Data Complexity in Machine Learning and Novel Classification Algorithms

Thesis by

Ling Li

In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

California Institute of Technology
Pasadena, California

2006
(Defended May 4, 2006)

Acknowledgments

I owe a great debt of gratitude to many people who have helped make the research work in this thesis possible.

First and foremost, I thank my advisor, Dr. Yaser Abu-Mostafa, for his guidance, support, and encouragement throughout my years at Caltech. I am also grateful for his wisdom, understanding, and friendship for almost everything in life.

It has been quite a pleasure working in the Learning Systems Group with my fellow members, Amrit Pratap and Hsuan-Tien Lin. I have enjoyed many valuable discussions, especially those taking place during the conferences. It is also with great appreciation that I mention my friends in the Vision Group, Marco Andreetto, Anelia Angelova, Claudio Fanti, Alex Holub, Pierre Moreels, and Dr. Pietro Perona. With their expertise, they have provided many helpful suggestions and opinions.

I appreciate the efforts of my thesis committee, Dr. Yaser Abu-Mostafa, Dr. Steven Low, Dr. Robert McEliece, and Dr. Pietro Perona, for their time reviewing the thesis. I also thank Dr. Leonard Schulman for spending time with me on my research.

My special gratitude goes to Lucinda Acosta and Dr. Alexander Nicholson, simply for being so helpful in many aspects of my life and research. I also owe a lot to Dr. Amir Atiya, Dr. Malik Magdon-Ismail, Dr. Alcherio Martinoli, and Dr. Joseph Sill for numerous career suggestions.

Finally, I thank my family, especially my wife Xin, for their love and support.

Abstract

This thesis summarizes four of my research projects in machine learning. One of them is on a theoretical challenge of defining and exploring complexity measures for data sets; the others are about new and improved classification algorithms.

We first investigate the role of data complexity in the context of binary classification problems. The universal data complexity is defined for a data set as the Kolmogorov complexity of the mapping enforced by that data set. It is closely related to several existing principles used in machine learning such as Occam's razor, the minimum description length, and the Bayesian approach. We demonstrate the application of the data complexity in two learning problems, data decomposition and data pruning. In data decomposition, we illustrate that a data set is best approximated by its principal subsets which are Pareto optimal with respect to the complexity and the set size. In data pruning, we show that outliers usually have high complexity contributions, and propose methods for estimating the complexity contribution. Experiments were carried out with a practical complexity measure on several toy problems.

We then propose a family of novel learning algorithms to directly minimize the 0/1 loss for perceptrons. A perceptron is a linear threshold classifier that separates examples with a hyperplane. Unlike most perceptron learning algorithms, which require smooth cost functions, our algorithms directly minimize the 0/1 loss, and usually achieve the lowest training error compared with other algorithms. The algorithms are also computationally efficient. Such advantages make them favorable for both standalone use and ensemble learning, on problems that are not linearly separable. Experiments show that our algorithms work very well with AdaBoost.

We also study ensemble methods that aggregate many base hypotheses in order to achieve better performance. AdaBoost is one such method for binary classification problems. The superior out-of-sample performance of AdaBoost has been attributed to the fact that it minimizes a cost function based on the margin, in that it can be viewed as a special case of AnyBoost, an abstract gradient descent algorithm. We provide a more sophisticated abstract boosting algorithm, CGBoost, based on conjugate gradient in function space. When the AdaBoost exponential cost function is optimized, CGBoost generally yields much lower cost and training error but higher test error, which implies that the exponential cost is vulnerable to overfitting. With the optimization power of CGBoost, we can adopt more "regularized" cost functions that have better out-of-sample performance but are difficult to optimize. Our experiments demonstrate that CGBoost generally outperforms AnyBoost in cost reduction. With suitable cost functions, CGBoost can have better out-of-sample performance.

A multiclass classification problem can be reduced to a collection of binary problems with the aid of a coding matrix. The quality of the final solution, which is an ensemble of base classifiers learned on the binary problems, is affected by both the performance of the base learner and the error-correcting ability of the coding matrix. A coding matrix with strong error-correcting ability may not be overall optimal if the binary problems are too hard for the base learner. Thus a trade-off between error-correcting and base learning should be sought. In this paper, we propose a new multiclass boosting algorithm that modifies the coding matrix according to the learning ability of the base learner. We show experimentally that our algorithm is very efficient in optimizing the multiclass margin cost, and outperforms existing multiclass algorithms such as AdaBoost.ECC and one-vs-one. The improvement is especially significant when the base learner is not very powerful.

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Data Complexity in Machine Learning

We investigate the role of data complexity in the context of binary classification problems. The universal data complexity is defined for a data set as the Kolmogorov complexity of the mapping enforced by the data set. It is closely related to several existing principles used in machine learning such as Occam's razor, the minimum description length, and the Bayesian approach. The data complexity can also be defined based on a learning model, which is more realistic for applications. We demonstrate the application of the data complexity in two learning problems, data decomposition and data pruning. In data decomposition, we illustrate that a data set is best approximated by its principal subsets which are Pareto optimal with respect to the complexity and the set size. In data pruning, we show that outliers usually have high complexity contributions, and propose methods for estimating the complexity contribution. Since in practice we have to approximate the ideal data complexity measures, we also discuss the impact of such approximations.

1.1 Introduction

Machine learning is about pattern[1] extraction. A typical example is an image classifier that automatically tells the existence of some specific object category, say cars, in an image. The classifier would be constructed based on a training set of labeled image

[1] In a very general sense, the word "pattern" here means hypothesis, rule, or structure.

examples. It is relatively easy for computers to "memorize" all the examples, but in order for the classifier to also be able to correctly label images that have not been seen so far, meaningful patterns about images in general and the object category in particular should be learned. The problem is "what kind of patterns should be extracted?"

Occam's razor states that entities should not be multiplied beyond necessity. In other words, if presented with multiple hypotheses that have indifferent predictions on the training set, one should select the simplest hypothesis. This preference for simpler hypotheses is actually incorporated, explicitly or implicitly, in many machine learning systems (see for example Quinlan, 1986; Rissanen, 1978; Vapnik, 1999). Blumer et al. (1987) showed theoretically that, under very general assumptions, Occam's razor produces hypotheses that can correctly predict unseen examples with high probability. Although experimental evidence was found against the utility of Occam's razor (Webb, 1996), it is still generally believed that the bias towards simpler hypotheses is justified for real-world problems (Schmidhuber, 1997). Following this line, one should look for patterns that are consistent with the examples, and simple.

But what exactly does "simple" mean? The Kolmogorov complexity (Li and Vitányi, 1997) provides a universal measure for the "simplicity" or complexity of patterns. It says that a pattern is simple if it can be generated by a short program or if it can be compressed, which essentially means that the pattern has some "regularity" in it. The Kolmogorov complexity is also closely related to the so-called universal probability distribution (Li and Vitányi, 1997; Solomonoff, 2003), which is able to approximate any computable distributions. The universal distribution assigns high probabilities to simple patterns, and thus implicitly prefers simple hypotheses.

While most research efforts integrating Occam's razor in machine learning systems have been focused on the simplicity of hypotheses, the other equivalently important part in learning systems, training sets, has received much less attention in the complexity aspect, probably because training sets are given instead of learned. However, except for some side information such as hints (Abu-Mostafa, 1995), the training set is the sole information source about the underlying learning problem. Analyzing the complexity of the training set, as we will do in this paper, can actually reveal much useful information about the underlying problem.

This paper is a summary of our initial work on data complexity in machine learning. We focus on binary classification problems, which are briefly introduced in Section 1.2. We define the data complexity of a training set essentially as the Kolmogorov complexity of the mapping relationship enforced by the set. Any hypothesis that is consistent with the training set would have a program length larger than or equal to that complexity value. The properties of the data complexity and its relationship to some related work are discussed in Section 1.3.

By studying in Section 1.4 the data complexity of every subset of the training set, one would find that some subsets are Pareto optimal with respect to the complexity and the size. We call these subsets the *principal subsets*. The full training set is best approximated by the principal subsets at different complexity levels, analogous to the way that a signal is best approximated by the partial sums of its Fourier series. Examples not included in a principal subset are regarded as outliers at the corresponding complexity level. Thus if the decomposition of the training set is known, a learning algorithm with a complexity budget can just train on a proper principal subset to avoid outliers.

However, locating principal subsets is usually computationally infeasible. Thus in Section 1.5 we discuss efficient ways to identify some principal subsets.

Similar to the Kolmogorov complexity, the ideal data complexity measures are either incomputable or infeasible for practical applications. Some practical complexity measure that approximates the ideal ones has to be used. Thus we also discuss the impact of such approximation to our proposed concepts and methods. For instance, a data pruning strategy based on linear regression is proposed in Section 1.5 for better robustness.

Some related work is also briefly reviewed at the end of every section. Conclusion and future work can be found in Section 1.6.

1.2 Learning Systems

In this section, we briefly introduce some concepts and notations in machine learning, especially for binary classification problems.

We assume that there exists an unknown function f, called the *target function* or simply the *target*, which is a deterministic mapping from the input space $\mathcal{X}$ to the output space $\mathcal{Y}$. We focus on *binary classification problems* in which $\mathcal{Y} = \{0, 1\}$. An *example* or *observation* (denoted by $\mathbf{z}$) is in the form of an input-output pair $(\mathbf{x}, y)$, where the input $\mathbf{x}$ is generated independently from an unknown probability distribution $P_{\mathcal{X}}$, and the output y is computed via $y = f(\mathbf{x})$. A *data set* or *training set* is a set of examples, and is usually denoted by $\mathcal{D} = \{\mathbf{z}_n = (\mathbf{x}_n, y_n)\}_{n=1}^{N}$ with $N = |\mathcal{D}|$, the size of $\mathcal{D}$.

A *hypothesis* is also a mapping from $\mathcal{X}$ to $\mathcal{Y}$. For classification problems, we usually define the *out-of-sample error* of a hypothesis h as the expected error rate,

$$\pi(h) = \mathop{\mathrm{E}}_{\mathbf{x} \sim P_{\mathcal{X}}} [\![h(\mathbf{x}) \neq f(\mathbf{x})]\!],$$

where the Boolean test $[\![\cdot]\!]$ is 1 if the condition is true and 0 otherwise. The goal of learning is to choose a hypothesis that has a low out-of-sample error. The set of all candidate hypotheses (denoted by $\mathcal{H}$) is called the *learning model* or *hypothesis class*, and usually consists of some parameterized functions.

Since both the distribution $P_{\mathcal{X}}$ and the target function f are unknown, the out-of-sample error is inaccessible, and the only information we can access is often limited in the training set $\mathcal{D}$. Thus, instead of looking for a hypothesis h with a low out-of-sample error, a learning algorithm may try to find an h that minimizes the number of errors on the training set,

$$e_{\mathcal{D}}(h) = \sum_{n=1}^{N} [\![h(\mathbf{x}_n) \neq y_n]\!].$$

A hypothesis is said to *replicate* or *be consistent with* the training set if it has zero errors on the training set.

However, having less errors on the training set by itself cannot guarantee a low out-of-sample error. For example, a lookup table that simply memorizes all the training examples has no ability to generalize on unseen inputs. Such an *overfitting* situation is usually caused by endowing the learning model with too much complexity or flexibility. Many techniques such as early stopping and regularization were proposed to avoid overfitting by carefully controlling the hypothesis complexity.

The Bayes rule states that the most probable hypothesis h given the training set $\mathcal{D}$ is the one that has high likelihood $\Pr\{\mathcal{D} \mid h\}$ and prior probability $\Pr\{h\}$,

$$\Pr\{h \mid \mathcal{D}\} = \frac{\Pr\{\mathcal{D} \mid h\} \Pr\{h\}}{\Pr\{\mathcal{D}\}}. \tag{1.1}$$

Having less errors on the training set makes a high likelihood, but it does not promise a high prior probability. Regularizing the hypothesis complexity is actually an application of Occam's razor, since we believe simple hypotheses should have high prior probabilities.

The problem of finding a generalizing hypothesis becomes harder when the examples contain noise. Due to various reasons, an example may be contaminated in the input and/or the output. When considering only binary classification problems, we take a simple view about the noise—we say an example $(\mathbf{x}, y)$ is an *outlier* or *noisy* if $y = 1 - f(\mathbf{x})$, no matter whether the actual noise is in the input or the output.

1.3 Data Complexity

In this section, we investigate the complexity of a data set in the context of machine learning. The Kolmogorov complexity and related theories (Li and Vitányi, 1997) are briefly reviewed at the beginning, with a focus on things most relevant to machine learning. Our complexity measures for a data set are then defined, and their properties are discussed. Since the ideal complexity measures are either incomputable or infeasible for practical applications, we also examine practical complexity measures that approximate the ideal ones. At the end of this section, other efforts in quantifying the complexity of a data set are briefly reviewed and compared.

1.3.1 Kolmogorov Complexity and Universal Distribution

Consider a universal Turing machine $\mathcal{U}$ with input alphabet $\{0, 1\}$ and tape alphabet $\{0, 1, \text{␣}\}$, where ␣ is the blank symbol. A binary string p is a (prefix-free) *program* for the Turing machine $\mathcal{U}$ if and only if $\mathcal{U}$ reads the entire string and halts. For a program p, we use $|p|$ to denote its length in bits, and $\mathcal{U}(p)$ the output of p executed on the Turing machine $\mathcal{U}$. It is possible to have an input string x on an auxiliary tape. In that case, the output of a program p is denoted as $\mathcal{U}(p, x)$.

Given a universal Turing machine $\mathcal{U}$, the Kolmogorov complexity measures the algorithmic complexity of an arbitrary binary string s by the length of the shortest program that outputs s on $\mathcal{U}$. That is, the (prefix) *Kolmogorov complexity* $K_{\mathcal{U}}(s)$ is defined as

$$K_{\mathcal{U}}(s) = \min \{|p| : \mathcal{U}(p) = s\} . \tag{1.2}$$

$K_{\mathcal{U}}(s)$ can be regarded as the length of the shortest description or encoding for the string s on the Turing machine $\mathcal{U}$. Since universal Turing machines can simulate each other, the choice of $\mathcal{U}$ in (1.2) would only affect the Kolmogorov complexity by at most a constant that only depends on $\mathcal{U}$. Thus we can drop the $\mathcal{U}$ and denote the Kolmogorov complexity by $K(s)$.

The *conditional Kolmogorov complexity* $K(s \mid x)$ is defined as the length of the shortest program that outputs s given the input string x on the auxiliary tape. That is,

$$K(s \mid x) = \min \{|p| : \mathcal{U}(p, x) = s\} . \tag{1.3}$$

In other words, the conditional Kolmogorov complexity measures how many additional bits of information are required to generate s given that x is already known. The Kolmogorov complexity is a special case of the conditional one where x is empty.

For an arbitrary binary string s, there are many programs for a Turing machine $\mathcal{U}$ that output s. If we assume a program p is randomly picked with probability $2^{-|p|}$, the probability that a random program would output s is

$$P_{\mathcal{U}}(s) = \sum_{p\,:\,\mathcal{U}(p)=s} 2^{-|p|}. \tag{1.4}$$

The sum of $P_{\mathcal{U}}$ of all binary strings is clearly bounded by 1 since no program can be the prefix of another. The $\mathcal{U}$ can also be dropped since the choice of $\mathcal{U}$ in (1.4) only affects the probability by no more than a constant factor independent of the string. This partly justifies why P is named the *universal distribution*. The other reason is that the universal distribution P dominates any computable distributions by up to a multiplicative constant, which makes P the *universal prior*.

The Kolmogorov complexity and the universal distribution are closely related, since we have $K(s) \approx -\log P(s)$ and $P(s) \approx 2^{-K(s)}$. The approximation is within a constant additive or multiplicative factor independent of s. This is intuitive since the shortest program for s gives the most weight in (1.4).

