Capacity Control in Boosting using a p-Convex Hull

Final Year Honours Thesis
Department of Engineering
The Australian National University

Jeremy Peter Barnes

Student ID 3015334

October 25, 1999

Supervised by Dr Bob Williamson

Abstract

AdaBoost is a machine learning algorithm that generates a hypothesis by combining a series of so-called weak hypotheses generated by another algorithm. The performance of AdaBoost is good to exceptional, particularly on low-noise data. However it has been observed that AdaBoost can generate overly complex hypotheses that overfit noisy data.

We consider modifying AdaBoost to allow the complexity of its combined hypotheses to be explicitly controlled via a "p" parameter. The resulting "p-boosting" algorithm is constrained to generate only combinations of weak hypotheses that lie on the p-convex hull of its weak hypotheses.

Several algorithms based upon this concept are developed. Simulations indicate that the best algorithm outperforms AdaBoost by a modest amount (at best 18% test error vs 25% for AdaBoost) on a low-dimensional noisy dataset. The generalisation ability over other noisy datasets was slightly improved on AdaBoost; AdaBoost performed better on low-noise datasets. A number of operational difficulties were observed, including very long training times. These difficulties appear to be due to details of the gradient descent optimisation technique used internally, and solutions are proposed for those where the cause was apparent.

The p-convex hull approach appears to be a useful improvement to the already excellent AdaBoost algorithm. This approach is complementary to other AdaBoost variants also designed to improve performance on noisy datasets, allowing the possibility of a combined approach superior to either.

Contents

List of Figures

Acknowledgements

I would like to express my appreciation for the contributions made to this project by the following people:

- *Peter Bartlett* for very patiently straightening out the many inconsistencies and gaps in my knowledge, and then showing me a much easier avenue of enquiry.
- *Gunnar Rätsch* for giving me another perspective on my proposed algorithms.
- *Mum and Dad* for proofreading my manuscript and providing me with many helpful suggestions.
- The maintainers of the datasets that I have used.
- *Bob Williamson* for an interesting, challenging and rewarding project; and for putting far more time and effort into supervision than obliged to.

Notation and Acronyms

Notation

Symbols	Meaning	Section
$\mathbf{x} \in \mathcal{I}$	$\mathbf{x}$ is a sample	fig 2.1
$y \in \mathcal{O}$	y is a label	”
$X = ((\mathbf{x}_1, y_1), \ldots))$	Labeled samples, length m (training dataset)	2.1
$\mathcal{I}, \mathcal{O}$	Domain and range (input and output space; usually $\mathcal{O} \in \{\pm 1\}$)	2.1.1
$h(\cdot) \in \mathcal{H} : \mathcal{I} \to \mathcal{O}$	A hypothesis (classifier)	2.1
$\mathcal{H}$	A set of hypotheses h	2.1.2
$f(\cdot) \in \mathcal{F} : \mathcal{I} \to \mathbb{R}$	A non-thresholded hypothesis; usually $h(\cdot) = \operatorname{sign}(f(\cdot))$	2.1.3
$\mathcal{F}$	A set of hypotheses f (also $\mathcal{H} = \operatorname{sign}(\mathcal{F})$)	”
$\mathbb{W} : \mathcal{I}^m \to (\mathcal{I} \to \mathcal{O})$	Learning machine ($\mathbb{W}(X) = h$)	”
$q = Q(y, \hat{y})$	Loss function	2.2.1
$R(h)$	True or expected risk	2.2.2
$R_{\underset{X}{\text{emp}}}(h)$	Empirical risk (samples X implicit if not specified)	2.2.3
$R^w_{\text{emp}}(h)$	Weighted empirical risk (training error)	2.2.4
$R^\gamma_{\text{emp}}(h)$	Margin risk	2.2.5
$\operatorname{VCdim}(\cdot)$	VC dimension	2.3.2
$\mathcal{N}(\mathcal{X}, \epsilon, X)$	Covering number at scale ϵ of $\mathcal{X}$ over samples X	2.3.3
$\mathcal{N}(\mathcal{X}, \epsilon, m)$	Uniform covering number of $\mathcal{X}$ at scale ϵ over m points	”
$\operatorname{co}_p(\mathcal{X})$, $\operatorname{co}(\mathcal{X})$	p-convex hull of set $\mathcal{X}$; $p = 1$ assumed if not specified	2.3.5
$\mathbb{B}$, $\overset{\mathbb{B}}{\Longrightarrow}$	AdaBoost, action of AdaBoost	fig 3.1
$F(\mathbf{x})$, $H(\mathbf{x})$	Boosting hypothesis, non-thresholded & thresholded, $H(\cdot) = \operatorname{sign}(F(\cdot))$	”
$b_1, \ldots, b_t$	AdaBoost classifier weights	”
$w_{t,1}, \ldots, w_{t,m}$	AdaBoost sample weights at iteration t	”

A hat ($\hat{}$) means “estimate”. An asterisk (*) means “optimal” in some sense.

Acronyms

Acronym	Meaning	First introduced
SLT	Statistical Learning Theory	2.3
ERM	Empirical Risk Minimisation	2.2.6
SRM	Structural Risk Minimisation	2.4.1
VC dimension	Vapnik-Chervonenkis dimension	2.3.2
AdaBoost	Adaptive Boosting algorithm	chapter 3

Chapter 1

Introduction

The project undertaken was an investigation of a method of overcoming a known problem (*overfitting*) of a particular machine learning algorithm (the *AdaBoost*[1] algorithm). The result is a series of machine learning algorithms called *p-boosting algorithms*. These algorithms are developed, analysed and tested in later chapters.

Part I: Background

This introduction provides an overview of the rest of the thesis, and covers the bigger picture, describing the field of machine learning and its applications to practical problems.

The next three chapters rapidly narrow the focus. Chapter 2 describes the field of *Statistical learning theory* (SLT)[2], which provides much of the theoretical foundation of (but by no means encompasses) the field of machine learning. Assumptions of *binary problems* and *classification problems* further restrict the scope of the investigation.

With the necessary background in place, the *AdaBoost* algorithm itself is the subject of chapter 3. This algorithm forms the starting point for the original work that is considered in further chapters. The *problem* of overfitting in boosting is investigated, as a prelude to the *solutions* investigated in the second part of the thesis.

Part II: New work

The key inventive step of the project is the use of a *p-convex hull* to reduce the problem of overfitting. Chapter 4 investigates the theoretical justification behind this line of reasoning in some detail. These abstract ideas are then developed into concrete algorithms, and properties of these algorithms are considered.

Chapter 5 describes how these algorithms were tested. Chapter 6 details the main results of the project, and discusses their significance and relation to the theory. A conclusion (chapter 7) completes the thesis, evaluating the results obtained in the context of the project and in the broader scope of the field of machine learning. Suggestions of avenues further enquiry which may prove fruitful are also made.

[1] Adaptive Boosting.

[2] A table of acronyms is provided in the preface (p. viii).

1.1 The scheme of things

This section provides an overview of the context of this project, starting very broadly and rapidly focusing on the immediate background.

1.1.1 Artificial intelligence

The broad subject area in which this work is contained is often described as *artificial intelligence*. This field essentially contains our efforts to make computers act in a similar manner to humans, encompassing a large body of history, philosophy and knowledge (see, for example [18]). We will ignore most of this, and concentrate on efforts to make computers "learn".

Early work in this field (from the 1950s to the 1980s) was based on the idea that intelligence can be emulated with a large set of rules. The systems generated as a result, known as *expert systems*, were huge databases of human-generated rules that were meant to represent the complete knowledge of a human expert over a particular problem domain. These systems were reasonably successful at first, but became unwieldy as the number of rules increased (a large amount of effort is required to generate even the 1000 rules that were typically used in large expert systems circa 1980).

The problem was more practical than theoretical: systems with large numbers of rules can theoretically learn any learnable problem to a desired accuracy; it is *generating* the rules that is hard. The next step, obvious in hindsight, was to invent machines that could generate their own rules; machines that could *learn*.

1.1.2 Machine learning

Learning machines came in two flavours. *Supervised learners* are shown examples of some kind of relationship, along with the "right" answers (generated from some form of supervisor) and attempt to learn a relationship such that their answer always matches that of the supervisor. An example is trying to learn the likely outcome of a court case based upon details of similar cases (the "supervisor" here is the information on the outcomes of cases on file). All algorithms discussed in this thesis are supervised algorithms. *Unsupervised algorithms* are given a set of data and asked to learn a pattern in the data (for example, identifying interesting clumps of stars from telescope images).

1.1.3 Statistical learning theory

Machine learning rests upon the theoretical foundation of statistical learning theory (SLT). It provides a body of knowledge that allows bounds on the performance of learning algorithms to be generated[3]. Results from statistical learning theory enable us to be confident that learning machines will learn in a manner that is close to optimal.

1.1.4 Voting methods

Boosting is an example of a *voting method*, a relatively new class of algorithm which operates in a bootstrap-like manner by combining many "weak"[4] hypotheses to generate a composite "strong" hypothesis. Their success has been remarkable; as a result of these algorithms the focus of machine

[3] Much of this theory can be attributed to V.N.Vapnik [23].

[4] Strictly, the term "weak" is used in an informal manner only; we call any non-voting method weak. However, it is usually true that these weak algorithms perform significantly worse than voting methods.

learning research in recent years has shifted from the classical algorithms (all of which are more or less equal when boosted) to developments in voting methods.

These algorithms were initially observed to be almost immune to many of the problems of overfitting (see section 1.2 below). Recent results have indicated that this is unfortunately not the case.

1.1.5 A real-world application

Consider a telephone company that is interested in predicting whether its customers are likely to "churn" (change telephone companies) in the near future. Such a telephone company will have a detailed database with customer statistics such as frequency of telephone usage, number of service calls, etc. This data can be used to train a learning machine. The resultant learning machine can then be used to predict whether current customers are likely to churn (classify customers into "churn" and "not churn" categories). The company can target these likely customers with special offers or improved service, thereby retaining their custom.

1.2 Issues in machine learning

Machine learning is a mathematically hard (ill-posed) problem, and there are many difficult issues involved. A real-world example of many of these problems[5] concerns efforts by the US army to construct a learning machine to detect tanks from photographs. The training data was two sets of photographs of terrain; one set including tanks and the other without. Each set of photographs was taken at a different time during the day. When the learning machine was trained, it learned to detect differences in the sun position rather than the presence of tanks! Although the learning machine could classify the training data correctly, its generalisation ability was poor.

Overfitting, simply put, is learning the specifics of a process too well to be able to apply the knowledge to a more general problem. In the context of machine learning, overfitting occurs when the learning machine adapts to peculiarities or noise in the input data, to the detriment of generalisation ability. (There are also issues concerning what is "good" training data–in this thesis, it is assumed that the training data is given and thus uncontrollable).

Statistical learning theory gives us a theoretical reason for overfitting, and also an insight into how it may be avoided. In particular, by limiting the complexity of the *set* of hypotheses that the learning machine may generate we can avoid overfitting. This technique is known as *capacity control*.

1.3 Original work

The original work of this project (first considered in [25]) is contained in chapters 4 through 7. The motivation behind this work is an observation on a theoretical bound on generalisation ability; in particular on a method of reducing overfitting in the AdaBoost algorithm by using p-convex hulls. This concept is first developed into several practical algorithms (chapter 4); these algorithms are then tested against the original AdaBoost algorithm and for compliance with the theory (chapters 5 through 7).

[5] Part of machine learning folklore.

Chapter 2

Statistical learning theory

This chapter provides a general theoretical framework upon which more specific theory in chapter 3 is constructed. The central questions that this chapter answers are: "What is the *best* learning machine for a particular problem?", "How do we choose it?", and "How well does it perform?".

Some caution is required in using the word "best". Learning machines are *mathematical models* of the systems they are trying to learn. As with any mathematical model they are necessarily an approximation of the underlying system. Thus, when we say "best" we mean "an approximation which suits our application well". Throughout this chapter, several quantitative measures of how well a model suits an application are developed. These measures are used to define procedures for choosing the "best" learning machine–however it is an important point that none of these is the "best" in any absolute sense. All are subjective to an extent, as they all rely on *a priori* assumptions.

We begin with a formal overview of the problem and notation used. Section 2.2 then considers how to compare learning machines, and develops an inductive procedure (*Empirical Risk Minimisation*–ERM) for selecting the "best" one. Section 2.3 develops bounds on how closely learning machines generated by ERM approach the optimal theoretical performance. Finally, section 2.4 considers the problem of overfitting and how to avoid this phenomenon by limiting the complexity of our learning machines. This leads to another inductive procedure, known as *structural risk minimisation* (SRM).

2.1 Formulation of the machine learning problem

Heuristically, a *learning machine* is

> ...an algorithm (usually implemented in software) that estimates an unknown mapping (dependency) between a system's inputs and outputs from the available data, namely from known (input, output) samples. [7]

This description motivates the formal definition of supervised learning, which is illustrated in figure 2.1. The *sample generator* produces samples $\mathbf{x} \in \mathcal{I}$ drawn from a fixed (but unknown) probability distribution $p(\mathbf{x})$ over $\mathcal{I}$[1], where $\mathcal{I}$ is the domain, or "input space", of the machine learning problem.

The *supervisor* h_{sup} implements the unknown dependency which the learning machine is trying to learn. Given a sample $\mathbf{x}$, it returns a label $y^* = h_{\text{sup}}(\mathbf{x})$ according to a *fixed but unknown* probability distribution D on $\mathcal{I} \times \mathcal{O}$, where $\mathcal{O}$ is the range or "output space" of the machine learning problem (all $y^* \in \mathcal{O}$)[2].

[1] In practice, these samples are normally *observed* rather than *generated*.

[2] Sometimes, samples are presented to a learning machine in a batch instead of serially. The formulation presented here is sufficiently general to cover this special case.

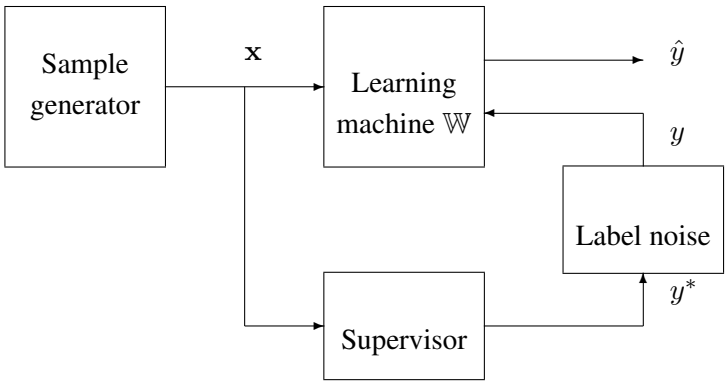


Figure 2.1: Supervised learning

We furthermore allow y^* to be subject to *label noise*, a random process that modifies y^* in some manner (we consider exactly *what* manner shortly) to give y.

The *learning machine* $\mathbb{W}$ receives $\mathbf{x}$ as input and updates its hypothesis h (which is an approximation to D, based on all data seen so far) based on the error $y - h(\mathbf{x})$. Thus, we see that the action of the learning machine is to update a hypothesis; this hypothesis is used to generate the estimates $\hat{y}$. (For a concrete example of a learning machine and the type of hypotheses that it can generate, appendix A describes a very simple learning machine called *decision stumps*.) The goal is for $\mathbb{W}$ to generate a hypothesis h such that $h(x) - y^* = 0$ for *all* samples $\mathbf{x}$, including samples it has not seen[3].

Matters are complicated somewhat by the presence of *label noise*. This noise q is modelled as a probability over all samples that the observed sample is wrong:[4]

$$q = \Pr_{\mathbf{x} \in \mathcal{I}}(y \neq y^*) \tag{2.1}$$

In practical applications, figure 2.1 tells only half of the story. Once training is complete[5], the final hypothesis h^* is used *unsupervised* on further (unseen) input samples. For example, a hypothesis generated by a learning machine that had been trained on the "churn" dataset described in chapter 1 would then be used to determine if new customers were likely to churn. Although this thesis concentrates primarily on *generating* h^*, it is important to keep in mind that h^* may then be used to make predictions on unseen samples. The training process is usually not an end in itself.

2.1.1 Domain and range of learning machines

The symbols $\mathcal{I}$ and $\mathcal{O}$ are used to specify the domain and range of the learning problem: all samples $\mathbf{x} \in \mathcal{I}$; all labels $y \in \mathcal{O}$. $\mathcal{I}$ depends upon number and domain of input variables in the problem; in most examples in this thesis $\mathcal{I} = [0, 1]^d$, where d is 1 or 2. Real world problems typically have much larger domains: the "churn" dataset introduced in section 1.1.5 has nine real, seven integer and two boolean attributes giving $\mathcal{I} = \mathbb{R}^9 \times \mathbb{N}^7 \times \{0, 1\}^2$.

$\mathcal{O}$ is a subset of $\mathbb{R}^o$, where o is the number of output variables. In most applications, there is only one output variable ($o = 1$) or multiple output variables are independent and can each be generated

[3] Further discussion of exactly what constitutes "learning behaviour" is beyond the scope of this thesis; see [1].

[4] Other more complex models of noise are sometimes used (such as *attribute noise* or models based upon full probability density functions). These are beyond the scope of this thesis.

[5] Section 2.4.1 discusses how to tell when training is complete

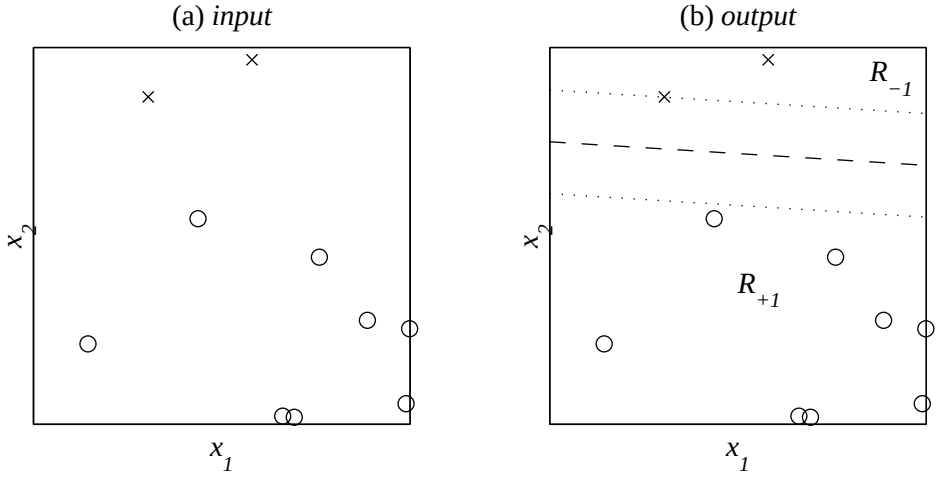


Figure 2.2: The classification problem

Part (a) shows the input X to a learning machine: a set of labelled points in $\mathbb{R}^2 \times \{\pm 1\}$. The $\times$ symbol represents $y = -1$ and the $\circ$ symbol $y = 1$. Part (b) shows the output: a decision boundary (dashed line) generated by a hypothesis $h(\mathbf{x})$. The two regions $\mathcal{S}_{-1}$ and $\mathcal{S}_{+1}$ are also shown. The dotted lines indicate the minimum margin *over the training samples (section 2.1.3). Note that there are many possible hypotheses (decision boundaries) that will correctly classify (separate) all of the input data. Here, $h((x_1, x_2)) = \text{sign}\{(x_2 - b) - mx_1\}$.*

by a separate learning machine. Thus we restrict our attention to $o = 1$.

The distinction between *classification* and *regression* problems is made on the size of $\mathcal{O}$. A learning machine is solving a *regression* problem when $|\mathcal{O}|$ is infinite. Usually, this means that $\mathcal{O}$ is some interval on $\mathbb{R}$. This thesis does not further consider regression problems.

When $|\mathcal{O}| < \infty$ we are solving a *classification* problem. (Learning machines which solve a classification problem are often called *classifiers*). Permissible y values are now a finite number of discrete *categories*:

$$\mathcal{O} = \{y_1, y_2, \ldots, y_o\} \tag{2.2}$$

A classifier operates as an equivalence relation, splitting the input space $\mathcal{I}$ into a set of disjoint regions $\{\mathcal{R}_{y_1} \ldots \mathcal{R}_{y_o}\}$, each of which corresponds to a value of the label. The "decision boundary" (illustrated in figure 2.2) is a useful representation of the boundary between these regions.

For the remainder of this thesis, we restrict our attention to *binary classification problems* where $\mathcal{O} = \{\pm 1\}$. When restricted in this manner, the terms "classifier" and "hypothesis" become equivalent, and are used interchangeably.

2.1.2 Representation of learning machines

As mentioned above, a learning machine $\mathbb{W}$ takes a labelled set of data $X = ((\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m))$, where $x_i \in \mathcal{I}$ and $y_i \in \mathcal{O}$, and outputs a hypothesis $h(\cdot) : \mathcal{I} \to \mathcal{O}$ chosen according to some deterministic procedure. It is often convenient to represent learning machines by the set

$$\mathcal{H} = \{h : (\exists X : \mathbb{W}(X) = h)\} \tag{2.3}$$

This set contains all hypotheses that the learning machine $\mathbb{W}$ could generate; note however that the procedure (algorithm) through which a h is selected from $\mathcal{H}$ is not explicit in this formulation[6].

2.1.3 Alternative formulation for binary classification problems

Many binary classifiers produce their output by thresholding a real valued function. It is often useful to deal with this continuous function directly, rather than with the discrete thresholded version (which contains less information). We write this relationship as

$$h(\mathbf{x}) = \text{sign}\left(f(\mathbf{x})\right) \tag{2.4}$$

where h is the thresholded hypothesis ($h : \mathcal{I} \to \{\pm 1\}$) and the function $f : \mathcal{I} \to \mathbb{R}$. (Note that f is only used *internally* in (2.4); the *external* representation h still has a discrete output). We then define the *margin* of a labeled sample as follows:

Definition 1 (Margin of a sample) *Given a labelled sample $(\mathbf{x}, y)$ and a real-valued hypothesis f, we define the* margin *of the sample as*

$$m_f(\mathbf{x}, y) = yf(\mathbf{x}) \tag{2.5}$$

which is a real valued function, positive if the hypothesis is correct on the given sample and negative otherwise.

In practice, it usually turns out that the magnitude of the margin at a point is an indication of how "confident" the hypothesis is of its classification. A large value of the margin indicates that there is little uncertainty in the classification of the point in question. Thus, we would expect that a hypothesis with large margins would have good generalisation performance. Geometrically, for "well behaved" functions, the distance between a point and the decision boundary will roughly correspond to the magnitude of the margin at that point (see figure 2.2)[7].

We will always call the real-valued hypotheses f and the thresholded hypotheses h, where $h = \text{sign}(f)$. We use similar notation for the sets of hypotheses: $\mathcal{H} = \text{sign}(\mathcal{F}) = \{\text{sign}(f) : f \in \mathcal{F}\}$. Many results later in this thesis make use of the margin formulation; in particular some generalisation performance bounds in section 2.3 are defined in terms of the margins, and we show that the limiting behaviour of the AdaBoost algorithm (chapter 3) is to maximise the minimum margin over a set of training samples.

2.2 Comparing and selecting hypotheses

We now consider how to construct a learning machine–in particular, how it can compare two hypotheses to determine which is "better".

[6] As a notational convenience, when we apply an operation to $\mathcal{H}$ (for example $\text{sign}(\mathcal{H})$), we actually apply that operation to all elements of that set.

[7] The discussion is simplified for the sake of clarity. It is possible to generate margins of arbitrarily large size by scaling f by a constant β. We are actually, therefore, interested in the *normalised margin* $m_F(\mathbf{x}, y)/|f|$ where $|f|$ is (for example) an upper bound over all samples of the magnitude of f. It is a minor difference in this thesis as the algorithms considered all normalise f anyway.

2.2.1 Loss functions

We first introduce the notion of a "loss function". The input to a loss function is an ordered pair (y, y^*) where y is the output of some hypothesis in response to a sample $\mathbf{x}$, and y^* is the "correct" output (that generated by the supervisor–see figure 2.1). The output of the loss function is a real number q which describes how "bad" this estimate is. In symbols, the loss function is written

$$q = Q(y, y^*) \tag{2.6}$$

where $q \in \mathbb{R}$ and $y^*, y \in \mathcal{O}$.

In this thesis we usually use the "misclassification loss function", which is defined as

$$Q(\hat{y}, y) = \begin{cases} 0 & \text{if } \hat{y} = y \\ 1 & \text{if } \hat{y} \neq y \end{cases} \tag{2.7}$$

This loss function is very easy to understand: zero if right, one if wrong. Adding together the loss functions over several samples simply counts the number of mistakes.

2.2.2 True risk

We can use our loss function Q to compare the performance of a number of hypotheses over the entire input space.

Definition 2 (True risk) *We define the* true risk *of a classifier $h(\mathbf{x})$ given a supervisor function $h_{\text{sup}}(\mathbf{x})$ as*

$$R(h) = \int_{\mathcal{I}} Q(h(\mathbf{x}), h_{\text{sup}}(\mathbf{x}))\, p(\mathbf{x})\, d\mathbf{x} \tag{2.8}$$

where Q is the misclassification loss function (2.7) and $p(\mathbf{x})$ is the probability density function of $\mathbf{x}$.

Using the loss function (2.7), the value of R ($0 \leq R \leq 1$) is easily seen to be the proportion of samples that are misclassified. If two classifiers are compared using this metric, then the one with the least probability of making a mistake will have a lower true risk. This is intuitively a sensible measure to use.

The goal of machine learning is generate learning machines that minimise the true risk. Unfortunately, it is impossible in principle to evaluate (2.8) as one does not know either $h(\mathbf{x})$ or $p(\mathbf{x})$. Instead, we know m (possibly noisy) observations $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)$.

In section 2.4 we will describe a method (structural risk minimisation) that comes close to minimising the true risk. However, for the time-being we will consider a measure that we *can* actually calculate: the *empirical risk*.

2.2.3 Empirical risk

The empirical risk is an estimate of true risk, that can be evaluated over a collection of training samples.

Definition 3 (Empirical risk) *Given m observations $X = ((\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m))$ and a hypothesis $h \in \mathcal{H}$, we define the* empirical risk *of h over X as*

$$R_{\text{emp}}(h) = \underset{X}{R_{\text{emp}}}(h) = \frac{1}{m} \sum_{i=1}^{m} Q(h(\mathbf{x}_i), y_i) \tag{2.9}$$

The law of large numbers ensures that as $m \to \infty$, the empirical risk approaches the true risk (assuming that Q is unbiased). Convergence for finite m is discussed in section 2.3.

2.2.4 Weighted empirical risk

The empirical risk can be extended to the case where the samples in X are not equally important (or reliable), by giving each sample a weight w_i.

Definition 4 (Weighted empirical risk) *Given m observations $X = ((\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m))$ a normalised vector of weights $W = (w_1, \ldots, w_m) \in \mathbb{R}^m$ where $\sum_i w_i = 1$ and a hypothesis $h \in \mathcal{H}$, we define the* weighted empirical risk *of h over X as*

$$R^W_{\text{emp}}(h) = \sum_{i=1}^{m} w_i Q(h(\mathbf{x}_i), y_i) \tag{2.10}$$

It is clear that when all samples are equally weighted ($w_i = 1/m$), this definition is equivalent to that of empirical risk. Weighted samples are used by the AdaBoost algorithm described in chapter 3.

2.2.5 Margin risk

We define another risk functional that will be of use later on. Recall from section 2.1.3 that the margin is a real-valued quantity that indicates how confident a classification is. Margins with positive values indicate that the classification of a point was correct.

We use these margins to introduce a stronger form of risk functional. Specifically, we include in our risk values not only those samples that were classified wrongly, but also those samples that were *nearly* classified wrongly.

Definition 5 (Margin risk) *Given a series of labelled training examples $X = ((\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m))$ and a margin $\gamma > 0$, we define the* margin risk *of a hypothesis $h = \text{sign}(f)$ as*

$$R^\gamma_{\text{emp}}(h) = \frac{1}{m} \left|\{i : y_i f(\mathbf{x}_i) \leq \gamma\}\right| \tag{2.11}$$

The margin risk is clearly the proportion of samples with a margin less than or equal to γ. It has the property that $R^\gamma_{\text{emp}}(f)$ is nondecreasing with increasing γ.

2.2.6 Empirical risk minimisation

We have been concerned thus far with a *single* hypothesis h. However the goal of machine learning algorithms is to pick the best h from a set $\mathcal{H}$ of possible hypotheses (specified *a priori*), given a series of training samples X. One way of doing so is the *empirical risk minimisation* inductive principle (ERM).

Definition 6 (Empirical risk minimisation) *Suppose we are given a set $\mathcal{H}$ of hypotheses, a loss function Q, and a set of training data $X = ((\mathbf{x}_1, y_1) \ldots (\mathbf{x}_m, y_m))$. Then the* empirical risk minimisation *inductive principle selects the learning machine $h^* \in \mathcal{H}$ that minimises the empirical risk:*

$$h^* = \underset{h \in \mathcal{H}}{\text{argmin}}\, R_{\text{emp}}(h) \tag{2.12}$$

It is an important detail that the set $\mathcal{H}$ of possible hypotheses is specified *outside* the ERM principle. The selection of this set corresponds to the input of *a priori* knowledge into the learning process—indeed it can be shown that there cannot be a useful completely general (no prior knowledge) learning algorithm[8].

[8] This is sometimes called the "no free lunch" principle.

2.3 Statistical learning theory

We now turn to a fundamental question of machine learning: given a hypothesis $h \in \mathcal{H}$ that was selected via ERM over a set X of training examples X, how well can we expect it to generalise to *unseen* examples? The theory that attempts to answer this question is known as *statistical learning theory* (SLT). This theory is the subject of several texts [23, 7, 2].

2.3.1 Performance bounds over finite $\mathcal{H}$

By treating the probability of an error on a particular sample as a binomial random variable and bounding the tail of this distribution, the probability of an error for a finite hypothesis class ($|\mathcal{H}| < \infty$) can be bounded. This procedure leads to the following theorem.

Theorem 1 (Upper bound for $|\mathcal{H}| < \infty$) *If $h \in \mathcal{H}$ minimises empirical risk on a training set X with m examples, then with probability at least $1 - \delta$*

$$R(h) \leq R_{\text{emp}}(h) + \sqrt{\frac{2}{m} \log\left(\frac{2\,|\mathcal{H}|}{\delta}\right)} \tag{2.13}$$

The requirement that $|\mathcal{H}| < \infty$ is very restrictive in practice; even a class of learning machines parameterised by one real number fails to meet this condition. The *Vapnik-Chervonenkis dimension* has less restrictive requirements, and thus turns out to be a more useful measure.

2.3.2 VC dimension

The VC (Vapnik-Chervonenkis) dimension is another measure of the complexity of a class of hypotheses $\mathcal{H}$ that is useful for many problems. Its definition relies on the following definition:

Definition 7 (Shattering) *Consider a series S of p points $(\mathbf{x}_1, \ldots, \mathbf{x}_p) \in \mathcal{I}^p$. Then a function class $\mathcal{H}$ is said to* shatter *S if and only if there exist functions $h_1, \ldots, h_n \in \mathcal{H}$ such that each of the $n = 2^p$ possible classifications of the points are produced:*

$$\{(h_i(\mathbf{x}_1), \ldots, h_i(\mathbf{x}_p)) : i = 1, \ldots, n\} = \{\pm 1\}^p \tag{2.14}$$

It is an important feature of the definition that *every* possible classification of the points must produced. If there exists even one classification that cannot be produced, then the definition is *not* satisfied. Figure 2.3 provides an example of the shattering of points in $\mathbb{R}^2$ plane by the set of straight lines in $\mathbb{R}^2$.

Definition 8 (VC dimension) *Given a function class $\mathcal{H}$ on $\mathcal{I} \to \mathcal{O}$, the VC dimension of the class is defined as*

$$\text{VCdim}(\mathcal{H}) = \max_d : (\exists S = \{\mathbf{x}_1, \ldots, \mathbf{x}_d\} \text{where } \mathcal{H} \text{ shatters } S) \tag{2.15}$$

where the set S contains no repetitions ($i \neq j \Rightarrow \mathbf{x}_i \neq \mathbf{x}_j$).

In other words, $\text{VCdim}(\mathcal{H})$ is the largest number d of distinct points that *can* be shattered by $\mathcal{H}$. Note that this definition only requires that there be *one* set of points that can be shattered; not that all sets of points may be shattered. As an extension of the ideas in figure 2.3, the VC dimension of lines in $\mathbb{R}^d$ is $d + 1$.

The VC dimension plays a role in a bound on the generalisation performance of a hypothesis selected via ERM.

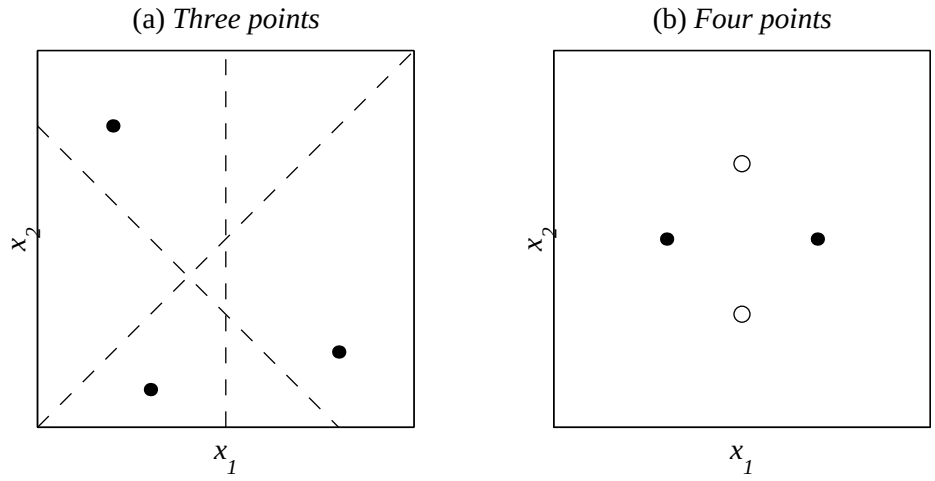


Figure 2.3: Shattering a set of points

The closed and open circles are data points; the lines are decision boundaries. Part (a) shows how three points on $\mathbb{R}^2$ can be shattered by linear decision boundaries. Part (b) shows that four points cannot be shattered by a linear decision boundary; in particular no straight line can separate the $\bullet$ points from the $\circ$ points (try it!)

Theorem 2 (VC upper bound [1]) *Let $\mathcal{H}$ be a class of functions mapping from a set $\mathcal{I}$ to $\mathcal{O} = \{\pm 1\}$ and having VC-dimension d. Then for any probability distribution on $\mathcal{I} \times \{\pm 1\}$, with probability $1-\delta$ over m random examples X, there exists a constant c such that the hypothesis h^* selected via ERM satisfies*

$$R(h^*) \leq R_{\text{emp}}(h^*) + \sqrt{\frac{c}{m}\left[d + \log\left(\frac{1}{\delta}\right)\right]} \tag{2.16}$$

There is also a lower bound with a similar form that applies to hypotheses selected by *any* inductive principle. These two bounds between them appear to tell the whole story: we know that the generalisation ability of hypotheses chosen via ERM lies within a confidence interval of a known size of the true risk, and we know that a hypothesis chosen by another principle cannot be asymptotically better. The story appears to be complete... until we introduce a new twist: for many function classes $\mathcal{H}$ the confidence interval may be large or even infinite, due to properties of the VC dimension which are now explored. Learning is still, nevertheless, possible.

