Chapter 1

APPROACHES IN MACHINE LEARNING

Jan van Leeuwen
Institute of Information and Computing Sciences, Utrecht University,
Padualaan 14, 3584 CH Utrecht, the Netherlands

Abstract
Machine learning deals with programs that learn from experience, i.e. programs that improve or adapt their performance on a certain task or group of tasks over time. In this tutorial, we outline some issues in machine learning that pertain to ambient and computational intelligence. As an example, we consider programs that are faced with the learning of tasks or concepts which are impossible to learn exactly in finitely bounded time. This leads to the study of programs that form hypotheses that are 'probably approximately correct' (PAC-learning), with high probability. We also survey a number of meta-learning techniques such as bagging and adaptive boosting, which can improve the performance of machine learning algorithms substantially.

Keywords: Machine learning, computational intelligence, models of learning, concept learning, learning in the limit, PAC learning, VC-dimension, meta-learning, bagging, boosting, AdaBoost, ensemble learning.

1. Algorithms that Learn

Ambient intelligence requires systems that can learn and adapt, or otherwise interact intelligently with the environment in which they operate ('situated intelligence'). The behaviour of these systems must be achieved by means of intelligent algorithms, usually for tasks that involve some kind of learning. Here are some examples of typical learning tasks:

- select the preferred lighting of a room,
- classify objects,
- recognize specific patterns in (streams of) images,
- identify the words in handwritten text,

- understand a spoken language,
- control systems based on sensor data,
- predict risks in safety-critical systems,
- detect errors in a network,
- diagnose abnormal situations in a system,
- prescribe actions or repairs, and
- discover useful common information in distributed data.

Learning is a very broad subject, with a rich tradition in computer science and in many other disciplines, from control theory to psychology. In this tutorial we restrict ourselves to issues in *machine learning*, with an emphasis on aspects of algorithmic modelling and complexity.

The goal of machine learning is to design programs that learn and/or discover, i.e. automatically improve their performance on certain tasks and/or adapt to changing circumstances over time. The result can be a 'learned' program which can carry out the task it was designed for, or a 'learning' program that will forever improve and adapt. In either case, machine learning poses challenging problems in terms of algorithmic approach, data representation, computational efficiency, and quality of the resulting program. Not surprisingly, the large variety of application domains and approaches has made machine learning into a broad field of theory and experimentation [Mitchell, 1997].

In this tutorial, some problems in designing *learning algorithms* are outlined. We will especially consider algorithms that learn (or: are trained) *on-line*, from examples or data that are provided one at a time. By a suitable feedback mechanism the algorithm can adjust its hypothesis or the model of 'reality' it has so far, before a next example or data item is processed. The crucial question is how good programs can become, especially if they are faced with the learning of tasks or concepts which are impossible to learn exactly in finite or bounded time.

To specify a learning problem, one needs a precise *model* that describes what is to be learned and how it is done, and what measures are to be used in analysing and comparing the performance of different solutions. In Section 2 we outline some elements of a model of learning that should always be specified for a learning task. In Section 3 we highlight some basic definitions of the theory of learning programs that form hypotheses that are 'probably approximately correct' [Kearns and Vazirani, 1994; Valiant, 1984]. In Section 4 we mention some of the results of this theory. (See also [Anthony, 1997].) In

Section 5 we discuss meta-learning techniques, especially bagging and boosting. For further introductions we refer to the literature [Cristianini and Shawe-Taylor, 2000; Mendelson and Smola, 2003; Mitchell, 1997; Poole *et al*, 1998] and to electronic sources [COLT].

2. Models of Learning

Learning algorithms are normally designed around a particular 'paradigm' for the learning process, i.e. the overall approach to learning. A computational learning model should be clear about the following aspects:

Learner: Who or what is doing the learning. In this tutorial: an algorithm or a computer program. Learning algorithms may be embedded in more general software systems e.g. involving systems of agents or may be embodied in physical objects like robots and ad-hoc networks of processors in intelligent environments.

Domain: What is being learned. In this tutorial: a function, or a concept. Among the many other possibilities are: the operation of a device, a tune, a game, a language, a preference, and so on. In the case of concepts, sets of concepts that are considered for learning are called *concept classes*.

Goal: Why the learning is done. The learning can be done to retrieve a set of rules from spurious data, to become a good simulator for some physical phenomenon, to take control over a system, and so on.

Representation: The way the objects to be learned are represented *c.q.* the way they are to be represented by the computer program. The *hypotheses* which the program develops while learning may be represented in the same way, or in a broader (or: more restricted) format.