The Bayes rule for learning (1.1) can be rewritten as

$$-\log \Pr\{h \mid \mathcal{D}\} = -\log \Pr\{\mathcal{D} \mid h\} - \log \Pr\{h\} + \log \Pr\{\mathcal{D}\}. \tag{1.5}$$

The most probable hypothesis h given the training set $\mathcal{D}$ would minimize $-\log \Pr\{h \mid \mathcal{D}\}$. Let's assume for now a hypothesis is also an encoded binary string. With the universal prior in place, $-\log \Pr\{h\}$ is roughly the code length for the hypothesis h, and $-\log \Pr\{\mathcal{D} \mid h\}$ is in general the minimal description length of $\mathcal{D}$ given h. This leads to the *minimum description length* (MDL) principle (Rissanen, 1978) which is a formalization of Occam's razor: the best hypothesis for a given data set is the one that minimizes the sum of the code length of the hypothesis and the code length of the data set when encoded by the hypothesis.

Both the Kolmogorov complexity and the universal distribution are incomputable.

1.3.2 Universal Data Complexity

As we have seen, for an arbitrary string, the Kolmogorov complexity $K(s)$ is a universal measure for the amount of information needed to replicate s, and $2^{-K(s)}$ is a universal prior probability of s. In machine learning, we care about similar perspectives: the amount of information needed to approximate a target function, and the prior distribution of target functions. Since a training set is usually the only information source about the target, we are thus interested in the amount of information needed to replicate a training set, and the prior distribution of training sets. In short, we want a complexity measure for a training set.

Unlike the Kolmogorov complexity of a string for which the exact replication of the string is mandatory, one special essence about "replicating" a training set is that the exact values of the inputs and outputs of examples do not matter. What we really want to replicate is the input-output relationship enforced by the training set, since this is where the target function is involved. The unknown input distribution $P_{\mathcal{X}}$ might be important for some machine learning problems. However, given the input part of the training examples, it is also irrelevant to our task.

At first glance the conditional Kolmogorov complexity may seem suitable for measuring the complexity of replicating a training set. Say for $\mathcal{D} = \{(\mathbf{x}_n, y_n)\}_{n=1}^{N}$, we collect the inputs and the outputs of all the examples, and apply the conditional Kolmogorov complexity to the outputs given the inputs, i.e.,

$$K(y_1, y_2, \ldots, y_N \mid \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N).$$

This conditional complexity, as defined in (1.3), finds the shortest program that takes as a whole all the inputs and generates as a whole all the outputs. In other words, the shortest program that maps all the inputs to all the outputs. The target function, if encoded properly as a program, can serve the mapping with an extra loop to take care of the N inputs, as will any hypotheses that can replicate the training set.

However, with this measure one has to assume some permutation of the examples. This is not only undesired but also detrimental in that some "clever" permutations would ruin the purpose of reflecting the amount of information in approximating the target. Say there are N_0 examples that have 0 as the output. With permutations that put examples having output 0 before those having output 1, the shortest program would probably just encode the numbers N_0 and $(N - N_0)$, and print a string of N_0 zeros and $(N - N_0)$ ones. The conditional Kolmogorov complexity would be approximately $\log N_0 + \log(N - N_0) + O(1)$, no matter how complicated the target might be.

Taking into consideration that the order of the examples should not play a role in the complexity measure, we define the *data complexity* as

Definition 1.1: *Given a fixed universal Turing machine $\mathcal{U}$, the data complexity of a data set $\mathcal{D}$ is*

$$C_{\mathcal{U}}(\mathcal{D}) = \min \{|p| : \forall (\mathbf{x}, y) \in \mathcal{D}, \mathcal{U}(p, \mathbf{x}) = y\}.$$

That is, the data complexity $C_{\mathcal{U}}(\mathcal{D})$ is the length of the shortest program that can correctly map every input in the data set $\mathcal{D}$ to its corresponding output.

Similar to the Kolmogorov complexity, the choice of the Turing machine can only affect the data complexity up to a constant. Formally, we have this invariance theorem.

Theorem 1.1: *For two universal Turing machines $\mathcal{U}_1$ and $\mathcal{U}_2$, there exists a constant c that only depends on $\mathcal{U}_1$ and $\mathcal{U}_2$, such that for any data set $\mathcal{D}$,*

$$|C_{\mathcal{U}_1}(\mathcal{D}) - C_{\mathcal{U}_2}(\mathcal{D})| \leq c. \tag{1.6}$$

PROOF: Let $\langle \mathcal{U}_1 \rangle$ be the encoding of $\mathcal{U}_1$ on $\mathcal{U}_2$. Any program p for $\mathcal{U}_1$ can be transformed to a program $\langle \mathcal{U}_1 \rangle\, p$ for $\mathcal{U}_2$. Thus $C_{\mathcal{U}_2}(\mathcal{D}) \leq C_{\mathcal{U}_1}(\mathcal{D}) + |\langle \mathcal{U}_1 \rangle|$. Let $\langle \mathcal{U}_2 \rangle$ be the encoding of $\mathcal{U}_2$ on $\mathcal{U}_1$. By symmetry, we have $C_{\mathcal{U}_1}(\mathcal{D}) \leq C_{\mathcal{U}_2}(\mathcal{D}) + |\langle \mathcal{U}_2 \rangle|$. So (1.6) holds for $c = \max \{|\langle \mathcal{U}_1 \rangle|, |\langle \mathcal{U}_2 \rangle|\}$.

Thus the data complexity is also universal, and we can drop the $\mathcal{U}$ and simply write $C(\mathcal{D})$.

Unfortunately, the data complexity is also not a computable function.

Lemma 1.1: *$C(\mathcal{D}) \leq K(\mathcal{D}) + c$ where c is a constant independent of $\mathcal{D}$.*

PROOF: Let p be the shortest program that outputs $\mathcal{D}$. Consider another program p' that takes an input $\mathbf{x}$, calls p to generate $\mathcal{D}$ on an auxiliary tape, searches $\mathbf{x}$ within the inputs of examples on the auxiliary tape, and returns the corresponding output if $\mathbf{x}$ is found and 0 otherwise. The program p' adds a "shell" with constant length c to the program p, and c is independent of $\mathcal{D}$. Thus $C(\mathcal{D}) \leq |p'| = |p| + c = K(\mathcal{D}) + c$.

Lemma 1.2: *The data complexity $C(\cdot)$ is not upper bounded.*

PROOF: Consider any target function $f\colon \{0,1\}^m \to \{0,1\}$ that accepts m-bit binary strings as inputs. A data set including all possible m-bit binary inputs and their outputs from the target f would fully decide the mapping from $\{0,1\}^m$ to $\{0,1\}$. Since the Kolmogorov complexity of such mapping (for all integer m) is not upper bounded (Abu-Mostafa, 1988b,a), neither is $C(\cdot)$.

Theorem 1.2: *The data complexity $C(\cdot)$ is incomputable.*

PROOF: We show this by contradiction. Assume there is a program p to compute $C(\mathcal{D})$ for any data set $\mathcal{D}$. Consider another program p' that accepts an integer input l, enumerates over all data sets, uses p to compute the data complexity for each data set, and stops and returns the first data set that has complexity at least l. Due to Lemma 1.2, the program p' will always halt. Denote the returned data set as $\mathcal{D}_l$. Since the program p' together with the input l can generate $\mathcal{D}_l$, we have $K(\mathcal{D}_l) \leq |p'| + K(l)$. By Lemma 1.1 and the fact that $C(\mathcal{D}_l) \geq l$, we obtain $l \leq K(l) + |p'| + c$, where c is the constant in Lemma 1.1. This is contradictory for l large enough since we know $K(l)$ is upper bounded by $\log l$ plus some constant.

With fixed inputs, a universal prior distribution can be defined on all the possible outputs, just similar to the universal prior distribution. However, the details will not be discussed in this paper.

1.3.3 Data Complexity with Learning Models

Using our notions in machine learning, the universal data complexity is the length of the shortest hypothesis that replicates the data set, given that the learning model is the set of all programs. However, it is not common that the learning model includes all possible programs. For a limited set of hypotheses, we can also define the data complexity.

Definition 1.2: *Given a learning model $\mathcal{H}$, the data complexity of a data set $\mathcal{D}$ is*

$$C_{\mathcal{H}}(\mathcal{D}) = \min \left\{ |h| : h \in \mathcal{H} \textit{ and } \forall\, (\mathbf{x}, y) \in \mathcal{D},\, h(\mathbf{x}) = y \right\}.$$

This definition is almost the same as Definition 1.1, except that program p has now been replaced with hypothesis $h \in \mathcal{H}$. An implicit assumption is that there is a way to measure the "program length" or complexity of any hypothesis in the learning model. Here we assume an encoding scheme for the learning model that maps a hypothesis to a prefix-free binary codeword. For example, the encoding scheme for feed-forward neural networks (Bishop, 1995) can be the concatenation of the number of network layers, the number of neurons in every layer, and the weights of every neuron, with each number represented by a self-delimited binary string. We also assume that a program $p_{\mathcal{H}}$, called the *interpreter* for the learning model $\mathcal{H}$, can take a codeword and emulate the encoded hypothesis. Thus $|h|$, the complexity of the hypothesis h, is defined as the length of its codeword.[2] It is easy to see that $C_{\mathcal{U}}(\mathcal{D}) \le |p_{\mathcal{H}}| + C_{\mathcal{H}}(\mathcal{D})$.

The data complexity as Definition 1.2 is in general not universal, i.e., it depends on the learning model and the encoding scheme, since full simulation of one learning model by another is not always possible. Even with the same learning model, two encoding schemes could differ in a way that it is impossible to bound the difference in the codeword lengths of the same hypothesis.

Definition 1.2 requires that some hypothesis in the learning model can replicate the data set. This is probably reasonable if the target is in the learning model and the

[2] This also includes the possibility of using a universal Turing machine as the interpreter and directly mapping a hypothesis to a program. In this case, $|h|$ is the program length.

data set is also noiseless. What if the target is not in the learning model or there is noise in the data set? A data set might not be consistent with any of the hypotheses, and thus the data complexity is not defined for it. Actually in the case of noisy data sets, even if there are hypotheses that are consistent, it is not desirable to use their complexity as the data complexity. The reason will be clear later in this subsection. In summary, we need another definition that can take care of replication errors.

Consider a hypothesis h that is consistent with all the examples except $(\mathbf{x}_1, y_1)$. We can construct a program p by memorizing input $\mathbf{x}_1$ with a lookup table entry:

$$p = \text{if input is } \mathbf{x}_1\text{, then output } y_1\text{; else run the interpreter } p_{\mathcal{H}} \text{ on } h.$$

Excluding the length of the interpreter, which is common to all hypotheses, the program p is just a little longer than h, but can perfectly replicate the data set. Actually, the increase of the program length is the length of the "if ... then ... else" structure plus the Kolmogorov complexity of $\mathbf{x}_1$ and y_1. For a hypothesis that has more than one error, several lookup table entries can be used.[3] If we assume the increase in the program length is a constant for every entry, we have

Definition 1.3: *Given a learning model $\mathcal{H}$ and a proper positive constant λ, the data complexity (with a lookup table) of a data set $\mathcal{D}$ is*

$$C_{\mathcal{H},\lambda}(\mathcal{D}) = \min \left\{ |h| + \lambda e_{\mathcal{D}}(h) \colon h \in \mathcal{H} \right\}.$$

The constant λ can be seen as the equivalent complexity of implementing one lookup table entry with the learning model. It can also be regarded as the complexity cost of one error. Definition 1.2 does not allow any errors, so the data complexity $C_{\mathcal{H}}(\cdot)$ is actually $C_{\mathcal{H},\infty}(\cdot)$.

[3] There are other general ways to advise that h fails to replicate the example $(\mathbf{x}_1, y_1)$. Here is another one:

$$p = \text{let } y = h(\text{input})\text{; if input is in } \{\mathbf{x}_1\}\text{, then output } 1 - y\text{; else output } y.$$

When there are several erroneous examples, $\{\mathbf{x}_1\}$ can be replaced with the set of the inputs of the erroneous examples. If only very basic operations are allowed for constructing p from h, all these ways lead to the same Definition 1.3 of the data complexity.

For positive and finite λ, the data complexity $C_{\mathcal{H},\lambda}(\mathcal{D})$ is actually computable. This is because the complexity is bounded by $\min\{|h| : h \in \mathcal{H}\} + \lambda\,|\mathcal{D}|$, and we can enumerate all codewords that are not longer than that bound.[4]

Given a learning model $\mathcal{H}$ and an encoding scheme, which determines the hypothesis complexity, we consider a prior of hypotheses where $\Pr\{h\} = 2^{-|h|}$. Let's also assume a Bernoulli noise model where the probability of an example being noisy is ε. This gives the likelihood as

$$\Pr\{\mathcal{D} \mid h\} = \varepsilon^{e_{\mathcal{D}}(h)}(1-\varepsilon)^{N-e_{\mathcal{D}}(h)} = (1-\varepsilon)^N \left(\varepsilon^{-1}-1\right)^{-e_{\mathcal{D}}(h)}.$$

And according to (1.5), we have

$$-\log \Pr\{h \mid \mathcal{D}\} = |h| + e_{\mathcal{D}}(h) \cdot \log\left(\varepsilon^{-1}-1\right) + c,$$

where $c = \log \Pr\{\mathcal{D}\} - N \log(1-\varepsilon)$ is a constant independent of the hypothesis h. To maximize the posterior probability $\Pr\{h \mid \mathcal{D}\}$ or to minimize $-\log \Pr\{h \mid \mathcal{D}\}$ is equivalent to minimize the sum of the hypothesis complexity and the error cost for $C_{\mathcal{H},\lambda}(\mathcal{D})$, with $\lambda = \log(\varepsilon^{-1}-1)$. And the case of $C_{\mathcal{H}}(\cdot)$ or $C_{\mathcal{H},\infty}(\cdot)$ corresponds to $\varepsilon = 0$. The Bayesian point of view justifies the use of λ, and also emphasizes that the encoding scheme should be based on a proper prior of hypotheses.

We also have this straightforward property:

Theorem 1.3: $C_{\mathcal{H},\lambda}(\mathcal{D}) \le C_{\mathcal{H},\lambda}(\mathcal{D} \cup \mathcal{D}') \le C_{\mathcal{H},\lambda}(\mathcal{D}) + \lambda\,|\mathcal{D}'|$.

The first inequality says that the data complexity is increasing when more examples are added. The second inequality states that the increase of the complexity is at most $\lambda\,|\mathcal{D}'|$, the cost of treating all the added examples with lookup table entries. The increase would be less if some of the added examples can be replicated by the shortest hypothesis for $\mathcal{D}$, or can form patterns which are shorter than lookup table entries. More will be discussed on these two cases when the complexity-error path is

[4]Well, we also assume that every hypothesis, simulated by the interpreter, always halts. This is true for any reasonable learning models.

introduced (Definition 1.6 on page 31).

For the rest of the paper, we will mostly work with Definition 1.2 and Definition 1.3, and we will use just $C(\cdot)$ for $C_{\mathcal{H},\lambda}(\cdot)$ or $C_{\mathcal{U}}(\cdot)$ when the meaning is clear from the context. Because lookup table entries are an integrated part of Definition 1.3, we will also simply use "hypothesis" to mean a hypothesis together with lookup table entries. Thus all the three data complexity definitions can be unified as the length of the shortest consistent hypothesis. We use $h_{\mathcal{D}}$ to denote one of the shortest hypotheses that can replicate $\mathcal{D}$.

We will also assume that any mechanisms for memorizing individual examples, no matter whether it is built-in or implemented as lookup table entries as in Definition 1.3, would cost the same complexity as a lookup table. In other words, if an example cannot help build patterns for other examples, adding it to a set would increase the data complexity by λ.

1.3.4 Practical Measures

Although we now have three data complexity measures, none of them is feasible in practice. The universal data complexity $C_{\mathcal{U}}(\cdot)$ is incomputable. The data complexity defined on a learning model $\mathcal{H}$, $C_{\mathcal{H}}(\cdot)$, may be computable for some $\mathcal{H}$, but finding a hypothesis that is consistent with the data set is usually NP-complete, not to mention finding a shortest one. The data complexity with a lookup table $C_{\mathcal{H},\lambda}(\cdot)$ seems the most promising to be used in practice. But it also suffers from the exponential time complexity in searching for a shortest hypothesis (with errors). We need to have some approximate complexity measure for practical applications.

