorithms have done). This is less satisfying since it gives no guarantee on the utility of using the final hypothesis of the learner. As we have seen it is also not needed.

Another aspect is raised by the difference between the two algorithms. As far as our definition is concerned both algorithms are good. However it is clear that in terms of efficiency the second one is preferable, and from a computational perspective the first algorithm is not very interesting. We will therefore incorporate efficiency as part of the requirements for successful learning. For
polynomials we have seen that learning can be done efficiently, but depending on the degree of the target polynomial. We can generalize this to allow the number of mistakes the algorithm makes to depend on the size of the function representation, taking size to be a complexity parameter. We can also generalize in another direction and consider functions in more than one variable. This generates another complexity parameter - the input dimension of the function. These aspects are formalized in the next section.

We next show that even without efficiency considerations there are limits to learnability. Let RS be the class of computable total functions from integers to integers. That is, for each \( f \in RS \) there is a Turing machine that on input \( i \) outputs \( f(i) \) in finite time. Note that a sequence is in RS if and only if its elements can be enumerated.

**Theorem 2.4** There is no algorithm that learns to predict sequences in RS in the limit.

**Proof:** The proof is by a diagonalization argument. Assume that there is a learning algorithm \( A \). Let \( B \) be the following algorithm generating a sequence in RS. The algorithm \( B \) uses a copy of \( A \) as a subroutine. In the \( i \)’th stage, \( B \) runs the copy of \( A \) on the sequence generated so far to compute what its prediction will be; let this value be \( k \). \( B \) outputs the value \( k + 1 \) as the \( i \)’th element. Clearly, if \( A \) is a computable function then so is \( B \) and hence \( B \in RS \). It is also clear that \( A \) makes an infinite number of mistakes on the sequence of \( B \). \( \square \)

Note that together with Theorem 2.2 this result shows that RS is not recursively enumerable (which is a well known result). Various interesting questions arise when considering partial functions (corresponding to Turing machines that may not halt on some inputs). However, we will not pursue this in the course. Instead we consider more restricted cases and stress efficiency as one of the primary requirements of learning.

### 3 On-Line Concept Learning

Concept learning describes a scenario where a learner sees some exemplars of a particular concept (e.g. bitmap pictures describing objects — and whether the object is a chair or not) and has to identify that concept in the sense that it can predict for future examples (pictures) whether they belong to the concept (describe a chair) or not. For sequence prediction we had a natural ordering on the presentation \( f(1), f(2), \ldots, \) but here there is no such natural ordering on examples. As before we will assume a worst case scenario and allow an adversary to pick the worst possible ordering of examples.

Let \( X \) be a set of elements (called the instance space or domain); a concept \( c \) is a subset of the instance space, such that \( x \in c \) is an element of the concept (called a positive example) and \( x \notin c \) is not in the concept (a negative example). It is convenient to think of concepts as Boolean functions, \( c \subseteq X \) corresponds to \( c : X \to \{0, 1\} \) where \( c(x) = 1 \) iff \( x \in c \). A concept class \( C \subseteq 2^X \) is simply a set of concepts.

An on-line concept learning game can be played between an adversary and a learner in stages as follows. First the adversary picks a concept \( c \in C \). Then, in the \( i \)’th stage (for \( i = 1, 2, 3, \ldots \)), the adversary picks \( x \in X \) and presents it to the learner. The learner guesses the value of \( c(x) \). It is then told the correct value, and if different from the guess we say that the learner made a mistake.

**Definition 3.1** An algorithm learns the class \( C \) in the limit if for every \( c \in C \) and for every sequence of examples chosen by the adversary there exits a finite \( M \) such that the algorithm does not make mistakes after the \( M \)’th stage.
Notice that we could be stricter and require the same $M$ to work for any example presentation but chose not to do so. Such a definition would be too limiting if the adversary is allowed to repeat examples.\footnote{In particular, even the elimination algorithm would not qualify as “learner in the limit” with such a definition.} We will see later that a different form of uniform bound - in terms of the number of mistakes - can be applied.