Algorithmic technology: The algorithmic framework to be used. Among the many different 'technologies' are: artificial neural networks, belief networks, case-based reasoning, decision trees, grammars, liquid state machines, probabilistic networks, rule learning, support vector machines, and threshold functions. One may also specify the specific learning paradigm or discovery tools to be used. Each algorithmic technology has its own learning strategy and its own range of application. There also are *multi-strategy* approaches.

Information source: The information (training data) the program uses for learning. This could have different forms: positive and negative examples (called *labeled examples*), answers to queries, feedback from certain actions, and so on. Functions and concepts are typically revealed in the form of labeled instances taken from an *instance space* X. One often identifies a concept with the set of all its positive instances, i.e. with a subset of X. An information source may be *noisy*, i.e. the training data may have errors. Examples may be *clustered* before use in training a program.

Training scenario: The description of the learning process. In this tutorial, mostly *on-line learning* is discussed. In an on-line learning scenario, the program is given examples one by one, and it recalculates its hypothesis of what it learns after each example. Examples may be drawn from a random source, according to some known or unknown probability distribution. An on-line scenario can also be *interactive,* in which case new examples are supplied depending on the performance of the program on previous examples. In contrast, in an off-line learning scenario the program receives all examples at once. One often distinguishes between

- *supervised learning:* the scenario in which a program is fed examples and must predict the label of every next example before a teacher tells the answer.
- *unsupervised learning:* the scenario in which the program must determine certain regularities or properties of the instances it receives e.g. from an unknown physical process, all by itself (without a teacher).

Training scenarios are typically finite. On the other hand, in *inductive inference* a program can be fed an unbounded amount of data. In *reinforcement learning* the inputs come from an unpredictable environment and positive or negative feedback is given at the end of every small sequence of learning steps e.g. in the process of learning an optimal strategy.

Prior knowledge: What is known in advance about the domain, e.g. about specific properties (mathematical or otherwise) of the concepts to be learned. This might help to limit the class of hypotheses that the program needs to consider during the learning, and thus to limit its ‘uncertainty’ about the unknown object it learns and to *converge* faster. The program may also use it to *bias* its choice of hypothesis.

Success criteria: The criteria for successful learning, i.e. for determining when the learning is completed or has otherwise converged sufficiently. Depending on the goal of the learning program, the program should be *fit* for its task. If the program is used e.g. in safety-critical environments, it must have reached sufficient accuracy in the training phase so it can decide or predict reliably during operation. A success criterion can be ‘measured’ by means of *test sets* or by theoretical analysis.

Performance: The amount of time, space and computational power needed in order to learn a certain task, and also the quality (accuracy) reached in the process. There is often a trade-off between the number of examples used to train a program and thus the computational resources used, and the capabilities of the program afterwards.

Computational learning models may depend on many more criteria and on specific theories of the learning process.

2.1 Classification of Learning Algorithms

Learning algorithms are designed for many purposes. Learning algorithms are implemented in web browsers, pc's, transaction systems, robots, cars, video servers, home environments and so on. The specifications of the underlying models of learning vary greatly and are highly dependent on the application context. Accordingly, many classifications of learning algorithms exist based on the underlying *learning strategy*, the type of *algorithmic technology* used, the ultimate *algorithmic ability* achieved, and/or the *application domain*.

2.2 Concept Learning

As an example of machine learning we consider *concept learning*. Given a (finite) instance space X, a concept c can be identified with a subset of X or, alternatively, with the Boolean function $c(x)$ that maps instances $x \in X$ to 1 if and only if $x \in c$ and to 0 if and only if $x \notin c$. Concept learning is concerned with retrieving the definition of a concept c of a given concept class C, from a *sample* of positive and negative examples. The information source supplies noise-free instances x and their *labels* $c(x) \in (0,1)$, corresponding to a certain concept c. In the training process, the program maintains a hypothesis $h = h(x)$ for c. The training scenario is an example of on-line, supervised learning:

Training scenario: The program is fed labelled instances $(x, c(x))$ one-by-one and tries to learn the unknown concept c that underlies it, i.e. the Boolean function $c(x)$ which classifies the examples. In any step, when given a next instance $x \in X$, the program first *predicts* a label, namely the label $h(x)$ based on its current hypothesis h. Then it is presented the true label $c(x)$. If $h(x) = c(x)$ then h is right and no changes are made. If $h(i) \neq c(x)$ then h is wrong: the program is said to have made a *mistake*. The program subsequently *revises* its hypothesis h, based on its knowledge of the examples so far.