A reasonable approximation to $C_{\mathcal{H}}(\cdot)$ or $C_{\mathcal{H},\lambda}(\cdot)$ can be obtained as a byproduct of the learning procedure. A learning algorithm usually minimizes the number of errors plus some regularization term over the learning model, and the regularization term is usually meant to approximate the complexity (encoding length) of a hypothesis. Thus some information about the learned hypothesis can be used as a practical data complexity measure. For example, the number of different literals used to construct a mixed DNF-CNF rule was used by Gamberger and Lavrač (1997). In the following text, we will deduce another practical data complexity measure based on the hard-margin support vector machine.

The hard-margin support vector machine (SVM) (Vapnik, 1999) is a learning algorithm that finds an optimal hyperplane to separate the training examples with maximal minimum margin. A hyperplane is defined as $\langle \mathbf{w}, \mathbf{x} \rangle - b = 0$, where $\mathbf{w}$ is the weight vector and b is the bias. Assuming the training set is linearly separable, SVM solves the optimization problem below:

$$\begin{aligned} \min_{\mathbf{w}, b} \quad & \|\mathbf{w}\|^2, \\ \text{subject to} \quad & y_n \left(\langle \mathbf{w}, \mathbf{x}_n \rangle - b \right) \geq 1, \quad n = 1, \ldots, N. \end{aligned} \tag{1.7}$$

The dual problem is

$$\begin{aligned} \min_{\alpha} \quad & \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j \langle \mathbf{x}_i, \mathbf{x}_j \rangle - \sum_{n=1}^{N} \alpha_n, \\ \text{subject to} \quad & \alpha_n \geq 0, \quad n = 1, \ldots, N, \\ & \sum_{n=1}^{N} y_n \alpha_n = 0. \end{aligned}$$

The optimal weight vector, given the optimal α^* for the dual problem, is a linear combination of the training input vectors,

$$\mathbf{w}^* = \sum_{n=1}^{N} y_n \alpha_n^* \mathbf{x}_n. \tag{1.8}$$

Note that only the so-called *support vectors*, for which the equality in (1.7) is achieved, can have nonzero coefficients α_n^* in (1.8).

For a linearly nonseparable training set, the kernel trick (Aizerman et al., 1964) is used to map input vectors to a high-dimensional space and an optimal separating hyperplane can be found there. Denote the inner product in the mapped space of two inputs $\mathbf{x}$ and $\mathbf{x}'$ as $\mathcal{K}(\mathbf{x}, \mathbf{x}')$, the so-called kernel operation. The dual problem with the kernel trick uses the kernel operation instead of the normal inner product, and the optimal hypothesis (a hyperplane in the mapped space but no longer a hyperplane in the input space) becomes

$$\sum_{n=1}^{N} y_n \alpha_n^* \mathcal{K}(\mathbf{x}_n, \mathbf{x}) - b^* = 0.$$

Since the mapped space is usually high-dimensional or even infinite-dimensional, it is reasonable to describe the SVM hypothesis by listing the support vectors and their coefficients. Thus the descriptive length is approximately $(Mc_1 + c_2 + c_3)$, where M is the number of support vectors, c_1 is the average Kolmogorov complexity of describing an input vector and a coefficient, c_2 is the Kolmogorov complexity of the bias, and c_3 is the descriptive length of the summation and the kernel operation. Since c_3 is a common part for all SVM hypotheses using the same kernel, and c_2 is relatively small compared to c_1, we can use just the number of support vectors, M, as the complexity measure for SVM hypotheses.

With some minor conditions, SVM with powerful kernels such as the stump kernel and the perceptron kernel (Lin and Li, 2005a,b) can always perfectly replicate a training set. Thus the measure based on SVM with such kernels fit well with Definition 1.2. In the experiments for this paper, we used the perceptron kernel, which usually has comparable learning performance to the popular Gaussian kernel, but do not require a parameter selection (Lin and Li, 2005b).

Note that SVM can also be trained incrementally (Cauwenberghs and Poggio, 2001). That is, if new examples are added after an SVM has already been learned on the training set, the hyperplane can be updated to accommodate the new examples in an efficient way. Such capability of incrementally computing the complexity can be quite useful in some applications, such as data pruning (Subsection 1.5.2) and deviation detection (Arning et al., 1996).

1.3.5 Related Work

Our data complexity definitions share some similarity to the randomness of decision problems. Abu-Mostafa (1988b,a) discussed decision problems where the input space of the target function was finite, and defined the randomness of a problem based on the Kolmogorov complexity of the target's truth table. The randomness can also be based on the length of the shortest program that implements the target function, which is essentially equivalent to the previous definition (Abu-Mostafa, 1988a). However, in our settings, the input space is infinite and the training set includes only finite examples; hence an entire truth table is infeasible. Thus the second way, which we have adopted, seems to be the only reasonable definition.

Definition 1.3, the data complexity with a lookup table, is also similar to the two-part code length of the minimum description length (MDL) principle (Rissanen, 1978; Grünwald, 2005). The two-part scheme explains the data via encoding a hypothesis for the data, and then encoding the data with the help of the hypothesis. The latter part usually takes care of the discrepancy information between the hypothesis and the data, just like in Definition 1.3. However, in our definition, the inputs of the data are not encoded, and we explicitly ignore the order of examples when considering the discrepancy.

The data complexity is also conceptually aligned with the CLCH value (complexity of the least complex correct hypothesis) proposed by Gamberger and Lavrač (1997). They required the complexity measure for hypotheses, which is the program length in this paper, to be "reasonable." That is, for two hypotheses, h_1 and h_2, where h_2 is obtained by "conjunctively or disjunctively adding conditions" to h_1, h_1 should have no larger complexity than h_2. However, this intuitively correct requirement is actually troublesome. For instance, h_1 recognizes all points in a fixed hexagon, and h_2 recognizes all points in a fixed triangle enclosed in that hexagon. Although h_2 can be obtained by adding more constraints on h_1, it is actually simpler than h_1. Besides, their definition of a set being "saturated" and the corresponding "saturation test" depend heavily on the training set being large enough to represent the target function, which might not be practical.

Except the usual complexity measures based on logic clauses, not many practical complexity measures have been studied. Schmidhuber (1997) implemented a variant of the general universal search (Levin, 1973) to find a neural network with a close-to-minimal Levin complexity. Although the implementation is only feasible on very simple toy problems, his experiments still showed that such search, favoring short hypotheses, led to excellent generalization performance, which reinforced the validity of Occam's razor in learning problems.

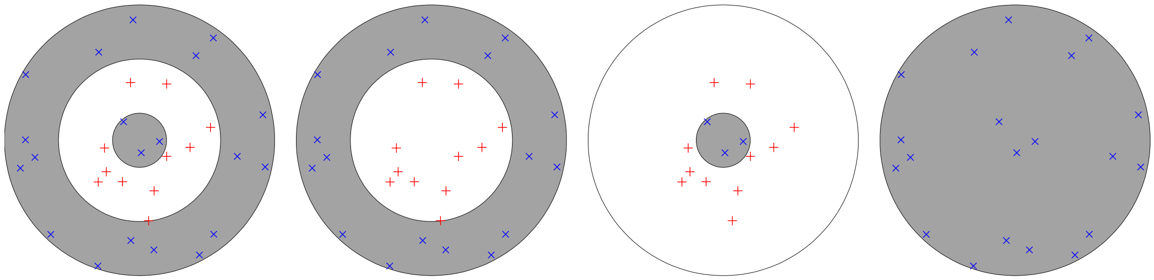
Wolpert and Macready (1999) proposed a very interesting complexity measure called self-dissimilarity. They observed that many complex systems tend to exhibit different structural patterns over different space and time scale. Thus the degrees of self-dissimilarity between the various scales with which a system is examined constitute a complexity signature of that system. It is mainly a complexity measure for a system, or a target function in our context, which can provide information at different scales, and is not straightforward to be applied to a data set.

1.4 Data Decomposition

In this section, we discuss the issue of approximating a data set with its subsets. Compared with the full data set, a subset is in general simpler (lower data complexity) but less informative (fewer examples). In addition, different subsets can form different patterns, and thus lead to different combinations of the data complexity and the subset size. We show that the principal subsets, defined later in this section, have the Pareto optimal combinations and best approximate the full set at different complexity levels. The concept of the principal subsets is also useful for data pruning (Section 1.5).

1.4.1 Complexity-Error Plots

When there is only one class of examples, the data complexity is a small constant. Only with examples from both classes can more interesting patterns be formed. Given



(a) (b) (c) (d)

Figure 1.1: Subsets of examples from the concentric disks problem: (a) all examples; (b) examples from the two outer rings; (c) examples within the middle disk; (d) all "$\times$" examples

a training set $\mathcal{D}$, different subsets of $\mathcal{D}$ may form different patterns, and thus lead to different complexity values.

Figure 1.1(a) shows a toy learning problem with a target consisting of three concentric disks. The target is depicted with the "white" and "gray" backgrounds in the plot—examples on the white background are classified as class 1, and examples on the gray background are classified as 0. The examples in the plot were randomly generated and are marked with "$+$" and "$\times$" according to their class labels. The other three plots in Figure 1.1 illustrate how different subsets of the examples can be explained by hypotheses of different complexity levels, and thus may have different complexity values. We also see that different subsets approximate the full set to different degrees.

For a given data set $\mathcal{D}$, we are interested in all possible combinations of the data complexity and the approximation accuracy of its subsets. Consider the following set of pairs:

$$\Omega_1 = \{(C(\mathcal{S}), |\mathcal{D}| - |\mathcal{S}|) : \mathcal{S} \subseteq \mathcal{D}\}\,. \tag{1.9}$$

Here we use $|\mathcal{D}| - |\mathcal{S}|$ as the approximation error of $\mathcal{S}$.[5] The set Ω_1 can be regarded as a plot of points on the 2-D plane. For each subset $\mathcal{S}$, there is a point in Ω_1 with the horizontal axis giving the data complexity and the vertical axis showing the approximation error. Such a plot is called the *subset-based complexity-error plot* (see

[5] If we regard $\mathcal{S}$ as a lookup table, the error of the lookup table on the full set is $|\mathcal{D}| - |\mathcal{S}|$.

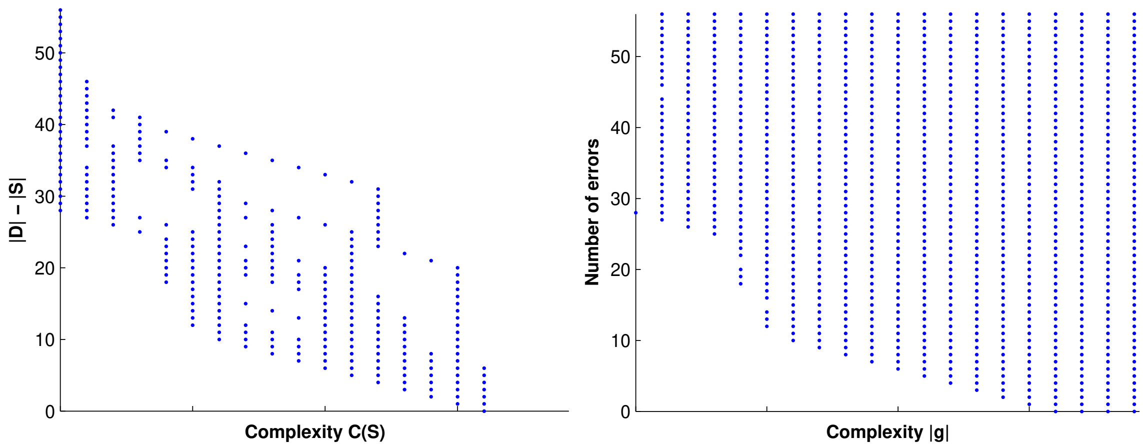


Figure 1.2: Fictional complexity-error plots for (left) Ω_1 and (right) Ω_2

Figure 1.2).

We can also consider another set built upon programs or hypotheses:

$$\Omega_2 = \left\{ \left(|h| \, , e_{\mathcal{D}}(h) \right) : h \in \mathcal{H} \right\}.$$

This set, Ω_2, has a point for each hypothesis h in the learning model, depicting the complexity of the hypothesis and the number of errors on the training set. It is called the *hypothesis-based complexity-error plot*. Note that the hypothesis h and the learning model $\mathcal{H}$ shall agree with the data complexity measure $C(\cdot)$ used in (1.9). For example, if the data complexity measure allows lookup tables, the learning model $\mathcal{H}$ would then includes hypotheses appended with lookup tables of all sizes.

The two plots in Figure 1.2 demonstrate for a fictional training set how the two sets of pairs look. Here are some observations for the two complexity-error plots:

1. For each point in Ω_1, there is at least one subset $\mathcal{S}$ associated with it. The point associated with $\mathcal{S}$ also shows the complexity of $h_{\mathcal{S}}$.

2. There is a subset $\mathcal{D}_h$ associated with each hypothesis h,

$$\mathcal{D}_h = \left\{ (\mathbf{x}, y) : \ (\mathbf{x}, y) \in \mathcal{D} \text{ and } h(\mathbf{x}) = y \right\}.$$

   And $e_{\mathcal{D}}(h) = |\mathcal{D}| - |\mathcal{D}_h|$. Thus the point associated with h also shows the approximation error of the subset $\mathcal{D}_h$.

3. The leftmost points in both plots are with subsets of only one class, since that gives the lowest complexity.

4. The points on the horizontal axis are associated with the full set.

5. For any point in Ω_1, there is a point in Ω_2 that has the same complexity value but a smaller or equal error value. This is because $|\mathcal{D}_{h_{\mathcal{S}}}| \geq |\mathcal{S}|$ when $\mathcal{S} \subseteq \mathcal{D}$.

6. For any point in Ω_2, there is a point in Ω_1 that has the same error value but a smaller or equal complexity value. This is because $C(\mathcal{D}_h) \leq |h|$.

1.4.2 Principal Points and Principal Subsets

The two complexity-error plots depict all the possible combinations of the data complexity and the approximation error for subsets of the training set. In general, if one subset or hypothesis gets more examples correct, it would be more complex. However, with the same data complexity, some subsets may contain more examples than others; and with the same size, some subsets may be simpler than others. Ideally, to approximate the full set, we want a subset to have the most examples but the least complexity.

With respect to the data complexity and the approximation error, some points in a complexity-error plot are optimal in the sense that no other points are better than them. They are called the *principal points*:

Definition 1.4: *A point (c, e) in a complexity-error plot is a principal point if and only if we have for any other point (c', e') in the plot, $c' > c$ or $e' > e$.*

In other words, a principal point is a Pareto optimal point, since there are no other points that have one coordinate smaller without making the other coordinate larger.

Although the subset-based complexity-error plot looks quite different from the hypothesis-based complexity-error plot, they actually have the same set of principal points.

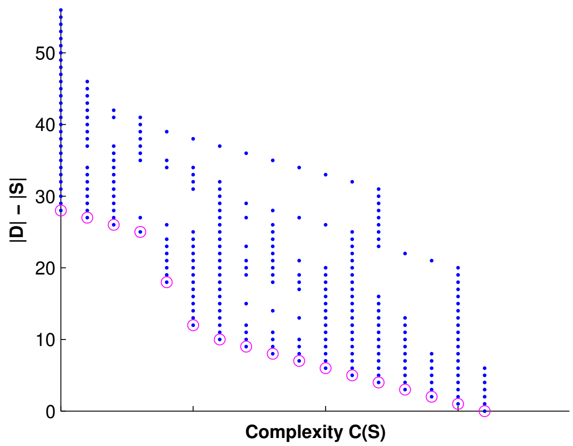


Figure 1.3: A fictional complexity-error plot with principal points circled

Theorem 1.4: *The subset-based and hypothesis-based complexity-error plots have the same set of principal points.*

Proof: The proof utilizes the observations in the previous subsection that relates points in these two plots. For any principal point $(c, e) \in \Omega_1$, there is a point $(c_2, e_2) \in \Omega_2$ with $c_2 = c$ and $e_2 \leq e$. If (c_2, e_2) is not a principal point in Ω_2, we can find a point $(c_2', e_2') \in \Omega_2$ such that $c_2' \leq c_2$, $e_2' \leq e_2$, and at least one inequality would be strict; otherwise let $c_2' = c_2$ and $e_2' = e_2$. For (c_2', e_2'), there is a point $(c', e') \in \Omega_1$ with $c' \leq c_2'$ and $e' = e_2'$. Overall we have $c' \leq c$ and $e' \leq e$, and either $e_2 < e$ or (c_2, e_2) not being a principal point will make at least one inequality be strict, which contradicts the assumption that (c, e) is a principal point in Ω_1. Thus $e = e_2$ and (c, e) is also a principal point in Ω_2. Likewise we can also prove that any principal point of Ω_2 is also a principal point in Ω_1.