2.3.3 Covering numbers

The problem with the VC dimension is that it is sensitive to behaviour on an arbitrarily small scale. Figure 2.4 shows an extreme example. We use *covering numbers* to avoid some of these problems.

Covering numbers are a way of measuring the effective "size" of a class of functions $\mathcal{H}$ at a given scale ϵ. The following definition is used by the definition of covering numbers:

Definition 9 (Restriction of a function [1]) *Consider a series of points $X = (\mathbf{x}_1, \ldots, \mathbf{x}_k) \in \mathcal{I}^k$, and a function u where $u : \mathcal{I} \to \mathbb{R}$. Then we define the* restriction of u to X *as a vector*

$$\mathbf{u} = u_{|X} = (u(\mathbf{x}_1), \ldots, u(\mathbf{x}_n)) \tag{2.17}$$

We now can define exactly what we mean by "to cover":

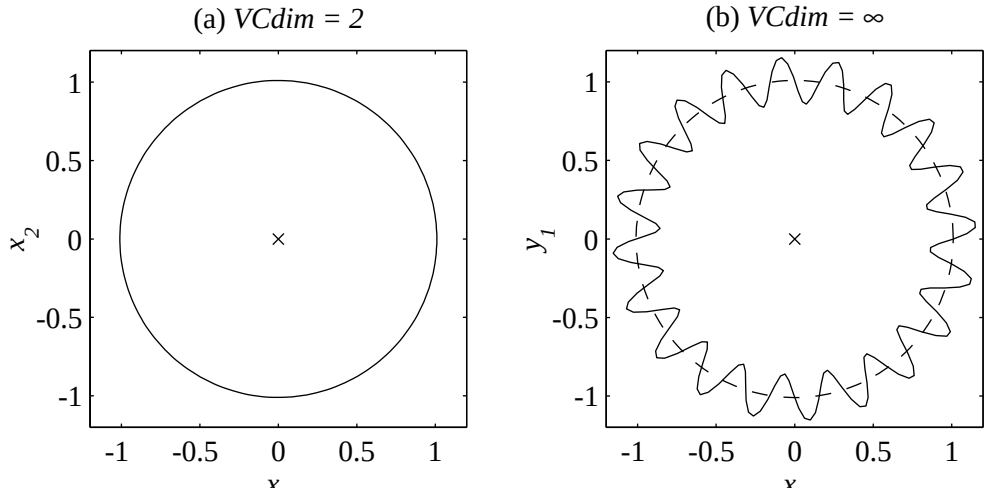


Figure 2.4: Problems with scale insensitive measures of complexity

We consider the decision boundaries of two hypothesis classes. Part (a) shows the decision boundary of class $\mathcal{H}_\circ$, which consists of circles centered at $(\bar{x}_1, \bar{x}_2)$ (indicated by $\times$) with a radius function $r(\theta) = 1$. These decision boundaries are simply circles; $\mathrm{VCdim}(\mathcal{H}_\circ) = 4$.
Part (b) shows a second hypothesis class $\mathcal{H}_*$. This class has a radius function $r(\theta) = 1 + \alpha \sin(\omega\theta)$. Despite $\mathcal{H}_\circ \to \mathcal{H}_*$ as $\alpha \to 0$, $\mathrm{VCdim}(\mathcal{H}_*) = \infty$ for all $\alpha > 0$ (a consequence of the sampling theorem; see [7]). In other words, two function classes with decision boundaries that are arbitrarily close have wildly differing VC dimensions. Clearly, in this case the VC dimension is not *a good measure of complexity.*

Definition 10 (Covering and covering numbers) *Suppose we are given two function classes $\mathcal{H}$ and $\mathcal{S}$ where $\mathcal{S} \subseteq \mathcal{H}$ and $f \in \mathcal{H} \Rightarrow (f : \mathcal{I} \to \mathbb{R})$, a set of points $X = (\mathbf{x}_1, \ldots, \mathbf{x}_n) : \mathbf{x}_i \in \mathcal{I}$, and a scale $\epsilon > 0$. Then we define the metric d_∞ as*

$$d_\infty(\mathbf{p}, \mathbf{q}) = \max_i |p_i - q_i| \tag{2.18}$$

(the usual ∞ norm). We say that $\mathcal{S}$ is an ϵ-cover for $\mathcal{H}$ with respect to X *if and only if*

$$(\forall u \in \mathcal{H})\ (\exists v \in \mathcal{S}) : d(u_{|X}, v_{|X}) < \epsilon \tag{2.19}$$

We further define the "d ϵ-covering number of $\mathcal{H}$ with respect to X" *as*

$$\mathcal{N}(\mathcal{H}, \epsilon, X) = \sup_{\mathcal{S} \subseteq \mathcal{H}} |\mathcal{S}| : \mathcal{S} \textit{ is an } \epsilon\textit{-cover for } \mathcal{H} \textit{ w.r.t. } X \tag{2.20}$$

Figure 2.5 illustrates the notion of covering; both geometrically and for a set of functions.

This definition suffices if we have a fixed X (for example, we want to develop bounds on the *empirical* risk). By dropping the dependence on X, and instead taking a maximum over all sets X of size m, we can obtain a result that is applicable to the *true* risk:

Definition 11 (Uniform covering numbers) *Given the same parameters as definition 10, we define the* uniform covering number *as*

$$\mathcal{N}(\mathcal{H}, \epsilon, m) = \max_{\forall X \in \mathcal{I}^m} \{\mathcal{N}(\mathcal{H}, \epsilon, d_\infty)\} \tag{2.21}$$

The uniform covering number is used in a similar manner to the VC dimension, to bound the generalisation performance of a learning algorithm.

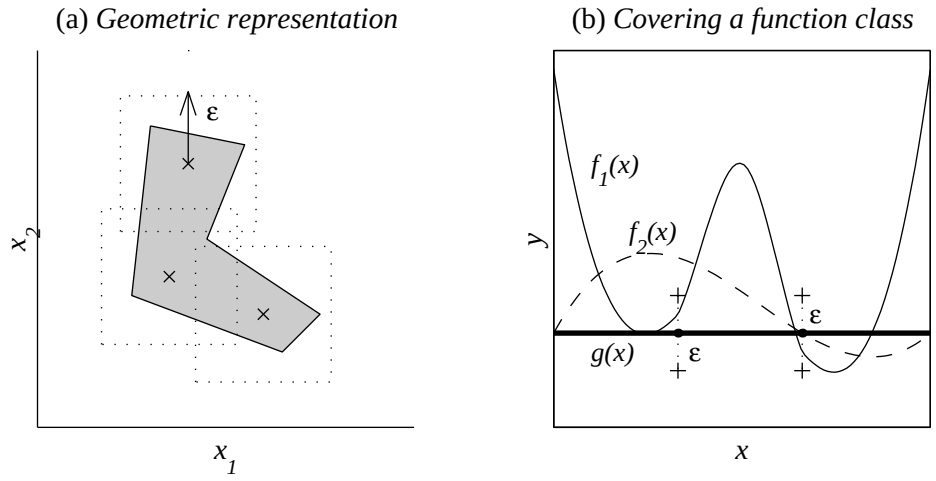


Figure 2.5: Illustration of covering numbers

Part (a) shows a geometric representation of a set of points (indicated by $\times$) covering a set in $\mathbb{R}^2$ (after [1]). Part (b) illustrates covering applied to functions. Function f_1 (thin solid line) is covered by function g (thick solid line) at the two points indicated with $\bullet$, as it is within ϵ at both points. However, function f_2 (dashed line) is not *covered by g as it is not within ϵ at the leftmost point. Note that in part (b) we are only looking at* single *functions; in the text we are considering* classes *of functions.*

2.3.4 Bounds using covering numbers

The result stated in this section is an important part of the background of this thesis. It is derived in Anthony and Bartlett [1] by bounding the expectation of the value of the uniform covering numbers; and converted in a straightforward manner to the form shown below.

Theorem 3 (Convergence bound using covering numbers) *Consider a hypothesis space $\mathcal{F} : f \in \mathcal{F} \Rightarrow (f : \mathcal{I} \to \mathbb{R})$, a series of m training samples $X \in (\mathcal{I} \times \mathcal{O})^m$, and a margin $0 \leq \gamma < 1/2$. Then with probability at least $1 - \delta$,*

$$R(f) \leq R^{\gamma}_{\text{emp}}(f) + \sqrt{\frac{8}{m} \log \left(\frac{2\mathcal{N}(\mathcal{H}, \gamma/2, 2m)}{\delta} \right)} \tag{2.22}$$

There are two important differences between the above theorem and the corresponding VC dimension bound (theorem 8). Firstly, theorem 3 uses a scale sensitive dimension (the uniform covering numbers) instead of the VC dimension. Secondly, it uses the margin risk (definition 5) instead of the empirical risk. These are linked by the γ parameter, which appears both as the margin threshold for the margin risk and as the scale for the uniform covering numbers.

By inspection of (2.22) we can see that the confidence interval will decrease as γ increases. However, increasing γ will have the opposite effect on $R^{\gamma}_{\text{emp}}(f)$ as the margin risk is nondecreasing with γ. As a result, there is a tradeoff here between the two terms. This is explored in section 2.4.

2.3.5 Application to p-convex hulls ($0 < p < 2$)

The theoretical motivation behind the algorithms developed in this thesis uses the covering number of a p-convex hull ($\text{co}_p(\mathcal{H})$). This section defines a p-convex hull and gives a qualitative result on the covering numbers for $0 < p < 2$. A full discussion appears in [25].

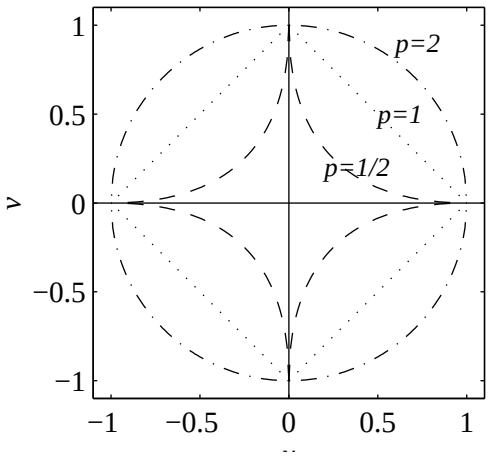


Figure 2.6: The p-convex hull of two variables

The figure illustrates the shape of a p-convex hull of two variables u and v for various values of p. The smaller p is, the closer to the axes the hull is.
We define two sets: the points on the boundary are termed "on" the hull, and those within the boundary "in" the hull. The equation of the curves is $(u^p + v^p)^{(1/p)} = 1$.

Definition 12 (p-convex hull) *The p-convex hull (strictly speaking, the p-absolutely convex hull) of a set $\mathcal{H}$ of functions (where $p > 0$) is defined as*

$$\mathrm{co}_p(\mathcal{H}) = \bigcup_{n\in\mathbb{N}} \left\{ \sum_{i=1}^{n} \alpha_i f_i : f_1, \ldots, f_n \in \mathcal{H}, \quad \alpha_1, \ldots, \alpha_n \in \mathbb{R}, \quad \sum_{i=1}^{n} |a_i|^p \leq 1 \right\} \tag{2.23}$$

Thus the p-convex hull of $\mathcal{H}$ is a subset of $\mathrm{lin}(\mathcal{H})$, the set of all possible linear combinations of functions in $\mathcal{H}$. While the p-convex hull of a set of *functions* is hard to visualise, the p-convex hull of points in a Euclidean space is a useful visual aid. See figure 2.6. This figure also illustrates that p-convex hulls define a *structure*: If $0 \leq p_1 \leq p_2 \leq \cdots \leq p_z$, then $\mathrm{co}_{p_1} \subseteq \mathrm{co}_{p_2} \subseteq \cdots \subseteq \mathrm{co}_{p_z}$. This property is used in section 2.4.1.

We now state a theorem that gives an approximate bound on the covering numbers of a p-convex hull. This result is central to the thesis.

Theorem 4 (Covering numbers of a p-convex hull [25]) *Consider a hypothesis class $\mathcal{H}$ where the covering numbers grow as*

$$\mathcal{N}(\mathcal{H}, \epsilon) \approx \left(\frac{1}{\epsilon}\right)^d \tag{2.24}$$

and a number $0 < p < 2$. Then for a fixed integer $m > 0$, the uniform covering number at scale $\epsilon > 0$ can be approximated by

$$\log_2 \mathcal{N}(\mathrm{co}_p(\mathcal{H}), \epsilon, m) \approx c(p)\, d\, \left(\frac{1}{\epsilon}\right)^{\frac{2p}{2-p}} \log_2 \left(\frac{1}{\epsilon}\right) \tag{2.25}$$

where $c(p)$ is a constant that depends only upon p.

It can be shown that equation (2.24) holds for finitely parameterised classes (a simple geometric argument may help: in $[0, 1]^d$, if the space is tiled with n hypercubes of size ϵ, then it takes 2^d hypercubes of size $\epsilon/2$; thus (2.24) holds).

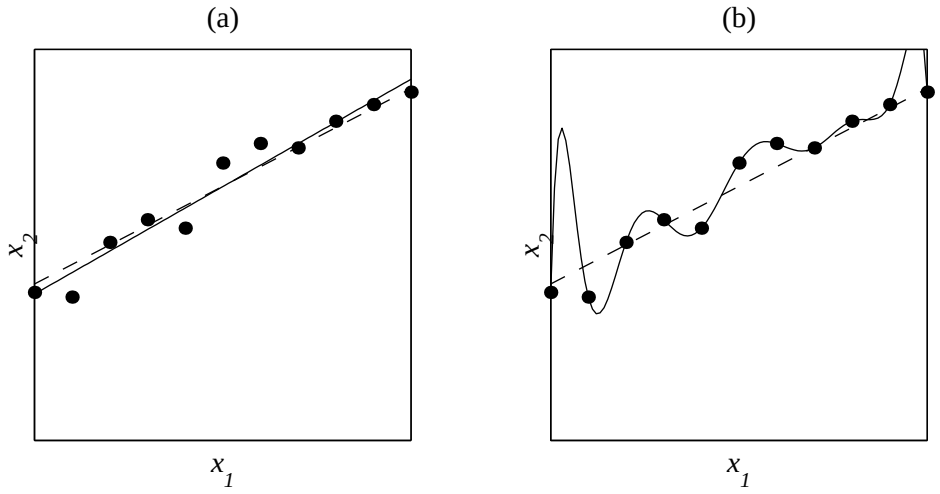


Figure 2.7: Overfitting with polynomials: two extreme examples

The • symbols are noisy data points; the dashed line is the underlying distribution. In part (a) an order 1 polynomial has been fitted. In part (b) an order 10 polynomial has been fitted. The fit for the first order polynomial is evidently closer to the underlying distribution. Thus, more complex models do not necessarily generalise better.

Theorem 4 can be made more precise, but at the expense of a *lot* more complexity. This extra complexity is unnecessary for our purposes; it is the *form* of this bound that motivates the work in this thesis. In particular, it should be noted that the approximate bound (2.25) decreases with decreasing p for $0 < p \leq 2$ (assuming that $c(p)$ remains nearly constant).

Finally, we emphasise that $\mathrm{co}_p(\mathcal{H})$ is usually a *lot* "richer" (contains a greater variety of hypotheses) than $\mathcal{H}$, especially for $p \geq 1$. Extending our hypothesis space in this manner is one way to markedly improve the empirical risk; indeed this is exactly what the Boosting algorithm (chapter 3) does.

2.4 Overfitting

We have now developed enough theory to analyse the problem that this thesis is attempting to solve: *overfitting*. The problem of overfitting is quite familiar to anyone who has had to fit a polynomial (say) to noisy data: to what extent should the data be trusted? By fitting a high-order polynomial (a complex model), we may fit the data perfectly, yet the underlying distribution poorly. We call this "overfitting". Conversely, by fitting a low-order polynomial (a less-complex model), we may fit the underlying distribution well but not our data. Figure 2.9 illustrates this point.

Overfitting is avoided by trading off model complexity against empirical risk. In this section, we will first describe how overfitting can be detected using two separate datasets (a training dataset and a test dataset), and give some examples of overfitting using the algorithms considered in this thesis. We will then consider two explanations of overfitting: a qualitative explanation based on general properties of model complexity, and a more quantitative explanation based on theory developed earlier in this chapter.

Given a set of m training samples X and a classifier $h^* \in \mathcal{H}$ that minimises empirical risk,

$$h^* = \underset{h \in \mathcal{H}}{\operatorname{argmin}} R_{\text{emp}}(h)$$

with probability of at least $1 - \delta$,

$$\underbrace{R(h)}_{\text{true risk}} \quad \leq \quad \underbrace{R_{\text{emp}}(\hat{h})}_{\text{empirical risk}} \quad + \quad \underbrace{b(\delta, |\mathcal{H}|, m)}_{\text{confidence interval}} \tag{2.26}$$

where the function $b(\delta, |\mathcal{H}|, m)$ has the following properties:

1. $b(\delta, \cdot, \cdot)$ is non-increasing with δ;
2. $b(\cdot, |\mathcal{H}|, \cdot)$ is nondecreasing with $|\mathcal{H}|$;
3. $b(\cdot, \cdot, m)$ is non-increasing with m.

Figure 2.8: General form of generalisation performance bounds

2.4.1 Theoretical overfitting and structural risk minimisation

Formally, we consider a series of hypothesis classes $\mathcal{H}_1, \ldots, \mathcal{H}_n$ where $n \leq \infty$ (recall that a hypothesis class contains all possible hypotheses of a learning algorithm) and some *appropriate* measure of complexity $|\cdot|$. (By "appropriate", we mean "suitable for the problem at hand". Depending upon the situation, we might choose $|\mathcal{H}_i| = \text{VCdim}(\mathcal{H}_i)$ or $|\mathcal{H}_i| = \mathcal{N}(\mathcal{H}_i, \epsilon)$, for example). We also assume without loss of generality that $\mathcal{H}_1 \subseteq \mathcal{H}_2 \subseteq \cdots \subseteq \mathcal{H}_n$, and thus using any sensible complexity measure, $|\mathcal{H}_1| \leq |\mathcal{H}_2| \leq \cdots \leq |\mathcal{H}_n|$. This sequence of increasingly complex hypothesis classes is called a *structure*.

Our goal is to find the optimal $\mathcal{H}_i$, in the sense that the true risk (2.8) is minimised. [9] The tools which enable us to proceed are the bounds on generalisation performance in section 2.3, which all have the general form and properties given in figure 2.8.

If there are more functions to choose from in $\mathcal{H}$, we are more likely to find one which fits the *training* data exactly. Another look at figure 2.7 will solidify this point. Thus, we require a large $|\mathcal{H}|$. Conversely, to minimise the model uncertainty term we want to make $|\mathcal{H}|$ as *small* as possible (a principle reminiscent of Occam's Razor: *the simplest solution is the best*).

Consequently, there is a tradeoff between the two terms in (2.26); we need to select an $\mathcal{H}$ with optimal complexity $|\mathcal{H}^*|$. Figure 2.9 illustrates the process[10]. The principle of minimising the RHS of (2.26) by choosing the optimal $|\mathcal{H}|$ is known as *structural risk minimisation*.

2.4.2 Implementing SRM

There are two methods that are used to implement SRM. The first is useful if we have a tight bound on $b(\delta, |\mathcal{H}|, m)$. As we can calculate R_{emp}, we can directly minimise (2.26).

In the problems considered in this thesis, the bound on b is very loose. We instead approximate the true risk directly by using a *testing dataset* X_T. The learning machine never sees the labels of the

[9] We have not yet attempted to tackle the *true risk* in this manner: previously we have been concerned only with the *empirical risk*.

[10] Figure 2.9 was generated from experimental results using the AdaBoost algorithm (chapter 3).

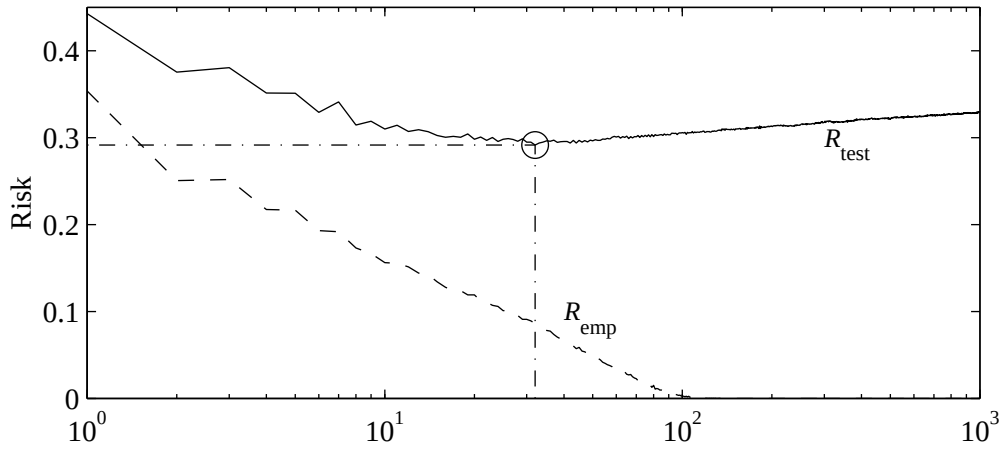


Figure 2.9: Structural risk minimisation

Training errors (empirical risk) and test errors (approximate true risk) for a dataset. $|\mathcal{H}|$ increases along the x axis. The optimal structure is identified with the dash-dot line and the $\circ$. The existence of a clear local minima indicates that overfitting has occurred.

samples in X_T; thus this dataset can be used to approximate the true risk:

$$R(f) \approx \underset{X_T}{R_{\text{emp}}(f)} \tag{2.27}$$

We then simply stop training when it appears that we have reached the optimal complexity (minimal test error)[11].

The main disadvantage of this method is that some data must be put aside for the test dataset; thus the learning algorithm does not train on all of the available data[12].

[11] There are many methods for doing this efficiently; these are beyond the scope of this discussion.

[12] Again, methods for overcoming this problem are beyond the scope of this thesis.

Chapter 3

AdaBoost and boosting algorithms

Boosting algorithms (AdaBoost being one of the first) are a very important recent development in machine learning that has significantly altered the field. The name "Boosting" is applied to a range of machine learning algorithms, all of which combine several "weak" hypotheses to form a combined "strong" hypothesis that significantly outperforms all of them.

In this chapter we describe the classic algorithm *AdaBoost* published by Freund and Schapire in 1996 [9], and present results on its properties and generalisation performance. We then show that AdaBoost implements the optimisation of a cost functional via gradient descent, an property that proves to provide a useful framework for the development of modified algorithms in chapter 4. We digress slightly to look at *normed* boosting algorithms, a minor modification, and conclude by again considering the inherent problem of overfitting.

3.1 The AdaBoost algorithm

In essence, the AdaBoost algorithm "bootstraps" itself onto another learning algorithm (the *weak learning algorithm*–the decision stumps algorithm described in appendix A is one such algorithm), improving its performance by using a "committee" (linear combination) of weak hypotheses.

AdaBoost performs *very* well; its combined hypotheses generally perform significantly better than *any* non-boosted algorithm (particularly for reliable (low-noise) data). Even the simplest of weak algorithms, when "boosted", will usually outperform the best un-boosted algorithms[1]. We consider questions of how and why AdaBoost works; in particular how it chooses the optimal linear combination of classifiers from the very large set of possible linear combinations.

Figure 3.1 gives an explicit description of AdaBoost. The key features are:

1. The AdaBoost algorithm operates in iterations. During iteration $t+1$, one classifier h_{t+1} is added to the combined hypothesis F_t, to give $F_{t+1} = F_t + b_t h_{t+1}$.

2. The weight b_t of each weak hypothesis (the *classifier weight*) depends upon the weighted empirical risk (definition 4) of that classifier over the training dataset. Section 3.1.1 describes these classifier weights in more detail.

3. Each sample in the training dataset is given a weight (*sample weight*) which is modified depending upon how "hard" that sample is to classify. Section 3.1.2 describes these sample weights in more detail.

[1] As a result, the focus of recent research effort has shifted from the development of *learning* algorithms (which are all more or less equivalent when boosted) to the development of better *boosting* algorithms.

Input: m examples $X = ((\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m))$ and a weak learning algorithm $\mathbb{W}$
Initialisation: $F_0(\mathbf{x}) = 0$; $\mathbf{w}_0 : w_{0,i} = 1/m$ for $i = 1 \ldots m$
Do for $t = 1 \ldots T$:
Update $F_t \overset{\mathbb{B}}{\Longrightarrow} F_{t+1}$:

1. Train weak learner $\mathbb{W}$ on the weighted sample set $((\mathbf{x}_1, y_1, w_{t,1}), \ldots, (\mathbf{x}_m, y_m, w_{t,m}))$ and obtain hypothesis $h_t : \mathcal{I} \to \{\pm 1\}$

2. Calculate the weighted empirical risk (training error) ϵ_t of h_t:

$$\epsilon_t = R_{\text{emp}}^{\mathbf{w}_t}(h) = \sum_{j:h(\mathbf{x}_j) \neq y_j} w_{t,j} \tag{3.1}$$

   If $\epsilon_t = 0$ (a single weak learner can correctly learn the relationship) or $\epsilon_t \geq 1/2$ (the weak learner is performing as badly as random guessing) then abort the training process, with $F_{t+1} = F_t$.

3. Calculate the classifier weight b_t:

$$b_t = -\frac{1}{2} \log \left(\frac{\epsilon_t}{1 - \epsilon_t} \right) \tag{3.2}$$

4. Update the sample weights $\mathbf{w}_t \Rightarrow \mathbf{w}_{t+1}$:

$$w_{t+1,i} = \begin{cases} \frac{w_{t,i}}{Z_t} \exp\{b_t\} & \text{if } f_t(\mathbf{x}_i) = y_i \\ \frac{w_{t,i}}{Z_t} \exp\{-b_t\} & \text{otherwise} \end{cases} \tag{3.3}$$

   where Z_t is a normalisation constant, such that $\sum_{i=1}^{m} w_{t+1,i} = 1$

Output:

$$H_T(\mathbf{x}) = \text{sign}(F_T(\mathbf{x})) = \text{sign}\left\{ \sum_{i=1}^{T} b_i h_i(\mathbf{x}) \right\} \tag{3.4}$$

Figure 3.1: The AdaBoost algorithm $\mathbb{B}$

The notation $w_{t,i}$ denotes the weight of sample i at iteration t. The process of training AdaBoost for one iteration is denoted $F_t \overset{\mathbb{B}}{\Longrightarrow} F_{t+1}$. Two AdaBoost hypotheses F_a and F_b are said to be *equivalent* ($F_a \equiv F_b$) if they produce the same output for any input, even if their internal state (b and w values) are not identical.

3.1.1 Classifier weights

On training iteration t, one weak learner $h_t(\cdot)$ is added to the linear combination (h_t is chosen by the weak learning algorithm). The coefficient b_t of h_t is calculated from the training error ϵ_t of h_t as

$$b_t = -\frac{1}{2} \log \left(\frac{\epsilon_t}{1 - \epsilon_t} \right) \tag{3.5}$$

which is plotted in figure 3.2. In section 3.2 we will show how (3.5) is derived. We will sometimes use the notation $\mathbf{b}$ to refer to the vector of classifier weights $\mathbf{b} = (b_1, \ldots, b_t)$.

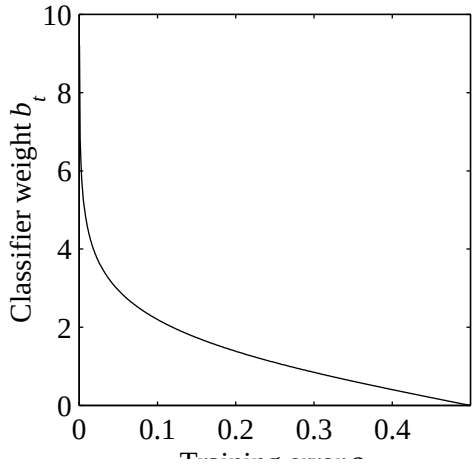


Figure 3.2: Classifier weights against weighted training error

When $\epsilon_t = 1/2$, the classifier only does as well as random guessing, and $b_t = 0$. As ϵ_t approaches zero, b_t increases without bound. The effect is that F becomes dominated by those weak hypotheses that performed well on the training samples (training is halted if $\epsilon_t = 0$ or $\epsilon_t \geq 1/2$).

3.1.2 Sample weights

At iteration t, each sample in the training set has a corresponding weight $w_{t,j}$ which is updated to reflect the difficulty of that sample. On each iteration the weights of samples which h_t classified incorrectly are increased (multiplied by e^{b_t}); those which h_t classified correctly are *divided* by the same amount. The whole lot are then normalised. As a result, the sample weight distribution is affected in the following manner (illustrated in figure 3.3):

- Training samples which are misclassified often, or are one of few points to get misclassified, increase their proportion of the weight (they are identified as "hard samples");
- Other samples decrease their proportion of the weight, and are effectively identified as "easy" samples.

The overall effect is for those samples near the decision boundary to increase in weight, and those far from it to decrease. This forces the algorithm to concentrate on the hard samples, and is one reason why it works well on low-noise datasets. Unfortunately, those samples corrupted by noise are usually hard also, and concentrating on those is not desirable. We explore the implications of this observation in section 3.6.

3.2 Boosting as gradient descent

Recent work has shown boosting to be an implementation of a gradient descent algorithm in an inner product space [16][2]. An inner product space requires both a universal set $\mathcal{X}$ and an inner product

[2]This discussion follows [16] rather closely with some changes in notation and omission of details such as stopping criteria.

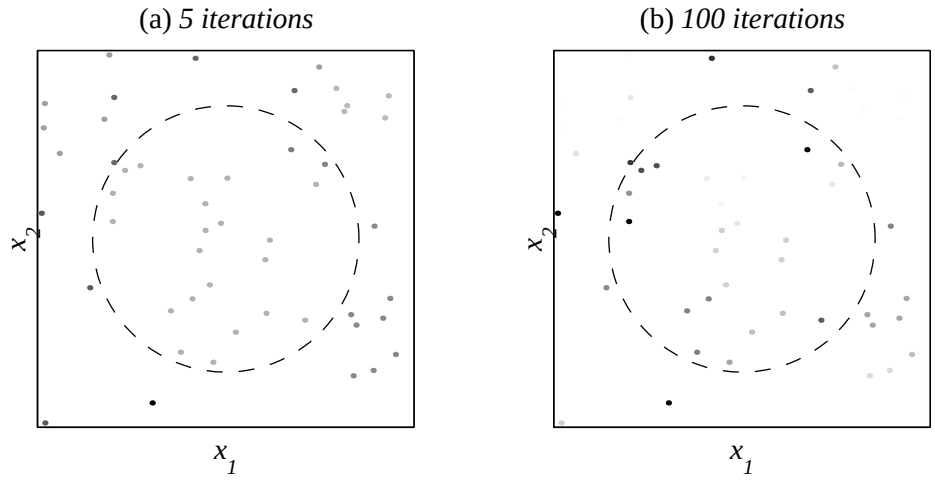


Figure 3.3: Evolution of sample weights

The figure shows the evolution of the sample weights from (a) 5 iterations to (b) 100 iterations of boosting over a set of training data in $\mathbb{R}^2$. The sample weight at a point is proportional to the intensity of the dot. It can be seen that as $t \to \infty$, the "easy" samples (far from the decision boundary–indicated by the dashed line) lose weight relative to the "hard" samples.

operator $\langle \cdot, \cdot \rangle$. We define

$$\mathcal{X} = \mathrm{co}(\mathcal{H}) = \bigcup_{n \in \mathbb{N}} \left\{ \sum_{i=1}^{n} b_i f_i : f_1, \ldots, f_n \in \mathcal{H}, \quad b_1, \ldots, b_n > 0, \quad \sum_{i=1}^{n} b_i = 1 \right\} \tag{3.6}$$

(which is the convex hull of $\mathcal{H}$), and

$$\langle F, G \rangle = \frac{1}{m} \sum_{i=1}^{m} F(\mathbf{x}_i) G(\mathbf{x}_i) \qquad F, G \in \mathcal{X} \tag{3.7}$$

where it should be noted that $\langle \cdot, \cdot \rangle$ implicitly depends upon a series of training samples

$$X = ((\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)) \tag{3.8}$$

We furthermore define the *cost functional* (which our gradient descent algorithm is trying to minimise) as

$$C(F) = \frac{1}{m} \sum_{i=1}^{m} c(y_i F(\mathbf{x}_i)) \tag{3.9}$$

which sums another function c (introduced below) over the *margin* of all training samples in X.

At iteration $t+1$ of gradient descent, we wish to choose a hypothesis h_{t+1} and a weight b_{t+1} such that $C(F_t + b_{t+1} h_{t+1}) < C(F_t)$. We do not choose h_{t+1} directly, however–that is done by the weak learning algorithm–instead, we manipulate the sample weights $\mathbf{w}_t$ in such a manner that (as we will see) the weak learning algorithm chooses the "right" h_{t+1} for us. The optimal h_{t+1}, called h^*_{t+1}, is the one that leads to the steepest reduction in the cost function:

$$h^*_{t+1} = -\nabla C(F)(\mathbf{x}) = \left. \frac{\partial C(F + \alpha 1_{\mathbf{x}})}{\partial \alpha} \right|_{\alpha = 0} \tag{3.10}$$

where $1_{\mathbf{x}}$ is the indicator function of $\mathbf{x}$; the use of which is necessary as (3.10) is known only where we have a sample. Since the optimal h^*_{t+1} will not necessarily be in $\mathcal{H}$, we choose the $h_{t+1} \in \mathcal{H}$ with

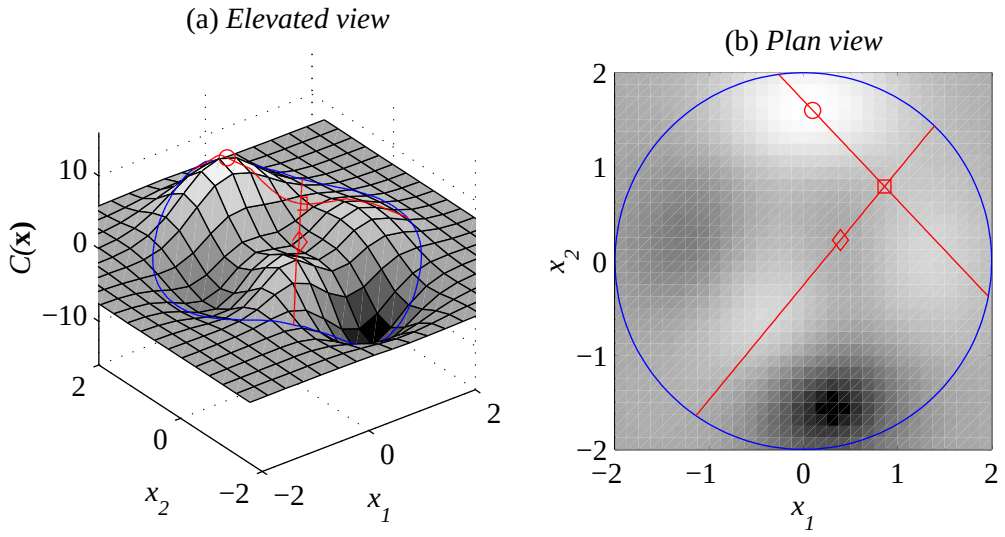


Figure 3.4: Schematic representation of gradient descent.
Points within the circle are in $\mathcal{H}$, the height of the surface the value of $C(\mathbf{x})$ at that point. (a) is a 3D view, (b) is plan view. Lines show line searches; markers minima. We descend from $\circ$ through $\square$ to $\Diamond$, where the algorithm terminates due to a local minima.

the greatest inner product $\langle -\nabla C(F), h^*_{t+1} \rangle$. More specifically, using the inner product space and cost functions chosen above, we are maximising

$$-\langle \nabla C(F_t), f_{t+1} \rangle = -\frac{1}{m^2} \sum_{i=1}^{m} y_i f_{t+1}(\mathbf{x}_i) c'(y_i F_t(\mathbf{x}_i)) \tag{3.11}$$

The following theorem states that the *weighted empirical risk* $R^{\mathbf{w}_t}_{\text{emp}}$ (section 2.2.4) is the "right" form of risk to minimise, and gives the weights $\mathbf{w}_t = (w_{t,1}, \ldots, w_{t,m})$:

Theorem 5 (Weighted empirical risk minimisation [16]) *Maximising (3.11) is equivalent to minimising the weighted empirical risk*

$$R^{\mathbf{w}_t}_{\text{emp}} = \sum_{i:y_i \neq h(\mathbf{x}_i)}^{m} w_{t,i} \tag{3.12}$$

where $\mathbf{w}_t$ is given by

$$w_{t,i} = \frac{c'(y_i\ F_t(\mathbf{x}_i))}{\sum_{j=1}^{m} c'(y_j\ F(\mathbf{x}_j))} \qquad i = 1 \ldots m \tag{3.13}$$

Proof: See [16]. We assume that $c(\alpha)$ is monotonically decreasing with respect to an increasing α (as any sensible cost function will be); the result follows easily from this assumption.