The goal is to let $h(x)$ become consistent with $c(x)$ for all x, by a suitable choice of learning algorithm. Any correct $h(x)$ for c is called a *classifier* for c.

The number of mistakes an algorithm makes in order to learn a concept is an important measure that has to be minimized, regardless of other aspects of computational complexity.

DEFINITION 1.1 *Let C be a finite class of concepts. For any learning algorithm A and concept $c \in C$, let $M_A(c)$ be the maximum number of mistakes A can make when learning c, over all possible training sequences for the concept. Let $Opt(C) = min_A(max_{c \in C} M_A(c))$, with the minimum taken over all learning algorithms for C that fit the given model.*

$Opt(C)$ is the optimum ('smallest') mistake bound for learning C. The following lemma shows that $Opt(C)$ is well-defined.

LEMMA 1.2 (LITTLESTONE, 1987) $Opt(C) \leq log_2(|C|)$.

PROOF. Consider the following algorithm A. The algorithm keeps a list L of all possible concepts $h \in C$ that are consistent with the examples that were input up until the present step. A starts with the list of all concepts in C. If a next instance x is supplied, A acts as follows:

1 Split L in sublists $L_1 = \{d \in L | d(x) = 1\}$ and $L_0 = \{d \in L | d(x) = 0\}$. If $|L_1| \geq |L_0|$ then A predicts 1, otherwise it predicts 0.

2 If a mistake is made, A deletes from L every concept d which gives x the wrong label, i.e. with $d(x) \neq c(x)$.

The resulting algorithm is called the 'Halving' or 'Majority' algorithm. It is easily argued that the algorithm must have reduced L to the concept to be found after making at most $log_2(|C|)$ mistakes. $\square$

DEFINITION 1.3 (GOLD, 1967) *An algorithm A is said to identify the concepts in C* in the limit *if for every $c \in C$ and every allowable training sequence for this concept, there is a finite m such that A makes no more mistakes after the m^{th} step. The class C is said to be learnable in the limit.*

COROLLARY 1.4 *Every (finite) class of concepts is learnable in the limit.*

3. Probably Approximately Correct Learning

As a further illustration of the theory of machine learning, we consider the learning problem for concepts that are impossible to learn 'exactly' in finite (bounded) time. In general, insufficient training leads to weak classifiers. Surprisingly, in many cases one can give bounds on the size of the training sets that are needed to reach a good *approximation* of the concept, with high probability. This theory of 'probably approximately correct' (PAC) learning was originated by Valiant [Valiant, 1984] in 1984, and is now a standard theme in *computational learning*.

3.1 PAC Model

Consider any concept class C and its instance space X. Consider the general case of learning a concept $c \in C$. A PAC learning algorithm works by learning from instances which are randomly generated upon the algorithm's request by an external source according to a certain (unknown) distribution $\mathcal{D}$ and which are labeled ($+$ or $-$) by an oracle (a teacher) that knows the concept c. The hypothesis h after m steps is a random variable depending on the sample of size m that the program happens to draw during a run. The performance of the algorithm is measured by the bound on m that is needed to have a high probability that h is 'close' to c regardless of the distribution $\mathcal{D}$.

DEFINITION 1.5 *The error probability of h w.r.t. concept c is: $Err_c(h) = Prob(c(x) \neq h(x))$ = 'the probability that there is an instance $x \in X$ that is classified incorrectly by h'.*

Note that in the common case that always $h \subseteq c$, $Err_c(h) = Prob(x \in c \wedge x \neq h)$. If the 'measure' of the set of instances on which h errs is small, then we call h ε-good.

DEFINITION 1.6 *A hypothesis h is said to be ε-good for $c \in C$ if the probability of an $x \in X$ with $c(x) \neq h(x)$ is smaller than ε: $Err_c(h) \leq \varepsilon$.*

Observe that different training runs, thus different samples, can lead to very different hypotheses. In other words, the hypothesis h is a *random variable* itself, ranging over all possible concepts $\in C$ that can result from samples of m instances.

3.2 When are Concept Classes PAC Learnable

As a criterion for successful learning one would like to take: $Err_c(h) \leq \varepsilon$ for every h that may be found by the algorithm, for a predefined tolerance ε. A weaker criterion is taken, accounting for the fact that h is a random variable. Let $Prob_{\mathcal{S}}$ denote the probability of an event taken over all possible samples of m examples. The success criterion is that

$$Prob_{\mathcal{S}}(Err_c(h) \leq \varepsilon) \geq 1 - \delta,$$

for predefined and presumably 'small' tolerances ε and δ. If the criterion is satisfied by the algorithm, then its hypothesis is said to be 'probably approximately correct', i.e. it is 'approximately correct' with probability at least $1 - \delta$.