We next consider learning Boolean functions over Boolean domains. Let $\{x_1, \ldots, x_n\}$ be a set of propositional variables (that is we will assign either 0 or 1 to them). A literal is a variable $x_i$ or its negation $\bar{x}_i$. The instance space $X = \{0, 1\}^n$, that captures all possible assignments to these variables, is the set of possible examples. The class $C_n$ is the class of conjunctions over literals in $\{x_1, \ldots, x_n, \bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n\}$. For example $c = x_1 \land \bar{x}_3$ is a conjunction that is true iff $x_1$ is assigned 1 and $x_3$ is assigned 0. It is convenient to consider conjunctions as sets of literals. For example we say that $\bar{x}_3$ is in $c = x_1 \land \bar{x}_3$.

**The Elimination Algorithm:** (learning the class $C_n$)

1. The algorithm maintains a hypothesis $h$ initialized as $h = x_1 \land x_2 \land \ldots \land x_n \land \bar{x}_1 \land \bar{x}_2 \land \ldots \land \bar{x}_n$. That is the conjunction of all variables and all their negations (and therefore evaluates to 0 on any assignment $x \in \{0, 1\}^n$).

2. Repeat the following on each example in turn:

   (a) On example $x$ the predict $h(x)$.

   (b) If no mistake is made then no update to $h$ is made.

   (c) If a mistake is made such that $h(x) = 0$ was predicted while $c(x) = 1$, then remove from $h$ all the literals that are false in $x$.

   (d) If a mistake is made such that $h(x) = 1$ was predicted while $c(x) = 0$, then stop and report failure of the learning process.

For example, assume that $n = 6, c = x_1 \land \bar{x}_2$, and $h = x_1 \land \bar{x}_2 \land x_3 \land x_4$. Then if presented with $x = 100111$ the algorithm predicts 0 according to $h(x)$. Since $c(x) = 1$ the algorithm responds to the mistake by removing $x_3$ which is false in $x$ from $h$, so that $h = x_1 \land \bar{x}_2 \land x_4$.

**Lemma 3.1** No literal in $c$ is ever removed from $h$.

**Proof:** A literal $l$ is removed from $h$ iff $l = 0$ in $x$ where $h(x) = 0$ and $c(x) = 1$. But if $l$ is in a conjunction $c$ and $c(x) = 1$ then $l$ must be assigned 1 in $x$. \hfill $\blacksquare$

Notice that if the condition of Step 2(d) is satisfied then the learning algorithm fails. The following lemma shows that this does not happen:

**Lemma 3.2** If the algorithm makes a mistake on $x$ then $c(x) = 1$.

**Proof:** By the previous lemma the literals in $h$ are always a superset of the literals in $c$. Therefore if $c(x) = 0$ then the literal that makes $c$ false is also in $h$ and $h(x) = 0$. So a mistake cannot be made for such $x$. \hfill $\blacksquare$

**Theorem 3.3** The elimination algorithm learns the class $C_n$ in the limit. For any $c \in C_n$ the algorithm makes at most $n + 1$ mistakes.
Proof: By the previous lemma the algorithm is well defined. Note that the first mistake removes \( n \) literals and every other mistake removes at least one literal (since \( h(x) = 0 \)). Since the size of \( h \) cannot be negative the theorem follows.

In fact, the theorem shows that there is a uniform bound \( M \) on the number of mistakes made on any \( c \in C_n \). While we have so far ignored computation time it is easy to see that the algorithm can be performed in time polynomial in \( n \). We redefine the learning problem capturing both the requirement for time efficiency and the requirement for a small number of mistakes on any concept in the class. For this let \( \text{size}(c) \) be the size of the description of a concept \( c \) under some reasonable encoding, and \( |x| \) be the size of the description of the example \( x \). For \( X = \{0,1\}^n \) this is just the dimension \( n \).