Having chosen our direction (hypothesis) h_{t+1}, we now choose a step size (classifier weight) b_{t+1}. AdaBoost uses a line search for the minimum of the cost functional along the line parallel with h_{t+1} through F_t:

$$b_{t+1} = \operatorname*{argmin}_{\alpha} \sum_{i=1}^{m} C(y_i F_t(\mathbf{x}_i) + y_i \alpha h_{t+1}(\mathbf{x}_i)) \tag{3.14}$$

The gradient descent process is illustrated in figure 3.4.

We will now show that AdaBoost implements gradient descent using the cost function $c(\lambda) = e^{-\lambda}$. We do this by showing that gradient descent produces the same b_t and $\mathbf{w}_t$ values as AdaBoost.

Theorem 6 (AdaBoost implements gradient descent [16]) *The AdaBoost algorithm implements gradient descent over a margin cost function*

$$c(\lambda) = e^{-\lambda} \tag{3.15}$$

Proof: We outline a skeleton, ignoring initial conditions and the stopping conditions. We will show that, given a state at iteration t, then boosting and gradient descent produce identical states at iteration $t+1$.

Once $\mathbb{W}$ has chosen h_{t+1} using weighted empirical risk minimisation, we choose $b_{t+1} = \alpha^*$ that minimises the cost functional along the line $F_t + \alpha h_{t+1}$. The value of the cost functional along this line is

$$C = \sum_{i=1}^{m} c\left(y_i F_t(\mathbf{x}_i) + y_i \alpha h_{t+1}(\mathbf{x}_i)\right) \tag{3.16}$$

Substituting in (3.15), we obtain

$$C = \sum_{i=1}^{m} \exp\left\{y_i F_t(\mathbf{x}_i) + y_i \alpha h_{t+1}(\mathbf{x}_i)\right\} \tag{3.17}$$

We minimise (3.17) by differentiating with respect to α and setting the derivative to zero. The task is simplified considerably by noting that only the second half of the exponential is a function of α. The result is that

$$\frac{\partial C}{\partial \alpha} = \sum_{i=1}^{m} y_i h_{t+1}(\mathbf{x}_i) \exp\left\{y_i F_t(\mathbf{x}_i) + y_i \alpha h_{t+1}(\mathbf{x}_i)\right\} = 0 \tag{3.18}$$

We make several algebraic simplifications. Noting that

$$y_i h_{t+1}(\mathbf{x}_i) = \begin{cases} 1 & \text{if } y_i = h_{t+1}(\mathbf{x}_i) \\ -1 & \text{if } y_i \neq h_{t+1}(\mathbf{x}_i) \end{cases} \tag{3.19}$$

and that $w_{t,i} = k \exp\{-y_i F_t(\mathbf{x}_i)\}$ where k is some (normalising) constant, we recast (3.18) into the form

$$\underbrace{\frac{\overbrace{\sum_{i:y_i = f_{t+1}(\mathbf{x}_i)} w_{t,i}}^{1-\epsilon_t}}{\sum_{i:y_i \neq f_{t+1}(\mathbf{x}_i)} w_{t,i}}}_{\epsilon_t} = \exp\left\{2\alpha^*\right\} \tag{3.20}$$

from which it is straightforward algebra to obtain $b_{t+1} = \alpha^*$, the classifier weight equation(3.5). Thus, gradient descent updates the internal state in the same manner as AdaBoost; therefore AdaBoost implements gradient descent.

The cost functional $c(\lambda) = e^{-\lambda}$ is an approximation to the misclassification risk (2.7) that is differentiable at all points and leads to a closed-form solution for the step size. It is plotted in figure 3.5. Other approximations have been used, a survey of which appears in [16].

3.2.1 Modifying gradient descent

We have shown that the AdaBoost algorithm implements gradient descent. There are four degrees of freedom which can be exploited to produce modified versions of gradient descent:

1. Choice of universal set? (AdaBoost: $\mathrm{co}(\mathcal{H})$)
2. Choice of inner product? (AdaBoost: equation (3.7))

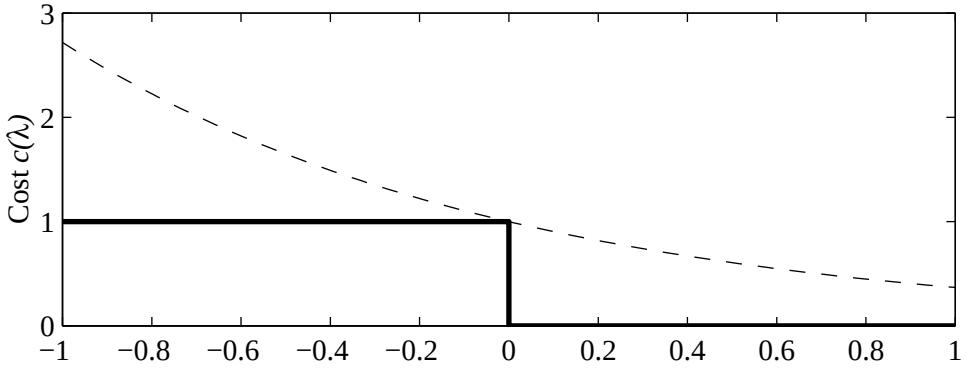


Figure 3.5: Approximation to the misclassification risk function

The misclassification risk is plotted in the dotted line, with the AdaBoost approximation plotted in the solid line. It is clear that the effect of this cost function will be to maximise the margins of the samples.

3. Choice of cost function? (AdaBoost: $c(\alpha) = e^{-\alpha}$)

4. Method of choosing step size? (AdaBoost: line search).

In chapter 4 we consider alternative choices for 1, 3 and 4 to generate algorithms with desirable properties.

3.3 AdaBoost and Boosting algorithms

Until this point we have only considered the AdaBoost algorithm. We now extend our scope to other AdaBoost-like algorithms. These algorithms are termed "boosting algorithms".

There has recently been some debate over what constitutes a "boosting algorithm"; see for example Duffy and Helmbold [8] who adopt a quite restrictive definition. In this thesis, we take a very broad definition, allowing any learning machine that is operating in an "AdaBoost-like" manner to be called a "boosting" algorithm. In particular, we don't require that the training error be guaranteed to reach zero.

3.4 Properties of AdaBoost

We have covered the intrinsic properties of the AdaBoost algorithm quite extensively. We now cover some extrinsic properties, allowing comparison of AdaBoost to other algorithms in chapter 6.

3.4.1 Convergence properties

Theorem 7 (AdaBoost converges to zero training error in finite time [10]) *Suppose that the AdaBoost algorithm $\mathbb{B}$ operating on weak-learning algorithm $\mathbb{W}$ generates hypotheses h_t with training errors $\epsilon_1, \ldots, \epsilon_T$ over a training set X. Furthermore, assume that each $\epsilon_t < 1/2$, and let $\beta_t = 1/2 - \epsilon_t$. Then the empirical risk of the* combined *hypothesis $H_T = \mathrm{sign}(F_T)$ is bounded by*

$$R_{\mathrm{emp}}(H_T) \leq \prod_{t=1}^{T} \sqrt{1 - 4\beta_t^2} \leq \exp\left\{-2\sum_{t=1}^{T} \beta_T^2\right\} \tag{3.21}$$

Proof: The straightforward proof of this theorem is contained in [10].

The following theorem shows that the total weight of the boosting algorithm is unbounded as the number of iterations increases. It is used to prove results on the minimum margin and final training error of boosting.

Theorem 8 (Sum of AdaBoost classifier weights is unbounded) *Suppose that we are given a set of training samples X and a weak-learner $\mathbb{W}$, and run AdaBoost for T training iterations. Then either*

- *AdaBoost terminates on an iteration $t_{\text{term}} < T$; or*
- *the 1-norm[3] of the classifier weight vector $\mathbf{b}$ is unbounded as $T \to \infty$:*

$$\lim_{T\to\infty} \|\mathbf{b}\|_1 = \infty \tag{3.22}$$

Proof: This is proved in [6]. The proof proceeds by contradiction, assuming a least upper bound on $\|\mathbf{b}\|_1$ and showing that this bound will only hold if training terminates.

3.4.2 AdaBoost maximises the minimum margin

This section shows that the limiting behaviour of AdaBoost is to choose the linear combination that maximises the minimum margin over its training samples.

Theorem 9 (AdaBoost maximises the minimum margin) *Given a particular hypothesis F and a training set X, define the minimum margin as*

$$m_{\min} = \min_{(\mathbf{x}_i, y_i)\in X} y_i F(x_i)$$

Then the AdaBoost algorithm will converge as $t \to \infty$ to the solution which maximises the minimum margin.

Proof: We present an outline (assuming that training doesn't terminate). We can write AdaBoost's hypothesis as $F_t = b_1 h_1 + \cdots + b_t h_t$. Then defining the *normalised hypotheses* $\bar{f}_i$ as

$$\bar{f}_i = \frac{b_i h_i}{\|b\|_1} \tag{3.23}$$

such that $\bar{F}_t = \bar{f}_1 + \cdots + \bar{f}_t$, the cost functional (3.17) can be rewritten as

$$C(F) = \sum_{i=1}^{m} \exp\{-y_i \bar{F}_t(\mathbf{x}_i)\}^{\|b\|_1} \tag{3.24}$$

We know from theorem 6 that AdaBoost is working to minimise the cost function. From theorem 8, $\|b\| \to \infty$; thus as $t \to \infty$ the largest value of $-y_i \bar{F}_t(x_i)$ will dominate the sum (3.24), and so $C(F) \to exp\{\max -y_i \bar{F}_t(x_i)\}$. Thus, to minimise $C(F)$ we must be making $\min y_i \bar{F}_t(x_i)$ as large as possible; that is we are maximising the minimum margin. Figure 3.6 illustrates the evolution of both the cost function and the margins of a dataset.

[3] For a vector $\mathbf{b} \in \mathbb{R}^n$, $\|\mathbf{b}\|_p = \left(\sum_{i=1}^n b_i^p\right)^{\frac{1}{p}}$ and $\|\mathbf{b}\|_1 = \sum_{i=1}^n b_i$.

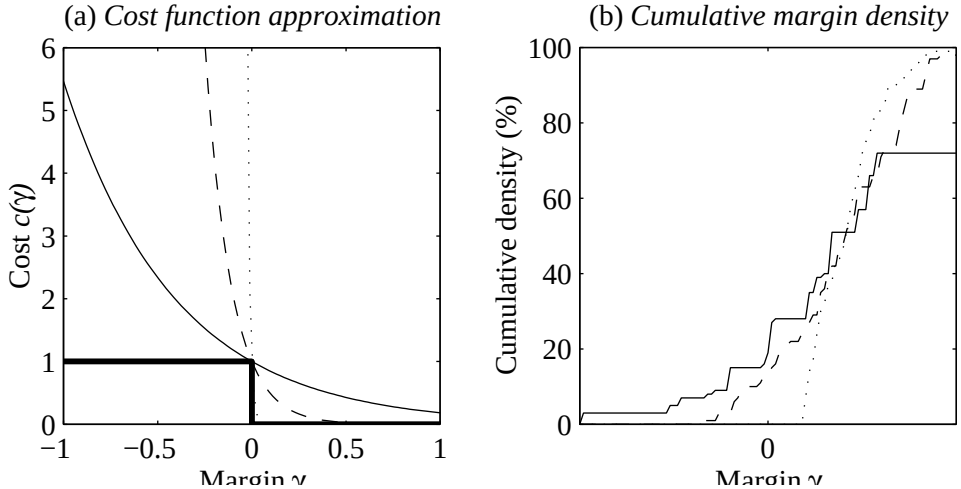


Figure 3.6: Cost function and margin evolution of AdaBoost

Each curve was generated by training AdaBoost on a small dataset. The lines indicate a snapshot at a particular time: 5 iterations (solid), 50 iterations (dashed) and 1000 iterations(dotted).
Part (a) shows the normalised *cost function* $c(\gamma) = e^{-\|\mathbf{b}\|\gamma}$. *It is clear that as* $t \to \infty$, *this cost function becomes very steep. Part (b) shows a cumulative distribution of the normalised margins (that is, the proportion of samples with margin less than or equal to* γ. *As time progresses, the minimum margin is pushed further to the right; at 1000 iterations the training error has evidently reached zero as all margins are positive. (Note that for clarity part (b) has its* x *axis normalised separately for each* p *value in order to stretch the margin distribution over the whole range.)*

3.4.3 Invariance of AdaBoost to uniform scaling of classifier weights

We present here an obvious theorem about the invariance of hypothesis weights. It has some implications on the hypothesis classes that the p-boosting algorithms discussed in chapter 4 use.

Theorem 10 *The hypothesis returned by the AdaBoost is independent of a scaling of the* b *values. Given an unthresholded hypothesis* F *and a constant* $\alpha > 0$,

$$H = \text{sign}(F) \equiv H' = \text{sign}(\alpha F) \tag{3.25}$$

Proof: This follows easily from the fact that the sign function is unaffected by a constant scaling.

3.5 Performance bounds for Boosting

The following theorem appears in Schapire et. al [20]. It gives a bound on generalisation error similar to those in chapter 2.

Theorem 11 (Performance bound for boosting (p=1)) *There is a constant* c *such that a combined hypothesis* $H = \text{sign}(F)$ *generated by the AdaBoost algorithm* $\mathbb{B}$ *from a base class* $\mathcal{H}$ *with VC dimension* d, *with probability at least* $1-\delta$ *over* m *independent training samples has true risk bounded by*

$$R(H) \leq R^{\gamma}_{\text{emp}}(F) + \sqrt{\frac{c}{m}\left[\frac{d\ln^2(m/d)}{\gamma^2} + \ln(1/\delta)\right]} \tag{3.26}$$

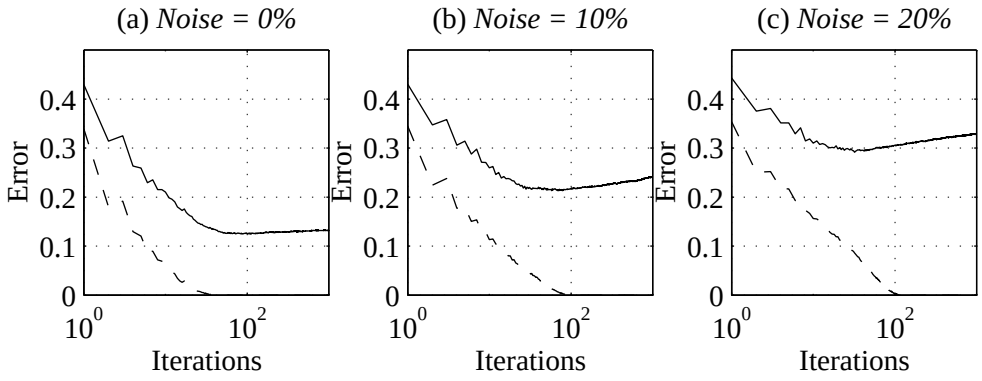


Figure 3.7: Experimental results for Boosting showing overfitting

Three plots of test/training error vs iteration number for Boosting. The dataset used was the "ring" distribution (see chapter 5) with 50 samples. Mild overfitting is visible on the 20% noise plot and becomes more pronounced as the level of noise increases. The solid curve is the test error (true risk); the dashed curve the training error.

3.6 Overfitting

Initial experiments with AdaBoost indicated that the algorithm was remarkably resistant to overfitting, despite the unlimited complexity of the hypothesis space [9]. However, further experiments [12, 3] revealed that training to 10^4 to 10^6 iterations, or adding label noise, would often induce overfitting. Figure 3.7 shows some examples of the effect. There have been several studies into the characteristics of AdaBoost that lead to overfitting [20, 12, 19]. A largely intuitive survey of the characteristics of AdaBoost that lead to overfitting is presented here.

That overfitting occurs in AdaBoost is a consequence of the property that AdaBoost maximises the minimum margin (theorem 9). Intuitively, this property causes the AdaBoost algorithm to concentrate on a few hard samples. Unfortunately, in a noisy dataset, these samples are usually all noise. Some algorithms are mentioned in the next section that explicitly allow for some samples to be misclassified, in order to reduce the tendency towards overfitting.

3.7 Previous work on regularising AdaBoost

Following the discovery that AdaBoost operated by maximising the minimum margin, several efforts were made to regularise AdaBoost to avoid overfitting. Algorithms such as *soft margins* [19] explicitly allow a certain number of training samples to have a small or negative margin; while algorithms such as DOOM and DOOM II [14, 15] do this implicitly by flattening out the cost function (figure 3.5) for low and negative margins. These approaches have been quite successful, both outperforming AdaBoost under some conditions (in particular on noisy data).

3.8 Normed boosting algorithms

Several authors [14, 6] have studied modified versions of AdaBoost, where the classifier weights are normalised *at each iteration*. After b_t is calculated for iteration t, the entire set of classifier weights b

are normalised:

$$b_i \Leftarrow \frac{b_i}{\|\mathbf{b}\|_p} \qquad i = 1 \dots t \tag{3.27}$$

where the norm is chosen via the p parameter.[4]

While it can be shown that these algorithms converge to the global minimum of the gradient descent cost functional (3.9), they do *not* have the property of guaranteed convergence to zero training error (theorem 7 does not hold); the reason being that $\sum_{i=1}^{t} b_{t,i}$ is bounded above (theorem 8 does not hold). As results in chapter 6 and the discussion in section 3.6 indicate, an algorithm does not have to converge to zero training error to generalise well.

The "sloppy" algorithm developed in chapter 4 is a generalisation of the normed boosting algorithm described in this section, where the value of p is any positive real.

[4] Mason et. al. considered mainly $p \in \{1, 2\}$ in [14].

Chapter 4

Development of p-boosting algorithms

This chapter describes the concepts behind and a theoretical justification of *p-boosting*, a generalisation of the AdaBoost algorithm. The treatment is somewhat abstract to begin with; further sections become more concrete as practical issues are considered.

4.1 Avoiding overfitting of boosting algorithms

Issues of overfitting have been addressed several times in this thesis. Recall that when given an algorithm, we avoid overfitting by halting training at the point where the generalisation error is at a minimum (figure 2.7). Theoretically, we see that guaranteed generalisation performance is the sum of the empirical risk (which *decreases* as $t \to \infty$) and a confidence interval (which *increases* as $t \to \infty$); figure 4.1 provides more detail.

We attack the problem by concentrating on the confidence interval. It was shown in chapter 3 that this confidence interval increases with the complexity of the hypothesis class $\mathcal{X}$. One obvious way, therefore, to avoid overfitting is to limit the complexity of the class of hypotheses that our learning algorithm can choose. This process is known as *capacity control*. Specifically, instead of choosing $\mathcal{X} = \mathrm{co}(\mathcal{H})$ as AdaBoost does, we choose $\mathcal{X} = \mathrm{co}_p(\mathcal{H})$ (definition 12) for some $p > 0$.[1] Then, for $p < 1$, we have $|\mathrm{co}_p(\mathcal{H})| < |\mathrm{co}(\mathcal{H})|$ (where, as in section 2.4.1, we choose $|\cdot|$ to be an appropriate measure of complexity, such as covering numbers). In effect, we have added a parameter p to our algorithm which can be adjusted to tune the capacity of $\mathcal{X}$. Using the language of section 2.4.1, the p parameter allows us to define a structure on $\mathcal{X}$.

By decreasing p, we decrease our confidence interval. However, the empirical risk is likely to *increase* as $p \to 0$. We have a tradeoff between the two terms; we need to use structural risk minimisation to find the optimal p value. Figure 4.2 illustrates this point.

4.1.1 Generalisation performance of p-convex boosting algorithms

We now combine the covering-number generalisation performance bound (2.22) with the approximate values of covering numbers for p-convex hulls (2.25), to obtain the following theorem on the generalisation performance of classifiers over p-convex hulls.

Theorem 12 (Generalisation performance over p-convex hulls) *Consider a set of hypotheses where the covering numbers of $\mathcal{H}$ grow as $\mathcal{N}(\mathcal{H}, \epsilon) \approx \left(\frac{1}{\epsilon}\right)^d$.*[2] *Then for any $H \in \mathrm{co}_p(\mathcal{H})$, where $H =$*

[1] For $p = 1$, the hypothesis space matches that of Boosting.

[2] See theorem 4, page 14.

Given a set of m training samples X, a classifier $F \in \mathcal{X}$ chosen by a boosting algorithm with probability of at least $1 - \delta$,

$$\underbrace{\Pr(\text{error}|\hat{f})}_{\text{true risk}} \leq \underbrace{R^{\gamma}_{\text{emp}}(\hat{f})}_{\text{empirical risk}} + \underbrace{b(\delta, |\mathcal{H}|, m)}_{\text{confidence interval}} \tag{4.1}$$

where the function $b(\delta, |\mathcal{H}|, m)$ has the following properties:

1. $b(\delta, \cdot, \cdot)$ is non-increasing with δ;
2. $b(\cdot, |\mathcal{H}|, \cdot)$ is nondecreasing with $|\mathcal{H}|$;
3. $b(\cdot, \cdot, m)$ is non-increasing with m.

Figure 4.1: Form of generalisation performance bounds for boosting (after figure 2.8)

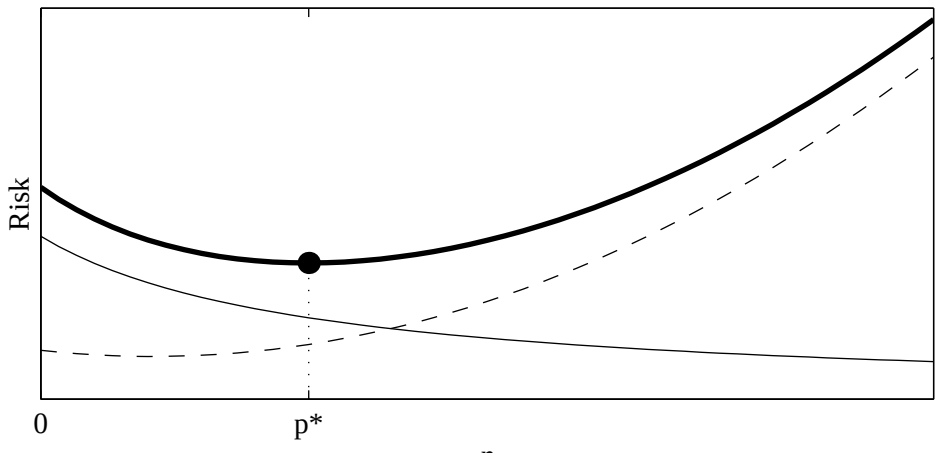


Figure 4.2: The effect of p on true risk

The true risk (thick line) is bounded by the sum of the empirical risk (solid line) and confidence interval (dashed line). The point chosen by the SRM principle is indicated; this is the optimal p value in the sense that it leads to the best generalisation performance.

$\operatorname{sign}(F)$, *the generalisation error over a set of* m *independent training samples with probability at least* $1-\delta$ *can be approximated by*

$$R(H) \lesssim R^{\gamma}_{\text{emp}}(F) + \sqrt{\frac{8}{m}\left[c(p)\, d\, \left(\frac{2}{\gamma}\right)^{\frac{2p}{2-p}} \log_2\left(\frac{2}{\gamma}\right) - \log_2 \delta\right]} \tag{4.2}$$

where $c(p)$ *is a constant that depends upon* p*.*

This theorem is necessarily approximate, due to the approximate nature of the theorems it is based upon and the loose nature of such generalisation bounds. However, it is clear that the confidence interval should decrease as $p \to 0$ (again assuming a nearly constant $c(p)$; and thus we can use the p parameter as an explicit parameter for capacity control.

4.2 Development of algorithms

Having established that boosting algorithms which operate on p-convex hulls theoretically have the desirable property of a tunable capacity via the p parameter, we turn to questions of how to modify AdaBoost to operate on a p-convex hull. We first describe a "naïve p-boosting" algorithm, which attempts to directly modify the classifier weights to produce this effect. After considering shortcomings of the naïve algorithm, we derive a gradient descent algorithm "strict p-boosting", using a restricted domain $\mathcal{X} = \operatorname{co}_p(\mathcal{H})$. Unfortunately, the cost functional optimised by this algorithm has undesirable properties which leads to the algorithm getting "stuck" for $p < 1$. As a result, we develop the "sloppy p-boosting" algorithm which sacrifices theoretical integrity for practical viability. Finally, we briefly consider a different type of p-boosting algorithm (still based on gradient descent) that uses regularisation to achieve $\mathcal{X} \approx \operatorname{co}_p(\mathcal{H})$. This algorithm is called the "gravity p-boosting" algorithm.

4.2.1 Naïve algorithm

The naïve algorithm (named with the benefit of hindsight) attempts to modify the b values of AdaBoost in a manner that will produce a "p-convex like" distribution of classifier weights.

The key feature of the algorithm is the calculation of the classifier weights b_t. Denoting the classifier weight that AdaBoost would have used as b'_t, we use the formula

$$b_t = (b'_t)^{\frac{1}{p}} = \left[-\frac{1}{2}\log\left(\frac{\epsilon_t}{1-\epsilon_t}\right)\right]^{\frac{1}{p}} \tag{4.3}$$

Thus, for $p < 1$ the hypotheses with low training error (and hence high classifier weights) have these weights "stretched" out. The effect of p is plotted figure 4.3.

There are two options for updating the sample weights w: based upon the value of b_t, or based upon b'_t (unchanged from AdaBoost). The first option yields the equation

$$w_{t+1,i} = \begin{cases} w_i|_t / Z_t \exp\left\{(b'_t)^{1/p}\right\} & \text{if } f_t(x_i) = y_i \\ w_i|_t / Z_t \exp\left\{-(b'_t)^{1/p}\right\} & \text{otherwise} \end{cases} \tag{4.4}$$

When using this first option, the $1/p$ power (which is greater than one for $p < 1$) is inside an exponential; and could (and does) lead to the value of the exponential getting extremely large (outside the limits of IEEE floating point numbers). Thus the sample weights are updated based upon b'_t (second option); they are identical to those chosen by the AdaBoost algorithm.

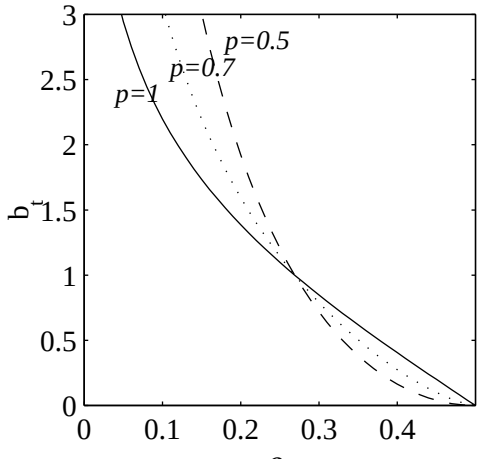


Figure 4.3: Effect of p on classifier weights for naïve algorithm
The three lines show b_t against ϵ_t for $p \in \{0.5, 0.7, 1.0\}$.

The naïvity of the algorithm becomes apparent when considered in the gradient descent framework. The sample weights are calculated the same as AdaBoost; thus given an identical initial state both algorithms would proceed in the same "direction". However the step sizes are different: AdaBoost performs a line search for the minimum in this direction and moves to that point; whereas the naïve algorithm also searches for that point *and then purposely avoids it!* Experiments with this algorithm prove to have indifferent performance.

4.3 Strict algorithm

The strict algorithm is the purest of all p-boosting algorithms considered in this thesis (although not the best performing). It performs gradient descent, using the same cost function and inner product as AdaBoost, but uses the the universal set $\mathcal{X} = \mathrm{co}_p(\mathcal{H})$. It is called the "strict" algorithm because it confines its line search to the p-convex hull (AdaBoost performs its line search along a line orthogonal to the combined hypothesis F_t). This idea is illustrated in figure 4.4.

4.3.1 Classifier weights

We derive an expression for the classifier weights, using the gradient descent framework of chapter 3. Algebraically, we are seeking a minimum for the expression

$$C(F_{t+1}) = \sum_{i=1}^{m} \exp\left\{-y_i F_{t+1}(\mathbf{x}_i)\right\} \tag{4.5}$$

subject to the constraint that

$$\|b\|_p = 1 \tag{4.6}$$

Substituting in the constraint, and performing some manipulations, we are looking for[3]

$$F_{t+1} = \frac{b_1 f_1 + \cdots + \alpha f_{t+1}}{(1+\alpha^p)^{1/p}} \tag{4.7}$$

[3]Note that here (unlike AdaBoost) *all* b values are updated (normalised) at each time iteration.

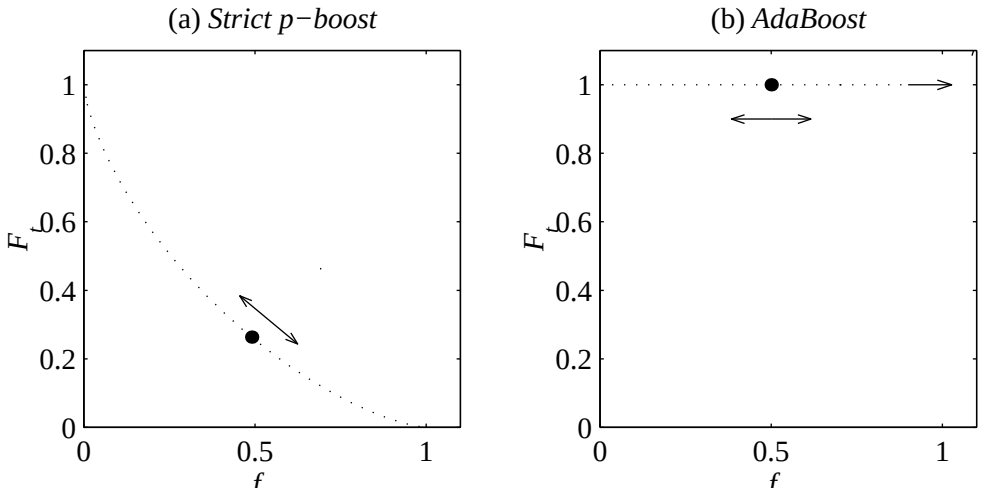


Figure 4.4: Line search for strict p-boosting algorithm

The strict p-boosting algorithm confines its line search to the p-convex hull of points, whereas AdaBoost searches orthogonal to the current combined hypothesis.

where

$$\alpha = \underset{\alpha}{\operatorname{argmin}} \sum_{i=1}^{m} \exp\left\{-y_i \left(\frac{F_t(\mathbf{x}_i) + \alpha h_{t+1}(\mathbf{x}_i)}{(1+\alpha^p)^{1/p}}\right)\right\} \tag{4.8}$$

Unfortunately, it is not possible to find a closed form solution for this expression as for AdaBoost; we instead have to be content with a numerical optimisation. The method chosen is Newton-Raphson iteration, which searches for $\partial C(F)/\partial\alpha = 0$ by repeatedly approximating it with a parabola until a tolerance is reached. Several technical points concerned with making the Newton-Raphson method robust are addressed in appendix B and [13].

4.3.2 Sample weights

As the strict algorithm is performing gradient descent, the sample weights are calculated using the results from theorem 5:

$$w_{t,i} = \frac{c'(y_i F_t(\mathbf{x}_i))}{\sum_{j=1}^{m} c'(y_j F(\mathbf{x}_j))} = \frac{-\exp\{y_i F_t(\mathbf{x}_i)\}}{\sum_{j=1}^{m} -\exp\{y_j F(\mathbf{x}_j)\}} \tag{4.9}$$

4.3.3 Existence of an optimal b_{t+1} for $p < 1$

As we shall see in chapter 6, when $p < 1$ the strict p-boosting algorithm suffers from a cost surface that is in many cases strictly increasing along the p-convex hull after a handful of iterations. The problem with a strictly increasing cost function is that it results in $F_{t+1} = F_t$, and the algorithm must be aborted (or continue in steady state endlessly).

In order to determine the mechanism through which this phenomenon occurs, a simple experiment was performed. This experiment trained the strict algorithm with $p = \frac{1}{2}$ on a simple dataset in $\mathbb{R}^2$ (randomly generated from the ring distribution; see appendix C) containing 10 samples. It was observed that sometimes there was no optimal solution at $t = 2$ (that is, *no* p-convex combination of the first two weak-learners produced a smaller cost than the first weak-learner only). It was furthermore observed that a sufficient condition for there to be no solution was:

1. The weak-learner h_1 classified 9 out of 10 training samples correctly;
2. The weak-learner h_2 classified 8 out of 10 training samples correctly, including the sample it classified *incorrectly* on the first round.

Adjusting the p parameter revealed that this behaviour always occurred for $p \leq 0.6695$, and did *not* occur for $p \geq 0.6698$. Thus under the conditions considered above, there appears to be a threshold value of p above which this behaviour is not observed.

Recalling that the cost functional $C(F)$ is the sum over the m training samples of the cost function approximation (4.5), it is useful to look at the contribution of each *sample* $S = (\mathbf{x}, y)$ to the total cost C, as a function of α and p:

$$c_S(\alpha, p) = \exp\left\{\frac{-yF_t(\mathbf{x}) - \alpha y h_{t+1}(\mathbf{x})}{(1+\alpha^p)^{\frac{1}{p}}}\right\} \tag{4.10}$$

When $\alpha = 0$, (4.10) reduces to

$$c_S(0, p) = \exp\{-yF_t(\mathbf{x})\} \tag{4.11}$$

Consider the meaning behind each part of (4.10). The term $yF_t(\mathbf{x})$ is simply the margin of the sample S at iteration t–we write this $m_{S,t}$. The term $y\alpha f_{t+1}(\mathbf{x})$ is equal to α if the weak-learner h_{t+1} classifies S correctly, or $-\alpha$ otherwise. The denominator $(1+\alpha^p)^{\frac{1}{p}}$ is equal to 1 at $\alpha = 0$ and approaches α as $\alpha \to \infty$.

When applied to our simple example, this analysis yields some insight into the processes that are occurring. Of the ten samples, seven were classified correctly by both weak-learning hypotheses; one was classified wrongly on the first iteration but correctly on the second; and two were classified correctly at first but wrongly on the second.