DEFINITION 1.7 (PAC-LEARNABLE) *A concept class C is said to be* PAC-learnable *if there is an algorithm A that follows the PAC learning model such that* *for every* $0 < \varepsilon, \delta < 1$ *there exists an m such that for every concept* $c \in C$ *and for every hypothesis h computed by A after sampling m times:*

$$Prob_{\mathcal{S}}(\text{ } h \text{ is } \varepsilon\text{-good for } c \text{ }) \geq 1 - \delta,$$

regardless of the distribution $\mathcal{D}$ *over* X.

As a performance measure we use the minimum sample size m needed to achieve success, for given tolerances $\varepsilon, \delta > 0$.

DEFINITION 1.8 (EFFICIENTLY PAC-LEARNABLE) *A concept class C is said to be* efficiently PAC-learnable *if, in the previous definition, the learning algorithm A runs in time polynomial in* $\frac{1}{\varepsilon}$ *and* $\frac{1}{\delta}$ *(and* $\ln |C|$ *if C is finite).*

The notions that we defined can be further specialized, e.g. by adding constraints on the representation of h. The notion of efficiency may then also include a term depending on the size of the representation.

3.3 Common PAC Learning

Let C be a concept class and $c \in C$. Consider a learning algorithm A and observe the 'probable quality' of the hypothesis h that A can compute as a function of the sample size m. Assume that A only considers *consistent* hypotheses, i.e. hypotheses h that coincide with c on all examples that were generated, at any point in time. Clearly, as m increases, we more and more 'narrow' the possibilities for h and thus increase the likelihood that h is ε-good.

DEFINITION 1.9 *After some number of samples m, the algorithm A is said to be* ε-close *if for every (consistent) hypothesis h that is still possible at this stage:* $Err_c(h) \leq \varepsilon$.

Let the total number of possible hypotheses h that A can possibly consider be finite and bounded by H.

LEMMA 1.10 *Consider the algorithm A after it has sampled m times. Then for any* $0 < \varepsilon < 1$*:*

$$Prob_{\mathcal{S}}(\text{ } A \text{ is not } \varepsilon\text{-close }) < He^{-\varepsilon m}.$$

PROOF.
After m random drawings, A fails to be ε-close if there is at least one possible consistent hypothesis h left with $Err_c(h) > \varepsilon$. Changing the perspective slightly, it follows that:

$$Prob_{\mathcal{S}}(\text{ A is not } \varepsilon\text{-close }) =$$

$$= Prob_{\mathcal{S}}(\text{ after } m \text{ drawings there is a consistent } h \text{ with } Err_c(h) > \varepsilon\ ) \leq$$
$$\leq \Sigma_{h \text{ with } Err_c(h) > \varepsilon} Prob_{\mathcal{S}}(\ h \text{ is consistent }) =$$
$$= \Sigma_{h \text{ with } Err_c(h) > \varepsilon} Prob_{\mathcal{S}}(\ h \text{ correctly labels all } m \text{ instances }) \leq$$
$$\leq \Sigma_{h \text{ with } Err_c(h) > \varepsilon} (1-\varepsilon)^m \leq$$
$$\leq \Sigma_{h \text{ with } Err_c(h) > \varepsilon} e^{-\varepsilon m} \leq$$
$$\leq He^{-\varepsilon m},$$

where we use that $(1-t) \leq e^t$. □

COROLLARY 1.11 *Consider the algorithm A after it has sampled m times, with h any hypothesis it can have built over the sample. Then for any* $0 < \varepsilon < 1$*:*

$$Prob_{\mathcal{S}}(\ h \text{ is } \varepsilon\text{-good }) \geq 1 - He^{-\varepsilon m}.$$

4. Classes of PAC Learners

We can now interpret the observations so far. Let C be a finite concept class. As we only consider consistent learners, it is fair to assume that C also serves as the set of all possible hypotheses that a program can consider.

DEFINITION 1.12 (OCCAM-ALGORITHM) *An* Occam-algorithm *is any on-line learning program A that follows the PAC-model such that (a) A only outputs hypotheses h that are consistent with the sample, and (b) the range of the possible hypotheses for A is C.*

The following theorem basically says that Occam-algorithms are PAC-learning algorithms, at least for finite concept classes.