**Definition 3.2** An algorithm efficiently learns the class \( C \) with mistake bound \( p(n,m) \) if there is a polynomial \( t(n,m) \) such that for every \( c \in C \) and for any sequence of examples
(1) at every stage the algorithm runs in time \( t(|x|, \text{size}(c)) \), and
(2) the algorithm makes at most \( p(|x|, \text{size}(c)) \) mistakes.

**Corollary 3.4** The elimination algorithm efficiently learns the class \( C_n \) with mistake bound \( (n+1) \).

Having seen that simple conjunctive concepts are learnable one might wonder whether more complex Boolean concepts are also learnable. We will devote some time to this question in the course. To illustrate, let \( D_n \) be the class of concepts described by disjunctive normal form (DNF) expressions. These are disjunctions of conjunctions as in \( c = (x_1 \land \overline{x_2} \land x_4) \lor (x_2 \land x_3) \). It is not known whether \( D_n \) is efficiently learnable (though it is known that it is not learnable in some restricted models). On the other hand, let \( T_n \) be the class of all Boolean circuits; then it is known that \( T_n \) is not efficiently learnable (under some complexity assumptions).

Finally, we discuss a weakness of the on-line model. Consider the domain of real numbers, and the class of closed intervals over this instance space. That is, a concept is an interval \([a,b]\) where \( a,b \in \mathbb{R} \) (perhaps capturing “medium size car”). Can this class be learned in the on-line model? While the answer is not obvious, it is easy to see that no deterministic algorithm will succeed. As in the proof for RS for each algorithm one can construct an interval on which it will fail. The main reason for this is the fact that examples and their order can be chosen adversarially. The PAC learning model (covered later in the course) maintains some worst case character but relaxes the adversarial nature to some extent. We will see that intervals are learnable in the PAC model.

## 4 Generic Mistake Bounds

For this section we assume that the concept class is finite and study general aspects of mistake bounds.

**Definition 4.1** For any learning algorithm \( A \) and any concept \( c \in C \) let \( M_A(c) \) be the maximum, over all possible input sequences, of the number of mistakes \( A \) makes when learning \( c \).

**Definition 4.2** The optimal mistake bound for \( C \) is defined as
\[
\text{Opt}(C) = \min_A \max_{c \in C} M_A(c)
\]

The minimum is over all deterministic learning algorithms for \( C \) regardless of computational efficiency.
Clearly $Opt(C) \leq |C| - 1$ and $Opt(C) \leq |X| - 1$. For $x \in X$ the label of $x$ is the classification provided during the learning process. We denote this label by $label(x)$.

**Definition 4.3** For any set of labeled examples $Y \subseteq X \times \{0,1\}$, consistent($Y$) is the set of all concepts in $C$ that assign correct labels to all examples in $Y$. The set consistent($Y$) is sometimes called the version space. (We assume here that $C$ is clear from the context and hence do not include it in the notation.) Formally,

$$consistent(Y) = \{c \in C \mid \forall (y, label(y)) \in Y, c(y) = label(y)\}$$

**Definition 4.4** For any set of concepts $Q$ and any example $x \in X$, $E()$ indicates how $Q$ splits according to the label assigned to $x$

$$E_0(Q, x) = \{c \in Q \mid c(x) = 0\}$$
$$E_1(Q, x) = \{c \in Q \mid c(x) = 1\}$$

**The Halving Algorithm:**

1. The algorithm maintains a variable CONS initialized $CONS = C$.

2. Repeat the following on each example in turn:

   (a) On example $x$ the algorithm first computes the sets $E_0(CONS, x)$ and $E_1(CONS, x)$.
   (b) if $|E_0(CONS, x)| < |E_1(CONS, x)|$ the algorithm predicts 1 and otherwise it predicts 0
   (c) When seeing the correct label $b \in \{0,1\}$ the algorithm sets $CONS = E_b(CONS, x)$.