When performing the line search for $t = 2$, the margin $m_{(\mathbf{x}_i,y_i),1}$ will evaluate to one of two values: $+1$ if $h_1(\mathbf{x}_i) = y_i$ and -1 otherwise. Likewise, the margin of the weak-learning hypothesis h_2 will be $\pm\alpha$ depending upon whether it classified the sample correctly or not. As a result, there are four possibilities for the sample cost function, depending upon the history of the first two weak learning hypotheses:

$$\begin{aligned} c_{(-1\Rightarrow-1),i}(p,\alpha) &= \exp\left\{\tfrac{1+\alpha}{(1+\alpha^p)^{\frac{1}{p}}}\right\} \quad c_{(-1\Rightarrow+1),i}(p,\alpha) = \exp\left\{\tfrac{1-\alpha}{(1+\alpha^p)^{\frac{1}{p}}}\right\} \\ c_{(+1\Rightarrow-1),i}(p,\alpha) &= \exp\left\{\tfrac{-1+\alpha}{(1+\alpha^p)^{\frac{1}{p}}}\right\} \quad c_{(+1\Rightarrow+1),i}(p,\alpha) = \exp\left\{\tfrac{-1-\alpha}{(1+\alpha^p)^{\frac{1}{p}}}\right\} \end{aligned} \tag{4.12}$$

where the subscript to c gives the history: $(u \Rightarrow v)$ means that hypothesis $H_1 = \text{sign}(F_1)$ had margin $yF_1(\mathbf{x}) = u$, and that the weak learning hypothesis h_2 had margin $yh_2(\mathbf{x}) = v$. These functions are plotted against alpha in figure 4.5.

The shape of the $(-1,+1)$ and $(+1,-1)$ curves is sensible: as α increases, samples which were wrong and are now classified correctly decrease in weight, while the opposite occurs for samples which were right and are now wrong. The main qualitative difference between $p = 0.5$ and $p = 1$ is the shape of the $(-1 \Rightarrow -1)$ and $(+1 \Rightarrow +1)$ curves: these are flat for $p = 1$ (indicating that the algorithm is indifferent to samples which do not change their margin), but form a hill for $p < 1$. These hills are obviously not conducive to minima.

Part (e) of figure 4.5 displays the cost functional

$$C(\alpha, p) = \sum_{i=1}^{10} c_{(\mathbf{x},y)}(\alpha, p) = 7c_{(+1\Rightarrow+1)} + 2c_{(+1\Rightarrow-1)} + c_{(-1\Rightarrow+1)} \tag{4.13}$$

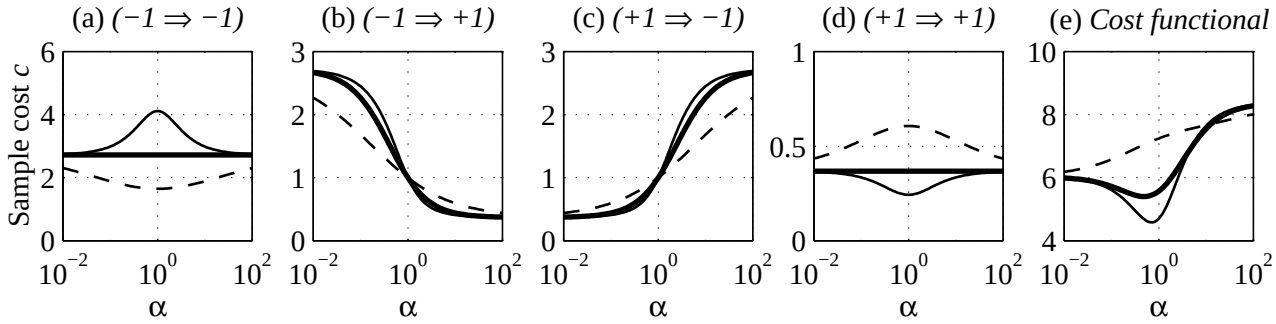


Figure 4.5: Sample cost contributions

Parts (a) through (d) show the sample cost at iteration $t = 2$ as a function of the history of the margins through weak-learners h_1 and h_2. The dashed line indicates $p = 0.5$, the thick line $p = 1$, and the thin solid line $p = 2$. Part (e) plots the cost functional *for the examples (10 samples) described in the text (in the example, $p = 0.5$ is used; the other curves are plotted for reference).*

for the same values of p. Note that a well-defined minimum exists for $p \geq 1$, but the cost function is monotonic increasing for $p = 0.5$. It is obvious that the contribution of $7c_{(+1\Rightarrow+1)}$ prevents a minimum from occurring.

While this analysis is by no means complete, some general observations can be made on likely reasons for the strict algorithm terminating early. Choosing $p < 1$ appears to produce a *maxima* in the per-sample cost functional for samples that had a positive margin on iteration t and were again classified correctly on iteration $t + 1$.

We have already seen from chapter 3 that as $t \to \infty$, the "easy" samples tend to be classified correctly very often (this result is true for AdaBoost, and by extrapolation true also for the strict algorithm, as both update their sample weights in the same manner). As a result, we would expect that many samples would have a cost function of the shape shown in figure 4.5 (d). On the other hand, samples that are "hard" tend to oscillate: as a result, there are few samples with a cost function in the shape of figure 4.5 (a) (which could balance out the effect). The net result is that as the number of training iterations increases, the cost functional becomes less and less likely to have a minima, and thus training terminates early.

4.4 Sloppy algorithm

Given the problems with the strict boost algorithm, it is reasonable to consider using an algorithm with a similar effect but a more desirable form for the cost function (in particular, one for which a minimum of the line search at $\alpha \neq 0$ is guaranteed to exist).

The *sloppy p-boosting algorithm* is one such algorithm. The simplification involves searching along the same line as AdaBoost, and then normalising to the p-convex hull *after* a suitable point has been found. In this way, the hypothesis F_t returned after round t is still $F_t \in \mathrm{co}_p(\mathcal{H})$, and the strictly increasing cost functions that troubled the strict algorithm are avoided. Figure 4.6 illustrates the differences in the line search between the sloppy and strict algorithms.

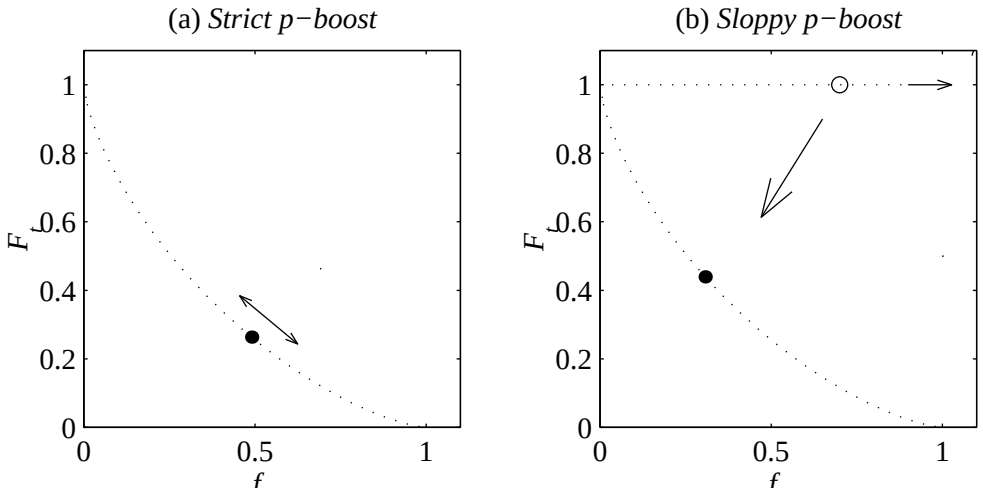


Figure 4.6: Line search for sloppy p-boosting algorithm

Comparison between the line search for the sloppy and strict algorithms. The strict algorithm searches directly along the p-convex hull. The sloppy algorithm searches along the same line as AdaBoost, and then normalises to put the point back onto the p-convex hull.

It should be noted that the sloppy algorithm is a straightforward generalisation of the normed boosting algorithms surveyed in chapter 3.

4.5 Gravity algorithm

The final variant of p-boosting considered in this thesis is somewhat different to the other two. This algorithm is less direct in its approach: rather than restricting the gradient descent algorithm to operate on a p-convex hull, it modifies the cost functional to "pull" the solution towards a low p-norm:

$$C(F) = \frac{1}{l}\sum_{i=1}^{l} \exp\left\{-\mathbf{y}_i F(\mathbf{x}_i)\right\} \quad + \quad \lambda\|b\|_p \tag{4.14}$$

where λ indicates how important a small p-norm is. This formulation also allows some latitude in the universal set $\mathcal{X}$ that is chosen: it would be sensible to choose $\mathcal{X} = \mathrm{co}_2(\mathcal{H})$ or even $\mathcal{X} = \mathrm{lin}(\mathcal{H})$ to ensure that no problems with gradient descent occur.

There are several problems with this algorithm. The most obvious is that there are now *two* parameters to optimise (p and λ). A less obvious problem is that rightmost term of (4.14) in its written form will be strictly increasing as $t \to \infty$, discounting the use of any standard optimisation method.

This algorithm was only briefly considered during the course of this project, due to the difficulties described above and lack of time. As a result, no experiments were performed using this algorithm; and it will not be considered further.

4.6 Hypotheses and learning machines

In this section, we return again to the distinction between a hypothesis and a learning machine. We first ask the question, "What is the difference between AdaBoost and the strict or sloppy algorithms

when $p = 1$?" This is an important question, as the hypothesis space for both algorithms is the same: $\mathcal{X} = \text{co}_1(\mathcal{H})$ (recall that AdaBoost normalises its hypothesis when training is completed). This line of reasoning is then continued to resolve a contradiction.

The difference between AdaBoost and the p-boosting algorithms when $p = 1$ is the slope of the cost function during training. Theorem 8 shows that the cost function for AdaBoost gets increasingly steep as t increases (due to $\|\mathbf{b}\|$ being unbounded). However, the p-boosting algorithms normalise $\|\mathbf{b}\|$ to 1 at every iteration; as a result the cost function is always the same shape. Thus, they are two quite different algorithms because the optimise over different cost functions.

We now extend these ideas further and generate an apparent contradiction, the resolution of which removes the potential for a great deal of confusion.

Theorem 10 on page 26 describes how AdaBoost is invariant to any nonzero proportional scaling of its classifier weights. This theorem also holds for the p-boosting algorithms, as they too generate their final hypothesis by thresholding a linear combination.

Consider two hypotheses $H_1 = \text{sign}(F_1)$ and $H_2 = \text{sign}(F_2)$ where $F_1 \in \text{co}_{p_1}(\mathcal{H})$, $F_2 \in \text{co}_{p_2}(\mathcal{H})$ and $p_1 \neq p_2$. It is clear that H_1 is invariant to any scaling βF_1 where $\beta > 0$. But if we choose

$$\beta = \frac{1}{\|F_1\|_{p_2}} \tag{4.15}$$

where $\|F_1\|_{p_2}$ means the p_2-norm of the weights of F_1, then suddenly

$$\|\beta F_1\|_{p_2} = \frac{\|F_1\|_{p_2}}{\|F_1\|_{p_2}} = 1 \tag{4.16}$$

and hence $\beta F_1 \in \text{co}_{p_2}(\mathcal{H})$! We have just shown that a hypothesis equivalent to H_1 (just a scaling of classifier weights) is on a *different* p-convex hull. Apparently, all p-convex hulls are equivalent; the choice of p is immaterial.

This conclusion is contradicted both by the theory developed in previous chapters and by the results presented in chapter 6. The way out is to observe that we have only considered scaling the *final* hypothesis F_1, not modifying the *algorithm*. The properties of a hypothesis depend upon how it was generated. Although the hypothesis class that H_1 and H_2 are in happen to be equivalent, each was generated by a distinct process (they were trained on $\text{co}_{p_1}(\mathcal{H})$ and $\text{co}_{p_2}(\mathcal{H})$ respectively). The p-convex hulls may all be equivalent–but the p-boosting *algorithms* are not.

If we try to extend the contradiction to the algorithms, we lose it altogether. Consider an algorithm which scaled H_1 by a factor β in (4.15) on every iteration. Then this algorithm is effectively constraining F_1 to a p_2-convex hull, which makes this algorithm identical to the p_2-boosting algorithm used to train F_2.

The performance bounds presented in the previous chapters also neatly avoid the contradiction, as they depend upon the hypothesis class *in which ERM was performed*, not the hypothesis class that the final hypothesis H_T happens to be in. Properties of a hypothesis depend upon the entire history of its training; nothing can be concluded without knowing something about how it was generated.

Chapter 5

Experiments

Extensive simulations of the p-boosting algorithms and AdaBoost were run on a microcomputer in order to compare their performance, both with the baseline AdaBoost algorithm, and with the theory developed in previous chapters. This chapter describes the experimental setup (including a brief description of the software developed) and the testing methodology to a level of detail sufficient to repeat the experiments.

5.1 Experimental setup

In this section a summary of the equipment and computer software used to perform the experiments is given. The full source code, datasets and test results are available on the CD-ROM attached inside the back cover of this thesis. Appendix D describes the contents of this CD-ROM in more detail (and provides information on how to access the online documentation for the software); and appendix C describes in detail the datasets used.

5.1.1 Hardware

The simulations were run on three microcomputers. The first was a 400MHz Celeron system with 256MB of memory, running RedHat Linux version 6.0 and `MATLAB` version 5.3.0.10183 (Release 11). The second was a 300MHz Celeron system with 128MB of memory, running RedHat Linux version 5.2 and the same version of `MATLAB`. The third was a 233MHz Cyrix M3 machine with 64MB of memory running Windows 95 version 4.00.1111 and `MATLAB` version 5.0.0.4073 (Student version).

5.1.2 Software

Although an incidental part of the project, the software package that was developed in order to perform the simulations is a significant piece of work in its own right. It was written from scratch to allow an accessible and efficient entry into the project area–which was not available at the commencement of the project. In particular, it includes many functions that allow visualisation of the algorithms. Use of this package will significantly lower the difficulty of beginning similar research or extending the ideas developed in this project.

The software was written as a `MATLAB` toolbox. Two weak learning algorithms (decision stumps and CART), several versions of the Boosting algorithm (including all mentioned in this thesis), an implementation of a Neural Network algorithm, an automated test harness, and assorted analysis and

visualisation functions are included. What follows is a brief description of the design and implementation of the software package as a whole.

5.1.3 Optimisation and numerical issues

The code was profiled extensively using MATLAB's inbuilt profiler, and efficient algorithms selected for the frequently executed sections. Tight sections of code, and certain data and code structures (for example, `for` loops), which are inefficient under MATLAB, were recoded in C to improve speed. (Details of how to interface this code were obtained from [21] and [22].) As each function was optimised, its output was compared with that of the un-optimised version to ensure equivalence of the two versions of code. The net effect of these optimisations was that simulations which would have required several years to run as initially coded were able to be run in a matter of days.

Minimising numerical errors was a major design goal. Several sections of code incorporate safeguards (such as periodic full recalculations in loops that are optimised via incremental calculations) to minimise the effect of numerical errors. In addition, 64 bit IEEE double precision floating point numbers are used exclusively.

In total, the source code was 270KB in size, comprising 235KB of MATLAB code and 35KB of C code. There are approximately 10000 lines in 191 MATLAB `.m` files and 1500 lines in 6 C source files. All source code was maintained under the CVS version control system.

5.2 Datasets

A total of seven datasets were used in the experiments. Four (ring0, ring10, ring20 and ring30) were synthetic datasets, randomly generated from a known distribution with noise added artificially. Two other datasets (sonar and wpbc[1]) were obtained from the UCI repository [4], and the final dataset (acacia) was ecological data used in a PhD thesis [17]. A summary of the features of each dataset appears in table 5.1 (Appendix C describes the datasets in more detail.) This group was selected to cover a broad range of situations: the four ring datasets allow the effect of noise to be isolated and contain a very high sample-to-attribute ratio; the sonar dataset is low-noise (generated from precise measurements) but contains a low sample-to-attribute ratio; and the wpbc and acacia datasets are examples of difficult and very difficult[2] real-world data.

5.3 Testing

In this section we detail the aims of the experiments, the procedure that was followed, and how the results were summarised and analysed.

5.3.1 Aims

The aims of the experiments were four-fold:

1. To test the generalisation performance of the p-boosting algorithms, and how this performance compares with the that of AdaBoost. It was expected that the p-boosting algorithms would outperform AdaBoost on noisy datasets.

[1] Wisconsin Prognostic Breast Cancer.

[2] Difficult and very difficult in that previous applications of machine learning algorithms to these datasets produce indifferent to poor results.

Dataset	$\mathcal{I}$	$\mathcal{O}$	Artificial noise	Size	Training samples	Test samples
ring0	$[0,1]^2$	$\{\pm 1\}$	0%	∞	50	5000
ring10	$[0,1]^2$	$\{\pm 1\}$	10%	∞	50	5000
ring20	$[0,1]^2$	$\{\pm 1\}$	20%	∞	50	5000
ring30	$[0,1]^2$	$\{\pm 1\}$	30%	∞	50	5000
sonar	$[0,1]^{60}$	$\{\pm 1\}$	0%	208	70	138
wpbc	$\subset \mathbb{R}^{33}$	$\{\pm 1\}$	0%	194	97	97
acacia	$\subset \mathbb{R}^{16}$	$\{\pm 1\}$	0%	204	102	102

Note that samples with missing attribute values were excluded from each dataset.

Table 5.1: Summary of dataset attributes

2. To verify that the qualitative behaviour (and quantitative behaviour wherever possible) of the p-boosting algorithms matched the theory developed in chapters 2, 3 and 4:

   - The p parameter controls the capacity of the function class. Thus, a generalisation error vs p plot should have a minimum at some optimal value p^* (figure 4.2).
   - The confidence interval in theorem 12 decreases as $p \to 0$. As a result, the true risk (test error) and empirical risk (training error) curves should match closely as $p \to 0$.

3. To verify that the algorithms were operating as designed. It was expected that a distribution of classifier weights would show that most of the weight is given to fewer and fewer classifiers for low p values.

4. To test the efficiency of the algorithms as compared to AdaBoost. An algorithm that generates a slightly better hypothesis but takes much longer to train may not be considered an improvement in some applications.

5.3.2 Method

A total of 31 tests were run, as detailed in table 5.2. Each test involved training a particular algorithm on a dataset (up to a maximum of *Iterations* training iterations). The test was repeated for *Trials* independent trials, each time with a newly generated (ring⋆) or randomly permuted and partitioned (sonar, wpbc, acacia) test and training dataset[3], and this whole procedure repeated for each p value.

The following data was recorded for each trial:

- All test parameters;
- Test and training datasets;
- Training and test error (unweighted) at each iteration;
- Margins of the training samples at both the iteration with the minimum *test* error and the final iteration;
- Classifier weights at the final iteration.

[3]Note that noise was added to the training ring⋆ datasets but *not* the test ring⋆ datasets.

Number	Algorithm	Dataset	Trials	Iterations	p values
1-4	AdaBoost	ring*	100	1000	-
5	AdaBoost	sonar	30	1000	-
6	AdaBoost	wpbc	50	10000	-
7	AdaBoost	acacia	50	10000	-
8-11	Naïve	ring*	50	1000	$\frac{1}{2} \leq p \leq 2; 10p \in \mathbb{N}$
12	Naïve	sonar	30	1000	$\frac{1}{2} \leq p \leq 2; 10p \in \mathbb{N}$
13-16	Strict	ring*	30	1000	$\frac{1}{2} \leq p \leq 2; 10p \in \mathbb{N}$
17	Strict	sonar	50	10000	$\frac{1}{2} \leq p \leq 2; 10p \in \mathbb{N}$
18	Strict	wpbc	20	5000	$\frac{1}{2} \leq p \leq 2; 10p \in \mathbb{N}$
19	Strict	acacia	20	10000	$\frac{1}{2} \leq p \leq 2; 10p \in \mathbb{N}$
21-24	Sloppy	ring*	30	1000	$\frac{1}{2} \leq p \leq 2; 10p \in \mathbb{N}$
25-28	Sloppy	ring*	30	10000	$\frac{1}{2} \leq p \leq 1; 10p \in \mathbb{N}$
29	Sloppy	sonar	50	10000	$\frac{1}{2} \leq p \leq 2; 10p \in \mathbb{N}$
30	Sloppy	wpbc	20	5000	$\frac{1}{2} \leq p \leq 2; 10p \in \mathbb{N}$
31	Sloppy	acacia	20	10000	$\frac{1}{2} \leq p \leq 2; 10p \in \mathbb{N}$

The weak learning algorithm is always *decision stumps*.

Table 5.2: Summary of experiments conducted

5.3.3 Analysis

More than two gigabytes of data was generated in the course of testing. This data was summarised across all trials for each (test, p-value) combination, and a summary file created. The summary file contained the following information:

- The mean and standard deviation of test and training error curves at each iteration;
- The number of trials that had not aborted before each iteration (recall that the algorithms would abort if the training error exceeded $\frac{1}{2}$, fell below 0, or if the cost function was strictly increasing);
- The value of the best (lowest) test error for each trial, and the number of training iterations after which this best error value occurred;

Each of the p-boosting algorithms had further statistics produced for each p value. These statistics were:

- The mean and standard deviation of the best test error at each p value, over all trials;
- The mean and standard deviation of the number of training iterations the best error was produced at, over all trials.

For each of the p-boosting algorithms, a crude form of structural risk minimisation (see section 2.4.1) was then performed, by selecting the p value with the *lowest mean best test error* to be p^* (see figure 6.3 on page 45). This is the p value that is used whenever comparisons of the performance of algorithms are made (it makes little sense to compare anything but the *best* performance!) The entire analysis described above was repeated for each dataset.

Chapter 6

Results and discussion

This chapter summarises the outcome of the experiments described in chapter 5, discusses their features, and compares them with the results expected from the theory in chapters 2 to 4.

An opening observation is that Boosting algorithms are very erratic in their behaviour. As a result, it is difficult to make conclusive judgments based on individual observations; at the very least statistical measures need to be used, and even so a huge amount of computing resources are required to obtain sufficient results to make confident conclusions.

6.1 Generalisation performance

Figure 6.1 is a high-level summary of results on the generalisation performance of the p-boosting variants as compared to AdaBoost. The graph compares the result from the hypothesis chosen via SRM (section 2.4.1) with AdaBoost.

It can be seen that although the average generalisation performance of the p-boosting algorithms is in most cases slightly better than that of AdaBoost, only in a few cases is the difference significant. The only dataset upon which the strict algorithm generalises significantly better than AdaBoost is acacia. The sloppy algorithm generalises significantly better on ring10, ring20, ring30 and acacia; while the naïve algorithm always outperforms AdaBoost by a small (never statistically significant) amount.

Thus, the sloppy algorithm appears to outperform AdaBoost on the noisy datasets considered (recall that the acacia dataset was chosen as it was a difficult dataset). This behaviour was expected–the algorithm was specifically designed to implement capacity control. It also appears that the naïve algorithm has little effect on generalisation performance (a somewhat surprising result–the discussion in chapter 4 concluded that the algorithm should be *worse* than AdaBoost. We shall return to this result shortly).

6.2 Training time

Figure 6.2 details how the training times of p-boosting algorithms compare with AdaBoost. It is quite clear that the sloppy and strict algorithms require slightly to substantially more training iterations than AdaBoost. Thus, the improved generalisation performance of these algorithms comes at the cost of increased training time. In section 6.4 we will examine the mechanisms that cause the increased training time, and suggest remedies.

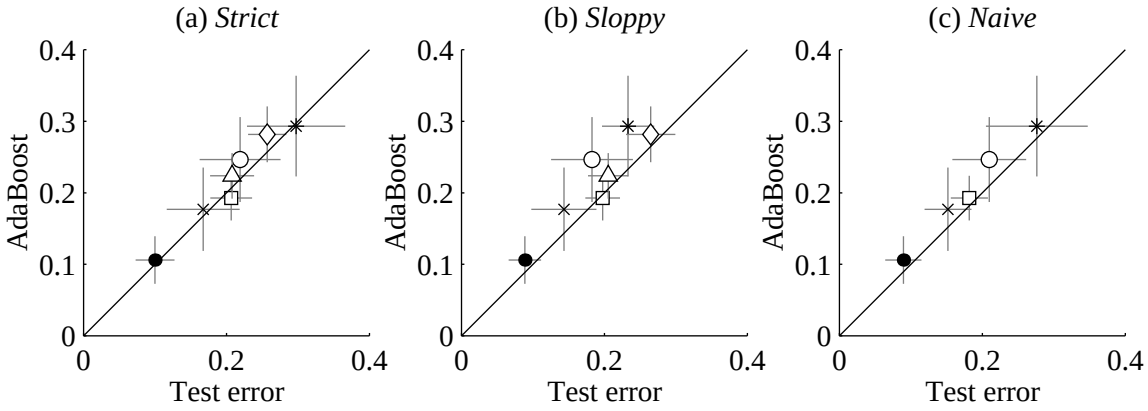


Figure 6.1: Comparison of AdaBoost and p-boosting test generalisation performance

The diagonal line indicates points where the test error of AdaBoost and the p-boosting algorithm are equal. Points above line indicate that the p-boosting algorithm generalises better than AdaBoost. The light bars indicate the spread of the trials, and are one standard deviation in length either side of the mean.

Key: • *ring0*, × *ring10*, ○ *ring20*, ∗ *ring30*, □ *sonar*, △ *wpbc*, ◊ *acacia*.

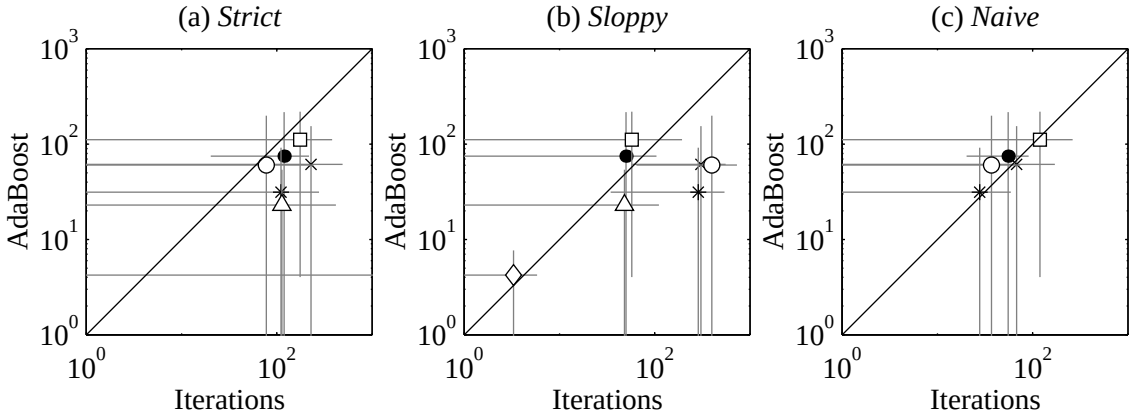


Figure 6.2: Comparison of AdaBoost and p-boosting training times

Both axes measure numbers of iterations. The y axis measures the average number of iterations required for AdaBoost to reach the minimum of the test error; the y axis is the same statistic for the p-boosting variant (at p^*). Points above the line indicate that AdaBoost requires more iterations to train than the p-boosting algorithm. Uneven error bars are due to the logarithmetic scale. Marker symbols indicate datasets:

Key: • *ring0*, × *ring10*, ○ *ring20*, ∗ *ring30*, □ *sonar*, △ *wpbc*, ◊ *acacia*.

6.3 Effect of p value on generalisation performance

Figure 6.3 expands the information in figure 6.1 to include information on *all* values of p (not just the best value p^*).

Consider first the sloppy algorithm (dashed line in figure 6.1). When trained on the noisy ring datasets, this algorithm appears to be implementing capacity control as expected, with the generalisation error reaching a minimum at an optimal value of p (compare with figure 4.2 on page 30). It is difficult to pinpoint exactly where this p^* value is due to the coarse resolution along the p scale, but it appears that $0.8 \leq p^* \leq 1.2$.

The results on other datasets are less clear: the low-noise datasets ring0 and sonar show a flat curve with no obvious minima (the slight downward slope indicating that such a minima may exist for $p > 2$); whereas the curves for wpbc and acacia both appear to be almost random.

The performance of the strict algorithm is very poor for $p < 1$ (due to the algorithm terminating training after few iterations); for $1 \leq p < 2$ it appears to reach a reasonable hypothesis (and a particularly good one in the case of the wpbc and acacia datasets). It shares the sloppy algorithm's general down-sloping characteristic on low-noise datasets, and also shares the existence of an optimal p value on the higher noise datasets. It is interesting to note that the p^* value differs substantially between the strict and sloppy algorithms (1.7 vs 0.9 on the ring30 dataset. This is a counter-intuitive result: the optimal capacity should be an intrinsic property of the *dataset*, not the algorithm, when both algorithms are operating from the same hypothesis class $\mathcal{X} = \mathrm{co}_p(\mathcal{H})$. One possible explanation is that deficiencies in the strict algorithm bias the error values for low p.

The performance of the naïve algorithm does not deviate significantly from that of AdaBoost. The p parameter appears to have little effect on this algorithm.

In summary, it appears that the strict and sloppy algorithms match the expected behaviour of an optimal p value minimising test error for the noisy datasets, but not for low-noise datasets (although there is an indication that maybe $p^* > 2$, and thus is off the scale of figure 6.3).

6.4 Training curves

Figure 6.4 shows a selection of training curves from the strict and sloppy algorithms, all trained on the ring30 dataset. There are three interesting observations concerning these curves. Firstly, part (a) shows that the training error for the strict algorithm, $p = 0.5$ is actually *increasing* with the number of iterations! This behaviour often recurs immediately before training is aborted; it is most likely a consequence of the difficult cost functional discussed in chapter 4.

Secondly, for $p = 0.5$ (parts (a) and (d)), the test error and training error curves follow each other quite closely. This behaviour has a theoretical explanation: for low values of p, the covering numbers of $\mathrm{co}_p(\mathcal{H})$ (and thus the confidence interval of figure 4.1) are small. We would therefore expect the test and training curves to match closely–which they do.

The third observation concerns the training of the sloppy algorithm. For $p \in \{0.5, 1\}$ the behaviour is quite erratic on a fine scale, but steady on average (random oscillations about a steady average value). It appears that this algorithm is "thrashing around" in the cost space, unable to find a direction (hypothesis) which will significantly reduce the cost functional. We will see shortly that the algorithm eventually will converge; the 10^3 iterations were not sufficient for this to happen. As a result, the results for the sloppy algorithm may be somewhat skewed towards higher test error values.

Figure 6.5 is a more detailed look at the mechanics of the training process for the AdaBoost

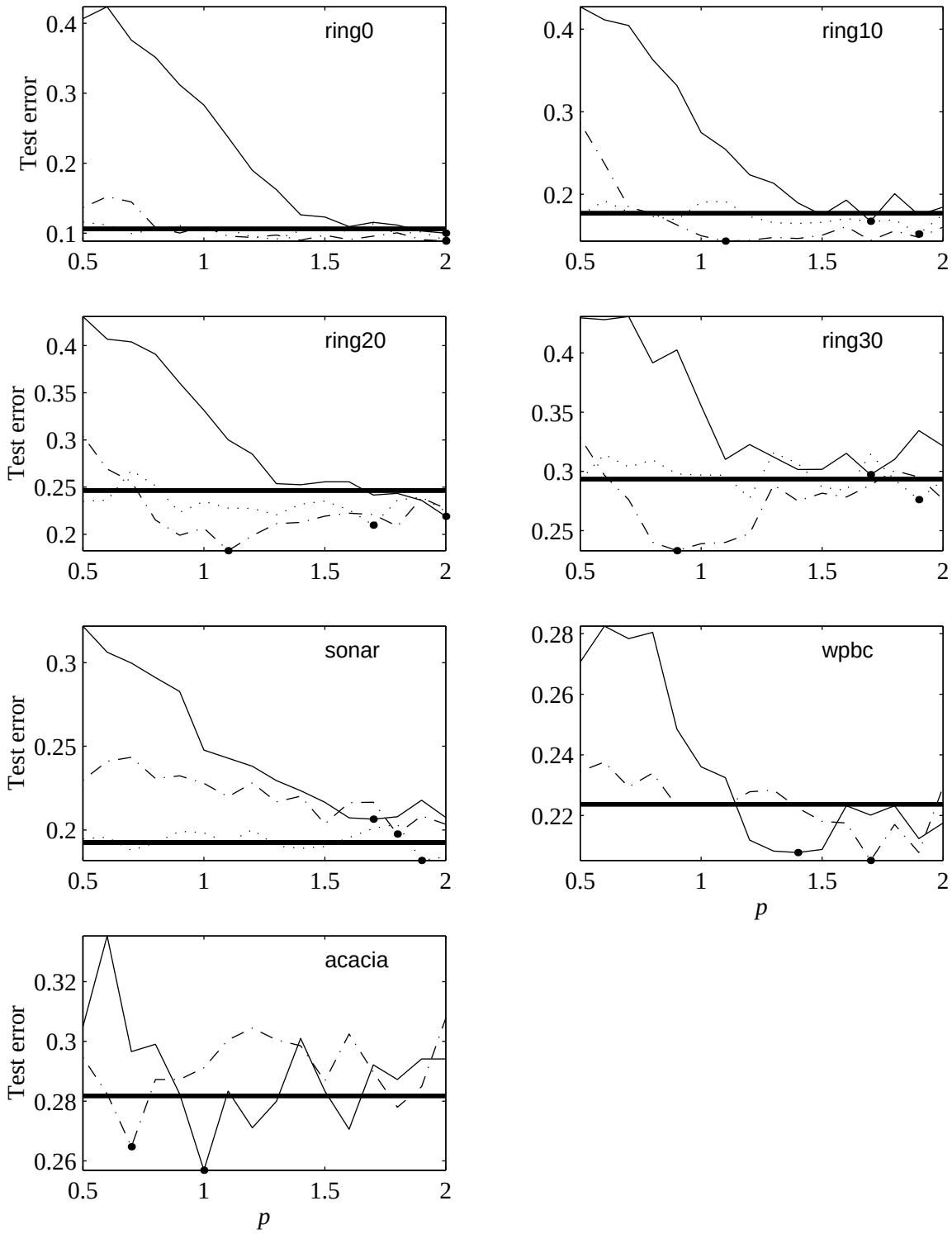


Figure 6.3: Effect of p on generalisation error

Average test error of each algorithm over all trials: AdaBoost (thick line), strict (thin solid), sloppy (dash-dot) and naïve (dotted) is shown for each dataset. The bullets • indicate the value p^* which is used in figure 6.1. Note that the sloppy algorithm was not trained on the wpbc *or* acacia *datasets. The y (test error) scale varies.*

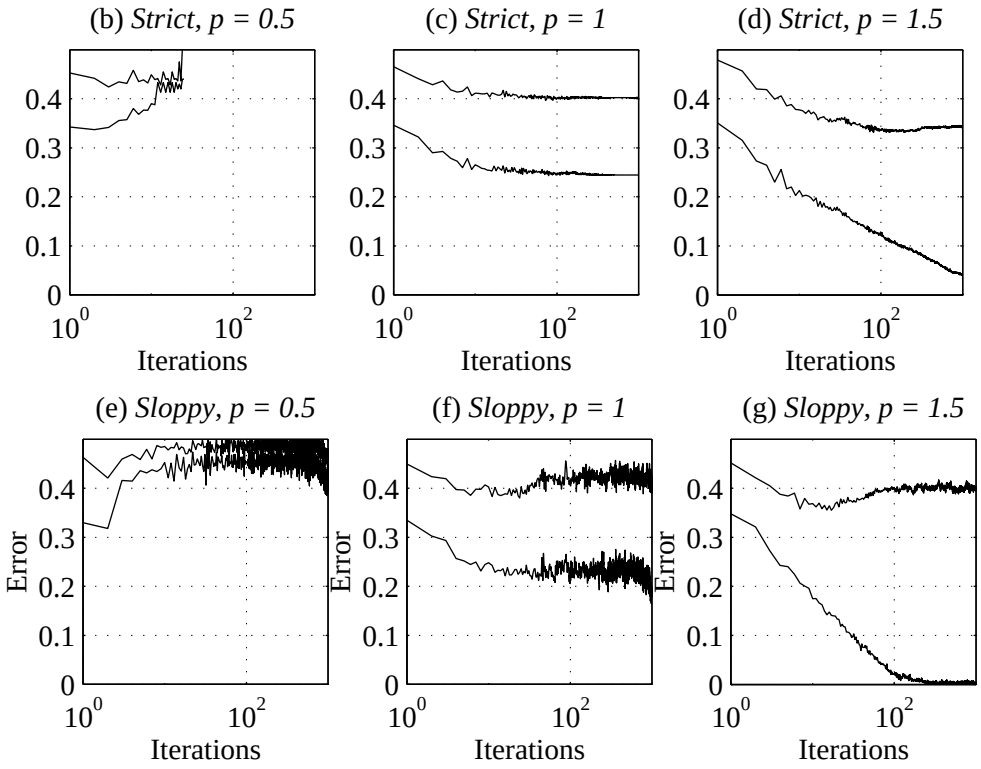


Figure 6.4: Selected test/training error curves

The curves show training error (grey line) and test error (black line) against iteration number. Each point of both curves is averaged over all trials that had not aborted at the specified iteration number.