Figure 1.3 shows the principal points in the subset-based complexity-error plot from the last fictional problem. Note that each principal point is associated with at least one subset and one hypothesis. Each principal point represents some optimal trade-off between the data complexity and the size. To increase the size of the associated subset, we have to go to a higher complexity level; to reduce the data complexity, we have to remove examples from the subset. Each principal point also represents some optimal trade-off between the hypothesis complexity and the hypothesis error.

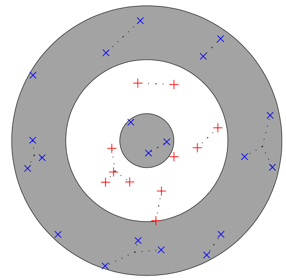


Figure 1.4: 15 clusters of the concentric disks problem, shown as groups of examples connected with dotted lines

To decrease the error on the training set, a more complex hypothesis should be sought; to use a simpler hypothesis, more errors would have to be tolerated. Thus the principal point at a given complexity level gives the optimal error level, and implicitly the optimal subset to learn, and the optimal hypothesis to pick.

A subset associated with a principal point is called a *principal subset*. The above arguments actually say that a principal subset is a best approximation to the full set at some complexity level.

1.4.3 Toy Problems

We verify the use of the data complexity in data decomposition with two toy problems. The practical data complexity measure is the number of support vectors in a hard-margin SVM with the perceptron kernel (see Subsection 1.3.4).

The first toy problem is the concentric disks problem with 31 random examples (see page 19). To have the subset-based complexity-error plot, we need to go over all the $(2^{31} - 1)$ subsets and compute the data complexity for each subset. To make the job computationally more feasible, we cluster the examples as depicted in Figure 1.4 by examples connected with dotted lines, and only examine subsets that consist of whole clusters.[6] The complexity-error plot using these 15 clusters, with principal points circled, is shown in Figure 1.5.

[6]The 15 clusters in Figure 1.4 were manually chosen based on example class and distance, such that each cluster contains only examples of the same class, and is not too close to examples of the other class. Although this could be done by some carefully crafted algorithm, we did it manually since the training set is small.

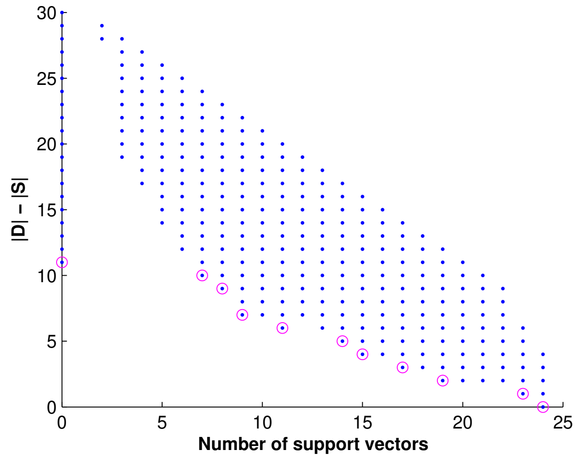
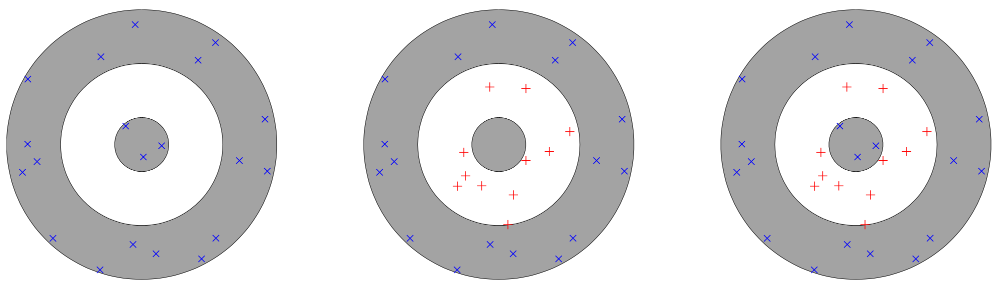


Figure 1.5: The complexity-error plot based on the clusters of the concentric disks problem (Figure 1.4), with the principal points circled



(a) $(0, 11)$ (b) $(17, 3)$ (c) $(24, 0)$

Figure 1.6: Three selected principal subsets of the concentric disks problem (complexity-error pairs are also listed)

There are 11 principal points associated with 17 principal subsets. In Figure 1.14 on page 45, we list all the 17 principal subsets, of which three selected ones are shown in Figure 1.6. The data complexity based on SVM-perceptron and the number of errors are also listed below the subset plots, and the white and gray background depicts the target function. Plot (a) shows the situation where a simplest hypothesis predicts the negative class regardless of the actual inputs. The subset is same as the one in Figure 1.1(d) on page 19. Plot (c) is the full training set with the highest data complexity, same as the one in Figure 1.1(a). The middle one, plot (b) or Figure 1.1(b), gives an intermediate situation such that the two classes of examples in the two outer rings are replicated, but the examples in the inner disk are deserted.

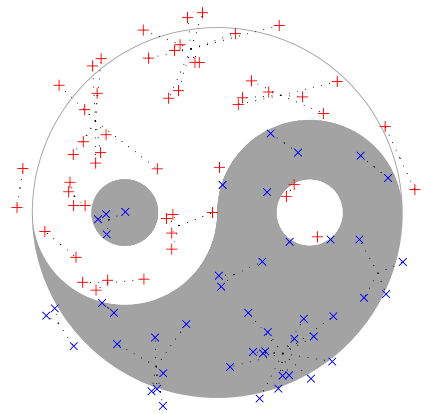


Figure 1.7: 32 clusters of the Yin-Yang problem, shown as groups of examples connected with dotted lines

This implies that, at that level of complexity, examples in the inner disk should rather be regarded as outliers than exceptions to the middle disk.

The second toy problem is about the Yin-Yang target function used by Li et al. (2005), which is also a 2-D binary classification problem. The background colors in Figure 1.7 depict how the Yin-Yang target classifies examples within a round plate centered at the origin; examples out of the plate belong to the Yang (white) class if it is in the upper half-plane. The training set consists of 100 examples randomly picked within a circle slightly larger than the plate. Clustering is also required for generating the complexity-error plot. Figure 1.7 also shows the 32 manually chosen clusters. The resulted complexity-error plot, based on the practical data complexity measure with SVM-perceptron, is show in Figure 1.8.

This time we get 25 principal points and 48 associated principal sets (see Figure 1.15 on page 46 and Figure 1.16 on page 47 for most of them). Here in Figure 1.9, we list and organize with arrows 11 principal sets that we think are representative. With the flow of arrows, the data complexity increases and the number of errors goes down. The first principal subset shows a simplest hypothesis classifying any input as the Yang (white) class. The one to the right shows a slightly more complex hypothesis that seems to separate the two classes by a line. From now on, the classification patterns fork in slightly different routes. The two subsets in the left route with points $(12, 15)$ and $(16, 11)$ continue the trend to separate two classes by relatively straight

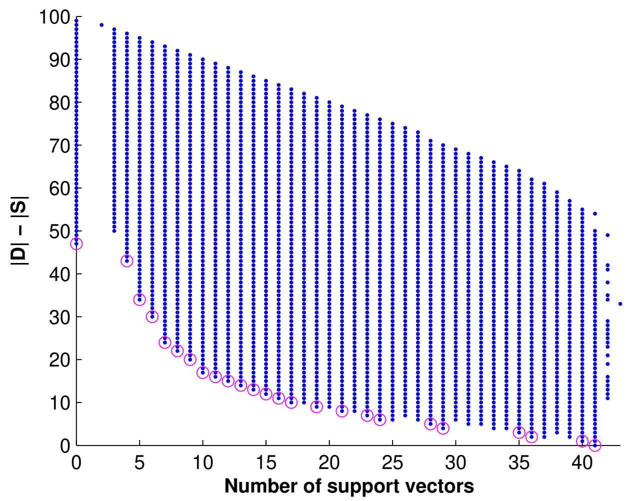


Figure 1.8: The complexity-error plot based on the clusters of the Yin-Yang problem (Figure 1.7), with the principal points circled

boundaries, only that more examples are included. Probably due to the specific random sample that we have as the training set, it is relatively "easier" or "cheaper" to replicate the examples in the small white disk than those in the small gray disk. This is reflected in both subsets in the left route. On the other hand, the hypotheses associated with the subsets in the right route, also with points $(12, 15)$ and $(16, 11)$, try to mimic the main S-shape boundary and ignore all examples in the two small disks. This gets to an extreme situation with the fourth subset in the right route (point $(23, 7)$), where all examples except the seven ones in the two small disks can be correctly replicated. With higher complexity values, the two routes merge at subsets (points $(23, 7)$ and $(29, 4)$ on the left) that include both examples in the small white disk but also examples around the S-shape boundary. Finally the examples in the small gray disk are also included.

1.4.4 Discussion

The concept of data decomposition is quite similar to approximating a signal function with partial sums of its Fourier series expansion. Each component in the Fourier series carries information about the original signal function at some frequency level. And the partial sum gives the best approximation to the signal function up to some

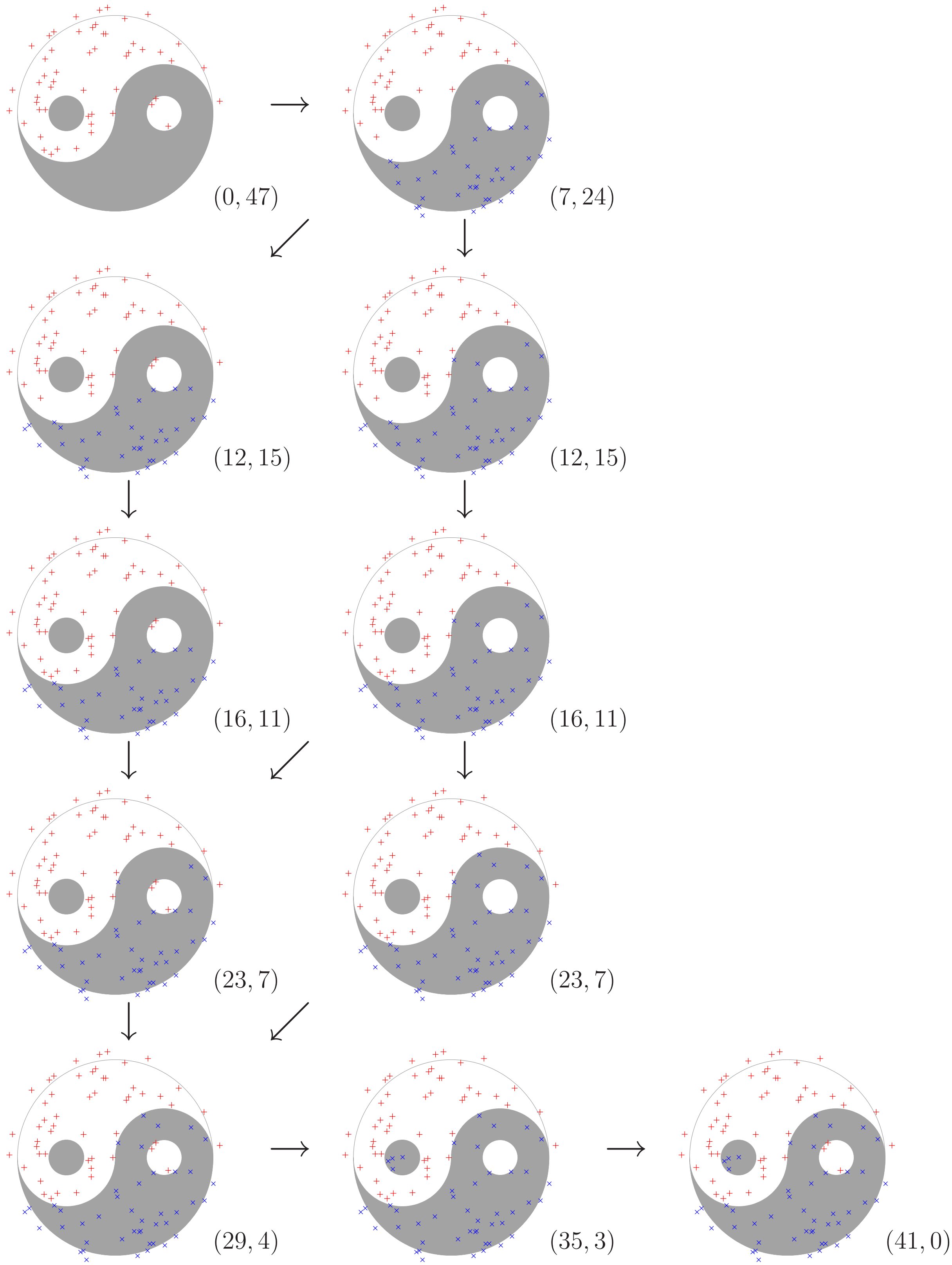


Figure 1.9: Eleven selected principal subsets of the Yin-Yang problem (complexity-error pairs are also listed)

frequency level. Higher precision can be obtained by adding more series components of higher frequency levels to the partial sum, and in the limit, the sum becomes the signal function itself.

Following this analogy, we would want to "decompose" the full data set $\mathcal{D}$ into a series of "components," denoted as δ_ℓ, which has information about the full set at different complexity levels quantified with ℓ. That is,

$$\mathcal{D} = \biguplus_{\ell=1}^{\infty} \delta_\ell,$$

where $\biguplus$ is some operation to add up the components. Although at this point we are still unclear exactly what the components δ_ℓ are and what the operation $\biguplus$ does, we would expect the partial sum,

$$\mathcal{D}_L = \biguplus_{\ell=1}^{L} \delta_\ell,$$

which should be a subset of $\mathcal{D}$, to be the best approximation to $\mathcal{D}$ within the complexity level L.

Our analysis about the complexity-error pairs of all the subsets concludes that $\mathcal{D}_L$ should be a principal subset. Since a smaller principal subset is not necessarily a subset of a larger principal subset, the decomposition component δ_ℓ is in general not a set of examples. We may consider δ_ℓ as a set of examples that should be added *and* a set of examples that should be removed at the complexity level ℓ.[7] That is, moving from one principal subset to another generally involves adding and removing examples.

This fact also leads to the inevitable high computational complexity of locating principal subsets. It is not straightforward to generate a new principal subset from a known one, and to determine whether a given subset is principal, exponentially many other subsets needed to be checked and compared. This issue will be reexamined in

[7] To even further generalize the data decomposition concept, we can formulate both the component δ_ℓ and the partial sum $\mathcal{D}_L$ as a set of input-belief pairs, where the belief replaces the original binary output and tells how much we believe what the output should be. However, this more generalized setting is not compatible with our data complexity measures for binary classification data sets, and will not be discussed in this paper.

Subsection 1.5.1. There we will see that the computational complexity is related to the number of the complexity-error paths that contain principal points.

1.4.5 Related Work

A general consensus is that not all examples in a training set are equivalently important to learning. For instance, examples contaminated by noise are harmful to learning. Even in cases where all the examples are noiseless, there are situations in which we want to deal with examples differently. For instance, in cases where none of the hypotheses can perfectly model the target, it is better to discard examples that cannot be classified correctly by any hypothesis as they may "confuse" the learning (Nicholson, 2002; Li et al., 2005).