THEOREM 1.13 *Let C be finite and learnable by an Occam-algorithm A. Then C is PAC-learnable by A. In fact, a sample size M with*

$$M > \frac{1}{\varepsilon}(ln\frac{1}{\delta} + ln|C|)$$

suffices to meet the success criterion, regardless of the underlying sampling distribution $\mathcal{D}$*.*

PROOF.

Let C be learnable by A. The algorithm satisfies all the requirements we need. Thus we can use the previous Corollary to assert that after A has drawn m samples,

$$Prob_{\mathcal{S}}(\text{ h is } \varepsilon\text{-good }) \geq 1 - He^{-\varepsilon m} \geq 1 - \delta,$$

provided that $m > \frac{1}{\varepsilon}(ln\frac{1}{\delta} + ln|C|)$. Thus C is PAC-learnable by A. $\square$

The sample size for an Occam-learner can thus remain polynomially bounded in $\frac{1}{\varepsilon}$, $\frac{1}{\delta}$ and $\ln|C|$. It follows that, if the Occam-learner makes only polynomially many steps per iteration, then the theorem implies that C is even *efficiently* PAC-learnable.

While for many concept classes one can show that they are PAC-learnable, it appears to be much harder sometimes to prove efficient PAC-learnability. The problem even hides in an unexpected part of the model, namely in the fact that it can be *NP*-hard to actually determine a hypothesis (in the desired representation) that is consistent with all examples from the sample set.

Several other versions of PAC-learning exist, including versions in which one no longer insists that the probably approximate correctness holds under every distribution $\mathcal{D}$.

4.1 Vapnik-Chervonenkis Dimension

Intuitively, the more complex a concept is, the harder it will be for a program to learn it. What could be a suitable notion of complexity to express this. Is there a suitable characteristic that marks the complexity of the concepts in a concept class C. A possible answer is found in the notion of Vapnik-Chervonenkis dimension, or simply VC-dimension.

DEFINITION 1.14 *A set of instances $S \subseteq X$ is said to be 'shattered' by concept class C if for every subset $S' \subseteq S$ there exists a concept $c \in C$ which separates S' from the rest of S, i.e. such that*

$$c(x) = \begin{cases} + & \text{if } x \in S', \\ - & \text{if } x \in S - S'. \end{cases}$$

DEFINITION 1.15 (VC-DIMENSION) *The VC-dimension of a concept class C, denoted by $VC(C)$, is the cardinality of the largest finite set $S \subseteq X$ that is shattered by C. If arbitrarily large finite subsets of X can be shattered, then $VC(C) = \infty$.*

VC-dimension appears to be related to the complexity of learning. Here is a first connection. Recall that $Opt(C)$ is the minimum number of mistakes that any program must make in the worst-case, when it is learning C in the limit. VC-dimension plays a role in identifying hard cases: it is lowerbound for $Opt(C)$.

THEOREM 1.16 (LITTLESTONE, 1987) *For any concept class C: $VC(C) \leq Opt(C)$.*

VC-dimension is difficult, even NP-hard to compute, but has proved to be an important notion especially for PAC-learning. Recall that finite concept classes that are learnable by an Occam-algorithm, are PAC-learnable. It turns out that this holds for *infinite* classes also, provided their VC-dimension is finite.

THEOREM 1.17 (VAPNIK, BLUMER ***et al.***) *Let C be any concept class and let its VC-dimension be $VC(C) = d < \infty$. Let C be learnable by an Occam-algorithm A. Then C is PAC-learnable by A. In fact, a sample size M with*

$$M > \frac{\gamma}{\varepsilon}(\ln\frac{1}{\delta} + d\ln\frac{1}{\varepsilon})$$

suffices to meet the success criterion, regardless of the underlying sampling distribution $\mathcal{D}$, for some fixed constant $\gamma > 0$.

VC-dimension can also be used to give a lowerbound on the required sample size for PAC-learning a concept class.

THEOREM 1.18 (EHRENFEUCHT ***et al.***) *Let C be a concept class and let its VC-dimension be $VC(C) = d < \infty$. Then any PAC-learning algorithm for C requires a sample size of at least $M = \Omega(\frac{1}{\varepsilon}(log\frac{1}{\delta} + d))$ to meet the success criterion.*

5. Meta-Learning Techniques

Algorithms that learn concepts may perform poorly because e.g. the available training (sample) set is small or better results require excessive running times. Meta-learning schemes attempt to turn weak learning algorithms into better ones. If one has several weak learners available, one could apply all of them and take the best classifier that can be obtained by combining their results. It might also be that only one (weak) learning algorithm is available. We discuss two meta-learning techniques: bagging, and boosting.