Let $M_H(C)$ denote the maximum number of mistakes made by the halving algorithm on any $c \in C$.

**Theorem 4.1** for any non-empty class $C$, $Opt(C) \leq M_H(C) \leq \log_2(|C|)$

**Proof:** The left inequality holds by definition. We prove the right inequality. The algorithm predicts using the larger of $E_0(CONS, x)$ and $E_1(CONS, x)$. Therefore, the size of CONS decreases by a factor of at least 2 after every mistake. Since CONS includes at least one concept (the correct one) there are at most $\log_2(|C|)$ mistakes.

While the bound is quite strong the result is not easy to apply as in many cases one cannot implement the halving algorithm efficiently. Moreover, for some classes this bound is not tight. For example take a class of singletons over instance space $X$ (each $c \in C$ includes exactly one example). Then it is easy to see that the halving algorithm makes at most one mistake regardless of the size of $X$. The optimal mistake bound can be characterized by the notion of mistake trees [7] but we will not cover it here.

**5 Tolerating Many Irrelevant Attributes**

Recall that the elimination algorithm learns conjunctions in the on-line model with a mistake bound of $n+1$, where $n$ is the number of propositional variables. Consider a situation where a conjunction depends on a small number, say $k$, of variables. To make this concrete imagine we have $n = 10^6$ but $k = 10$. In this case all other $n-k$ variables are irrelevant for the classification of any example but the mistake bound scales linearly with $n$. 

On the other hand, since $|C| = O(n^k)$ the halving algorithm will make $O(k \log n)$ mistakes. But we would need a direct way to implement it efficiently in order to use this result. We now show that a similar result can be obtained by using linear threshold elements (perceptrons) as the hypothesis language. Further motivation for using perceptrons is given in the next section.

A linear threshold element over $n$ variables is represented using an $(n+1)$-tuple of real numbers, $(w_1, \ldots, w_n, \theta)$. For $x \in \{0, 1\}^n$, $f(x) = 1$ iff $\sum_{i=1}^n w_i x_i > \theta$.

The Algorithm Winnow: (learns monotone disjunctions)

1. The algorithm maintains a hypothesis $h$ in the form of a linear threshold element. The parameters for $h$ are initialized as: $\theta = n$ and $w_i = 1$ for all $i$.

2. Repeat the following on each example in turn:
   (a) On example $x$ the algorithm predicts $h(x)$.
   (b) If a mistake is made such that $h(x) = 1$ was predicted while $c(x) = 0$, then for all $i$ s.t. $x_i = 1$ set $w_i = w_i/2$. We call this a demotion step.
   (c) If a mistake is made such that $h(x) = 0$ was predicted while $c(x) = 1$, then for all $i$ s.t. $x_i = 1$ set $w_i = 2w_i$. We call this a promotion step.

In fact the formula $w_i = w_i \cdot 2^{c(x) - h(x)}$ captures the update in both cases.

Note that the update for the weights make sense. In the first case, since $c(x) = 0$ and $c$ is a monotone disjunction we know that non of the variables in $c$ is assigned 1 in $x$. So variables that have value 1 in $x$ are irrelevant. The demotion step reduces the weight of such variables. In the second case at least one of the variables of $c$ is assigned 1 in $x$, but its weight in $h$ is not large enough and it is certainly less than $n$. Therefore increasing the weight is reasonable. This intuition is made precise in the following argument. From the discussion we have:

**Lemma 5.1**

1. In each promotion step at least one of the variables of $c$ is promoted.
2. For all $i$ and at all times $w_i \leq 2n$.
3. If $x_i$ is a literal in $c$ then $w_i$ is never demoted.

**Lemma 5.2** If $c$ is a monotone disjunction of $k$ variables then the total number of promotion steps is at most $k \log(2n)$.