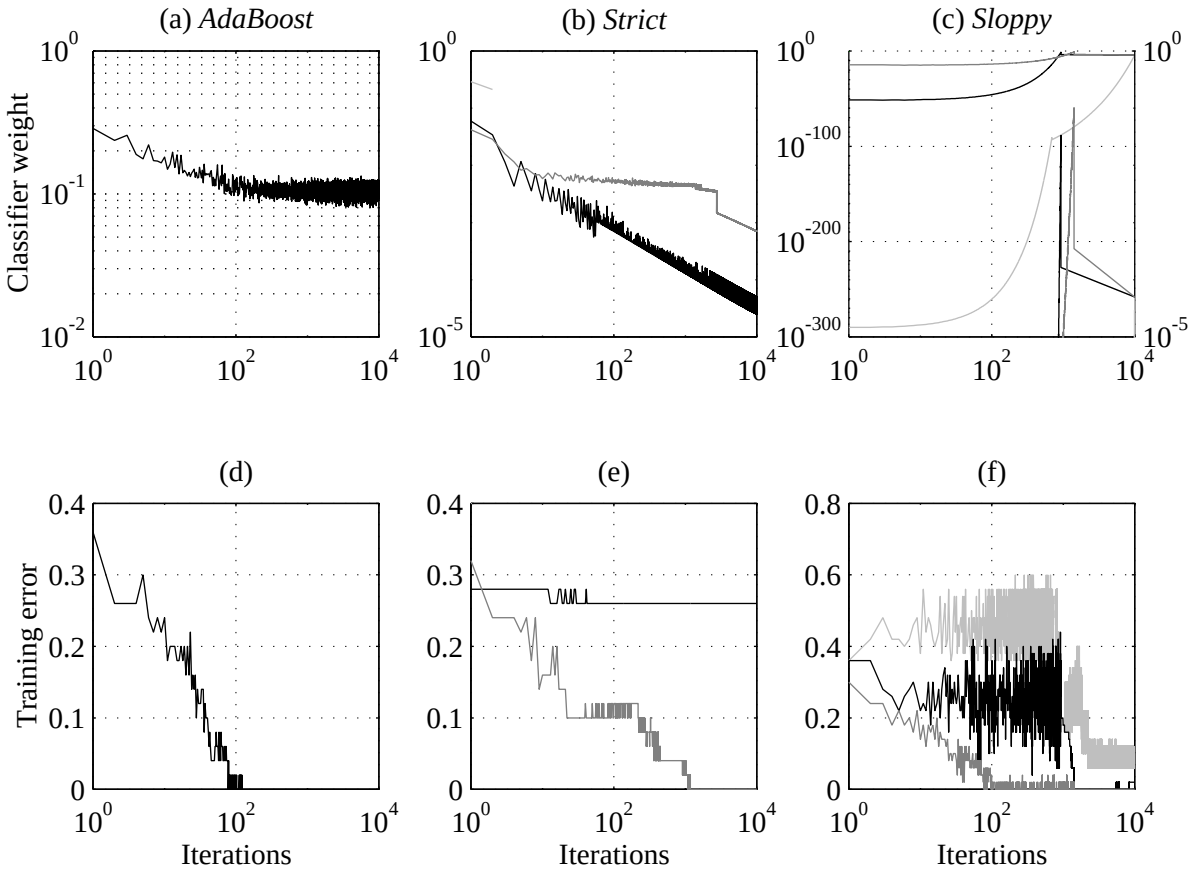


Figure 6.5: Details of the training process

The graphs detail a single *training run (not average behaviour). All three algorithms were trained to 10000 iterations, using the* ring30 *distribution, for* $p = 1$ *(black),* $p = 0.5$ *(dark gray) and* $p = 1.5$ *(light grey). Parts (a) to (c) show classifier weights; parts (d) through (f) are the corresponding training error curves. Note that (c) contains the* same *data plotted on two different scales–the two steep curves belong to the right hand scale. Note also the log scale on classifier weight graph.*

algorithm (as a reference) and the strict and sloppy algorithms.

We first describe the behaviour of AdaBoost, to provide a basis for comparison with the other algorithms. Part (a) shows that the classifier weights of the AdaBoost algorithm decrease as a constant power of t (note the logarithmetic scale)[1] until training error reaches zero, and then oscillate about this same value. Thus, the early weights are slightly larger than the later weights, but all are of a significant size.

Contrast this behaviour with that of the strict algorithm in part (b). Considering the $p = 1$ curve in part (b)[2] it is clear than these weights continue decreasing as $b_t = t^{-\lambda}$. The nearly flat training error curve is thus explained by the classifier weights getting too small for the weak hypotheses h_t as t increases to have any effect on the combined hypothesis H_t. The $p = 1.5$ curve in part (b) matches the shape of the AdaBoost curve until approximately 2000 iterations, where the weights also begin

[1] A straight line on a log-log plot indicates a relationship of the form $y = x^{\kappa}$, where κ is the slope of the line.

[2] The $p = 0.5$ curve aborted training after 2 iterations.

to drop off. The magnitude of the weights in the flat section are however much smaller than those of AdaBoost; it may be a consequence of this fact that the strict algorithm takes about 10 times longer to converge than AdaBoost.

Part (c) contains some very interesting behaviour which goes some way towards explaining the noisy part of the training error curve. Noting that the scale on the y axis reaches 10^{-300}, it is clear that most of the classifier weights are so small as to be practically zero. The second obvious feature is the "kink" in the $p = 1$ and $p = 0.5$ curves at approximately 800 iterations; this kink corresponds to the end of the noisy sections of part (f) and a (comparatively) rapid decrease in training error. The curves in the lower right corner of (c) ($p = 1$ and $p = 1.5$) are the *same* data plotted on the magnified scale (visible on the right hand scale of the graph; the range is 10^0 to 10^{-5}. The sharpness of the peaks indicate that only a few iterations have a significant amount of weight (in other words, b_t is sparse). This behaviour was expected by design.

In summary, these results indicate that the *concept* of using a p-convex hull is a good one: we saw in figure 6.1 that the final hypotheses generated by the sloppy algorithm often generalise better than AdaBoost, and figure 6.5 shows that this final hypothesis is indeed sparse in its parameter weights. The problem is in the *implementation*: a very large number of iterations are required to generate this sparse hypothesis; when the weights of hypotheses generated by most iterations are several *hundred* orders of magnitude below being significant.

6.4.1 Remedies for inefficient training

Observations in the previous section would suggest that the starting point for the gradient descent algorithm is particularly bad; as a result a lot of iterations are wasted in getting to a good starting point (the kink in part (c)) from where the optimisation can converge quite quickly. Another optimisation strategy that is less sensitive to starting point may prove to be more efficient (such as annealing); this algorithm could prove to be *very* useful if the slow initial training could be avoided.

Of course, it is not at all clear how to "start at a different point", as the algorithms as currently implemented start in a 1-dimensional space and add one dimension per iteration.

Alternatively, it may be possible to accelerate the training of the sloppy algorithm by choosing a more aggressive cost function. For example, the cost function

$$c(\alpha) = e^{-\alpha t^{\lambda}} \tag{6.1}$$

where t is the iteration number, and $\lambda > 0$ a constant parameter would serve to accelerate training, where λ controls the degree of acceleration ($\lambda = 0$ is equivalent to the sloppy algorithm). The main feature of this cost function is that it gets steeper as the number of iterations increases, and thus has the property of guaranteed zero training error. However, it may be that this algorithm would have its own problems. Insufficient time was available to further investigate this refinement.

6.5 Further observations

The following discussion concerns noteworthy features that were observed but have not been described previously.

6.5.1 Hard datasets

The full set of training curves in appendix E show that very rarely were the learning machines considered here (including AdaBoost) able to train for more than about 100 iterations on the acacia dataset. In the case of AdaBoost and the strict algorithm, the weak learner was not sufficiently good to continue generating hypotheses with a training error of less than $\frac{1}{2}$.

The wpbc dataset was also difficult, with only about one half of the trials making it through 1000 training iterations without terminating. A stronger weak learning algorithm (such as a deeper decision tree), or a method of restarting training after it has aborted (one such is described in [3]) could be used to avoid termination of the algorithm. Only the strict algorithm terminated on any other datasets.

6.5.2 Naïve p-boost

Observation of figures 6.1 and 6.3 shows that the performance of the naïve algorithm closely matches that of the AdaBoost algorithm; the variation between the two being insignificant (the two algorithms are equivalent when $p = 1$. This is a surprising result–in chapter 4 it was shown that the naïve algorithm systematically chooses a *less than optimal solution* by modifying distance chosen by the line search.

Thus, we observe that AdaBoost is quite robust to modifications to the *classifier* weights (of course, these modifications were in the specific form of a monotonic transform–arbitrary modifications would likely have an adverse effect). It would be expected, therefore, that the *sample* weights must cause the success of the algorithm. This assertion was also made by Breiman:

> After testing [an algorithm equivalent to AdaBoost] I suspected that its success lay not in its specific form but in its adaptive resampling property, where increasing weight was placed on those cases more frequently misclassified. [5]

Chapter 7

Conclusion

This thesis has considered variants of the AdaBoost machine learning algorithm, with the desirable property of explicit capacity control, implemented by confining the combined hypotheses to a p-convex hull of underlying hypotheses. Theoretical results indicate that these algorithms should outperform AdaBoost on noisy datasets.

Several "p-boosting" algorithms were developed: a naïve approach (which showed no noticeable improvement over AdaBoost), a "strict" algorithm confined strictly to the p-convex hull at all times, and a "sloppy" variant less restricted during the line search. A software toolbox was constructed to enable experimental data to be collected, and this toolbox used to perform extensive simulations.

The strict algorithm showed a small improvement over AdaBoost over the difficult acacia dataset, but in general performed poorly (particularly for $p < 1$). The poor performance was caused by the cost functional proving to be strictly increasing, or to have a minimum very close to the previous hypothesis (in effect the gradient descent algorithm did not move significantly from its starting point). Analysis of the features of the cost functional showed that "easy" samples produced hill-shaped sample cost functions that would contribute to a strictly a increasing cost functional.

The sloppy algorithm performed well on noisy datasets, with the shape of the capacity-vs-generalisation error curve matching the theoretical prediction, and the generalisation error at the optimal p value significantly (up to 25%) improved on AdaBoost (which already performs *very* well). While it appears that this algorithm did find a good solution, which was sparse as predicted by the theory, it did so in a very inefficient manner (particularly for $p < 1.2$), with up to several thousand iterations spent adding hypotheses with weights several hundred orders of magnitude below a significant level. A form of the cost function designed to accelerate the training process is proposed as a potential solution; alternatively a different optimisation strategy could be used.

It can be concluded that the *concept* of using a p-convex hull has merit as a method of increasing the noise-tolerance of the already very strong AdaBoost algorithm, but the algorithms that were developed to *implement* the idea were lacking.

No quantitative comparisons between p-boosting algorithms and other AdaBoost derivatives with similar goals (soft margins [19], DOOM I/II [16, 14]) were made. However, these approaches do not appear incompatible with the p-convex hull approach; it is likely that a combined approach would be superior to both.

Appendix A

Decision Stumps

This appendix studies a simple example of a practical learning machine, that has some nice properties (and was used to perform all of the experiments in this thesis). The *decision stumps* algorithm is a very simple learning machine. As a stand-alone tool it is almost useless; but when combined with the boosting algorithm it can generate very good hypotheses. Additionally, it has a fixed VC dimension which aids in the analysis of learning algorithms based upon it.

A.1 The decision stump learning machine

The decision stump algorithm divides the input space into two disjoint regions, with the boundary between them running perpendicular to one of the axes (the decision boundary is an axis-orthogonal hyperplane). Each region is given a label. Figure A.1 shows the decision boundaries of two decision stump classifiers.

The algorithm constructs a list of possible split locations and exhaustively searches this list for the minimum of the cost function (misclassification risk). Possible split locations are determined by projecting every data point onto each axis in turn, and bisecting each pair of consecutive projections to determine the candidate point. Figure A.2 illustrates the process.

A.2 Properties of decision stumps

Theorem 13 (VC dimension of decision stumps) *The set $\mathcal{S}$ of decision stump classifiers on an domain $\mathcal{I} \subseteq \mathbb{R}^d$ has a VC dimension of bounded by*

$$\mathrm{VCdim}(\mathcal{S}) \leq d + 1 \tag{A.1}$$

and, for a training set X of size $m \gg d$ with sufficient variation in its $\mathbf{x}$ parameters, has a constant VC dimension.

Proof: The set $\mathcal{S}$ of all axis orthogonal hyperplanes is contained within the set $\mathcal{H}$ of d dimensional hyperplanes (which has VC dimension $d + 1$.

That the VC dimension is constant is intuitively sensible but tricky to prove formally. As no subsequent result *relies* upon this property, we will not elaborate further.

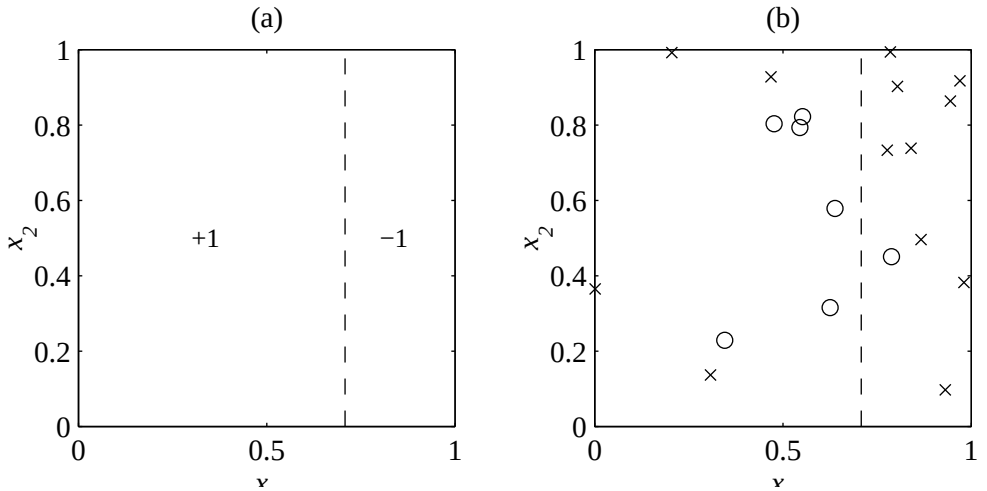


Figure A.1: A decision stump classifier

Part (a) shows the decision boundary of a decision stump classifier, and the label of each region. Part (b) the data it was trained on, where $\times = -1$ *and* $\circ = +1$.

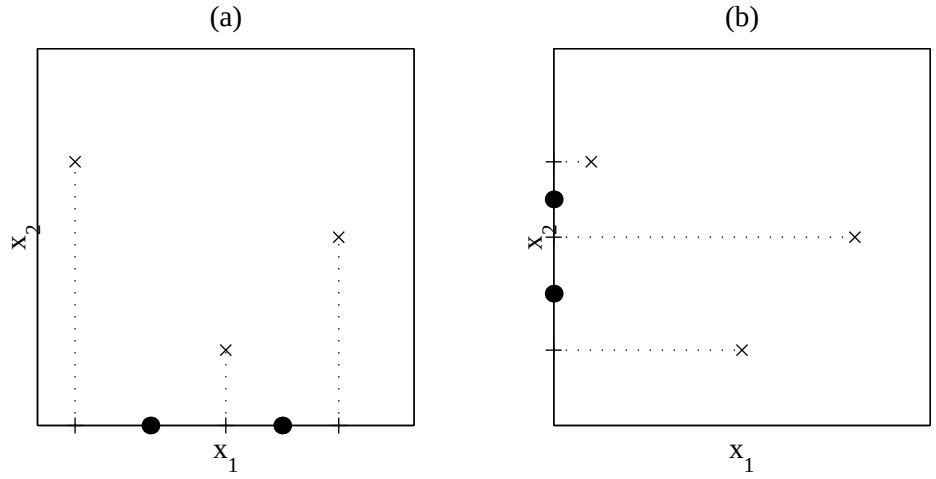


Figure A.2: Candidate points for decision stump split

Data points ($\times$) *are projected onto the axes. Each consecutive pair of projections is then bisected, to generate candidate split points* ($\bullet$).

Appendix B

Details of Newton-Raphson method

This appendix details Newton-Raphson search implemented in the sloppy and strict algorithms.

The following code sequence, which is edited to remove debugging code, is obtained from the file `trainagain.m` in the `normboost` object folder.

```
function [obj_r, context] = trainagain(obj)

if (aborted(obj))
   obj_r = obj;
   warning('trainagain: attempt to train when training aborted');
   return;
end

x_data = x(obj);  y_data = y(obj);  w_data = w(obj);  p = norm(obj);

% create and train a new classifier
new_c = train(weaklearner(obj), x_data, y_data, w_data);

% find the training error
new_error = training_error(new_c);

% see what this algorithm does to our data
new_y = classify(new_c, x_data);


% find if we need to abort
if (new_error == 0)
   % Zero error -- one classifier can do perfectly by itself
   obj_r = abort(obj);
   return;
end

if (new_error >= 0.5)
   % Error of 0.5 -- random guessing does just as well
   obj_r = abort(obj);
   return;
end

% Calculate alpha
if (iterations(obj) == 0)
   % First iteration -- set alpha to 1 and marg to the margins of the
   % trained weak learner.

   alpha = 1.0;
   marg = (y_data*2-1) .* (new_y*2-1);
```

```
   new_b = [1.0];

else
   % Calculate alpha using a line search.  This uses the Newton-Raphson method
   % to find the minimum.

   % Initialisation
   new_alpha = 0.5 / iterations(obj);
   d = 1;
   min_d = inf;
   max_d = -inf;
   last_c = 1000000;
   tolerance = 0.001;


   % Iterate
   iter = 0;
   while ((abs(d) >= tolerance) & (iter < 1000))
      alpha = new_alpha;

      [c, d, d2] = eval_cf(obj, new_c, alpha);
      min_d = min([d min_d]);
      max_d = max([d max_d]);

      % Handle d2=0 in a very crude manner, which allows us to continue
      if (d2 < eps)
 d2 = 0.01;
      end

      new_alpha = alpha - (d / d2);

      % Never let alpha move more than half the distance to zero.  This
      % can lead to alpha < zero, which causes all kinds of problems.
      min_alpha = alpha / 2;
      new_alpha = max([min_alpha new_alpha]);

      % This stopping rule is designed to detect the situation where there
      % is no solution as dc/dalpha > 0 for all alpha.  It halts the
      % training if there is no change in the cost function, and we have
      % not found a zero crossing of the derivative.
      if ((abs(c - last_c) < tolerance) & (min_d*max_d > 0))
 break;
      end
      last_c = c;

      iter = iter + 1;
   end

   % Test for lack of convergence
   if (iter >= 100)
      warning('trainagain: failed to converge after 100 iterations');
   end

   % Test for minimum/maximum at our found point.  We do this on the sign
   % of the second derivative: if positive, we found a minimum; if
   % negative we found a maximum.
   [c, d, d2, marg] = eval_cf(obj, new_c, alpha);

   if (d2 <= 0.0)
```

```
      alpha = 0;
      obj = abort(obj);
   end

   % Calculate our new classifier weights
   old_b = classifier_weights(obj);
   new_b = [old_b alpha] ./ pnorm([old_b alpha], p);

   one_norm = pnorm(new_b, 1)
end

% Update the sample weights.  These are calculated as the derivatives
% of the cost _function_ (not functional) and then normalised with a
% 1-norm equal to 1.
%
% For a cost function of exp(-x), dc/dx = -exp(-x).  The minus gets
% normalised out, and can be ignored.

new_w = exp(-marg);
new_w = new_w ./ sum(new_w);


% Update our weights
obj = add_iteration(obj, new_c, new_b, new_w);
obj.margins = marg;

obj_r = obj;
```

Appendix C

Description of datasets used

This appendix contains a brief description of the datasets used in the experiments. Details of the problem domain, descriptions of the attributes, and the results of previous studies using these datasets are given.

C.1 Ring dataset

The ring dataset is generated from a two dimensional mathematical distribution on the unit square. The sample probability density is constant within the unit square:

$$p((x_1, x_2)) = \begin{cases} 1 & \text{if } (x_1, x_2) \in [0,1]^2 \\ 0 & \text{otherwise} \end{cases} \tag{C.1}$$

and the label value is $+1$ within a circle of radius $\sqrt{1/8}$ centred at $(\frac{1}{2}, \frac{1}{2})$, or -1 otherwise. The decision boundary of this dataset is plotted in figure C.1.

The ring10, ring20 and ring30 datasets have noise added artificially: a proportion of samples n is selected *without replacement* from the generated data, and each of these samples has its label "flipped".

No previous experiments have been performed using this data. However, as no noise is added to the *test* datasets used, it is possible for the test error to reach 0 if the algorithm generalises perfectly.

C.2 Acacia dataset

The acacia dataset is an example of ecological/geographical data. The attributes of the dataset are listed in table C.1; there were 204 samples. The intended application of the dataset was to generate a learning machine that could detect the presence or otherwise of certain species from site and satellite data.

The acacia dataset considered in this thesis used a subset of the data; in particular columns 19 through 21 were ignored; the two-class classification problem being detecting the presence of *Acacia mabellae* (column 19) from the site attributes (columns 1 through 18). Column 3 (the site number) was removed from the dataset, leaving a total of 17 attributes. (The *Acacia mabellae* species was chosen because it had a roughly even number of positive and negative examples (119 and 85).

Previous work with this dataset [17, 24] used a Support Vector Machine [23]. The average test error over 100 trials for a partition of 144 training samples and 60 test samples was 32.6%.

C.3 Sonar dataset

This dataset (from the UCI repository [4]) contains data obtained by bouncing sonar signals off two objects: a metal cylinder and rock, under similar conditions. The sonar signal is preprocessed into

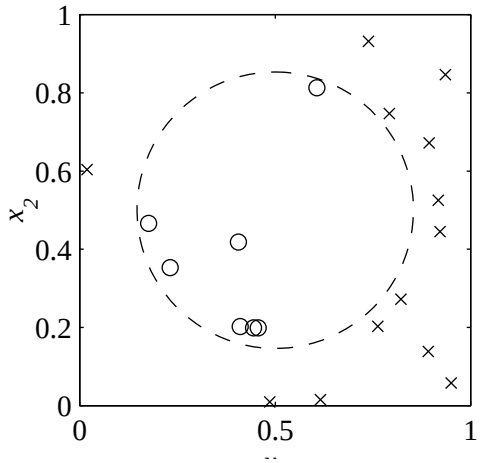


Figure C.1: The ring distribution
The markers are sample data points ($\times = +1$, $\bullet = -1$). The decision boundary is plotted in the dashed line.

No.	Description
1	easting
2	northing
3	site number
4	elevation
5	drainage area
6	slope (percent)
7	flow path length
8	landsat band 1
9	landsat band 2
10	landsat band 3
11	landsat band 4
12	landsat band 5
13	landsat band 6
14	landsat band 7
15	nutrient supply index
16	geology
17	net radiation
18	*Acacia mabellae* present
19	*Breynia oblongifolia* present
20	*Eucalyptus maculata* present
21	*Eucalyptus tereticornis* present

Table C.1: Attributes of the acacia dataset

60 values in the range $[0, 1]$ which each measure the energy in a particular frequency band. The experiment was repeated with the cylinders rotated at different angles.

The two class classification problem is to detect whether the object is a rock or metal cylinder. There are 208 examples in the dataset, comprising 97 cylinders and 111 rocks.

Previous experiments using this dataset [11] using a Neural Network over 13 trials of 192 training samples and 16 test samples resulted in a test error of 15.3%.

C.4 Wisconsin prognostic breast cancer dataset

The wpbc dataset is also from the UCI repository [4], and comes from the medical domain. It contains prognostic information for breast cancer patients of a Wisconsin doctor[1].

A total of 32 attributes are recorded, which contain various summarised information on the status of the patient, their previous history, and cell nuclei in breast tissue samples. The machine learning problem is to predict whether the breast cancer will recur. The dataset contains a total of 198 samples, of which 151 of the samples are (non-recurring) and 47 positive (recurring). Of these 198 samples, 4 have a missing attribute; these samples are ignored to give a total of 194 samples.

[1] Of course, the patient details are not identifiable through the dataset.

Appendix D

Contents of the attached CD-ROM

The attached disk is a standard 650MB cd-recordable that should be readable in any CD-ROM drive. It contains a Microsoft Joliet file system, readable by the Windows 95 and Linux[1] operating systems.

The root directory of the CD-ROM contains the following subdirectories and files:

- `cvsroot` contains the CVS repository for all software developed.
- `datasets` contains `MATLAB` versions of all datasets used. By setting the `DATASET_SAVE_PATH` global variable in `MATLAB` to point to this directory, the `dataset` class will be able to find these datasets.
- `data` contains the test and summary files for all of the tests that were completed (the raw data was in the order of 2.5 Gigabytes compressed and could not be included).
- `thesis` contains a checked-out version of the `thesis` CVS module (this document).
- `matlab` contains a checked-out version of the `matlab` CVS module. By adding this directory and all of its subdirectories that do not begin with an @ character to the `MATLAB` search path, it should be possible to execute the code and use the `MATLAB` toolbox.
- `license.txt` describes the conditions under which the information on the CD-ROM may be used, modified and distributed.
- `matlab_install.txt` describes in detail how to install and build the `MATLAB` toolbox. It also describes the necessary environment (both external and internal to `MATLAB`) that is required to use the software.
- `matlab_tutorial.txt` is an introduction to the `MATLAB` toolbox, describing its structure and capabilities.
- `accessing_tests.txt` describes how to view the test result summaries included in the `data` subdirectory.

[1] With the appropriate kernel options installed.

Appendix E

Detailed results

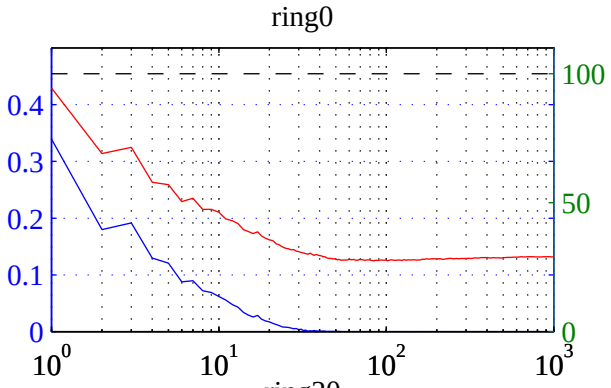
This appendix includes all test/training curves, and scatterplots of the best test error against p, and the best test iteration against p. Axis labels are neglected in order to avoid clutter.

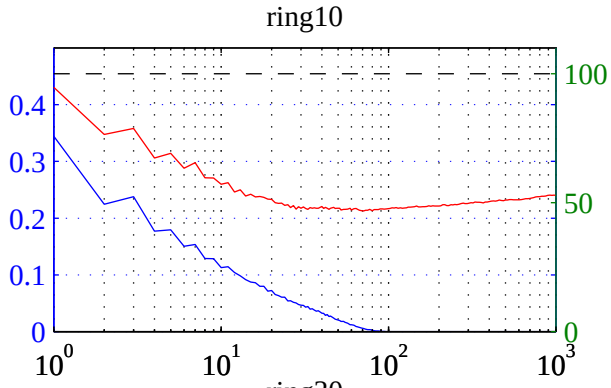
The training curves are averages over all trials. The x axis is trial number; the y axis is test or training error. Test error is the black curve; training error is the grey curve. The number of trials completing a certain number of iterations is plotted in the dashed line, on the right hand scale.

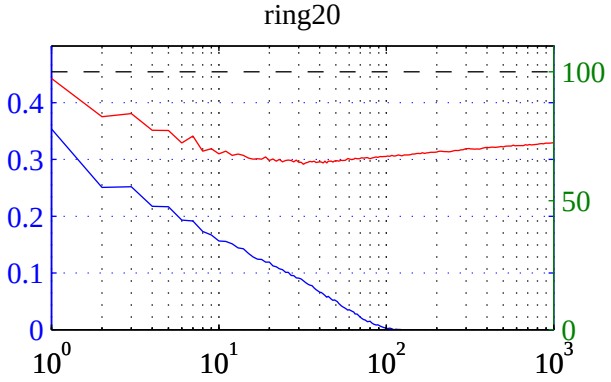
The best test error and best test iteration curves show the scatter over all trials. The x axis is the p value; the y axis the best test error or number of iterations at which this occurs.

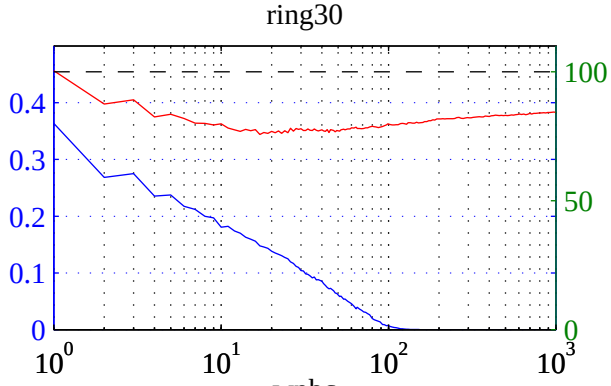
The thin lines indicate averages (these are also plotted in figure 6.3). The dotted line indicates one standard deviation above and below. The thick line indicates AdaBoost's result over that training set, with the thick dotted line one standard deviation above and below.