There have been many different criteria to discriminate examples as different categories: consistent vs. inconsistent (Brodley and Friedl, 1999; Hodge and Austin, 2004), easy vs. hard (Merler et al., 2004), typical vs. informative (Guyon et al., 1996), etc. Li et al. (2005) unified some of the criteria with the concept of intrinsic margin and group examples as typical, critical, and noisy.

The approach mentioned in this paper takes a quite different view for categorizing examples. There are no natively good or bad examples. Examples are different only because they demand different amount of data complexity for describing them together with other examples. Although we usually think an example is noisy if it demands too much complexity, the amount can be different depending on what other examples are also included in the set. Thus an example that seems noisy at some level of complexity, or with some subset of examples, could be very innocent at another level of complexity or with another subset of examples.

Hammer et al. (2004) studied Pareto optimal patterns in logical analysis of data. The preferences on patterns were deliberately picked, which is quite different from our data complexity measures, so that a Pareto optimal pattern could be found efficiently. Nevertheless, they also favored simpler patterns or patterns that were consistent with more examples. Their experimental results showed that Pareto optimal patterns led to superior learning performance.

The so-called function decomposition (Zupan et al., 1997, 2001) is a method that learns the target function in terms of a hierarchy of intermediate hypotheses, which effectively decomposes a learning problem into smaller and simpler subproblems. Although the idea and approaches are quite different from data decomposition, it shares a similar motivation for simple hypotheses. The current methods of function decomposition are usually restricted to problems that have discrete/nominal input features.

1.5 Data Pruning

Every example in a training set carries its own piece of information about the target function. However, if some examples are corrupted with noise, the information they provide may be misleading. Even in cases where all the examples are noiseless, some examples may form patterns that are too complex for hypotheses in the learning model, and thus may still be detrimental to learning. The process of identifying and removing such outliers and too complex examples from the training set is called *data pruning*. In this section, we apply the data complexity for data pruning.

We have known that given the decomposition of a training set, it is straightforward to select a principal subset according to a complexity budget. However, we also know that, even with a computable practical complexity measure, it is usually prohibitively expensive to find the decomposition. Fortunately, there are ways to approximately identify some principal subsets with affordable computational requirements.

We first show that, with the ideal data complexity measures, outliers or too complex examples can be identified efficiently. Then we show that some more robust methods are needed for practical data complexity measures. Our methods involve a new concept of complexity contribution and a linear regression model for estimating the complexity contributions. The examples with high complexity contributions are deemed as noisy for learning.

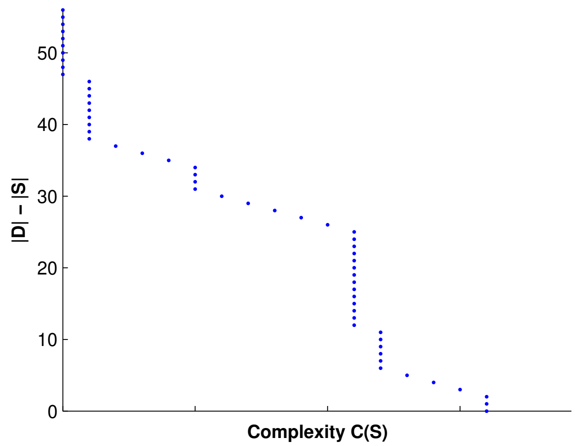


Figure 1.10: A fictional complexity-error path

1.5.1 Rightmost Segment

If we start from an empty set, and gradually grow the set by adding examples from a full set $\mathcal{D}$, we observe that the set becomes more and more complex, and reveals more and more details of $\mathcal{D}$, until finally its data complexity reaches $C(\mathcal{D})$. This leads to the definitions of subset paths and complexity-error paths:

Definition 1.5: *A subset path of set $\mathcal{D}$ is an ordered list of sets $(\mathcal{D}_0, \mathcal{D}_1, \ldots, \mathcal{D}_N)$ where $|\mathcal{D}_n| = n$ and $\mathcal{D}_n \subset \mathcal{D}_{n+1}$ for $0 \le n < N$, and $\mathcal{D}_N = \mathcal{D}$.*

Definition 1.6: *A complexity-error path of $\mathcal{D}$ is a set of pairs $\{(C(\mathcal{D}_n), |\mathcal{D}| - |\mathcal{D}_n|)\}$ where $0 \le n \le N$ and $(\mathcal{D}_0, \mathcal{D}_1, \ldots, \mathcal{D}_N)$ is a subset path of $\mathcal{D}$.*

Along a subset path, the data complexity increases and the approximation error decreases. So the complexity-error path, if plotted on a 2-D plane, would go down and right, like the one in Figure 1.10.

Visually, a complexity-error path consists of *segments* of different slopes. Say from $\mathcal{D}_n$ to $\mathcal{D}_{n+1}$, the newly added example can already be replicated by $h_{\mathcal{D}_n}$, one of the shortest hypotheses associated with $\mathcal{D}_n$. This means that no new patterns are necessary to accommodate the newly added example, and $C(\mathcal{D}_{n+1})$ is the same as $C(\mathcal{D}_n)$. Such a case is depicted as those vertical segments in Figure 1.10. If unfortunately that is not the case, a lookup table entry may be appended and the data complexity goes up by λ. This is shown as the segments of slope $-\lambda^{-1}$. It is also possible that some lookup table entries together can be covered by a pattern with reduced program length, or that the new example can be accounted for with patterns totally different from those associated with $\mathcal{D}_n$. For both cases, the data complexity increases by an amount less than λ.

A vertical segment is usually "good"—newly added examples agree with the current hypothesis. A segment of slope $-\lambda^{-1}$ is usually "bad"—newly added examples may be outliers according to the current subset since they can only be memorized.

The subset-based complexity-error plot can be regarded as the collection of all complexity-error paths of $\mathcal{D}$. The principal points comprise segments from probably more than one complexity-error path. Due to the mixing of complexity-error paths, a segment of slope $-\lambda^{-1}$ of the principal points may or may not imply that outliers are added. Fortunately, the *rightmost segment*, as defined below, can still help identify outliers.

Definition 1.7: *Given a data set $\mathcal{D}$, the rightmost segment of the principal points consists of all (c, e) such that (c, e) is a principal point and $c + \lambda e = C(\mathcal{D})$.*

In the following text, we will analyze the properties of the rightmost segment, and show how to use them to identify outliers.

Let's look at any shortest hypothesis for the training set $\mathcal{D}$, $h_{\mathcal{D}}$. Suppose $h_{\mathcal{D}}$ has lookup table entries for a subset $\mathcal{D}_{\mathrm{b}}$ of N_{b} examples, and the rest part of $h_{\mathcal{D}}$ can replicate $\mathcal{D}_{\mathrm{g}} = \mathcal{D} \backslash \mathcal{D}_{\mathrm{b}}$, the subset of $\mathcal{D}$ with just those N_{b} examples removed. Intuitively, examples in $\mathcal{D}_{\mathrm{b}}$ are outliers since they are merely memorized, and examples in the pruned set $\mathcal{D}_{\mathrm{g}}$ are mostly innocent. The question is, without knowing the shortest hypothesis $h_{\mathcal{D}}$, "can we find out the examples in $\mathcal{D}_{\mathrm{b}}$?"

Note that according to the structure of $h_{\mathcal{D}}$, we have $C(\mathcal{D}_{\mathrm{g}}) \le |h_{\mathcal{D}}| - \lambda N_{\mathrm{b}}$ and $C(\mathcal{D}) = |h_{\mathcal{D}}|$. Thus $\mathcal{D}_{\mathrm{g}}$ is on the rightmost segment, i.e., $C(\mathcal{D}) = C(\mathcal{D}_{\mathrm{g}} \cup \mathcal{D}_{\mathrm{b}}) = C(\mathcal{D}_{\mathrm{g}}) + \lambda N_{\mathrm{b}}$, and removing $\mathcal{D}_{\mathrm{b}}$ from $\mathcal{D}$ would reduce the data complexity by $\lambda\, |\mathcal{D}_{\mathrm{b}}|$. This is due to the Lemma 1.3 below. Furthermore, removing any subset $\mathcal{D}'_{\mathrm{b}} \subseteq \mathcal{D}_{\mathrm{b}}$ from $\mathcal{D}$ would reduce the data complexity by $\lambda\, |\mathcal{D}'_{\mathrm{b}}|$.

Lemma 1.3: *Assume* $C(\mathcal{D}_{\mathrm{g}} \cup \mathcal{D}_{\mathrm{b}}) \geq C(\mathcal{D}_{\mathrm{g}}) + \lambda |\mathcal{D}_{\mathrm{b}}|$. *We have for any* $\mathcal{D}'_{\mathrm{b}} \subseteq \mathcal{D}_{\mathrm{b}}$, $C(\mathcal{D}_{\mathrm{g}} \cup \mathcal{D}'_{\mathrm{b}}) = C(\mathcal{D}_{\mathrm{g}}) + \lambda |\mathcal{D}'_{\mathrm{b}}|$.

PROOF: From Theorem 1.3 on page 13, $C(\mathcal{D}_{\mathrm{g}} \cup \mathcal{D}'_{\mathrm{b}}) \leq C(\mathcal{D}_{\mathrm{g}}) + \lambda |\mathcal{D}'_{\mathrm{b}}|$, and

$$C(\mathcal{D}_{\mathrm{g}} \cup \mathcal{D}_{\mathrm{b}}) = C(\mathcal{D}_{\mathrm{g}} \cup \mathcal{D}'_{\mathrm{b}} \cup (\mathcal{D}_{\mathrm{b}} \backslash \mathcal{D}'_{\mathrm{b}})) \leq C(\mathcal{D}_{\mathrm{g}} \cup \mathcal{D}'_{\mathrm{b}}) + \lambda \left(|\mathcal{D}_{\mathrm{b}}| - |\mathcal{D}'_{\mathrm{b}}| \right).$$

With the assumption $C(\mathcal{D}_{\mathrm{g}} \cup \mathcal{D}_{\mathrm{b}}) \geq C(\mathcal{D}_{\mathrm{g}}) + \lambda |\mathcal{D}_{\mathrm{b}}|$, we have $C(\mathcal{D}_{\mathrm{g}} \cup \mathcal{D}'_{\mathrm{b}}) \geq C(\mathcal{D}_{\mathrm{g}}) + \lambda |\mathcal{D}'_{\mathrm{b}}|$.

Geometrically, the lemma says that $\mathcal{D} \backslash \mathcal{D}'_{\mathrm{b}}$ would always be on the rightmost segment if $\mathcal{D}'_{\mathrm{b}} \subseteq \mathcal{D}_{\mathrm{b}}$.

This also assures that removing any example $\mathbf{z} \in \mathcal{D}_{\mathrm{b}}$ from $\mathcal{D}$ would reduce the data complexity by λ. But the inverse is not always true. That is, given an example $\mathbf{z}$ such that $C(\mathcal{D} \backslash \{\mathbf{z}\}) = C(\mathcal{D}) - \lambda$, $\mathbf{z}$ might not be in $\mathcal{D}_{\mathrm{b}}$. This is because there might be several shortest hypotheses $h_{\mathcal{D}}$ with different lookup table entries and subsets $\mathcal{D}_{\mathrm{b}}$. An example in one such $\mathcal{D}_{\mathrm{b}}$ causes the data complexity to decrease by λ, but may not be in another $\mathcal{D}_{\mathrm{b}}$. With that said, we can still prove that an example that reduces the data complexity by λ can only be from $\mathcal{D}_{\mathrm{b}}$ associated with some $h_{\mathcal{D}}$.

Theorem 1.5: *Assume* $\mathcal{D} = \mathcal{D}_{\mathrm{g}} \cup \mathcal{D}_{\mathrm{b}}$, $\mathcal{D}_{\mathrm{g}} \cap \mathcal{D}_{\mathrm{b}} = \emptyset$, *the following propositions are equivalent:*

1. *There is a shortest hypothesis* $h_{\mathcal{D}}$ *for set* $\mathcal{D}$ *that has lookup table entries for examples in* $\mathcal{D}_{\mathrm{b}}$*;*

2. $C(\mathcal{D}) = C(\mathcal{D}_{\mathrm{g}}) + \lambda |\mathcal{D}_{\mathrm{b}}|$*;*

3. *For any subset* $\mathcal{D}'_{\mathrm{b}} \subseteq \mathcal{D}_{\mathrm{b}}$, $C(\mathcal{D}_{\mathrm{g}} \cup \mathcal{D}'_{\mathrm{b}}) = C(\mathcal{D}_{\mathrm{g}}) + \lambda |\mathcal{D}'_{\mathrm{b}}|$.

PROOF: We have seen $1 \Rightarrow 2$ and $2 \Rightarrow 3$. Here we prove $3 \Rightarrow 1$. Pick any shortest hypothesis $h_{\mathcal{D}_{\mathrm{g}}}$ for $\mathcal{D}_{\mathrm{g}}$. For any subset $\mathcal{D}'_{\mathrm{b}}$, add lookup table entries for examples in $\mathcal{D}'_{\mathrm{b}}$ to $h_{\mathcal{D}_{\mathrm{g}}}$ and we get $h_{\mathcal{D}_{\mathrm{g}} \cup \mathcal{D}'_{\mathrm{b}}}$, with $\left| h_{\mathcal{D}_{\mathrm{g}} \cup \mathcal{D}'_{\mathrm{b}}} \right| = \left| h_{\mathcal{D}_{\mathrm{g}}} \right| + \lambda |\mathcal{D}'_{\mathrm{b}}| = C(\mathcal{D}_{\mathrm{g}} \cup \mathcal{D}'_{\mathrm{b}})$. Thus $h_{\mathcal{D}_{\mathrm{g}} \cup \mathcal{D}'_{\mathrm{b}}}$ is a shortest hypothesis for $\mathcal{D}_{\mathrm{g}} \cup \mathcal{D}'_{\mathrm{b}}$. We let $\mathcal{D}'_{\mathrm{b}} = \mathcal{D}_{\mathrm{b}}$ to get the proposition 1.

Thus to identify $\mathcal{D}_{\mathrm{b}}$, we may try all subsets to see which satisfies proposition 2. Alternatively, we can also use a greedy method to remove examples from $\mathcal{D}$ as long as the reduction of the complexity is λ.

1.5.2 Complexity Contribution

From our analysis of the rightmost segment, removing an example from a training set would reduce the data complexity by some amount, and a large amount of complexity reduction (λ) implies that the removed example is an outlier. For convenience, define the *complexity contribution* of an examples as

Definition 1.8: *Given a data set $\mathcal{D}$ and an example $\mathbf{z} \in \mathcal{D}$, the complexity contribution of $\mathbf{z}$ to $\mathcal{D}$ is*

$$\gamma_{\mathcal{D}}(\mathbf{z}) = C(\mathcal{D}) - C(\mathcal{D} \setminus \{\mathbf{z}\}).$$

The greedy method introduced in the last subsection just repeatedly removes examples with complexity contribution equal λ.

However, this strategy does not work with practical data complexity measures. Usually, an approximation of the data complexity is based on a learning model and uses the descriptive length of a hypothesis learned from the training set. It is usually not minimal even within the learning model. In addition, the approximation is also noisy in the sense that data sets of similar data complexity may have quite different approximation values.

For the purpose of a more robust strategy for data pruning, we may look at the complexity contribution of an example to more than one data set. If most of the contributions are high, the example is likely to be an outlier; if most of the contributions are close to zero, the example is probably noiseless. Thus, for instance, we can use the average complexity contribution over different data sets as an indication of the outliers. In general, we can use a linear regression model for robustly estimating the complexity contributions.