5.1 Bagging

Bagging [Breiman, 1996] stands for '*b*ootstrap *agg*regat*ing*' and is a typical example of an *ensemble* technique: several classifiers are computed and combined into one. Let X be the given instance (sample) space. Define a bootstrap sample to be any sample X' of some fixed size n obtained by sampling X uniformly at random *with* replacement, thus with duplicates allowed. Applications normally have $n = |X|$. Bagging now typically proceeds as follows, using X as the instance space.

For $s = 1, \ldots, b$ do:

- construct a bootstrap sample X_s
- train the base learner on the sample space X_s

- let the resulting hypothesis (concept) be $h_s(x) : X \to \{-1,+1\}$.

Output as 'aggregated' classifier:

$h_A(x) =$ the majority vote of the $h_s(x)$ for $s = 1 \ldots b$.

Bagging is of interest because bootstrap samples can avoid 'outlying' cases in the training set. Note that an element $x \in X$ has a probability of only $1 - (1 - \frac{1}{n})^n \approx 1 - \frac{1}{e} \approx 63\%$ of being chosen into a given X_s. Other bootstrapping techniques exist and, depending on the on the application domain, other forms of aggregation may be used. Bagging can be very effective, even for small values of b (up to 50).

5.2 Boosting Weak PAC Learners

A 'weak' learning algorithm may be easy to design and quickly trained, but it may have a poor expected performance. Boosting refers to a class of techniques for turning such algorithms into arbitrarily more accurate ones.

Boosting was first studied in the context of PAC learning [Schapire, 1990]. Suppose we have an algorithm A that learns concepts $c \in C$, and that has the property that for some $\varepsilon < \frac{1}{2}$ the hypothesis h that is produced always satisfies $Prob_{\mathcal{S}}($ h is ε-good for c $) \geq \gamma$, for some 'small' $\gamma > 0$. One can boost A as follows. Call A on the same instance space k times, with k such that $(1 - \gamma)^k \leq \frac{\delta}{2}$. Let h_i denote the hypothesis generated by A during the i-th run. The probability that *none* of the hypotheses h_i found is ε-good for c is at most $\frac{\delta}{2}$. Consider $h_1, \ldots, h_k$ and test each of them on a sample of size m, with m chosen large enough so the probability that the *observed* error on the sample is not within ε from $Err_c(h_i)$ is at most $\frac{\delta}{2k}$, for each i. Now output the hypothesis $h = h_i$ that makes the *smallest* number of errors on its sample. Then the probability that h is not 2ε-good for c is at most: $\frac{\delta}{2} + k \cdot \frac{\delta}{2k} = \delta$. Thus, A is automatically boosted into a learner with a much better confidence bound. In general, one can even relax the condition on ε.

DEFINITION 1.19 (WEAK PAC-LEARNABLE) *A concept class C is said to be* weakly PAC-learnable *if there is an algorithm A that follows the PAC learning model such that*

> *for some polynomials p, q and $0 < \varepsilon_0 = \frac{1}{2} - \frac{1}{p(n)}$ there exists an m such that for every concept $c \in C$ and for every hypothesis h computed by A after sampling m times:*

$$Prob_{\mathcal{S}}(\text{ } h \text{ is } \varepsilon_0\text{-good for } c \text{ }) \geq \frac{1}{q(n)},$$

regardless of the distribution $\mathcal{D}$ over X.

THEOREM 1.20 (SCHAPIRE) *A concept class is (efficiently) weakly PAC-learnable if and only if it is (efficiently) PAC-learnable.*

A different boosting technique for weak PAC learners was given by Freund [Freund, 1995] and also follows from the technique below.

5.3 Adaptive Boosting

If one assumes that the distribution $\mathcal{D}$ over the instance space is not fixed and that one can 'tune' the sampling during the learning process, one might use training scenarios for the weak learner where a larger weight is given to examples on which the algorithm did poorly in a previous run. (Thus outlyers are not circumvented, as opposed to bagging.) This has given rise to the '*ada*ptive *boost*ing' or AdaBoost algorithm, of which various forms exist (see e.g. [Freund and Schapire, 1997; Schapire and Singer, 1999]). One form is the following:

> Let the sampling space be $Y = \{(x_1, c_1), \ldots (x_n, c_n)\}$ with $x_i \in X$ and $c_i \in \{-1, +1\}$ (c_i is the label of instance x_i according to concept c).
>
> Let $\mathcal{D}_1(i) = \frac{1}{n}$ (the uniform distribution).
>
> For $s = 1, \ldots, T$ do:
>
> – train the weak learner while sampling according to distribution $\mathcal{D}_s$
>
> – let the resulting hypothesis (concept) be h_s
>
> – choose α_s (we will later see that $\alpha_s \geq 0$)
>
> – update the distribution for sampling
>
> $$\mathcal{D}_{s+1}(i) \leftarrow \frac{\mathcal{D}_s(i)e^{-\alpha_s c_i h_s(x_i)}}{Z_s}$$
>
> where Z_s is a normalization factor chosen so $\mathcal{D}_{s+1}$ is a probability distribution on X.
>
> Output as final classifier: $h_B(x) = sign(\sum_{s=1}^{T} \alpha_s h_s(x))$.

The AdaBoost algorithm contains weighting factors α_s that should be chosen appropriately as the algorithm proceeds. Once we know how to choose them, the values of $Z_s = \sum_{i=1}^{n} \mathcal{D}_s(i)e^{-\alpha_s c_i h_s(x_i)}$ follow inductively. A key property is the following bound on the error probability $Err_{uniform}(h_B)$ of $h_B(x)$.

LEMMA 1.21 *The error in the classifier resulting from the AdaBoost algorithm satisfies:*

$$Err_{uniform}(h_B) \leq \prod_{s=1}^{T} Z_s.$$

PROOF.

By induction one sees that

$$\mathcal{D}_{T+1}(i) = \mathcal{D}_1 \frac{e^{-\sum_s \alpha_s c_i h_s(x_i)}}{\prod_s Z_s} = \frac{e^{-c_i \sum_s \alpha_s h_s(x_i)}}{n \cdot \prod_s Z_s},$$

which implies that

$$\tfrac{1}{n} \cdot e^{-c_i \sum_s \alpha_s h_s(x_i)} = (\textstyle\prod_{s=1}^{T} Z_s)\mathcal{D}_{T+1}(i).$$

Now consider the term $\sum_s \alpha_s h_s(x_i)$, whose sign determines the value of $h_B(x_i)$. If $h_B(x_i) \neq c_i$, then $c_i \cdot \sum_s \alpha_s h_s(x_i) \leq 0$ and thus $e^{-c_i \sum_s \alpha_s h_s(x_i)} \geq 1$. This implies that

$$Err_{uniform}(h_B) = \tfrac{1}{n}|\{i | h_A(x_i) \neq c_i\}| \leq \tfrac{1}{n}\textstyle\sum_i e^{-c_i \sum_s \alpha_s h_s(x_i)} = \sum_i (\prod_{s=1}^{T} Z_s)\mathcal{D}_{T+1}(i) = \prod_{s=1}^{T} Z_s.$$

□

This result suggests that in every round, the factors α_s must be chosen such that Z_s is minimized. Freund and Schapire [Freund and Schapire, 1997] analysed several possible choices. Let $\varepsilon_s = Err_{\mathcal{D}_s}(h_s) = Prob_{\mathcal{D}_s}(h_s(x) \neq c(x))$ be the error probability of the s-th hypothesis. A good choice for α_s is

$$\alpha_s = \frac{1}{2}\ln\left(\frac{1-\varepsilon_s}{\varepsilon_s}\right).$$

Assuming, as we may, that the weak learner at least guarantees that $\varepsilon_s \leq \frac{1}{2}$, we have $\alpha_s \geq 0$ for all s. Bounding the Z_s one can show:

THEOREM 1.22 (FREUND AND SCHAPIRE) *With the given choice of* α_s, *the error probability in the classifier resulting from the AdaBoost algorithm satisfies:*

$$Err_{uniform}(h_B) \leq e^{-2\sum_s(\frac{1}{2}-\varepsilon_s)^2}.$$

Let $\varepsilon_s < \frac{1}{2} - \theta$ for all s, meaning that the base learner is guaranteed to be at least slightly better than fully random. In this case it follows that $Err_{uniform}(h_B) \leq e^{-2\theta^2 T}$ and thus AdaBoost gives a result whose error probability decreases exponentially with T, showing it is indeed a boosting algorithm.

The AdaBoost algorithm has been studied from many different angles. For generalizations and further results see [Schapire, 2002]. In recent variants one attempts to reduce the algorithm's tendency to overfit [Kwek and Nguyen, 2002]. Breiman [Breiman, 1999] showed that AdaBoost is an instance of a larger class of '*a*daptive *r*eweighting and *c*ombin*ing*' (arcing) algorithms and gives a game-theoretic argument to prove their convergence. Several other adaptive boosting techniques have been proposed, see e.g. Freund [Freund, 2001]. An extensive treatment of ensemble learning and boosting is given by e.g. [Meir and Ratsch, 2003].