**Proof:** In every promotion step the weight for at least one of the variables of $c$ is doubled, such weights are never demoted, and the weights are bounded by $2n$. 

**Lemma 5.3** In any run of the algorithm the number of demotion steps $e$ is bounded by the number of promotion steps $p$ as follows: $e \leq 2p + 2$.

**Proof:** Let $W = \sum_{i=1}^n w_i$. Since for an demotion step $h(x) = 1$, we have $\sum_{i=1}^n w_i x_i = \sum_{x_i=1} w_i > n$. Therefore each such step reduces $W$ by at least $n/2$. On the other hand, for a promotion step $h(x) = 0$, and we have $\sum_{i=1}^n w_i x_i = \sum_{x_i=1} w_i \leq n$. Therefore each such step increases $W$ by at most $n$. Since initially $W = n$ and $W$ is never negative the result follows.

**Theorem 5.4** The Winnow algorithm efficiently learns the class of monotone disjunctions with mistake bound $p(n, k) = 3k \log(2n) + 2$. 


Proof: Since the algorithm simply evaluates the linear threshold element it can be implemented efficiently in every stage. Now since every mistake is either a promotion or a demotion step the mistake bound follows.

A slightly better bound can be achieved by setting weights to 0 in demotion steps instead of dividing by 2 but the resulting algorithm is more sensitive to “noise” in the examples.

The next section presents the well known perceptron algorithm for learning linear threshold elements. The Winnow algorithm is a variant of the perceptron algorithm. (Winnow was first analyzed about 3 decades after the perceptron.) Neither algorithm dominates the other in terms of mistake bounds. Winnow provides better bounds in cases where there are many irrelevant variables.

The result above for Winnow gives a dramatic improvement over the elimination algorithm. Instead of linear dependence we get logarithmic dependence on the number of irrelevant variables. The result may seem restricted in that it is stated for monotone disjunctions only, but one can extend it for learning general disjunctions and conjunctions. In fact, like the perceptron learning algorithm, Winnow can learn arbitrary threshold functions with a mistake bound that depends on the “separation” of the examples [7].

6 The Perceptron Algorithm

The interest in perceptrons stems to some extent from their use in neural modeling, but they also have interesting learning algorithms, and can express a variety of Boolean functions. For example a disjunction \( x_1 \lor x_2 \) can be expressed as \((1 \cdot x_1 + 1 \cdot x_2 \geq 1)\), a conjunction \( x_1 \land x_2 \) can be expressed as \((1 \cdot x_1 + 1 \cdot x_2 \geq 2)\), and the majority function (at least half of the input variables are assigned 1) can be expressed as \((\sum x_i \geq \frac{n}{2})\). The latter does not have small expressions in some Boolean representations such as DNF.

Recall that a linear threshold element over \( n \) variables is represented using an \((n+1)\)-tuple of real numbers, \((w_1, \ldots, w_n, \theta)\). For \( x \in \mathbb{R}^n \), \( f(x) = 1 \) iff \( \sum_{i=1}^{n} w_i x_i \geq \theta \). Let LT be the class of Boolean functions represented as linear threshold elements.\(^4\) We next use the following notation and assumptions.

- Without loss of generality we assume that \( \theta = 0 \). We can always adjust for that by adding a new variable \( x_{n+1} \) (that is always assigned 1) with \( w_{n+1} = -\theta \) to get the same concept.
- We adopt a vector notation so that \( \vec{w} = (w_1, \ldots, w_n) \), the inner product is \( \vec{x} \cdot \vec{y} = \sum x_i y_i \), and \( |\vec{x}|^2 = \vec{x} \cdot \vec{x} = \sum x_i x_i \). Also \( \vec{w} - \vec{x} = (w_1 - x_1, w_2 - x_2, \ldots, w_n - x_n) \), and for a constant \( a \), \( a\vec{w} = (aw_1, aw_2, \ldots, aw_n) \).
- We translate the labels to get values in \( \{-1, 1\} \) as follows. Recall that \( c(\vec{x}) \in \{0, 1\} \). We define \( l(\vec{x}) = (-1)^{1-c(\vec{x})} \) so that 0 is mapped to -1 and 1 is mapped to 1. There is nothing deep in this; it will serve to simplify the notation in arguments.
- We cannot prove a general mistake bound without assumptions. What we need is a notion of “separation” of the data. We assume that there exist \( \delta > 0 \) and \( \vec{v} \), s.t. \( |\vec{v}|^2 = 1 \) with