Assume that for every training example $\mathbf{z}_n$ there is a real number $\tilde{\gamma}_n$ that is the expected complexity contribution of $\mathbf{z}_n$. To be more formal, we assume that, if a subset $\mathcal{S}$ of $\mathcal{D}$ has $\mathbf{z}_n \in \mathcal{S}$ and $s_1 < |\mathcal{S}| \le s_2$, we have

$$C(\mathcal{S}) - C(\mathcal{S} \setminus \{\mathbf{z}_n\}) = \tilde{\gamma}_n + \varepsilon,$$

where $0 \le s_1 < s_2 \le N$ are two size constants, and ε is a random variable with mean 0 representing the measure noise. In general, if $\mathcal{S}' \subset \mathcal{S} \subseteq \mathcal{D}$, $s_1 \le |\mathcal{S}'|$, and $|\mathcal{S}| \le s_2$, we assume

$$C(\mathcal{S}) - C(\mathcal{S}') = \sum_{\mathbf{z}_n \in \mathcal{S} \setminus \mathcal{S}'} \tilde{\gamma}_n + \varepsilon.$$

With this assumption, we can set up linear equations of $\tilde{\gamma}_n$'s with different pairs of subsets $\mathcal{S}$ and $\mathcal{S}'$. If we just pick subset pairs in random, we would roughly need two data complexity measurings for each linear equation. To save the number of data complexity measurings, we try to reuse the subsets for setting up the equations. One way is to construct equations with subset paths, as detailed below.

Denote $s = s_2 - s_1$. For an N-permutation $(i_1, i_2, \ldots, i_N)$, define $\mathcal{S}_n = \{\mathbf{z}_{i_1}, \mathbf{z}_{i_2}, \ldots, \mathbf{z}_{i_n}\}$ for $0 \le n \le N$, which form a subset path $(\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_N)$. After getting the data complexity values for the subsets $\mathcal{S}_{s_1}$ until $\mathcal{S}_{s_2}$, we construct s linear equations for $1 \le m \le s$,

$$C(\mathcal{S}_{s_1+m}) - C(\mathcal{S}_{s_1}) = \sum_{n=s_1+1}^{s_1+m} \tilde{\gamma}_{i_n} + \varepsilon.$$

Thus we only need $(s+1)$ data complexity measurings for s linear equations. And if the practical measure supports incremental measuring, such as the number of support vectors in an SVM (Subsection 1.3.4), we have extra computational savings. With many different N-permutations, we would have many such equations. Let's write all the equations in vector form

$$\boldsymbol{\Delta} = \mathbf{P}\tilde{\boldsymbol{\gamma}} + \boldsymbol{\varepsilon}, \tag{1.10}$$

where $\boldsymbol{\Delta}$ is a column vector of complexity difference between subsets, $\mathbf{P}$ is a matrix and each row of $\mathbf{P}$ is an indication vector of which examples causes the complexity difference, $\tilde{\boldsymbol{\gamma}}$ is a column vector $[\tilde{\gamma}_1, \tilde{\gamma}_2, \ldots, \tilde{\gamma}_N]^T$, and $\boldsymbol{\varepsilon}$ is also a column vector of corresponding noise. For instance, the vector form of the s linear equations constructed from the permutation $(1, 2, \ldots, N)$ is

$$\begin{bmatrix} C(\mathcal{S}_{s_1+1}) - C(\mathcal{S}_{s_1}) \\ C(\mathcal{S}_{s_1+2}) - C(\mathcal{S}_{s_1}) \\ \vdots \\ C(\mathcal{S}_{s_2}) - C(\mathcal{S}_{s_1}) \end{bmatrix} = \begin{bmatrix} \overbrace{0 \ldots 0}^{s_1 \text{ columns}} \, 1 \, 0 \, 0 \ldots 0 \, \overbrace{0 \ldots 0}^{(N-s_2) \text{ columns}} \\ 0 \ldots 0 \, 1 \, 1 \, 0 \ldots 0 \, 0 \ldots 0 \\ \vdots \\ 0 \ldots 0 \, 1 \, 1 \, 1 \ldots 1 \, 0 \ldots 0 \end{bmatrix} \begin{bmatrix} \tilde{\gamma}_1 \\ \tilde{\gamma}_2 \\ \vdots \\ \tilde{\gamma}_N \end{bmatrix} + \boldsymbol{\varepsilon}. \tag{1.11}$$

For a different permutation, the columns of the indication matrix shuffle according to the permutation, and the rest is pretty much the same. The actual vector $\mathbf{\Delta}$ and matrix $\mathbf{P}$ contain many block matrices from different permutations.

If we further assume some joint distribution of the measure noise ε, we may locate the optimal $\tilde{\gamma}_n$ for these equations. For example, if we assume the noise is normally distributed with $\mathbf{\Sigma} = \mathrm{E}\left[\boldsymbol{\varepsilon}\boldsymbol{\varepsilon}^T\right]$ as the covariance matrix, the best linear unbiased estimator for (1.10) is

$$\tilde{\boldsymbol{\gamma}} = \left(\mathbf{P}^T\mathbf{\Sigma}^{-1}\mathbf{P}\right)^{-1}\mathbf{P}^T\mathbf{\Sigma}^{-1}\mathbf{\Delta}. \tag{1.12}$$

Notice that for a specific covariance matrix, the estimate is as simple as the average complexity contribution:

Theorem 1.6: *Assume the measure noise is independent across different permutations, and has covariance matrix*

$$\mathbf{\Sigma}_1 = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & 2 & \cdots & 2 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 2 & \cdots & s \end{bmatrix} \sigma^2$$

within equations of a same permutation. Thus the covariance matrix $\mathbf{\Sigma}$ for all the equations would be a block diagonal matrix with diagonal blocks being $\mathbf{\Sigma}_1$. The best linear unbiased estimator (1.12) is actually

$$\tilde{\gamma}_n = \mathrm{E}\left[C(\mathcal{S}_{i+1}) - C(\mathcal{S}_i) \mid s_1 \le |\mathcal{S}_i| < s_2\right], \tag{1.13}$$

where, for a particular permutation, $\mathcal{S}_i$ is the largest in the subset path generated by the permutation that does not include $\mathbf{z}_n$, and the expectation is over all the permutations used in constructing the equations such that $\mathcal{S}_i$ has the proper size.

Proof: Let's first focus on equations constructed from a same permutation. Without loss of generality, we take those in (1.11) for our proof. Define an s-by-s square matrix

$$\mathbf{A}_1 = \begin{bmatrix} 1 & & & \\ -1 & 1 & & \\ & \ddots & \ddots & \\ & & -1 & 1 \end{bmatrix} \sigma^{-1}.$$

Left-multiplying both sides of (1.11) with $\mathbf{A}_1$, we get

$$\begin{bmatrix} C(\mathcal{S}_{s_1+1}) - C(\mathcal{S}_{s_1}) \\ C(\mathcal{S}_{s_1+2}) - C(\mathcal{S}_{s_1+1}) \\ \vdots \\ C(\mathcal{S}_{s_2}) - C(\mathcal{S}_{s_2-1}) \end{bmatrix} = \begin{bmatrix} \overbrace{0 \dots 0}^{s_1 \text{ columns}} \, 1 \, 0 \, 0 \dots 0 \, \overbrace{0 \dots 0}^{(N-s_2) \text{ columns}} \\ 0 \dots 0 \, 0 \, 1 \, 0 \dots 0 \, 0 \dots 0 \\ \vdots \\ 0 \dots 0 \, 0 \, 0 \dots 0 \, 1 \, 0 \dots 0 \end{bmatrix} \begin{bmatrix} \tilde{\gamma}_1 \\ \tilde{\gamma}_2 \\ \vdots \\ \tilde{\gamma}_N \end{bmatrix} + \mathbf{A}_1 \boldsymbol{\varepsilon}.$$

It is easily verified that the noise covariance, $\mathbf{A}_1 \mathbf{\Sigma}_1 \mathbf{A}_1^T$, is now the identity matrix. The optimal solution for this permutation only would be $\tilde{\gamma}_{s_1+m} = C(\mathcal{S}_{s_1+m}) - C(\mathcal{S}_{s_1+m-1})$ for $1 \leq m \leq s$. Consider a block diagonal matrix $\mathbf{A}$ that has as many diagonal blocks as the permutations, and each diagonal block is $\mathbf{A}_1$. Left-multiplying both sides of (1.10) with $\mathbf{A}$ transforms all the equations into some form of

$$C(\mathcal{S}_{i+1}) - C(\mathcal{S}_i) = \tilde{\gamma}_n + \varepsilon',$$

with $\mathcal{S}_{i+1} = \mathcal{S}_i \cup \{\mathbf{z}_n\}$ and the noise covariance matrix being the identity matrix. Thus the optimal linear solution would be $\tilde{\gamma}_n$ equal the average of such complexity contributions. Since our construction only allows $\mathcal{S}_i$ to have a size between s_1 and s_2, we have the estimator (1.13).

Such assumption about the noise covariance matrix is somewhere between a uniform assumption and a fully-correlated assumption. The uniform assumption regards the noise in each equation as independent and of the same magnitude. The fully-correlated assumption fine-tunes the model to assume that, associated with each $\tilde{\gamma}_n$, there is a random noise variable with mean 0 and variance σ^2, and the noise of the equation is the sum of the random noise variables associated with $\tilde{\gamma}_n$'s in the equation. The noise covariance of two equations would then be proportional to the number of common $\tilde{\gamma}_n$'s in these two equations. That is, $\mathbf{\Sigma} = \mathbf{P}\mathbf{P}^T\sigma^2$. However, since usually there are more equations than unknown variables, $\mathbf{\Sigma}$ is singular, which gives trouble in solving the equations via (1.12). Although the actual noise covariance would depend many factors including the practical complexity measure, it happens that the assumption that leads to average complexity contribution (1.13) works well in practice, as we will see in the next subsection.

1.5.3 Experiments

We test with the Yin-Yang target the concept of complexity contribution and the methods for estimating the complexity contribution. The experimental settings are similar to those used by Li et al. (2005). That is, a data set of size 400 is randomly generated, and the outputs of 40 examples (the last 10% indices) are further flipped as injected outliers.

We first verify that outliers would have higher complexity contributions on average than noiseless examples. To do so, we pick a random subset path and compute the complexity increase along the path. This is repeated many times and the complexity increase is averaged. Figure 1.11 shows such average complexity contribution of noisy and noiseless examples versus the subset size. Here are some observations:

- Overall, the 40 noisy examples have apparently much higher average complexity contribution than the 360 noiseless ones.
- When the subset size is really small (≤ 6), the noisy and the noiseless examples are indistinguishable with respect to complexity contribution. There has to be

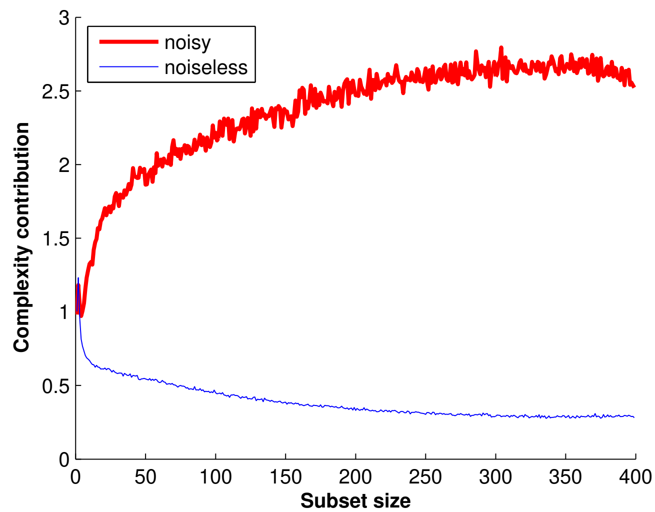


Figure 1.11: Average complexity contribution of the noisy and the noiseless examples in the Yin-Yang data

more information about the target in order to tell which examples are noisy and which are not.

- The average contribution of the noiseless examples becomes smaller when the subset gets larger. This is because when the subset has more details about the target function, newly added noiseless examples would have less chance to increase the data complexity.

- The average contribution of the noisy examples becomes larger when the subset gets larger, but it also seems converging to some value around 2.5. The reason of the contribution increase is related to that of the contribution decrease of the noiseless examples. When there is more correct information about the target function, an outlier would cause more complexity increase than it would when there is less information.

- The two contribution curves have noticeable bends at the right ends. These are artifacts due to the lack of distinct subsets when the subset size is close to the full size.

One way to use the complexity contribution for data pruning is to set a threshold θ and claim any example $\mathbf{z}_n$ noisy if $\tilde{\gamma}_n \geq \theta$. For the uniform variance assumption and

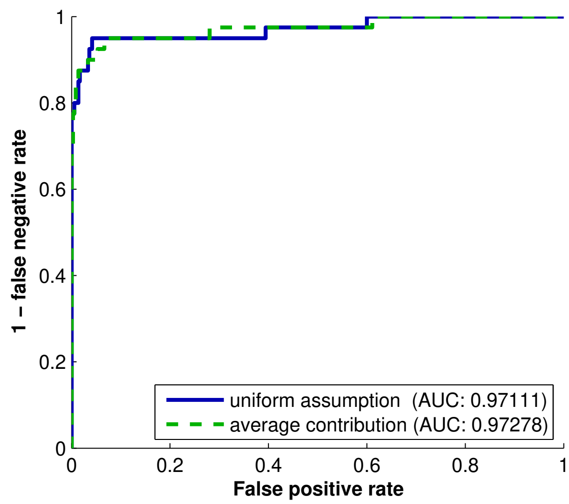


Figure 1.12: ROC curves of two estimators on the Yin-Yang data

the assumption leading to the average complexity contribution, which were discussed in Subsection 1.5.2, we set $s_1 = 50$, $s_2 = 400$, and solve equations created from 200,000 random permutations. We plot their receiver operating characteristic (ROC) in Figure 1.12. The ROC summarizes how the false negative rate (portion of noisy examples being claimed as noiseless) changes with the false positive rate (portion of noiseless examples being claimed as noisy) as the threshold θ varies. Both methods achieve large area under the ROC curve (AUC), a criterion to compare different ROC curves.

Similar to the data categorization proposed by Li et al. (2005), we also group all the training examples into three categories: typical, critical, and noisy. With two ad hoc thresholds $\theta_1 = \frac{2}{3}$ and $\theta_2 = 1$, the typical examples are those with $\tilde{\gamma}_n < \theta_1$, and we hope they are actually noiseless and far from the class boundary; the noisy examples are those with $\tilde{\gamma}_n > \theta_2$, and we hope they are actually outliers; the critical examples are those have $\tilde{\gamma}_n$ between θ_1 and θ_2, and we hope they are close to the class boundary. Figure 1.13 is the fingerprint plot which visually shows the categorization. The examples are positioned according to their signed distance to the decision boundary on the vertical axis and their index n in the training set on the horizontal axis. Critical and noisy examples are shown as empty circles and filled squares, respectively. We can see that most of the outliers (last 10% of the

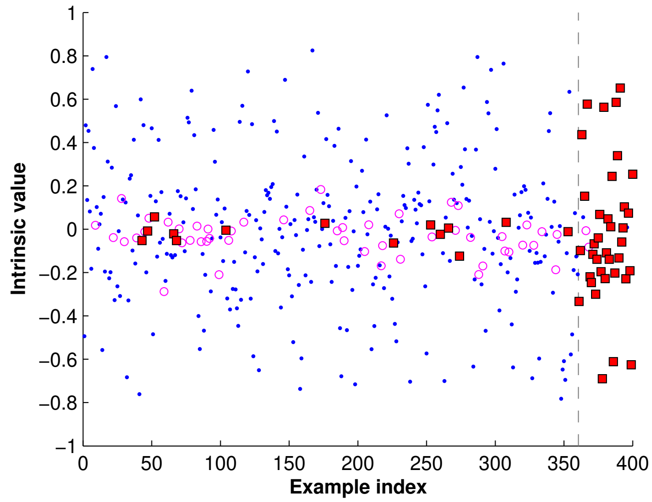


Figure 1.13: Fingerprint plot of the Yin-Yang data with the average contribution: · typical; ○ critical; ■ noisy

examples) are categorized as noisy, and some examples around the zero distance value are categorized as critical. We also have some imperfections—some of the critical examples are categorized as outliers and vice versa.