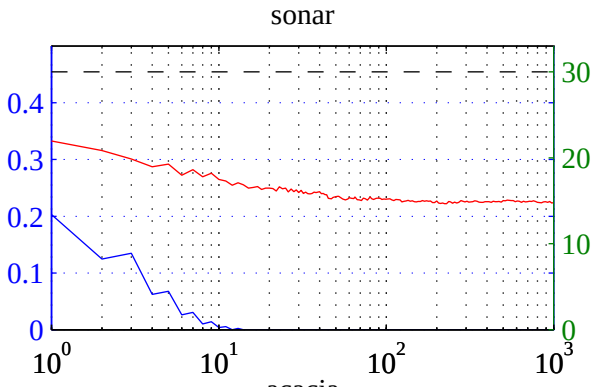
E.1 AdaBoost

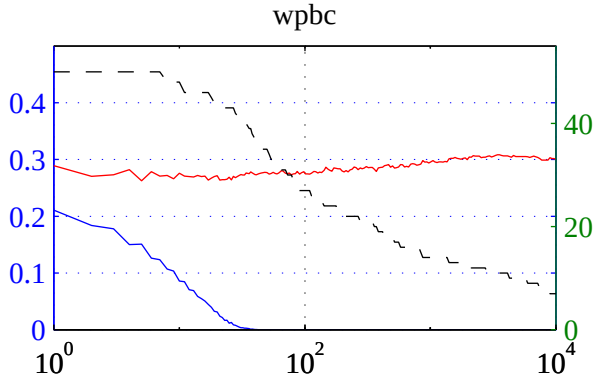

ring0
0.4
0.3
0.2
0.1
0
100
50
0
10^0
10^1
10^2
10^3

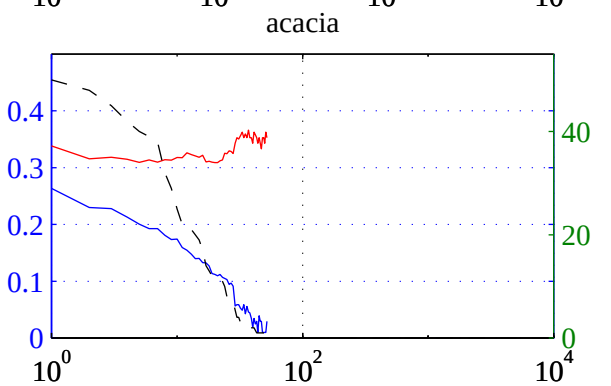

ring10
0.4
0.3
0.2
0.1
0
100
50
0
10^0
10^1
10^2
10^3


ring20
0.4
0.3
0.2
0.1
0
100
50
0
10^0
10^1
10^2
10^3


ring30
0.4
0.3
0.2
0.1
0
100
50
0
10^0
10^1
10^2
10^3

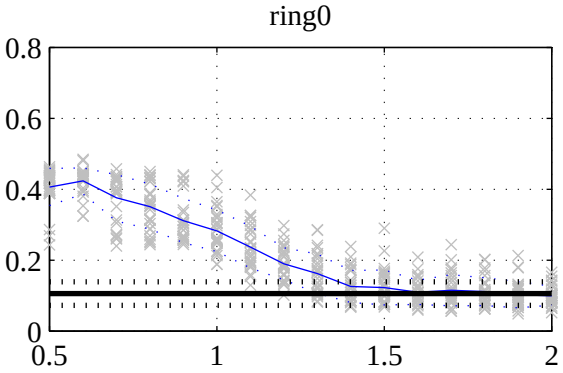

sonar
0.4
0.3
0.2
0.1
0
30
20
10
0
10^0
10^1
10^2
10^3

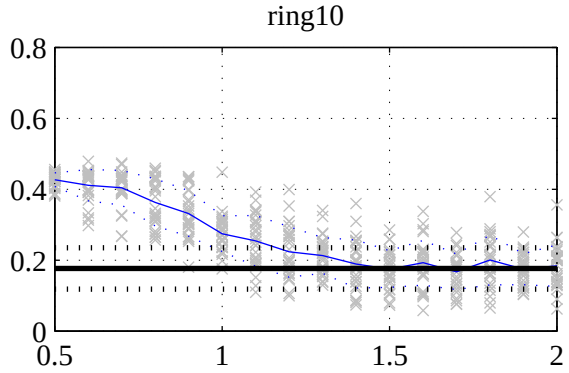

wpbc
0.4
0.3
0.2
0.1
0
40
20
0
10^0
10^2
10^4

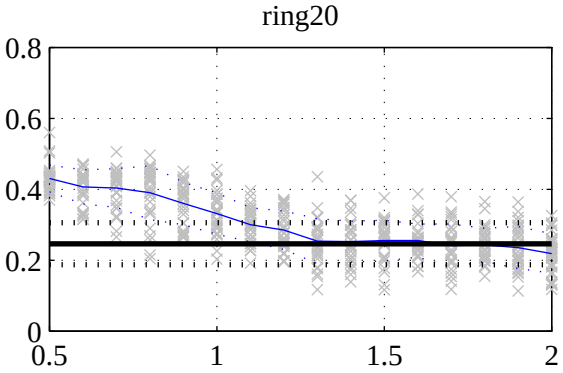

acacia
0.4
0.3
0.2
0.1
0
40
20
0
10^0
10^2
10^4

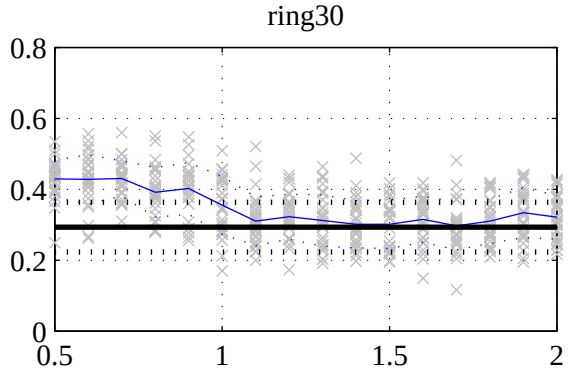
E.2 Strict

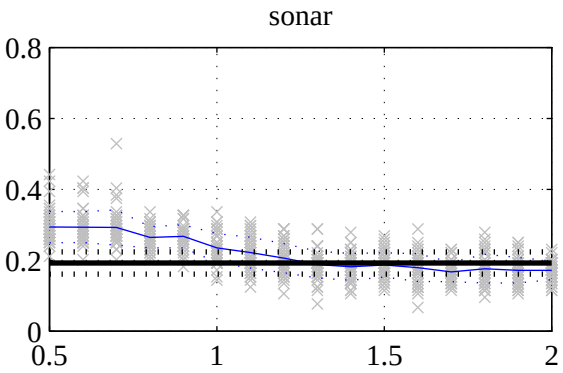
Best test error vs p

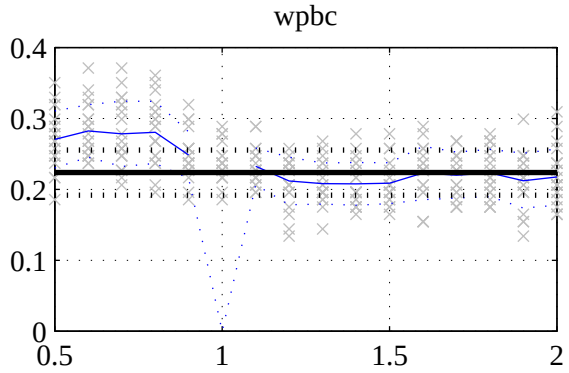


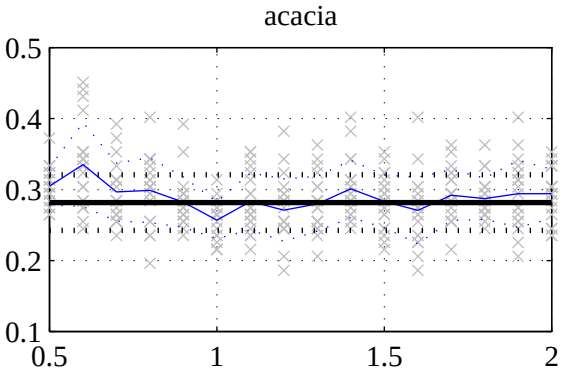




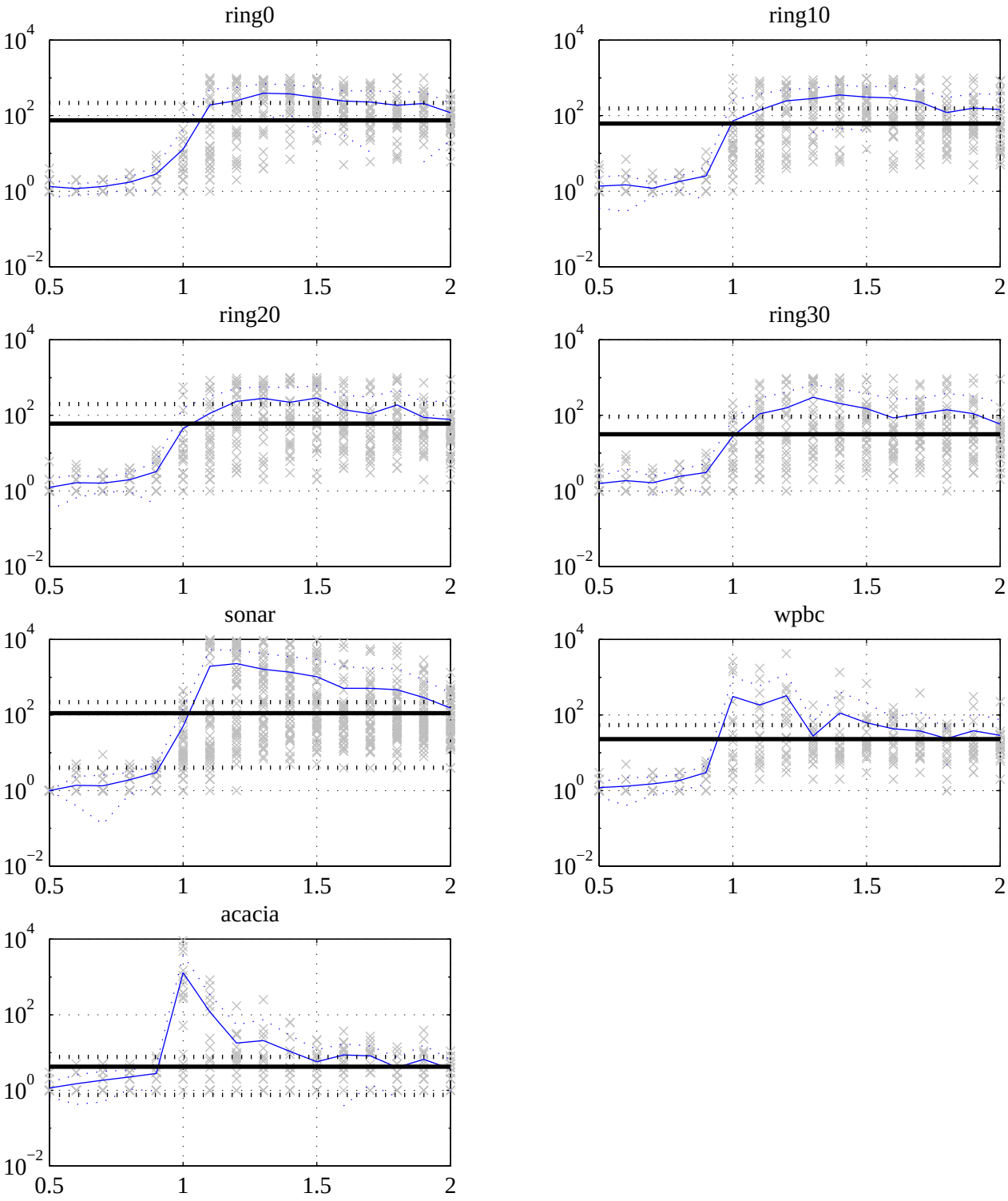






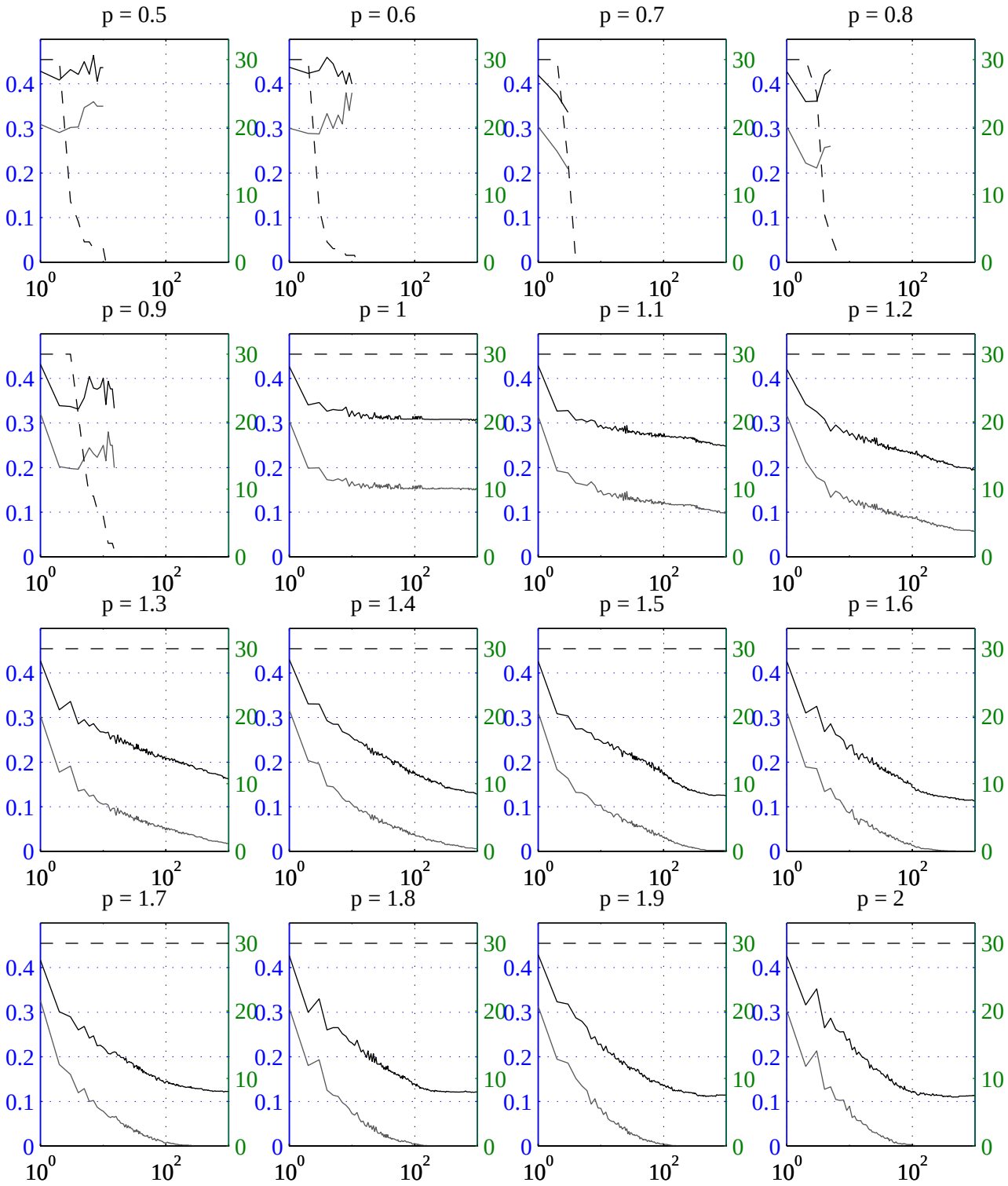


Best test iter vs p

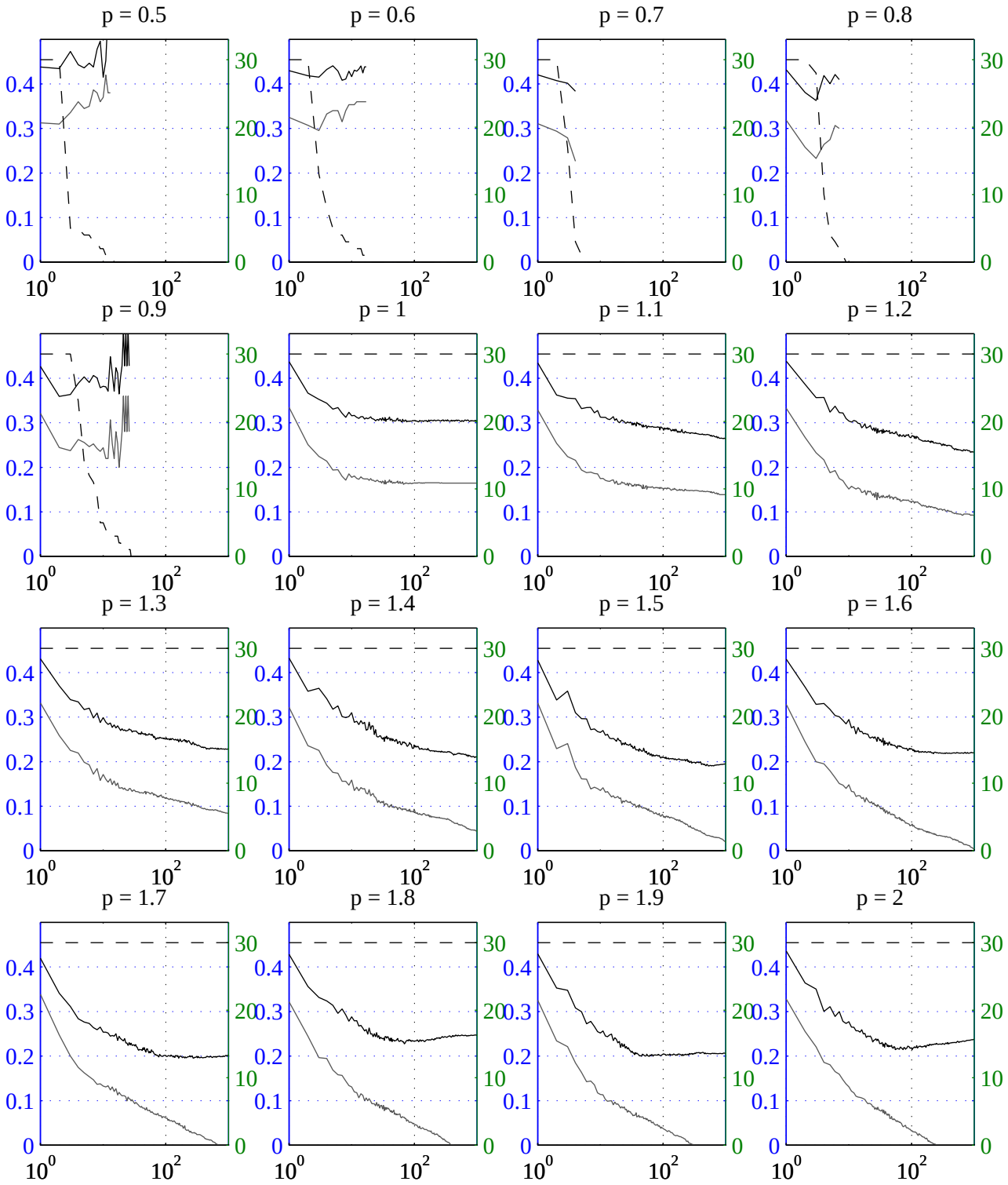


Training curves

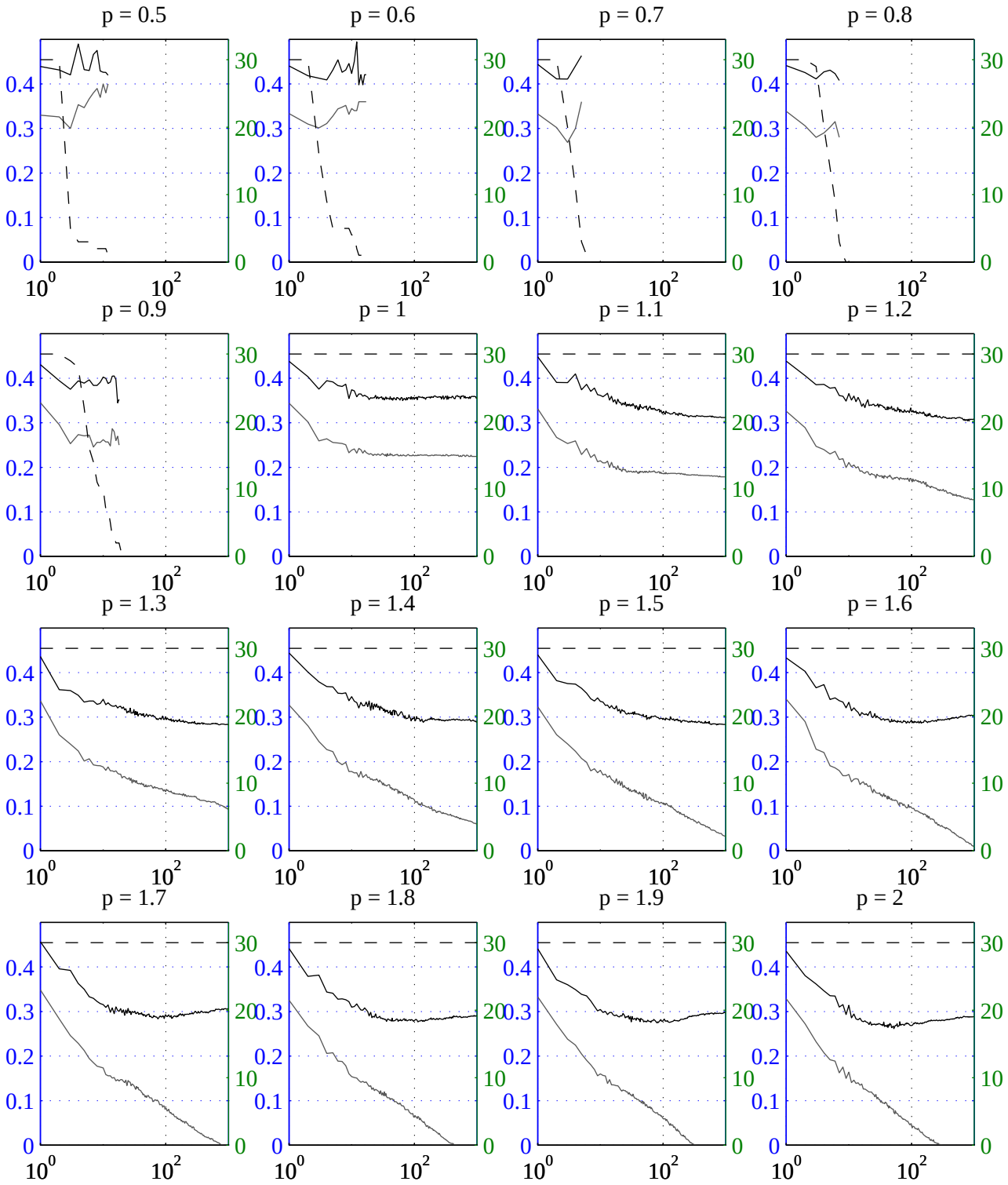
ring0



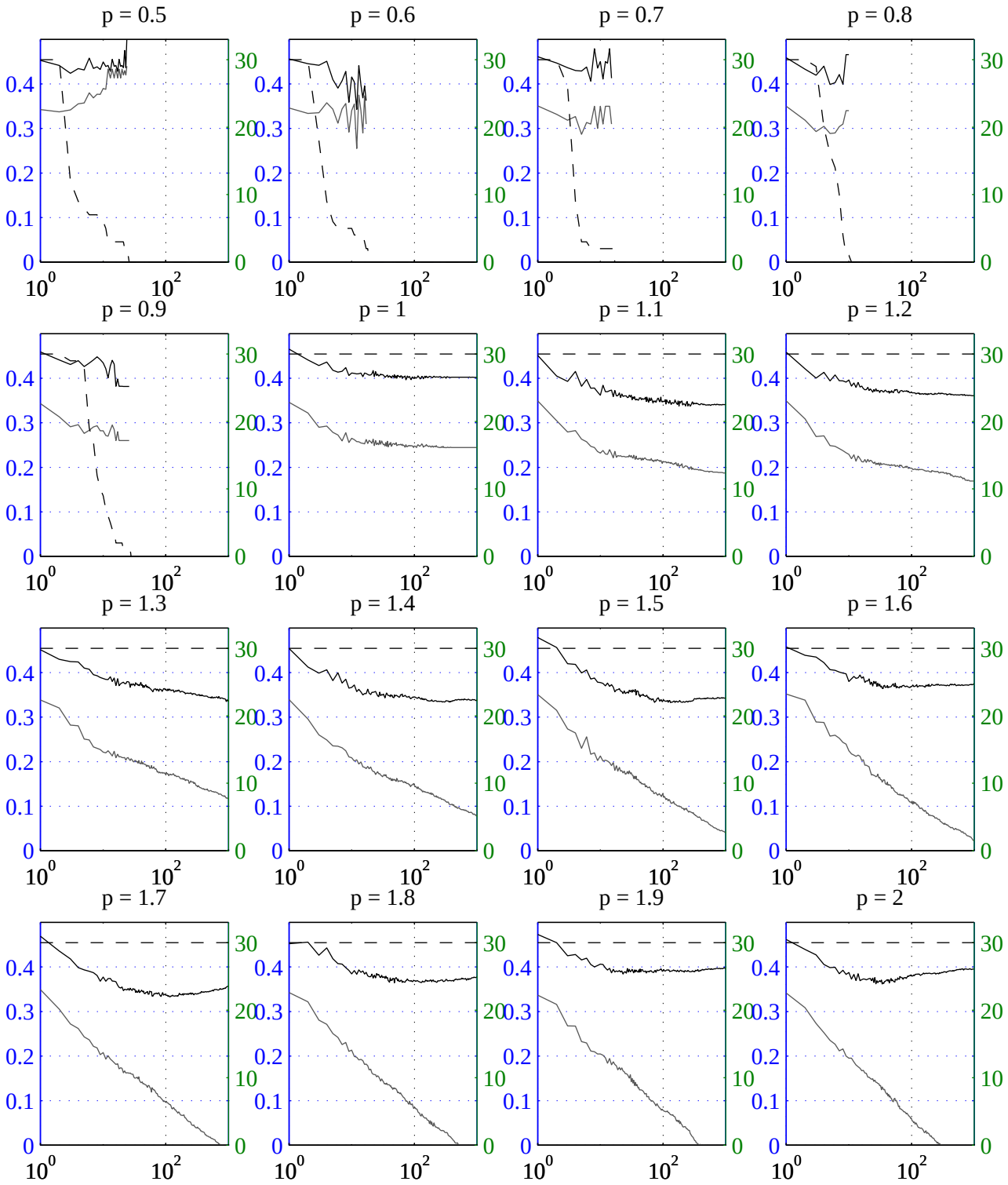
ring10


p = 0.5
p = 0.6
p = 0.7
p = 0.8
p = 0.9
p = 1
p = 1.1
p = 1.2
p = 1.3
p = 1.4
p = 1.5
p = 1.6
p = 1.7
p = 1.8
p = 1.9
p = 2
0
0.1
0.2
0.3
0.4
10
20
30
10^0
10^2

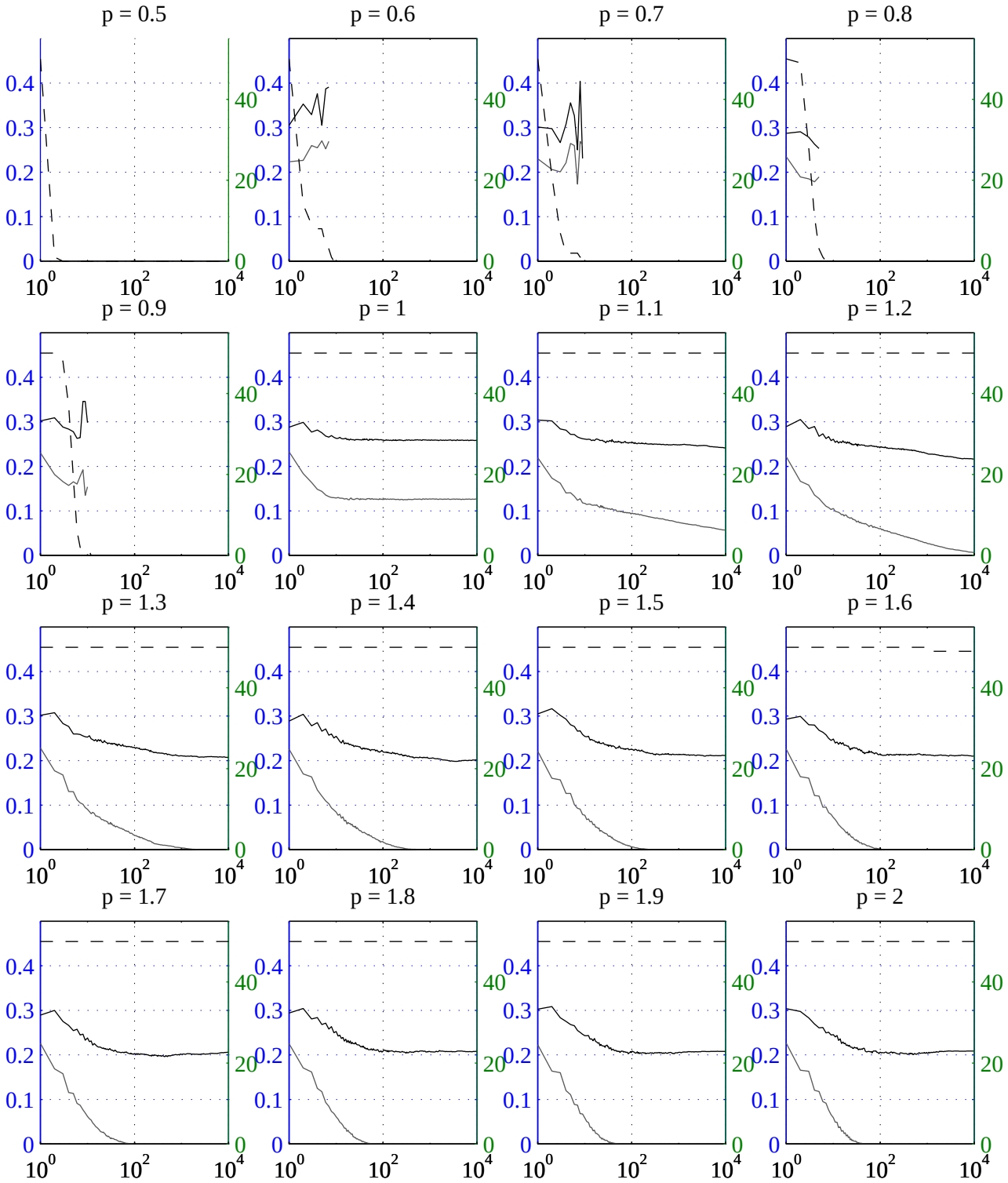
ring20


p = 0.5
p = 0.6
p = 0.7
p = 0.8
p = 0.9
p = 1
p = 1.1
p = 1.2
p = 1.3
p = 1.4
p = 1.5
p = 1.6
p = 1.7
p = 1.8
p = 1.9
p = 2
0
0.1
0.2
0.3
0.4
10
20
30
10^0
10^2

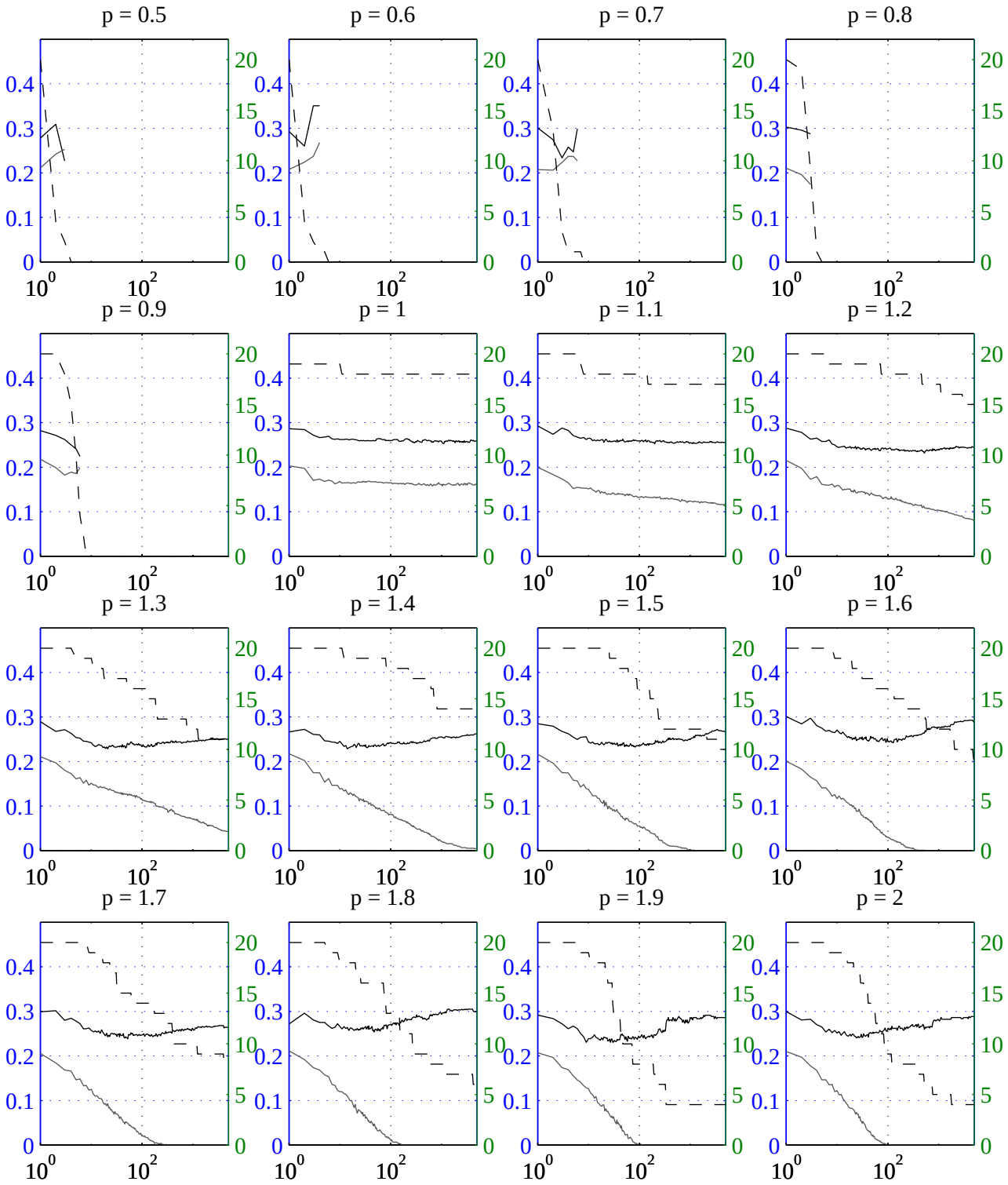
ring30


p = 0.5
p = 0.6
p = 0.7
p = 0.8
p = 0.9
p = 1
p = 1.1
p = 1.2
p = 1.3
p = 1.4
p = 1.5
p = 1.6
p = 1.7
p = 1.8
p = 1.9
p = 2
0
0.1
0.2
0.3
0.4
10
20
30
10^0
10^2