---

\(^4\)Note that a set of labeled examples induces a set of linear inequalities on weight values. For example if \( n = 3 \) and \( f(101) = 1 \) then we know that \( w_1 + w_3 > \theta \). Therefore, given a set of labeled examples, one can use Linear Programming algorithms (see e.g. [9] for details) to find a solution for these constraints, that is, a set of weights such that all examples are classified correctly. Later in the course we will call such algorithms (that find a compact hypothesis consistent with the examples) Occam algorithms and prove convergence results for them. The advantage in the current argument is that we get an on-line and efficient algorithm.
the following property. For all examples presented in the learning sequence, if \( \vec{x} \) is a positive example then \( \vec{v} \cdot \vec{x} \geq \delta \), and if \( \vec{x} \) is a negative example then \( \vec{v} \cdot \vec{x} \leq -\delta \). Thus, as far as the given examples are concerned there is a strict gap between positive and negative examples. Using the \( l() \) notation we get \( l(\vec{x})(\vec{v} \cdot \vec{x}) \geq \delta \). If this holds we say that the sequence has separation \( \delta \) via \( \vec{v} \).

The requirement that \( |\vec{v}|^2 = 1 \) can be accommodated by scaling \( \vec{v} \) and scaling \( \delta \) accordingly. However this effects the mistake bound for the algorithm.

The Perceptron Algorithm : (learns the class LT)

1. The algorithm maintains a hypothesis \( h \) in the form of a linear threshold element. In the current notation we have \( h(\vec{x}) = 1 \) iff \( \vec{w} \cdot \vec{x} \geq 0 \). The parameters for \( h \) are initialized as follows: \( \theta = 0 \) and \( w_i = 0 \) for all \( i \).

2. Repeat the following on each example in turn:

   (a) On example \( \vec{x} \) the algorithm predicts \( h(\vec{x}) \).

   (b) If a mistake is made such that \( h(\vec{x}) = 1 \) was predicted while \( c(\vec{x}) = 0 \), then \( \vec{w} = \vec{w} - \vec{x} \).

   (c) If a mistake is made such that \( h(\vec{x}) = 0 \) was predicted while \( c(\vec{x}) = 1 \), then \( \vec{w} = \vec{w} + \vec{x} \).

Note the similarity of Winnow to this algorithm. While here we have an additive update after every mistake, in Winnow we scale the weights multiplicatively. Note also that using the \( l() \) notation we can express steps (b) and (c) together as: If a mistake is made then \( \vec{w} = \vec{w} + l(\vec{x})\vec{x} \).

**Theorem 6.1** For any \( \vec{v} \) and \( \delta \) and for any sequence of examples with separation \( \delta \) via \( \vec{v} \), and such that for each example \( \vec{x} \) in the sequence \( |\vec{x}| \leq R \), the number of mistakes made by the algorithm is at most \( \frac{R^2}{\delta^2} \).