1.5.4 Discussion

With the estimated complexity contribution $\tilde{\gamma}_n$, we hope that a noiseless example would have a relatively low contribution and an outlier would have a relatively high contribution. However, whether an outlier can be distinguished using the complexity contribution heavily depends on the choice of the subsets used in the linear equations as well as many other factors.

For instance, the smaller Yin-Yang data set (Figure 1.7 on page 25) contains several examples in the small gray disk. It is reflected in the selected principal subsets (Figure 1.9 on page 27, also Figure 1.16 on page 47) that those examples are included in principal subsets of only relatively high data complexity ($\geq$ 35). Since those examples constitute a small percentage of the training set, a random subset of size smaller than, say, half of the full training set size, has a small probability to have most of those examples, and would usually only have patterns shown in principal subsets with data complexity lower than 35. Thus those examples would have large complexity contributions with high probability. For similar reasons, if some outliers happen to be in the small gray disk of the Yin-Yang target, with small subsets they may be regarded as innocent.

It is still unclear what conditions can assure that an outlier would have a high average complexity contribution.

1.5.5 Related Work

The complexity contribution quantifies to what degree an example may affect learning with respect to the data complexity. It is similar to the information gain concept behind informative examples used by Guyon et al. (1996) for outlier detection.

Some other outlier detection methods also exploit practical data complexity measures. Gamberger and Lavrač (1997) used a saturation test which is quite similar to our greedy method. Arning et al. (1996) looked for the greatest reduction in complexity by removing a subset of examples, which can be approximately explained by the proposition 2 in Theorem 1.5.

Statistics community has studied outlier detection extensively (Barnett and Lewis, 1994). They usually assume an underlying statistical model and define outliers based on discordance tests, many of which can be described as some simple distance-based check (Knorr and Ng, 1997).

Learning algorithms can also be used for data pruning. For example, Angelova et al. (2005) combined many classifiers with naive Bayes learning for identifying troublesome examples. Some learning algorithms, such as boosting, can produce information about the hardness of an example as a byproduct, which can be used for outlier detection (Merler et al., 2004; Li et al., 2005).

Angiulli et al. (2004) encoded background knowledge in the form of a first-order logic theory and outliers were defined as examples for which no logical justification can be found in the theory. They also showed that such outlier detection was intrinsically intractable due to its high computational complexity.

1.6 Conclusion

We have defined three ideal complexity measures for a data set. The universal data complexity is the length of the shortest program that can replicate the data set. The data complexity for a learning model finds a consistent hypothesis with the shortest encoding. And the data complexity with a lookup table also takes hypothesis errors into consideration. All these complexity measures are closely related to learning principles such as Occam's razor.

We have demonstrated the usage of the data complexity in two machine learning problems, data decomposition and data pruning. In data decomposition, we have illustrated that the principal subsets best approximate the full data set; in data pruning, we have proved that outliers are examples with high complexity contributions. We have also proposed and tested methods for estimating the complexity contribution.

Underneath the concept and the applications of the data complexity is the desire for generalization, the central issue of machine learning. Theoretically, if the correct prior and the exact noise model are known, we can encode the hypotheses and the errors in a way such that the shortest hypothesis generalizes the best. If the prior is unknown, the universal prior is a good guess for any computable priors, and the shortest hypothesis would still have high chance to generalize well. In practice, we also make a reasonable guess on the prior since practical approximations are usually designed with Occam's razor in mind.

Many approaches in this paper require intensive computational efforts, which is inevitable when the shortest hypothesis is sought. However, for practical applications, more computationally feasible solutions should be studied.

Acknowledgments

We wish to thank Xin Yu, Amrit Pratap, and Hsuan-Tien Lin for great suggestions and helpful comments. This work was partially supported by the Caltech SISL Graduate Fellowship.

1.A Principal Sets

Here we collect all the principal subsets of the concentric disks problem (Figure 1.14) and almost all the principal subsets of the Yin-Yang problem (Figures 1.15 and 1.16), which are described in Subsection 1.4.3. The corresponding complexity-error plots can be found in Figure 1.5 on page 24 and Figure 1.8 on page 26. For the Yin-Yang problem, each of the principal points $(11, 16)$, $(15, 12)$, and $(16, 11)$ has 4 or more associated principal subsets, but only two are shown for space reason. The number of support vectors in SVM-perceptron is the practical data complexity measure used.

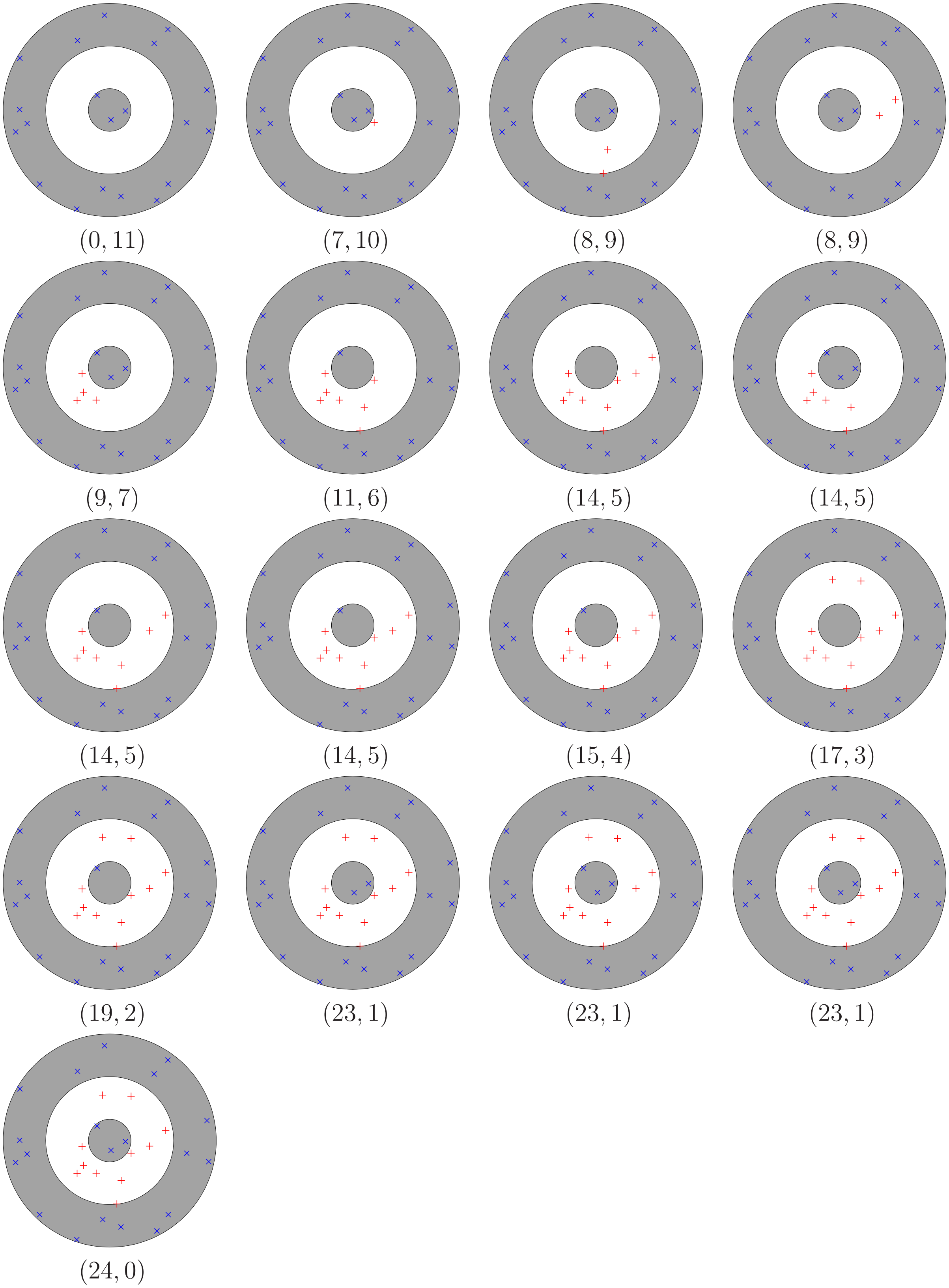


Figure 1.14: All principal subsets of the concentric disks problem (complexity-error pairs are also listed)

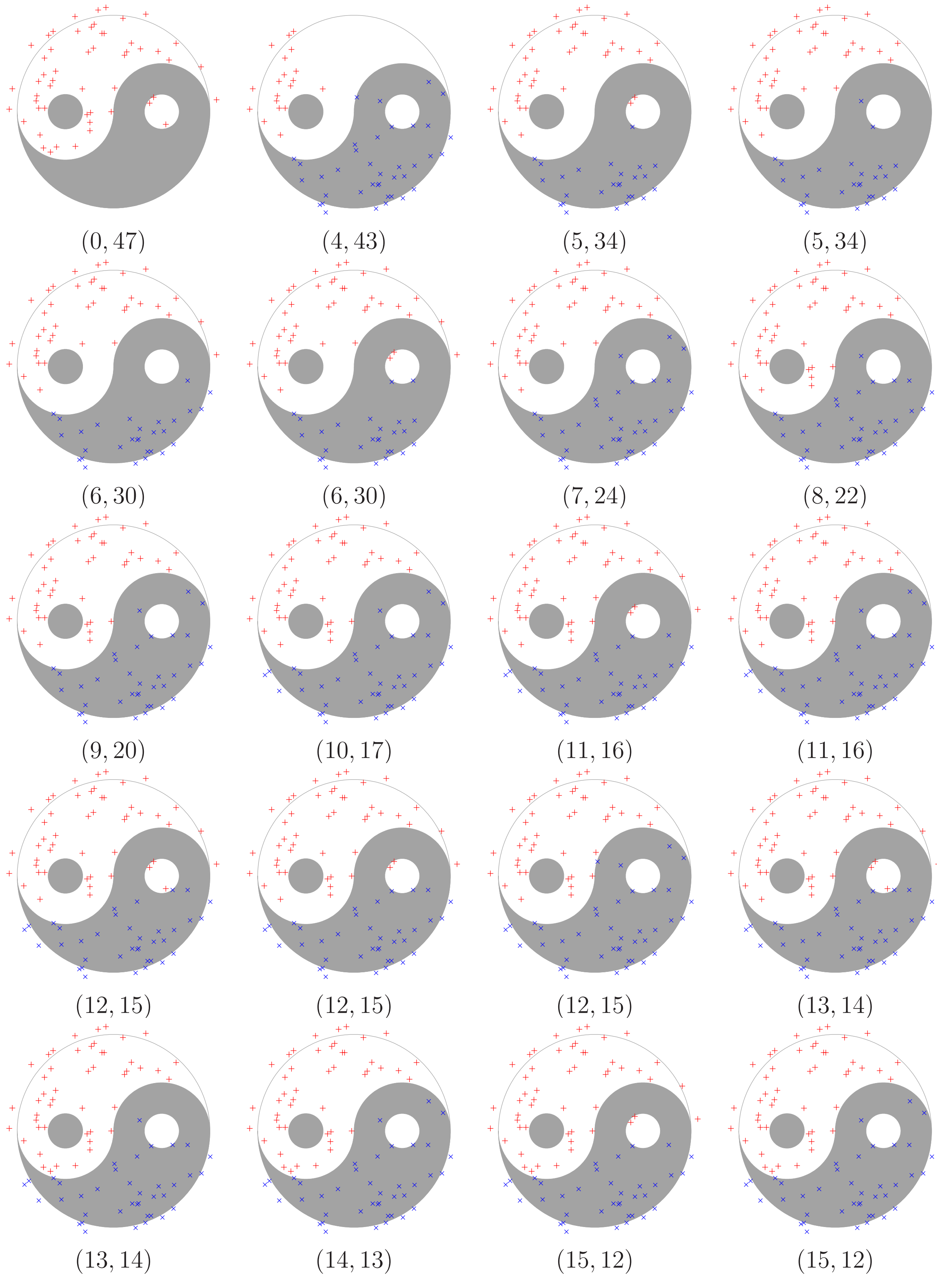


Figure 1.15: Principal subsets of the Yin-Yang data, part I

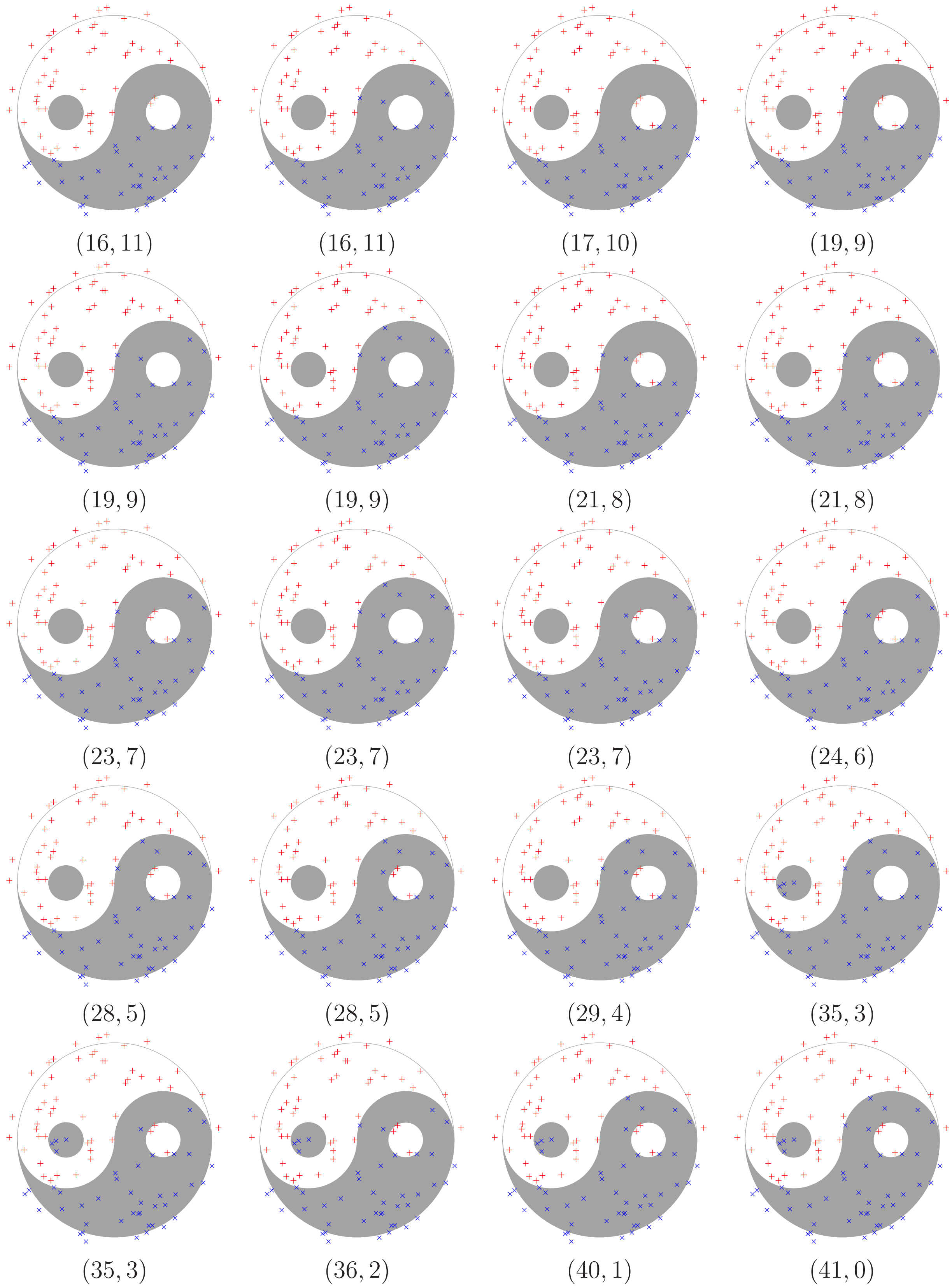


Figure 1.16: Principal subsets of the Yin-Yang data, part II

Chapter 2

Perceptron Learning with Random Coordinate Descent

A perceptron is a linear threshold classifier that separates examples with a hyperplane. It is perhaps the simplest learning model that is used standalone. In this paper, we propose a family of random coordinate descent algorithms for perceptron learning on binary classification problems. Unlike most perceptron learning algorithms, which require smooth cost functions, our algorithms directly minimize the training error, and usually achieve the lowest training error compared with other algorithms. The algorithms are also computationally efficient. Such advantages make them favorable for both standalone use and ensemble learning, on problems that are not linearly separable. Experiments show that our algorithms work very well with AdaBoost, and achieve the lowest test errors for half of the data sets.