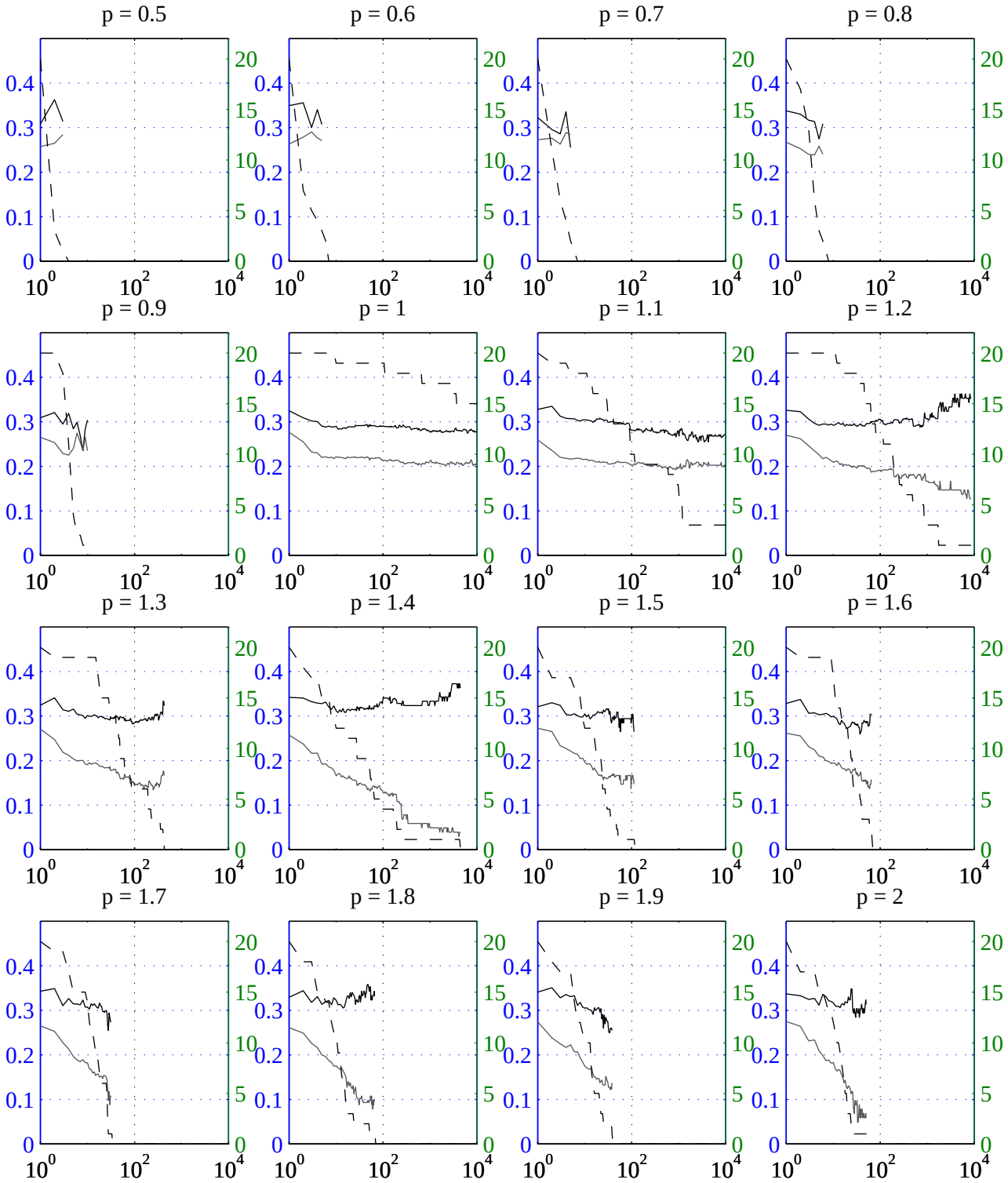
sonar



wpbc

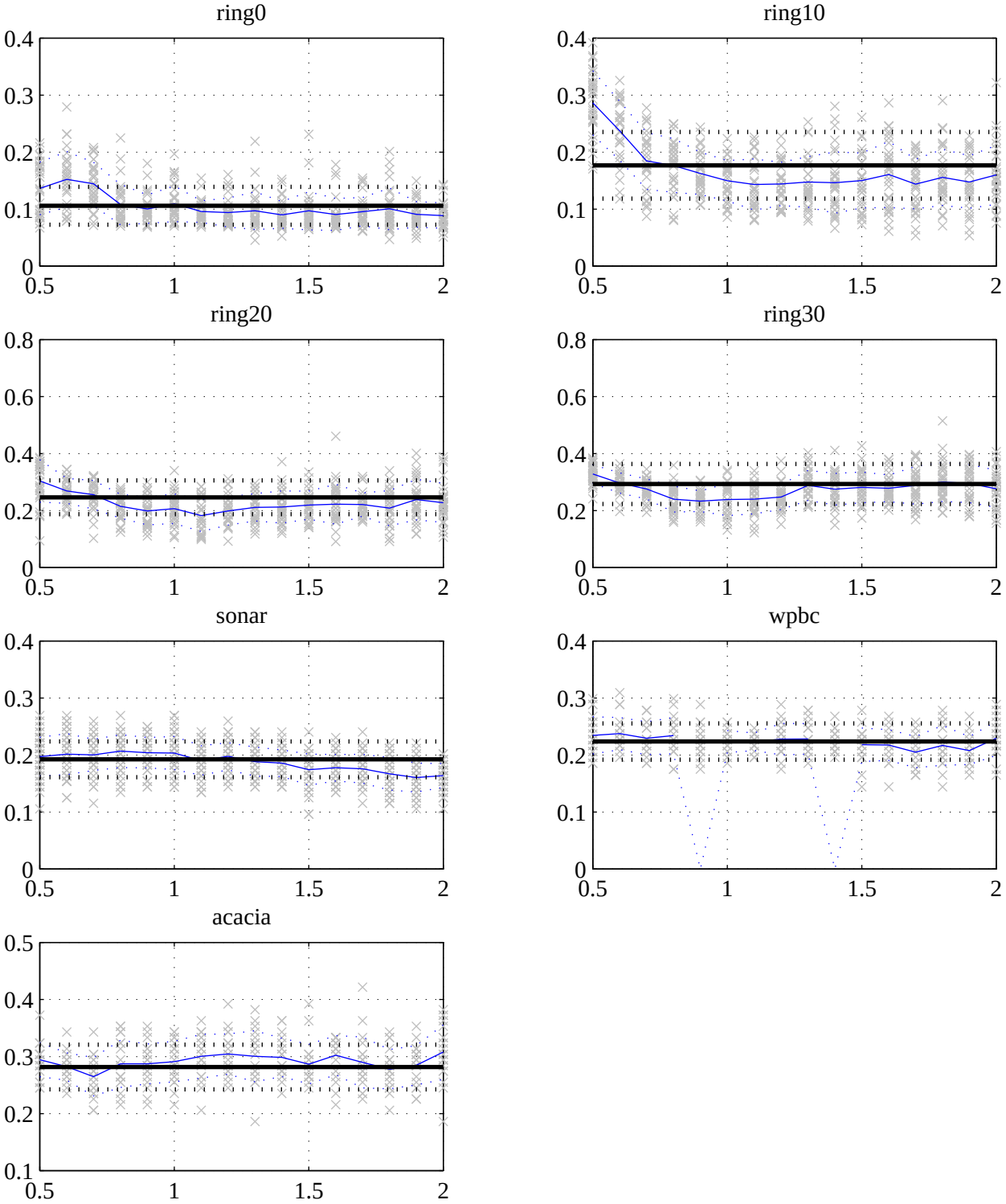


acacia

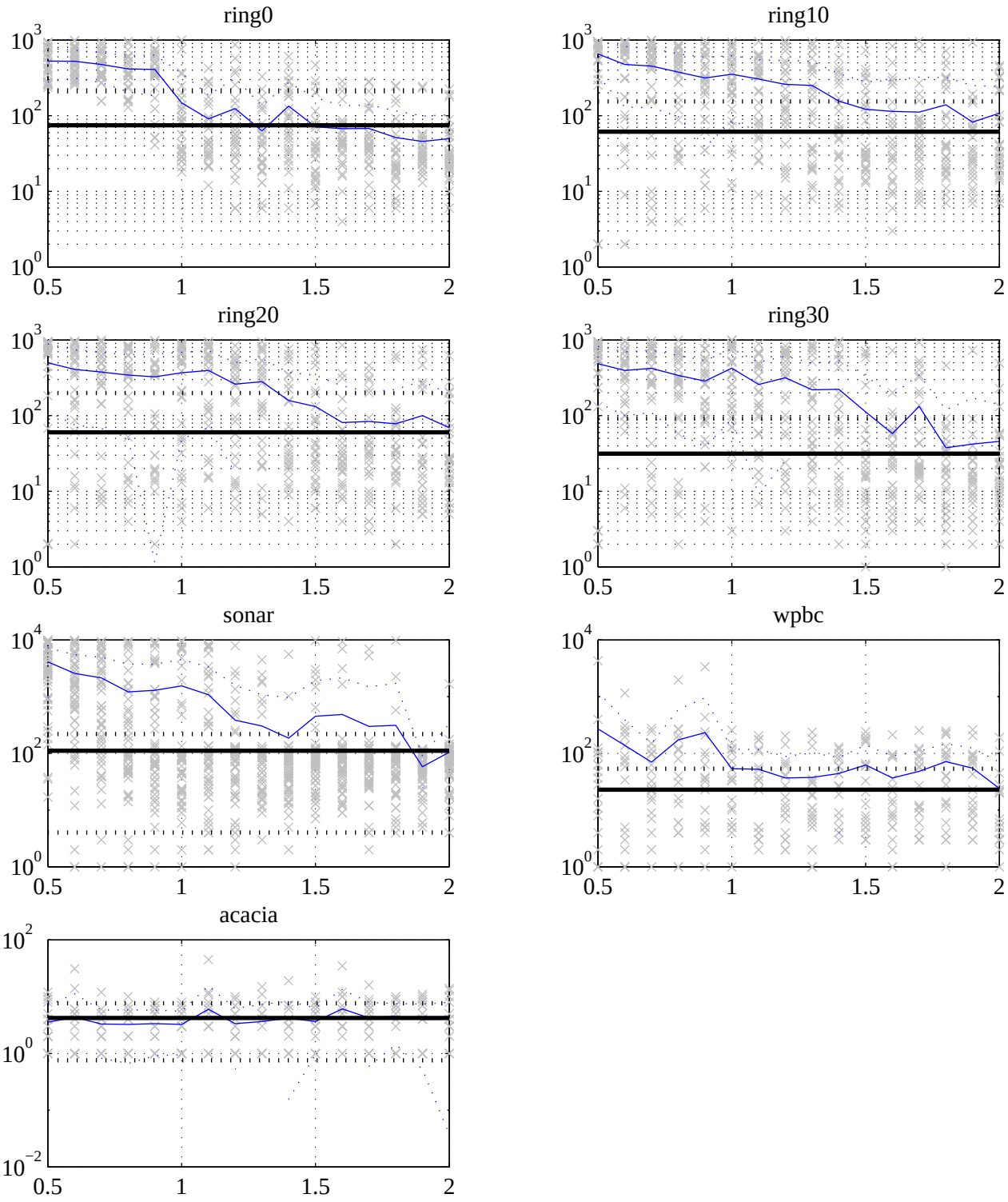


E.3 Sloppy

Best test error vs p

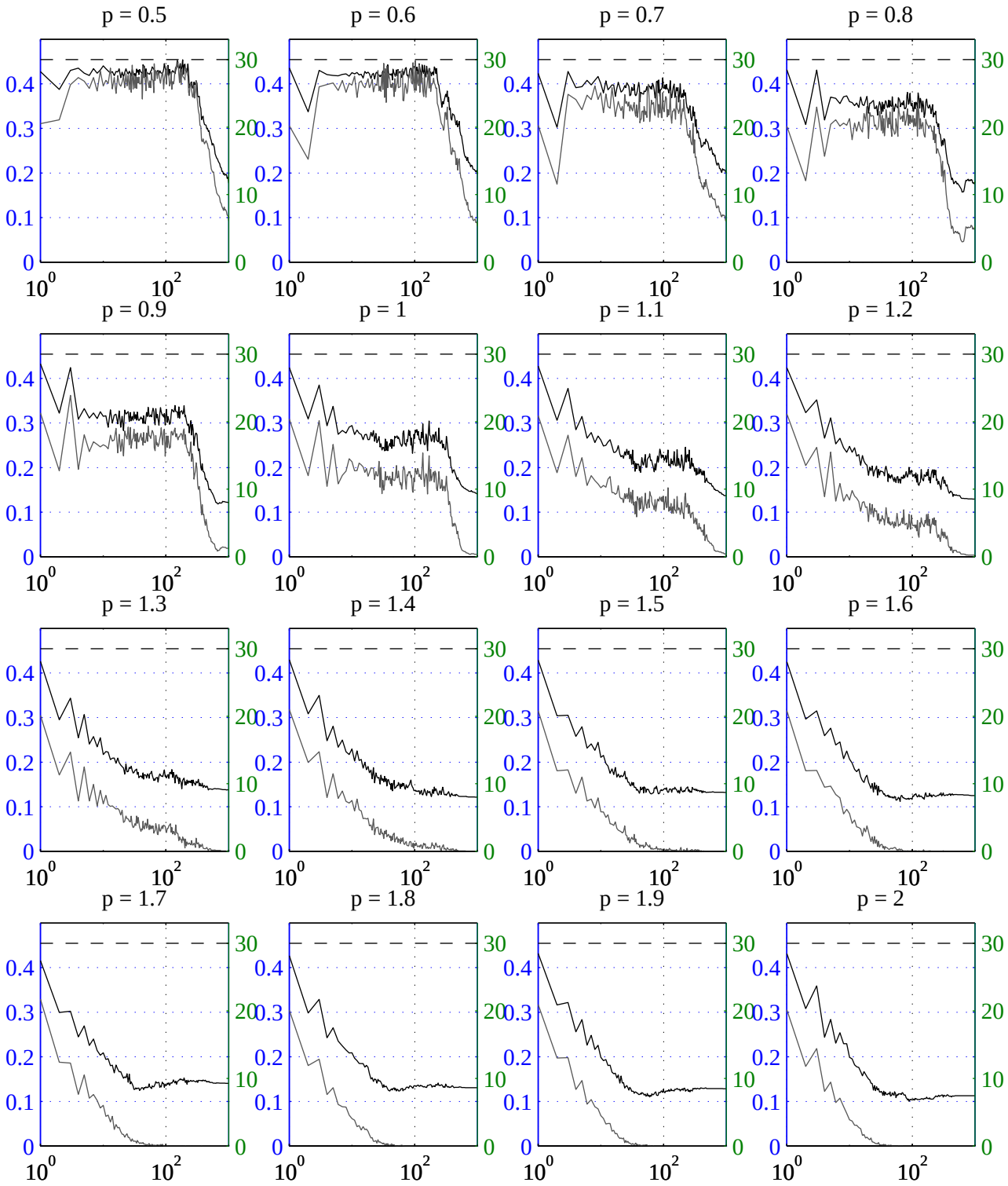


Best test iter vs p

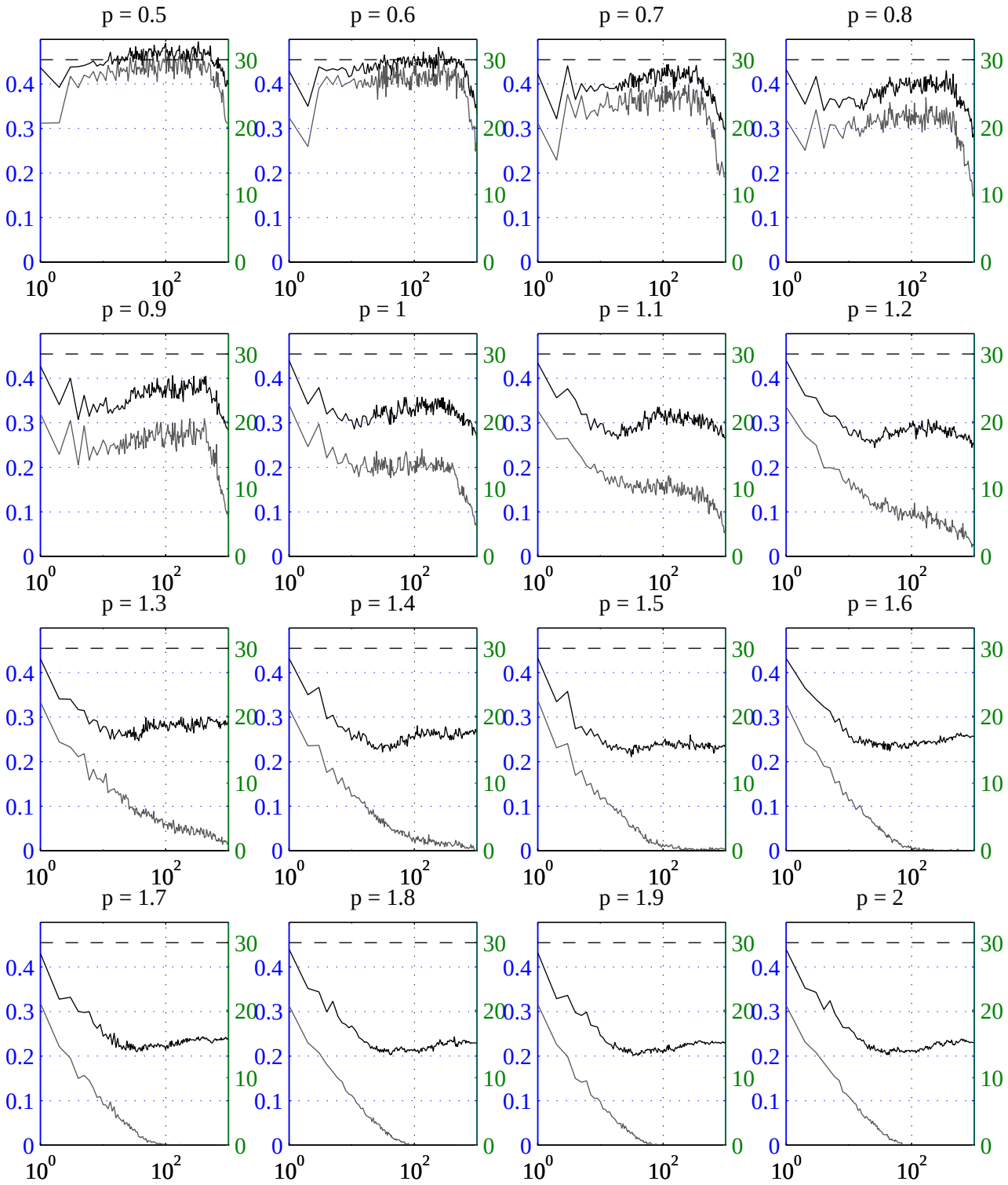


Training curves

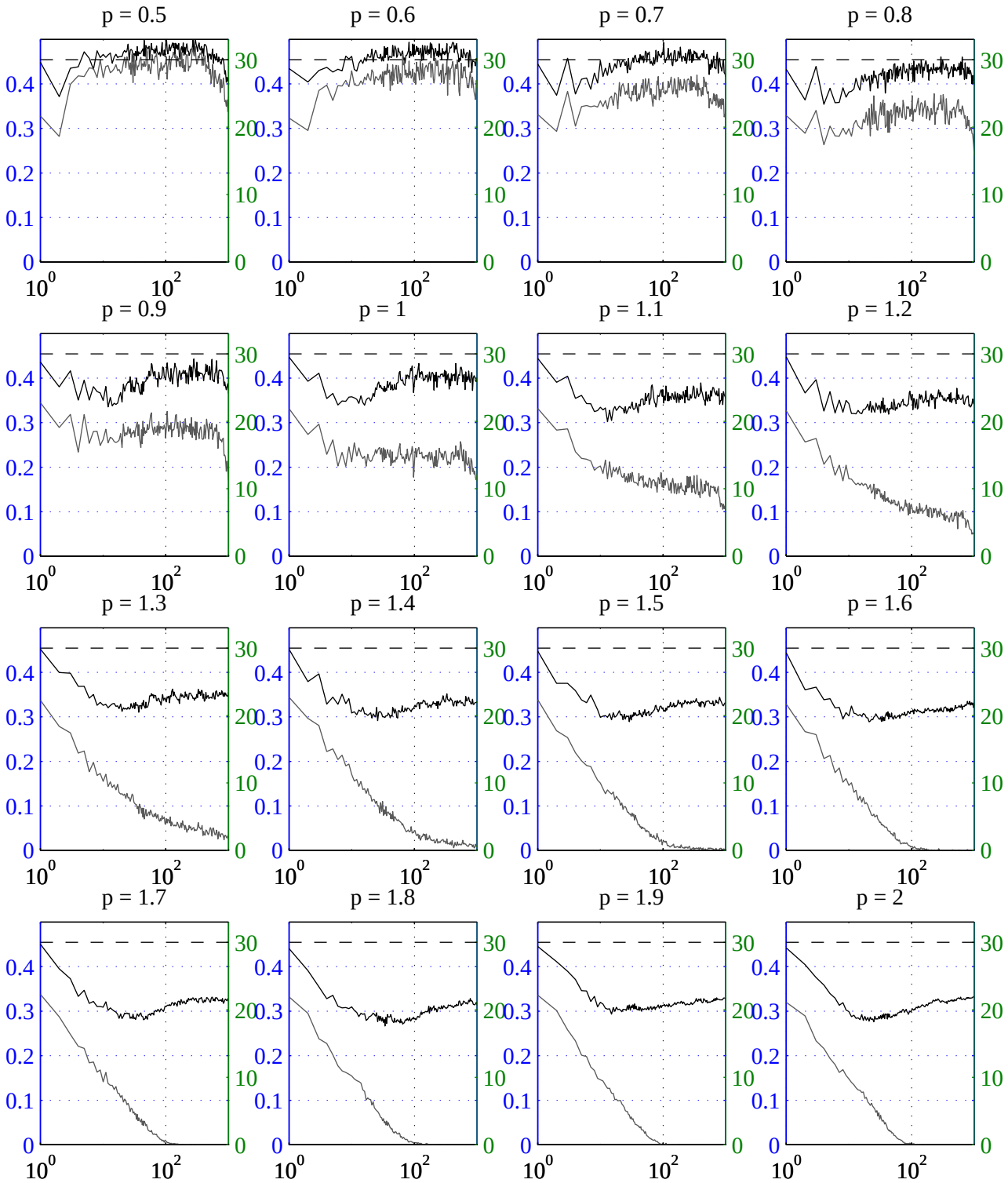
ring0



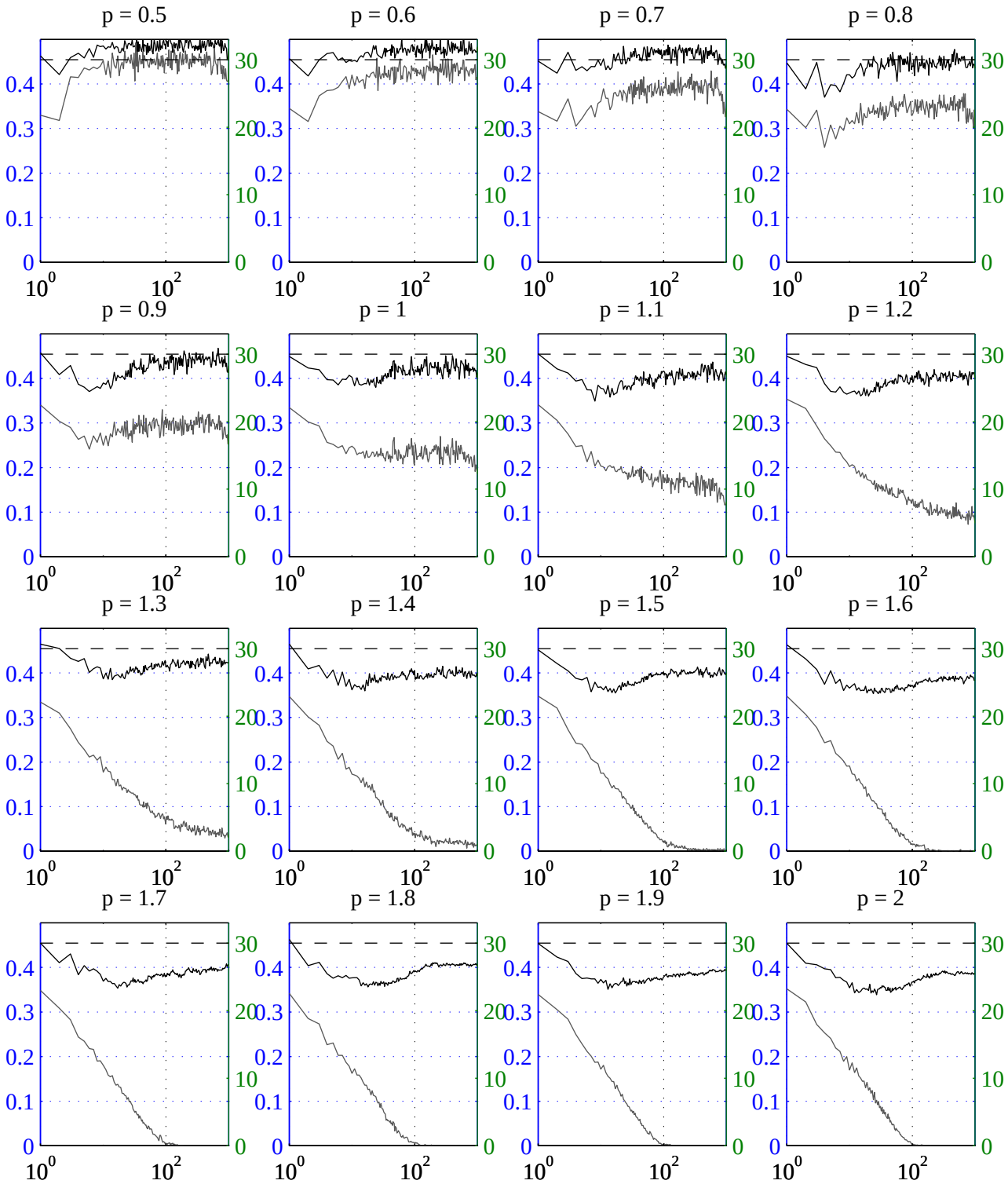
ring10


p = 0.5
p = 0.6
p = 0.7
p = 0.8
p = 0.9
p = 1
p = 1.1
p = 1.2
p = 1.3
p = 1.4
p = 1.5
p = 1.6
p = 1.7
p = 1.8
p = 1.9
p = 2
0
0.1
0.2
0.3
0.4
10
20
30
10^0
10^2

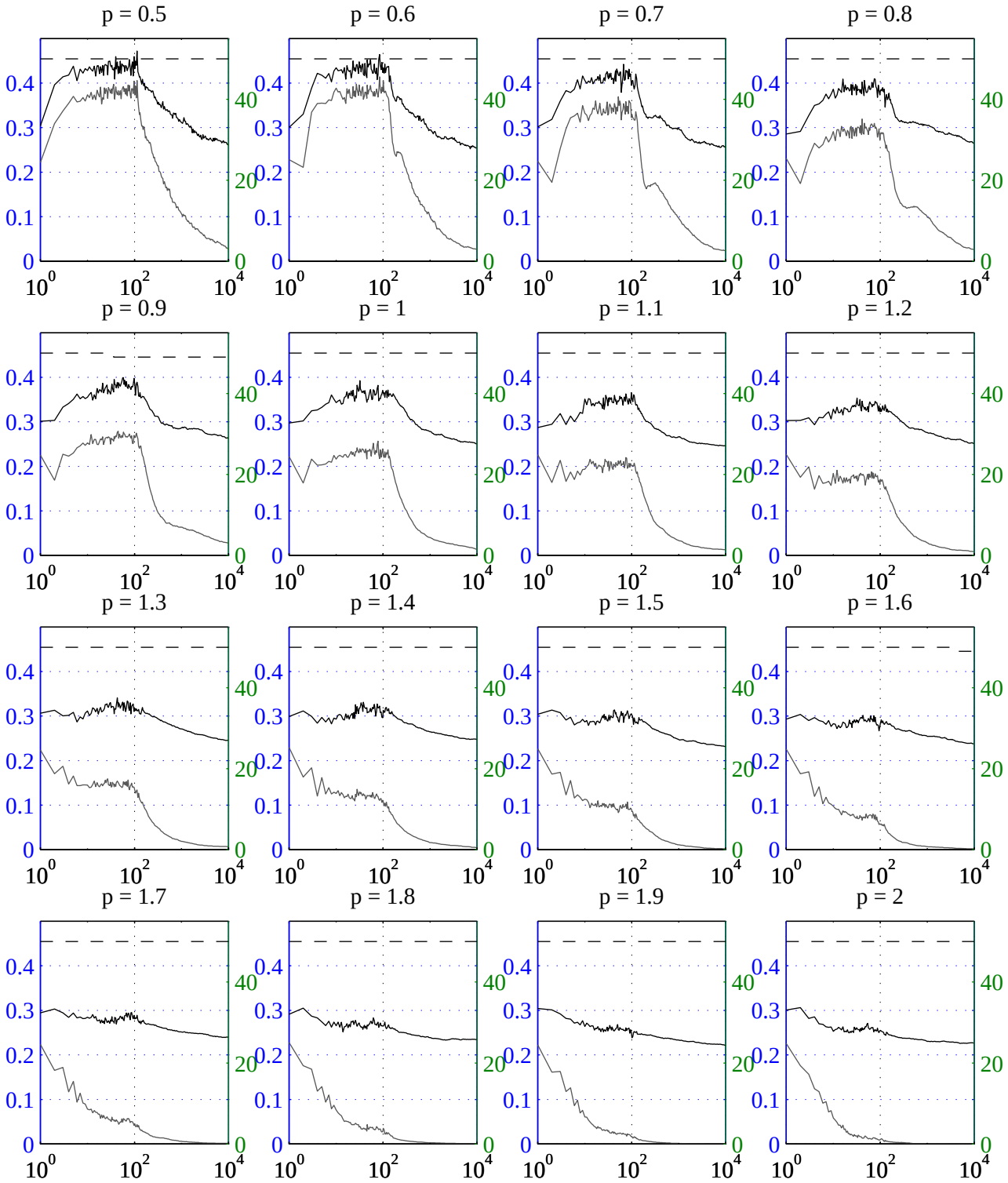
ring20


p = 0.5
p = 0.6
p = 0.7
p = 0.8
p = 0.9
p = 1
p = 1.1
p = 1.2
p = 1.3
p = 1.4
p = 1.5
p = 1.6
p = 1.7
p = 1.8
p = 1.9
p = 2
0
0.1
0.2
0.3
0.4
10
20
30
10^0
10^2

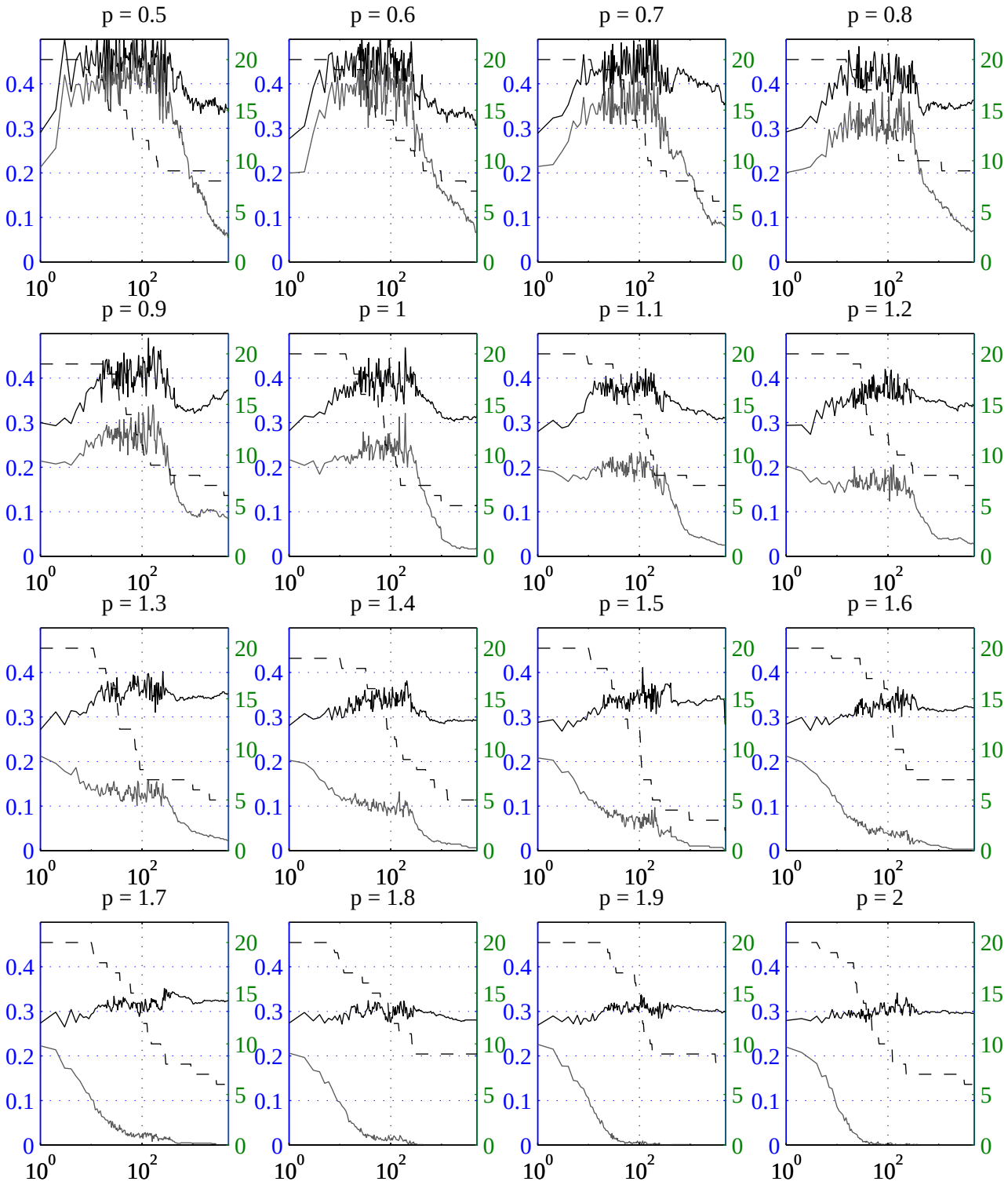
ring30


p = 0.5
p = 0.6
p = 0.7
p = 0.8
p = 0.9
p = 1
p = 1.1
p = 1.2
p = 1.3
p = 1.4
p = 1.5
p = 1.6
p = 1.7
p = 1.8
p = 1.9
p = 2
0
0.1
0.2
0.3
0.4
10
20
30
10^0
10^2

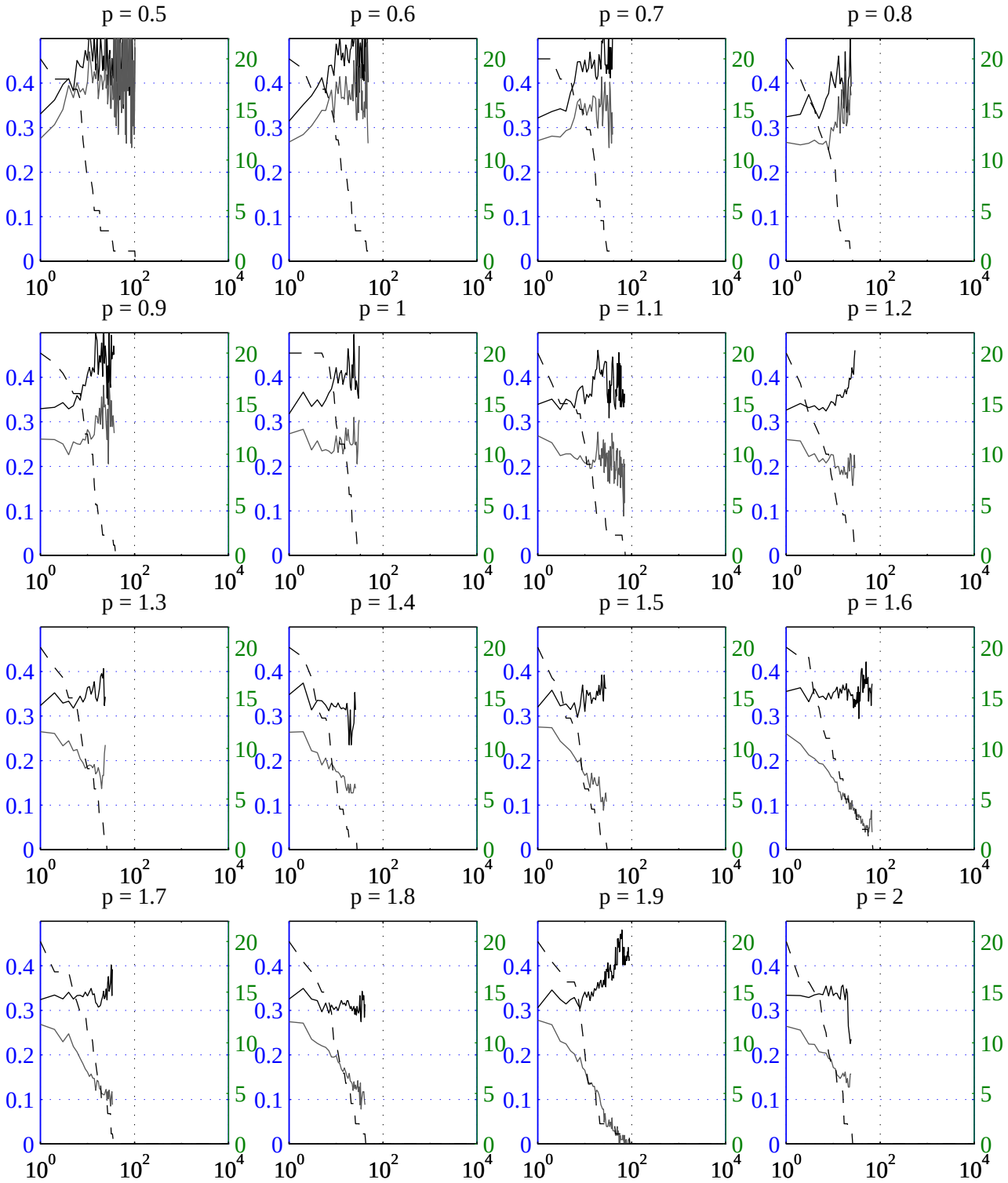
sonar


p = 0.5
p = 0.6
p = 0.7
p = 0.8
p = 0.9
p = 1
p = 1.1
p = 1.2
p = 1.3
p = 1.4
p = 1.5
p = 1.6
p = 1.7
p = 1.8
p = 1.9
p = 2
0
0.1
0.2
0.3
0.4
0
20
40
10^0
10^2
10^4