6. Conclusion

In creating intelligent environments, many challenges arise. The supporting systems will be 'everywhere' around us, always connected and always 'on', and they permanently interact with their environment, influencing it and being influenced by it. Ambient intelligence thus leads to the need of designing programs that learn and adapt, with a multi-medial scope. We presented a number of key approaches in machine learning for the design of effective learning algorithms. *Algorithmic learning theory* and *discovery science* are rapidly developing. These areas will contribute many invaluable techniques for the design of ambient intelligent systems.

References

M. Anthony. Probabilistic analysis of learning in artificial neural networks: the PAC model and its variants. In: *Neural Computing Surveys* Vol 1, 1997, pp. 1-47 (see also: http://www.icsi.berkeley.edu/ jagota/NCS).

A. Blumer, A. Ehrenfeucht, D. Haussler, and M.K. Warmuth. Learnability and the Vapnik-Chervonenkis dimension. *Journal of the ACM* 36 (1989) 929-965.

L. Breiman. Bagging predictors. *Machine Learning* 24 (1996) 123-140.

L. Breiman. Prediction games and arcing algorithms. *Neural Computation* 11 (1999) 1493-1517.

COLT. Computational learning theory resources. *website at* http://www.learningtheory.org.

N. Cristianini, J. Shawe-Taylor. *Support vector machines and other kernel-based learning methods*. Cambridge University Press, Cambridge (UK), 2000.

A. Ehrenfeucht, D. Haussler, M. Kearns, and L. Valiant. A general lower bound on the number of examples needed for learning. *Information and Computation* 82 (1989) 247-261.

Y. Freund. Boosting a weak learning algorithm by majority. *Information and Computation* 121 (1995) 256-285.

Y. Freund. An adaptive version of the boost by majority algorithm. *Machine learning* 43 (2001) 293-318.

Y. Freund, R.E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and Systems Sciences* 55 (1997) 119-139.

E.M. Gold. Language identification in the limit. *Information and Control* 10 (1967) 447-474.

M.J. Kearns and U.V. Vazirani. *An introduction to computational learning theory*. The MIT Press, Cambridge, MA, 1994.

S. Kwek, C. Nguyen. *i*Boost: boosting using an *i*nstance-based exponential weighting scheme. In: T. Elomaa, H. Mannila, and H. Toivonen (Eds.), *Machine Learning: ECML 2002*, Proc. 13th European Conference, Lecture Notes in Artificial Intelligence vol 2430, Springer-Verlag, Berlin, 2002, pp. 245-257.

N. Littlestone. Learning quickly when irrelevant attributes abound: a new linear-threshold algorithm. *Machine Learning* 2 (1987) 285 - 318.

R. Meir and G. Ratsch. An introduction to boosting and leveraging. In: S. Mendelson and A.J. Smola (Eds), *ibid*, pp 118-183.

S. Mendelson, A.J. Smola (Eds). *Advanced lectures on machine learning*. Lecture Notes in Artificial Intelligence vol 2600, Springer-Verlag, Berlin, 2003.

T.M. Mitchell. *Machine learning*. WCB/McGraw-Hill, Boston, MA, 1997.

G. Paliouras, V. Karkaletsis, and C.D. Spyropoulos (Eds.). *Machine learning and its applications*, Advanced Lectures. Lecture Notes in Artificial Intelligence vol 2049, Springer-Verlag, Berlin, 2001.

D. Poole, A. Mackworth, and R. Goebel. *Computational intelligence - a logical approach*. Oxford University Press, New York, 1998.

R.E. Schapire. The strength of weak learnability. *Machine learning* 5 (1990) 197-227.

R.E. Schapire. The boosting approach to machine learning - An overview. In: *MSRI Workshop on Nonlinear Estimation and Classification*, 2002 (available at: http://www.research.att.com/ schapire/publist.html).

R.E. Schapire, Y. Singer. Improved boosting algorithms using confidence-rated predictions. *Machine Learning* 37 (1999) 297-336.

M. Skurichina, R.P.W. Duin. Bagging, boosting and the random subspace method for linear classifiers. *Pattern Analysis & Applications* 5 (2002) 121-135.

L.G. Valiant. A theory of the learnable. Comm. ACM 27 (1984) 1134-1142.