**Proof:** Let \( W = |\vec{w} - \frac{R^2}{\delta} \vec{v}|^2 \) and consider how the value of \( W \) changes when the algorithm makes a mistake. Denote by \( W_b, W_a \) and \( \vec{w}_b, \vec{w}_a \) the values of \( W, w \) before and after the mistake respectively, and let \( l = l(\vec{x}) \).

\[
W_a = |\vec{w}_a - \frac{R^2}{\delta} \vec{v}|^2 = |\vec{w}_b - \frac{R^2}{\delta} \vec{v} + l\vec{x}|^2 = |\vec{w}_b - \frac{R^2}{\delta} \vec{v}|^2 + 2l\vec{x} \cdot (\vec{w}_b - \frac{R^2}{\delta} \vec{v}) + l^2|\vec{x}|^2
\]

\[
= |\vec{w}_b - \frac{R^2}{\delta} \vec{v}|^2 + 2l(\vec{x} \cdot \vec{w}_b) - 2\frac{R^2}{\delta} l(\vec{x} \cdot \vec{v}) + l^2|\vec{x}|^2 \leq W_b - 2\frac{R^2}{\delta} \delta + R^2 = W_b - R^2
\]

where in the last inequality use the following: (1) Since a mistake was made \( l \) and \( \vec{x} \cdot \vec{w}_b \) differ in sign so that \( l(\vec{x} \cdot \vec{w}_b) < 0 \) (replace with 0) (2) by the condition of the theorem \( l(\vec{x} \cdot \vec{v}) > \delta \) (replace with \( R^2 \)) (3) for all examples \( |\vec{x}|^2 \leq R^2 \) (replace with \( R^2 \)).

Now since \( |\vec{v}|^2 = 1 \) and initially \( \vec{w} = \vec{0} \) we have \( W = \frac{R^4}{\delta^2} \). Since \( W > 0 \) by definition, and it decreases with every mistake there are at most \( \frac{R^2}{\delta^2} \) mistakes.

Note that after every mistake the algorithm adds or subtracts the example vector itself from the weight vector. Since the initial value of the weight vector is \( \vec{0} \), at any point in the algorithm the weight vector can be expressed as a weighted sum of examples. We rewrite the algorithm using this observation.
The Perceptron Algorithm: (dual form)

1. The algorithm maintains a hypothesis $h$ in the form of a linear threshold element. For any (potential) example $\vec{x}$ let $\alpha(\vec{x})$ be the number of mistakes that were made on $\vec{x}$. Initially $\alpha(\vec{x}) = 0$ for every $\vec{x}$.

   Let $M$ be the set of examples on which mistakes were made so far. Initially $M = \emptyset$.

   In the current notation we have $h(\vec{x}) = 1$ iff $\sum_{\vec{z} \in M} l(\vec{z})\alpha(\vec{z})\vec{z} \cdot \vec{x} \geq 0$.

2. Repeat the following on each example in turn:

   (a) On example $\vec{x}$ the algorithm predicts $h(\vec{x})$.

   (b) If a mistake is made then
       $M = M \cup \{\vec{x}\}$ (add $\vec{x}$ to $M$ if not already there) and
       $\alpha(\vec{x}) = \alpha(\vec{x}) + 1$ (update count).

Since we do not maintain the value of the weight vector directly, the dual form of the algorithm is computationally slower than the original (but the prediction sequence is obviously the same). Notice however that we also have that $h(\vec{x}) = 1$ iff $\sum_{\vec{z} \in M} l(\vec{z})\alpha(\vec{z})(\vec{z} \cdot \vec{x}) \geq 0$ and therefore the prediction depends on the sample only via inner products $(\vec{z} \cdot \vec{x})$. We can therefore use the dual form of the algorithm in situations where we do not have a representation of the examples in their desired vector form but we can compute inner product for them in another way. This is the idea of kernel methods (e.g. [4]) which will be discussed later in the course.

7 Bibliographic Notes

This material is based on various articles and lecture notes and partly on lectures by Leslie Valiant at Harvard University. The model of learning in the limit is due to Gold [5]. Our presentation is rather limited in scope and is based on the introduction of [2]. The results for the on-line mistake counting model due to Littlestone are discussed in [7, 1, 10]. The perceptron algorithm due to Rosenblatt [11] is discussed in [8, 10, 4].

References