wpbc

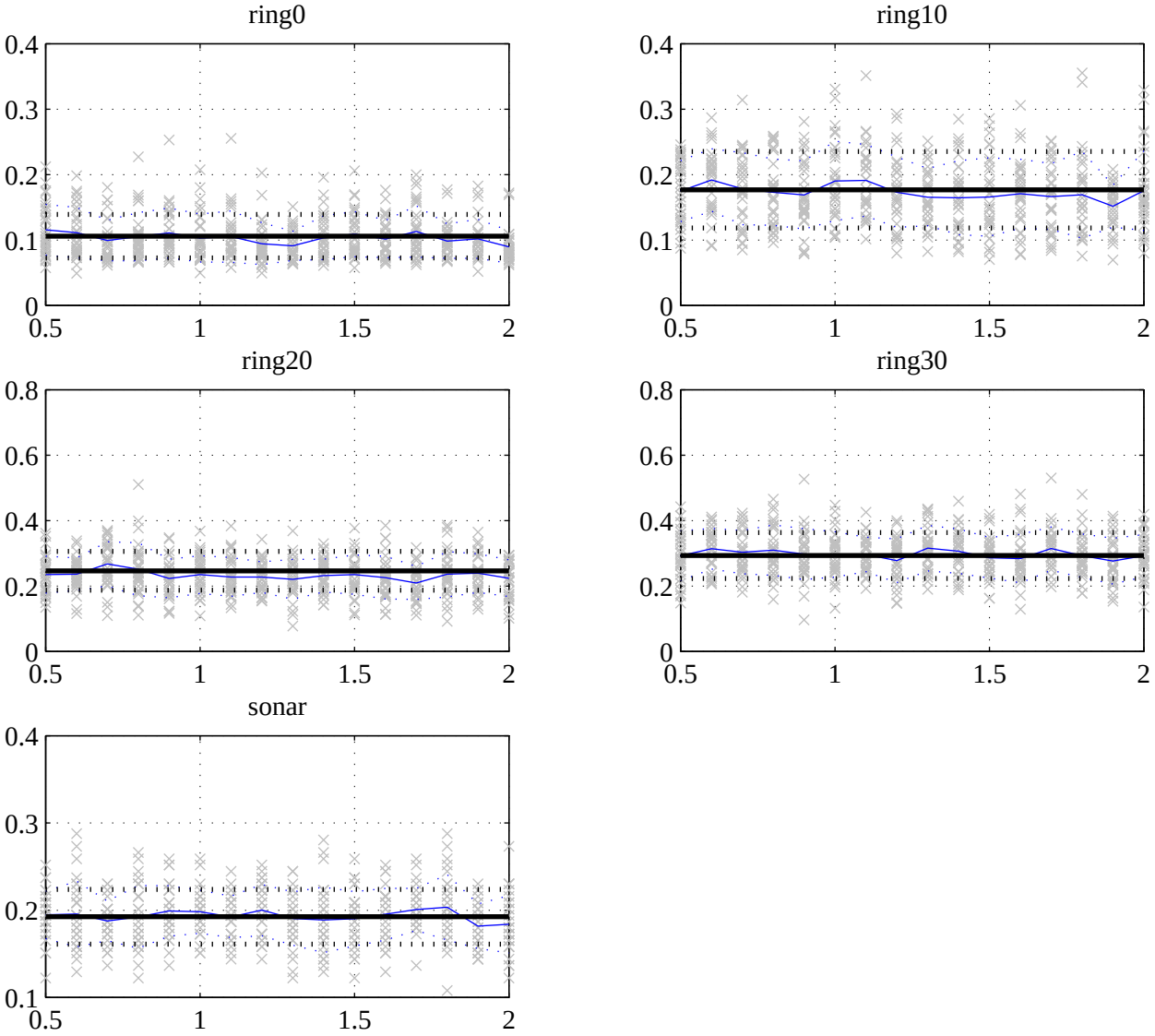


acacia

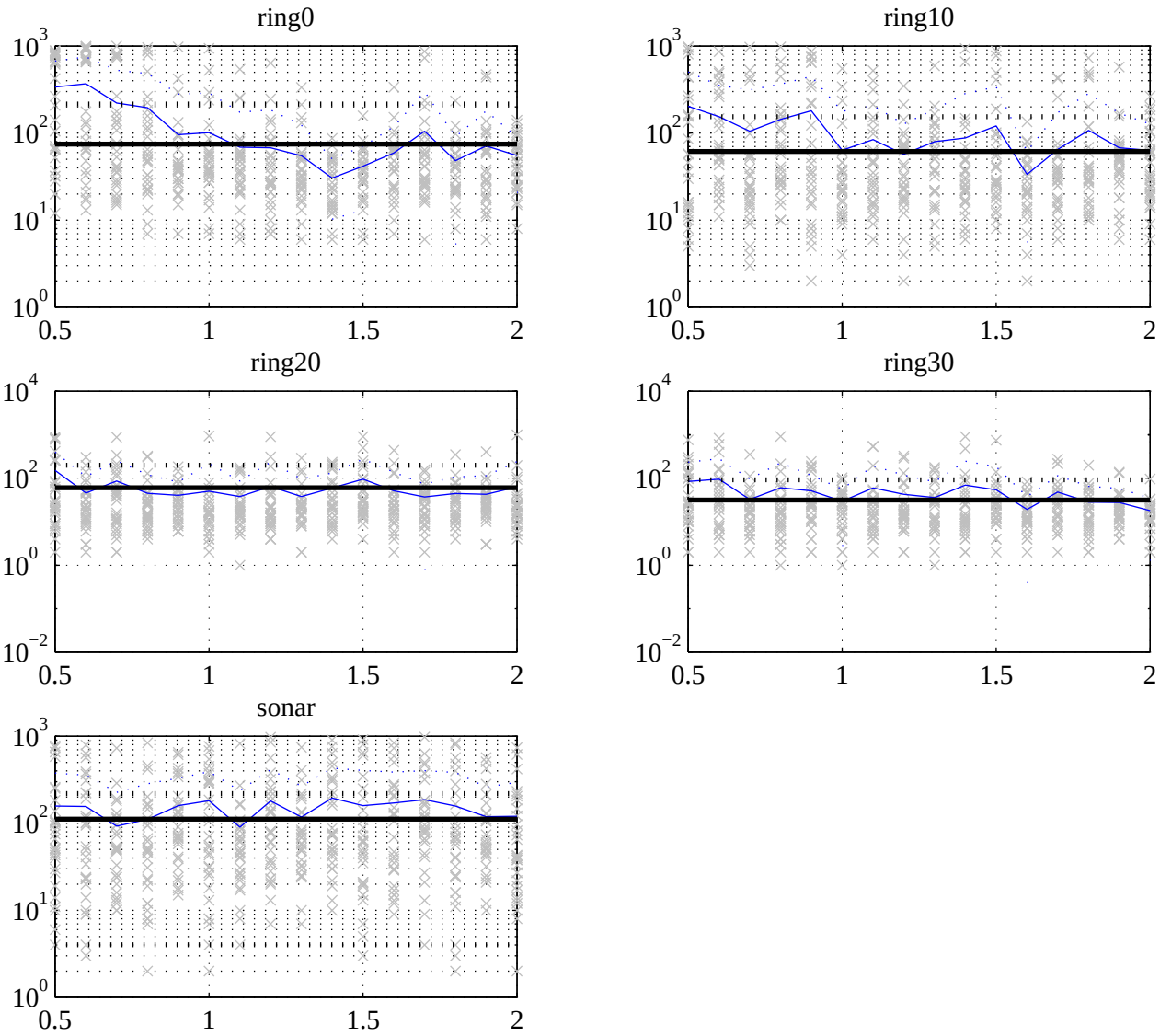


E.4 Naïve

Best test error vs p

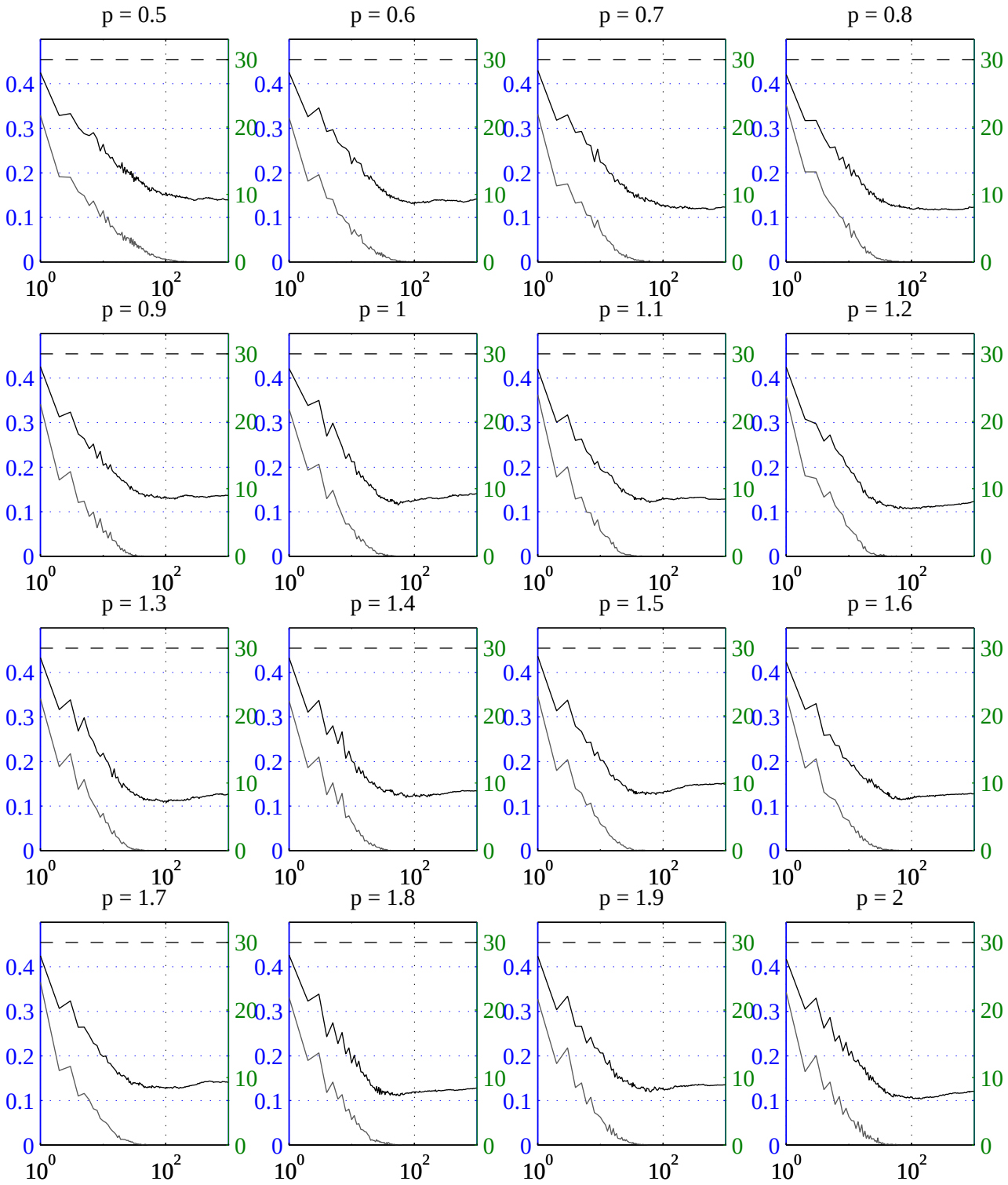


Best test iter vs p

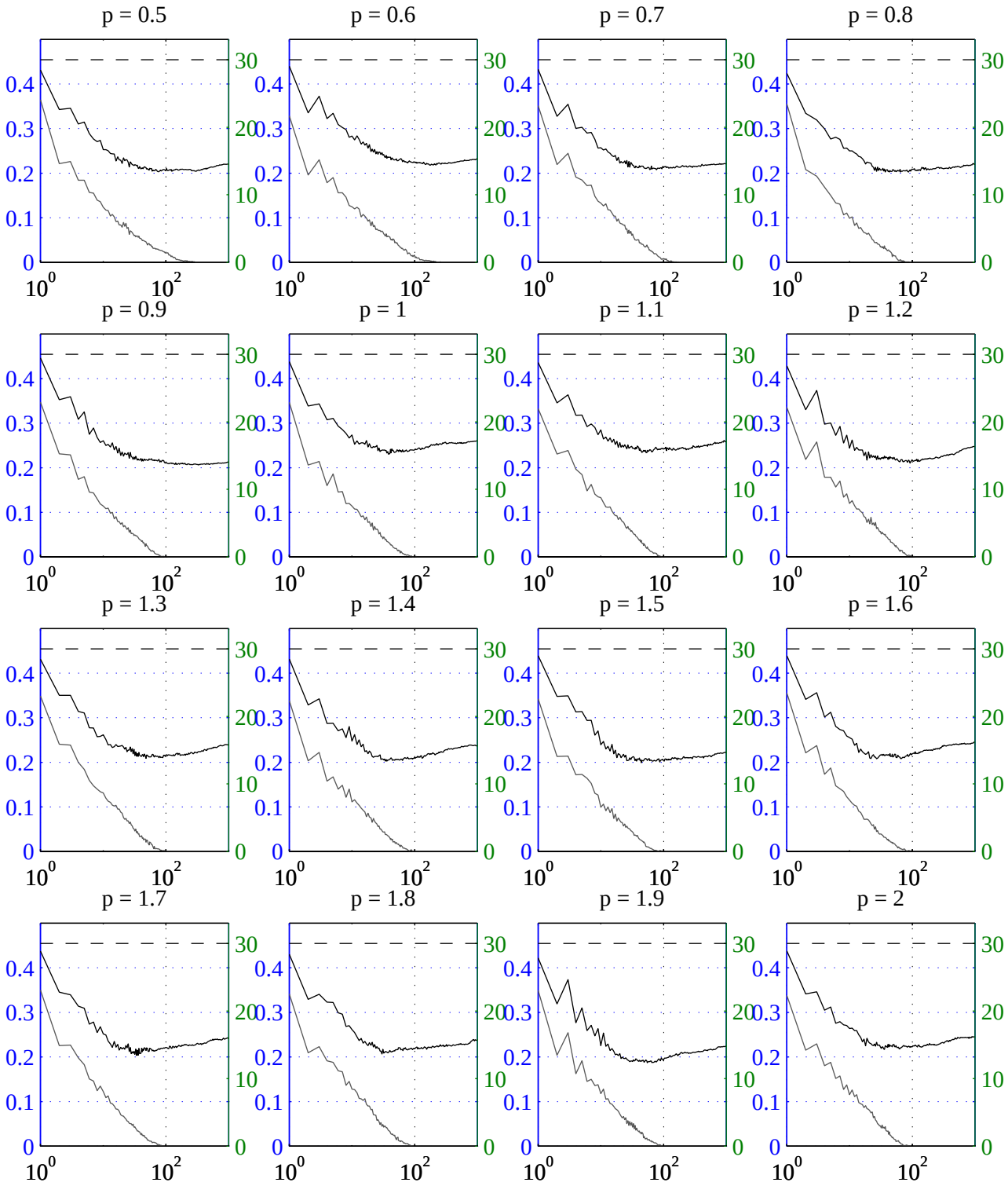


Training curves

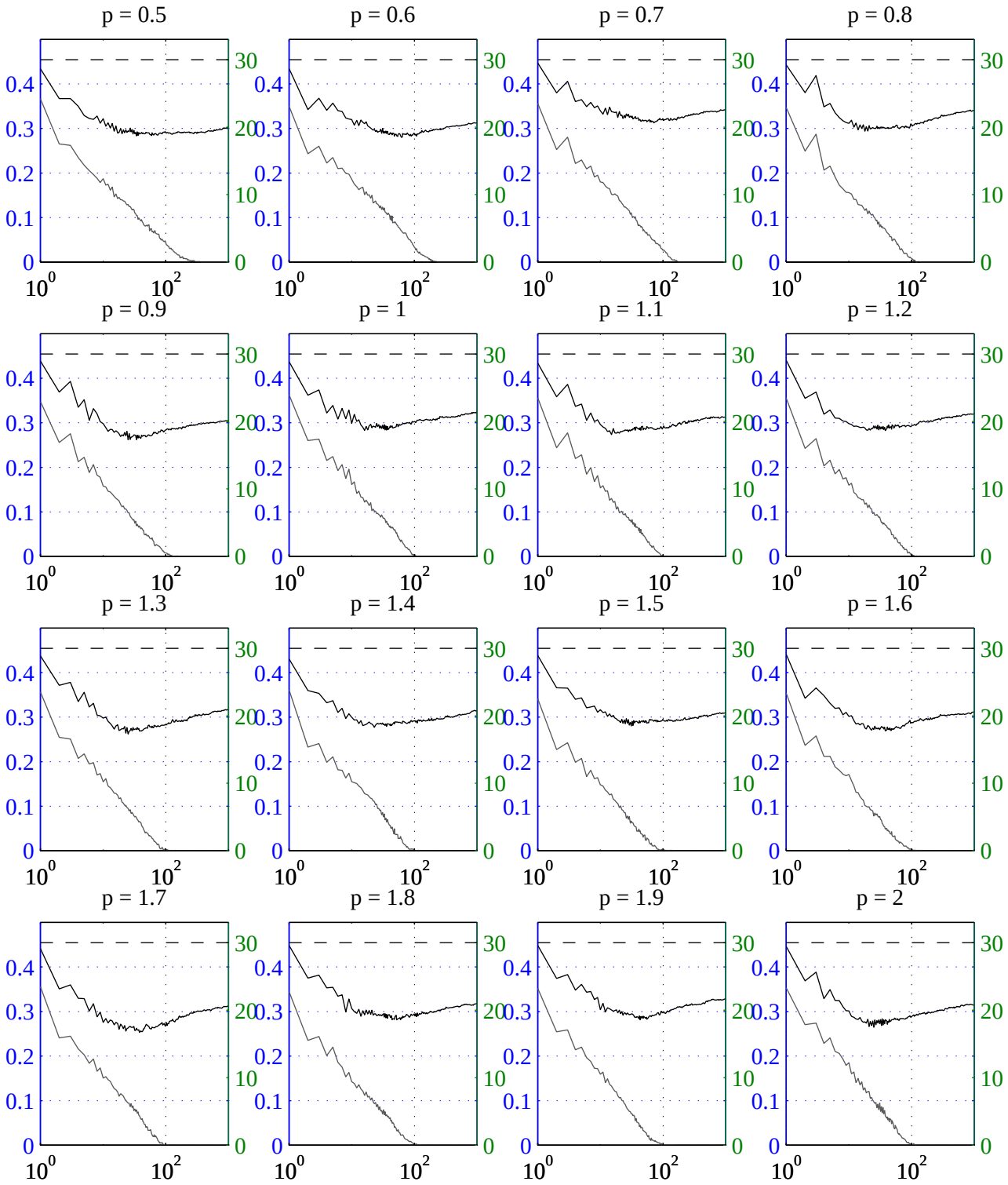
ring0



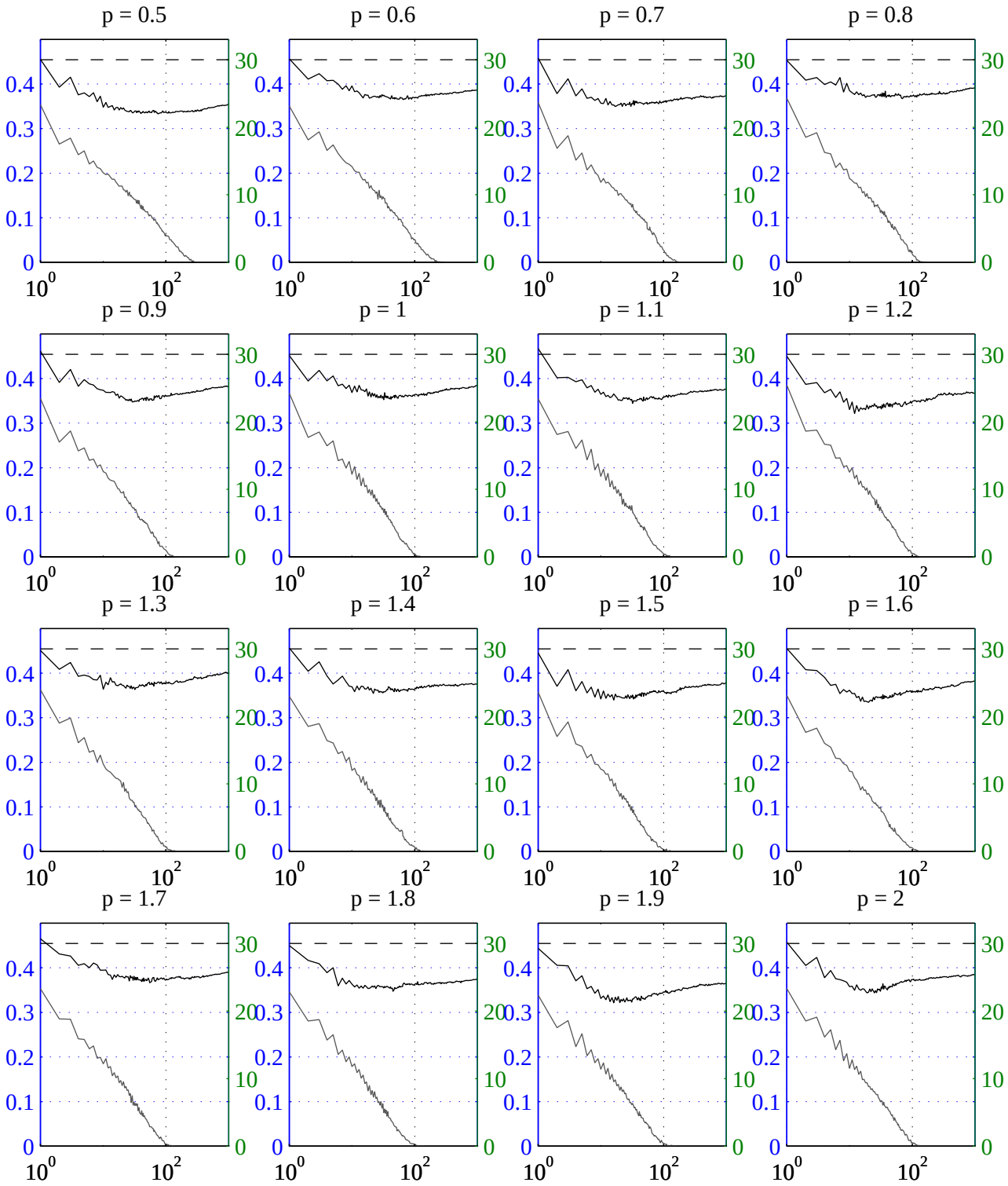
ring10


p = 0.5
p = 0.6
p = 0.7
p = 0.8
p = 0.9
p = 1
p = 1.1
p = 1.2
p = 1.3
p = 1.4
p = 1.5
p = 1.6
p = 1.7
p = 1.8
p = 1.9
p = 2
0
0.1
0.2
0.3
0.4
10
20
30
10^0
10^2

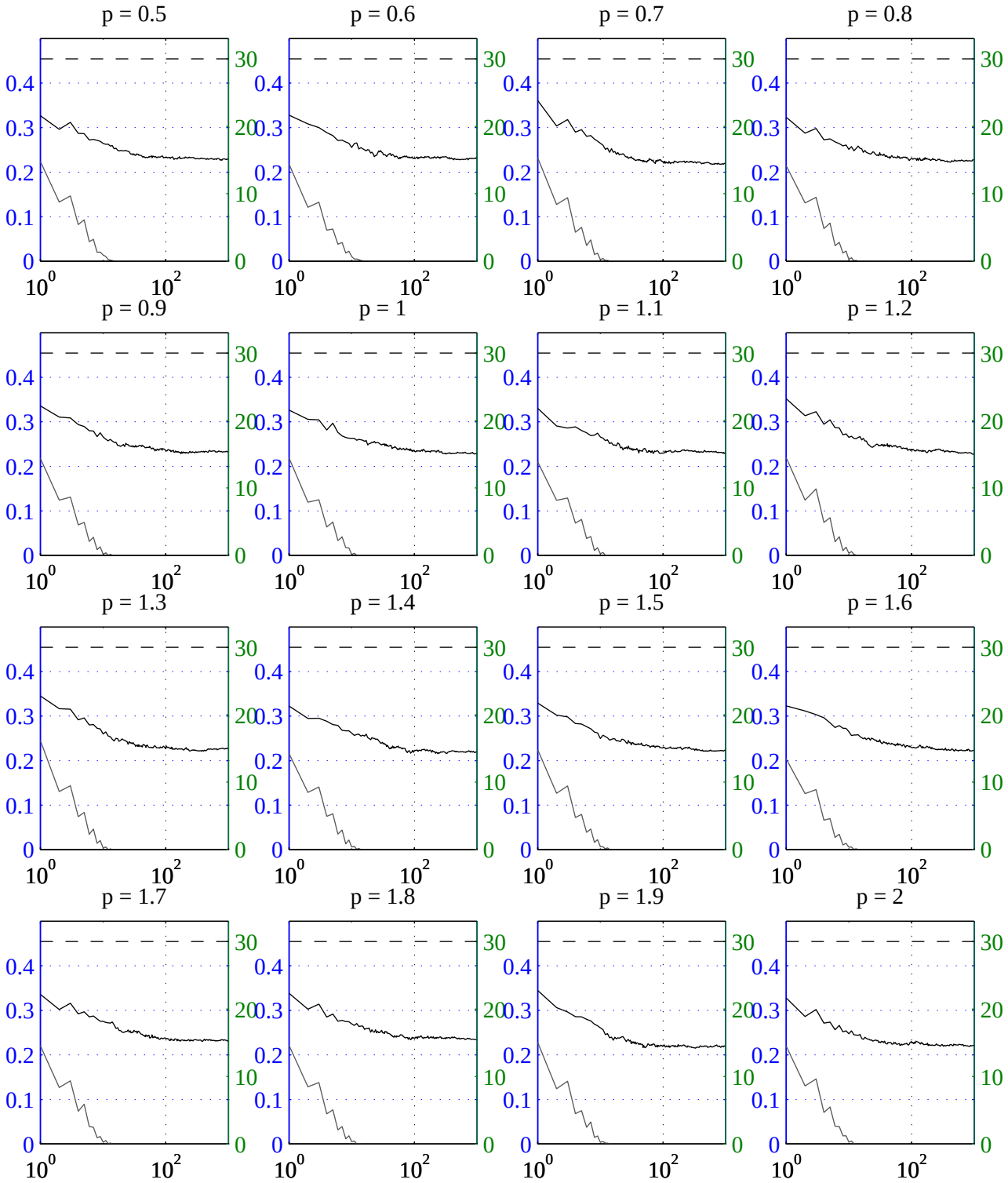
ring20


p = 0.5
p = 0.6
p = 0.7
p = 0.8
p = 0.9
p = 1
p = 1.1
p = 1.2
p = 1.3
p = 1.4
p = 1.5
p = 1.6
p = 1.7
p = 1.8
p = 1.9
p = 2
0
0.1
0.2
0.3
0.4
10
20
30
10^0
10^2

ring30


p = 0.5
p = 0.6
p = 0.7
p = 0.8
p = 0.9
p = 1
p = 1.1
p = 1.2
p = 1.3
p = 1.4
p = 1.5
p = 1.6
p = 1.7
p = 1.8
p = 1.9
p = 2
0
0.1
0.2
0.3
0.4
10
20
30
10^0
10^2

sonar



Appendix F

Project proposal

DRAFT Honours Project Proposal: Capacity Control in Boosting via an Adjustable b Function [1]

Jeremy Barnes
Supervisor: Dr Bob Williamson

F.1 Introduction

The aim of this document is to provide some detail on the scope and implementation of this honours project. A brief background on the project is given; followed by an explicit statement of the expected outcomes; and a timeline.

F.2 Background

Boosting is a method used to improve the generalisation ability of a "weak" learning algorithm. It is implemented by generating many instances of the algorithm, each trained on different data. This data is chosen such that examples which are commonly misclassified are emphasised. This forces the learning algorithm to work well with the difficult data, and have better overall performance. The output of the boosting algorithm is a weighted combination of the outputs of the weak learning algorithms (weighted by their performance).

Adjusting the capacity of a learning algorithm controls the size of the set of possible generalisations. It is important to control capacity to avoid overfitting (fitting the noise rather than the underlying function). Although boosting is particularly good at avoiding overfitting, implementing capacity control can improve the performance of the boosted algorithm [ref] and avoid eventual overfitting.

The b function is used within the boosting algorithm. It is calculated for each instance of the weak learning algorithm, from the learning error of the weak learning algorithm (the learning error is the proportion of samples which are misclassified). In the case of a binary classification problem, it has the form

$$b_t = \log \frac{\epsilon_t}{1 - \epsilon_t} \qquad (0 \leq \epsilon_t \leq 1) \tag{F.1}$$

where ϵ_t is the learning error of weak learning algorithm t.

This function describes the "worth" of this instance of the weak learning algorithm. A plot is shown in figure 1.

Weak learners with a value of ϵ_t around $1/2$ are worthless, as random guessing performs just as well. As $epsilon_t$ moves away from $1/2$, the algorithms become more useful (and show an increase in b_t). Algorithms that approach $\epsilon_t \to \pm 1$ are given a b_t that approaches $\pm\infty$.

[1] This project was originally called, "Aspects of Statistical Learning Theory and Support Vector Machines"

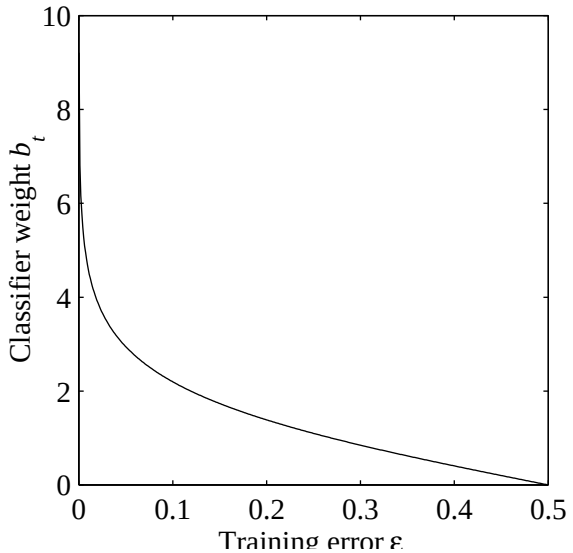


Figure F.1: The b_t function versus error ϵ_t

This project will investigate modifying the b function to become

$$b'_t = \text{sign}(b_t)|b_t|^{1/p} \tag{F.2}$$

The new parameter p allows control over how aggressively weak learners are penalised. This should allow capacity control via adjustment of this parameter.

F.3 Outcomes

The project will be successful if the following outcomes are attained:

- Enough experimental evidence is gathered to gain an appreciation of the effect of the parameter on the boosting algorithm.
- From this evidence, some guiding principles for improving the performance of the boosting algo-rithm using the parameter are developed.
- Theoretical justification of these results is obtained.

In addition to these goals, it is hoped that progress in one or more of the following areas will be made:

- Performance of the algorithm in problems with varying amounts of noise;
- Improved versions on the p-boosting algorithm using knowledge gained from the theory;
- Improving the computational efficiency of boosting using the p-boosting algorithm;
- Comparisons between p-boosting and other methods of capacity control in boosting;
- Extension of the results to include arbitrary classification problems;
- Extension of the results to include regression problems;
- Extension of the results to include other enhancements to the boosting algorithm, such as ”soft margins” [19].

Month	Tasks
December 1998	Reading
January 1999	Reading
February	Reading
March	Writing project proposal
	19th Project proposal due
	Obtaining or writing code for experiments
April	Writing, debugging code
	12th Code running, first test results
	Obtaining datasets, running tests, refining code
May	Running tests
	Developing and testing hypotheses
	Reading on theory
	31st First draft of "background" section of thesis
June	Initial attempts at theoretical analysis
	Running tests required for theoretical verification
	Attempting to reach closure on some aspects
	Exam period
July	Attempting to reach closure on some aspects
	Writing progress report
	19th Progress report due
	Preparing for seminar
	26th Seminar
August	Final work on theory
	Final experimental results to assist theory
	30th Sufficient work done to complete thesis
September	Work on spin-off or extension aspects
	Preparing for demonstration
	27th All experimental and theoretical results obtained
	Writing thesis
October	Writing thesis
	7th Draft thesis due
	Revising thesis
	27th Thesis due
	28th Demonstration

Table F.1: Project timeline

In order to limit the project to a reasonable scope, initially the following constraints will be placed:

- Only binary classification problems will be considered.
- Small datasets will be used.
- Simple weak learning algorithms will be used.

F.4 Timeline

This section includes a month-by-month description of what I intend to complete on the project. It is given in table F.1 on page 89.
Notes:

- I intend to work on the thesis gradually throughout the year, so that I do not need to write it all at the end.

- I have allowed 2-3 weeks to work on ”extension” aspects of the problem that are interesting but not vital to a strong thesis. It would not be a serious problem if none of this work can be included in the thesis. These 2-3 weeks also give me some slack if things are not going as planned or if I become burnt out and need a break.

Bibliography

[1] Martin Anthony and Peter Bartlett. *Learning in Neural Networks: Theoretical Foundations*. Cambridge University Press, 1998.

[2] Peter Bartlett and John Shawe-Taylor. Generalization performance of support vector machines and other pattern classifiers. Book chapter. Available at http://discus.anu.edu.au/ bartlett/papers/TR98b.ps.Z, 1999.

[3] Eric Bauer and Ron Kohavi. An empirical comparison of voting classification algorithms: Bagging, boosting and variants. *Machine Learning*, 36(1/2):105–139, 1999.

[4] C. Blake, E. Keogh, and C. J. Merz. UCI repository of machine learning databases, 1998.

[5] Leo Breiman. Bias, variance and arcing classifiers. Technical Report 460, Statistics Department, University of California, Berkeley, April 1996.

[6] Leo Breiman. Arcing the edge. Technical Report 486, Statistics Department, University of California, Berkeley, 1997.

[7] Vladamir Cherkassky and Filip Mulier. *Learning from Data: Concepts, Theory and Methods*. John Wiley and Sons, 1998.

[8] Nigel Duffy and David Helmbold. Potential boosters? In *Proceedings of NIPS99*, 1999.

[9] Yoav Freund and Robert E. Schapire. Experiments with a new boosting algorithm. In *Machine Learning: Proceedings of the Thirteenth International Conference*, pages 148–156, 1996.

[10] Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1997.

[11] R. P. Gorman and T. J. Sejnowski. Analysis of hidden units in a layered network trained to classify sonar targets. *Neural Networks*, 1:75–89, 1988.

[12] Adam J. Grove and Dale Schuurmans. Boosting in the limit: Maximising the margin of learned ensembles. In *Proceedings, Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pages 692–699. MIT Press, 1998.

[13] Michael T. Heath. *Scientific Computing: An Introductory Survey*. McGraw-Hill Book Co., Singapore, 1997.

[14] Llew Mason, Peter Bartlett, and Jonathon Baxter. Direct optimization of margins improves generalization in combined classifiers. Technical report, Department of Systems Engineering, The Australian National University, 1999.

[15] Llew Mason, Peter Bartlett, and Jonathon Baxter. Improved generalization through explicit optimization of margins. *Machine Learning*, 1999.

[16] Llew Mason, Johnathon Baxter, Peter Bartlett, and Marcus Frean. Boosting algorithms as gradient descent in function space. Preprint, 1999.

[17] Karen Payne. *Recent Advances in the Problem of Induction and their Relationship to Ecological Modelling*. PhD thesis, The Australian National University, September 1997.

[18] Roger Penrose. *The Emperor's New Mind*. Oxford University Press, 1989.

[19] G. Rätsch, T. Onoda, and K.-R. Müller. Soft margins for AdaBoost. 1998.

[20] Robert E. Schapire, Yoav Freund, Peter Bartlett, and Wee Sun Lee. Boosting the margin: A new explanation for the effectiveness of voting methods. In *Proc. 14th International Conference on Machine Learning*, pages 322–330. Morgan Kaufmann, 1997.

[21] The MathWorks, Inc. *MATLAB Application Program Interface Guide*. The Mathworks, Inc., Internet: http://www.mathworks.com/, 1996.

[22] The MathWorks, Inc. *MATLAB Application Program Interface Reference*. The Mathworks, Inc., Internet: http://www.mathworks.com/, 1996.

[23] Vladamir N. Vapnik. *Statistical Learning Theory*. John Wiley and Sons, 1998.

[24] Robert C. Williamson. Application of a support vector machine to a geographical data set. Unpublished private manuscript, January 1997.

[25] Robert C. Williamson, Alex J. Smola, and Bernhard Schölkopf. A maximum margin miscellany. Preprint, 1999.