2.1 Introduction

The perceptron was first introduced by Rosenblatt (1958) as a probabilistic model for information processing in the brain. Presented with an input vector $\mathbf{x}$, a perceptron calculates a weighted sum of $\mathbf{x}$, the inner product of $\mathbf{x}$ and its weight vector $\mathbf{w}$. If the sum is above some threshold, the perceptron outputs 1; otherwise it outputs -1.

Since a perceptron separates examples with a hyperplane in the input space, it is only capable of learning linearly separable problems.[1] For problems with more

[1] In this paper, phrases "linearly separable" and "separable" are interchangeable, and "nonsepa-

complex patterns, layers of perceptrons have to be connected to form an artificial neural network, and the back-propagation algorithm can be used for learning (Bishop, 1995).

If perfect learning is not required (i.e., nonzero training error is acceptable), the perceptron, as a standalone learning model, is actually quite useful. For instance, Shavlik et al. (1991) reported that the perceptron performed quite well under some qualifications, "hardly distinguishable from the more complicated learning algorithms" such as the feed-forward neural networks. Compared to another simple linear classifier, the decision stump (Holte, 1993), the perceptron is almost as fast to compute, but is more powerful in the sense that it can combine different input features.

Given a data set of examples labeled 1 or -1, the task of perceptron learning usually means finding a hyperplane that separates the examples of different labels with minimal error. When the data set is separable, the task is relatively easy and many algorithms can find the separating hyperplane. For example, the perceptron learning rule (Rosenblatt, 1962) is guaranteed to converge to a separating solution in a finite number of iterations. The support vector machine (SVM) can even find the optimal separating hyperplane that maximizes the minimal margin, by solving a quadratic programming problem (Vapnik, 1998).

However, these algorithms behave poorly when the data set is nonseparable, a more common situation in real-world problems. The perceptron learning rule will not converge, and is very unstable in the sense that the hyperplane might change from an optimal one to a worst-possible one in just one trial (Gallant, 1990). The quadratic programming problem of the hard-margin SVM is unsolvable; even if the soft-margin SVM is used, the solution may be heavily affected by examples that have the most negative margins, and may not be optimal for training error. It is also arguable which criterion, the margin or the training error, is more suitable for nonseparable problems.

There are many other perceptron learning algorithms, some of which will be introduced briefly in the next section. Although those algorithms appear quite different,

rable" means "not linearly separable."

they usually optimize some cost functions that are differentiable. The training error, although a very simple cost function, has never been minimized directly by those algorithms.

In this paper, we introduce a family of new perceptron learning algorithms that directly minimizes the training error. The essential idea is random coordinate descent, i.e., iteratively optimizing the cost function along randomly picked descent directions. An efficient update procedure is used to exactly minimize the training error along the picked direction. Both the randomness in the direction picking and the exact minimization of the training error help escape from local minima, and thus our algorithms usually achieve the best training error compared with other perceptron learning algorithms.

Although many real-world data sets are simple (Holte, 1993), it is by no means true that a single perceptron is complex enough for all problems. Sometimes more sophisticated learning models are required, and they may be constructed based on perceptrons. For example, the kernel trick used in SVM (Vapnik, 1998) allows the input features to be mapped into some high-dimensional space and a perceptron to be learned there. Another approach is to aggregate many perceptrons together to form a voted ensemble. Our algorithms can work with the kernel trick, but this will be the topic of another paper. In this paper, we explore AdaBoost (Freund and Schapire, 1996) to construct ensembles of perceptrons. We will show that our algorithms, unlike many other algorithms that are not good at reducing the training error, work very well with AdaBoost.

The paper is organized as follows: Some of the existing perceptron learning algorithms are briefly discussed in Section 2.2. Our random coordinate descent algorithms will be introduced in Section 2.3. We throughly compare our algorithms with several other perceptron learning algorithms in Section 2.4, either as standalone learners, or working with AdaBoost. We then conclude in Section 2.5.

2.2 Related Work

We assume that the input space is a subset of $\mathbb{R}^m$. A perceptron has a weight vector $\mathbf{w}$ and a bias term b (i.e., the negative threshold). For simplicity, we use the notations $\mathbf{w} = (w_0, w_1, \ldots, w_m)$ and $w_0 = b$ to avoid treating $\mathbf{w}$ and b separately. Each input vector $\mathbf{x}$ is also a real-valued vector in $\mathbb{R}^{m+1}$, with $x_0 = 1$. The perceptron labels the input vector $\mathbf{x}$ by computing the inner product between $\mathbf{w}$ and $\mathbf{x}$,

$$g(\mathbf{x}) = \operatorname{sign}\left(\langle \mathbf{w}, \mathbf{x} \rangle\right).$$

Given a training set $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$ where $y_i \in \{-1, 1\}$ is the class label, the perceptron learning rule proposed by Rosenblatt (1962) updates the perceptron weight vector when a classification error happens. That is, for an example $(\mathbf{x}, y)$, $\mathbf{w}$ is updated if $g(\mathbf{x}) \neq y$,

$$\mathbf{w}^{\text{updated}} = \mathbf{w} + y\mathbf{x}. \tag{2.1}$$

This learning rule is applied repeatedly to examples in the training set. If the training set is linearly separable, the perceptron convergence theorem (Rosenblatt, 1962) guarantees that a zero-error weight vector can be found in a finite number of update steps. However, if the training set is nonseparable, the algorithm will never converge and there is no guarantee that the weight vector obtained after any arbitrary number of steps can generalize well.

The pocket algorithm with ratchet (Gallant, 1990) (Algorithm 2.1) solves the stability problem of perceptron learning at the cost of more computational effort. It runs the learning rule (2.1) while keeping "in its pocket" an extra weight vector, which is the best-till-now solution. Whenever the perceptron weight vector is better than the pocket weight vector, the perceptron one replaces the pocket one. The ratchet check (step 9 in Algorithm 2.1) ensures that the training error of the pocket weight vector will only strictly decrease. Although the pocket algorithm can find an optimal weight vector that minimizes the training error with arbitrarily high probability, in practice, the number of trials required to produce an optimal weight vector is

Algorithm 2.1: The pocket algorithm with ratchet (Gallant, 1990) (note that the training error calculation, step 8, may be skipped when the weight has not changed)

Input: A training set $\{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$; Number of epochs T.

1: Initialize $\mathbf{w}$ {usually this means setting $\mathbf{w} \leftarrow \mathbf{0}$}
2: $\gamma \leftarrow 0$, $\mathbf{w}_{\mathrm{p}} \leftarrow \mathbf{w}$, $\gamma_{\mathrm{p}} \leftarrow 0$, $e_{\mathrm{p}} \leftarrow 1$
3: **for** $T \times N$ trials **do**
4:   Randomly pick an example $(\mathbf{x}_k, y_k)$
5:   **if** $y_k \langle \mathbf{w}, \mathbf{x}_k \rangle > 0$ **then** {$\mathbf{w}$ correctly classifies $(\mathbf{x}_k, y_k)$}
6:     $\gamma \leftarrow \gamma + 1$
7:     **if** $\gamma > \gamma_{\mathrm{p}}$ **then**
8:       $e \leftarrow \frac{1}{N} \sum_i [y_i \langle \mathbf{w}, \mathbf{x}_i \rangle \leq 0]$ {the training error of $\mathbf{w}$}
9:       **if** $e < e_{\mathrm{p}}$ **then**
10:         $\mathbf{w}_{\mathrm{p}} \leftarrow \mathbf{w}$, $e_{\mathrm{p}} \leftarrow e$, $\gamma_{\mathrm{p}} \leftarrow \gamma$
11:       **end if**
12:     **end if**
13:   **else** {$\mathbf{w}$ wrongly classifies $(\mathbf{x}_k, y_k)$}
14:     $\mathbf{w} \leftarrow \mathbf{w} + y_k \mathbf{x}_k$ {the Rosenblatt's update rule (2.1)}
15:     $\gamma \leftarrow 0$
16:   **end if**
17: **end for**
18: **return** $\mathbf{w}_{\mathrm{p}}$ as the perceptron weight vector

prohibitively large (Gallant, 1990).

In contrast with the pocket algorithm, which uses only the best weight vector, Freund and Schapire (1999) suggested combining all the weight vectors that occur in a normal perceptron learning by a majority vote. Each vector is weighted by its survival time, the number of trials before the vector is updated. Although this algorithm does not generate a linear classifier, one variant that uses averaging instead of voting—the averaged-perceptron algorithm (Algorithm 2.2)—does produce a linear classifier. Since their experiments showed that the voted- and averaged-perceptron algorithms had no significant difference in terms of performance, we will only consider the averaged-perceptron algorithm in this paper.

It is interesting to note that the perceptron learning rule (2.1) is actually the sequential gradient descent on a cost function known as the perceptron criterion,

$$C(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^{N} \max \{0, -y_i \langle \mathbf{w}, \mathbf{x}_i \rangle\} . \tag{2.2}$$

Algorithm 2.2: The averaged-perceptron algorithm (Freund and Schapire, 1999)

Input: A training set $\{(\mathbf{x}_i, y_i)\}$; Number of epochs T.

$t \leftarrow 1$, $\gamma_t \leftarrow 0$, initialize $\mathbf{w}_t$ {usually this means setting $\mathbf{w}_t \leftarrow \mathbf{0}$}
for T epochs **do**
    for $k = 1$ to N **do**
        if $y_k \langle \mathbf{w}_t, \mathbf{x}_k \rangle > 0$ **then** {$\mathbf{w}_t$ correctly classifies $(\mathbf{x}_k, y_k)$}
            $\gamma_t \leftarrow \gamma_t + 1$
        else {$\mathbf{w}_t$ wrongly classifies $(\mathbf{x}_k, y_k)$}
            $t \leftarrow t + 1$
            $\mathbf{w}_t \leftarrow \mathbf{w}_{t-1} + y_k \mathbf{x}_k$ {the Rosenblatt's update rule (2.1)}
            $\gamma_t \leftarrow 1$
        end if
    end for
end for
return $\sum_{i=1}^{t} \gamma_i \mathbf{w}_i$ as the perceptron weight vector

The pocket algorithm aims at minimizing the training error, but adopts the gradient of the perceptron criterion for weight update, and thus is not efficient. Using the same update rule, the averaged-perceptron algorithm also tries to minimize the perceptron criterion, but is heavily regularized via averaging.

Besides the perceptron criterion, there are algorithms that adopt other cost functions, such as the sum-of-squares error (also called the least-squares error), and minimize them by stochastic gradient descent (Zhang, 2004). Most cost functions for binary classification problems can be expressed as the sample sum of the example margin cost. That is,

$$C(\mathbf{w}) = \sum_{i=1}^{N} \varphi_i c \left(y_i \langle \mathbf{w}, \mathbf{x}_i \rangle \right),$$

where φ_i is the sample weight for example $(\mathbf{x}_i, y_i)$, $y_i \langle \mathbf{w}, \mathbf{x}_i \rangle$ is the unnormalized margin of the example, and $c \colon \mathbb{R} \to \mathbb{R}^+$ is a margin cost function. Several margin cost functions are listed in Table 2.1. In order to apply gradient descent, the margin cost has to be differentiable. Thus gradient descent type algorithms cannot work on the training error cost function, where $c(\rho) = [\rho \le 0]$. Another problem with such approaches is that the optimization process usually sticks at some local minima, and cannot go close to the optimal solutions.

The minimal (normalized) margin, which is the minimal distance from the exam-

Table 2.1: Several cost functions in the form of $C(\mathbf{w}) = \sum_{i=1}^{N} \varphi_i c\left(y_i \langle \mathbf{w}, \mathbf{x}_i \rangle\right)$

cost function	$c(\rho)$
perceptron criterion	$\max\{0, -\rho\}$
SVM hinge loss	$\max\{0, 1-\rho\}$
least-squares error	$(1-\rho)^2$
modified least-squares	$(\max\{0, 1-\rho\})^2$
0/1 loss training error	$[\rho \leq 0]$

ples to the separating hyperplane, plays an important role in bounding the number of mistakes made by a normal perceptron learning (Freund and Schapire, 1999). Usually the larger the margin is, the smaller the bound is, and the better the perceptron generalizes. Thus many algorithms aim at maximizing the minimal margin. For example, SVM tries to minimize the magnitude of the weight vector $\|\mathbf{w}\|$ while keeping the unnormalized margin bounded from below (Vapnik, 1998). The averaged-perceptron may achieve a better margin distribution through averaging, similar to how AdaBoost improves the base learner (Schapire et al., 1998).

The relaxed online maximum margin algorithm (ROMMA) is another algorithm that approximately maximizes the margin (Li and Long, 2002). Each update of ROMMA tries to minimize $\|\mathbf{w}\|$ according to some relaxed constraints. When the data set is separable, a certain way of running ROMMA converges to the maximum margin solution. However, there is yet no theoretical analysis on the behavior of the algorithm when the data set is nonseparable.

It is arguable that the margin is the right criterion to optimize when the data set is nonseparable. Outliers, which usually have very negative margins, may heavily affect the solution if we insist on maximizing the minimal margin. The training error, on the contrary, suffers less from outliers since the error count is the same no matter how negative the margins are.

2.3 Random Coordinate Descent

There are two elements in the perceptron learning rule (2.1) that may be altered for possibly better learning. The first one is the descent direction, which is $y\mathbf{x}$ in (2.1), and the second is the descent step, which is always 1 in (2.1). If we replace them with a vector $\mathbf{d}$ and a scalar α, respectively, the learning rule becomes

$$\mathbf{w}^{\text{updated}} = \mathbf{w} + \alpha\mathbf{d}. \tag{2.3}$$

Different choices on $\mathbf{d}$ and α may lead to different perceptron learning rules. In this section, we propose a family of new algorithms with proper choices of $\mathbf{d}$ and α to directly minimize the training error.

2.3.1 Finding Optimal Descent Step

We will discuss how to choose the descent directions later in Subsection 2.3.2. For now, let us assume that a descent direction $\mathbf{d}$ has been picked. We will find the the descent step α to minimize the training error along the direction $\mathbf{d}$. That is, we need to solve this subproblem:

$$\min_{\alpha \in \mathbb{R}} e\left(g_{\mathbf{w}+\alpha\mathbf{d}}\right) = \sum_{i=1}^{N} \varphi_i \left[y_i \left\langle \mathbf{w} + \alpha\mathbf{d}, \mathbf{x}_i \right\rangle \leq 0 \right].$$

Let us first look at how the error on example $(\mathbf{x}, y)$ is decided for the weight vector $(\mathbf{w} + \alpha\mathbf{d})$. Denote $\langle \mathbf{d}, \mathbf{x} \rangle$ by δ.

- When $\delta \neq 0$,

  $$\langle \mathbf{w} + \alpha\mathbf{d}, \mathbf{x} \rangle = \langle \mathbf{w}, \mathbf{x} \rangle + \alpha\delta = \delta \left(\delta^{-1} \langle \mathbf{w}, \mathbf{x} \rangle + \alpha \right). \tag{2.4}$$

  Thus

  $$g_{\mathbf{w}+\alpha\mathbf{d}}(\mathbf{x}) = \operatorname{sign}(\delta) \cdot \operatorname{sign}\left(\delta^{-1} \langle \mathbf{w}, \mathbf{x} \rangle + \alpha \right).$$

  This means that the error of $g_{\mathbf{w}+\alpha\mathbf{d}}$ on example $(\mathbf{x}, y)$ is the same as the error of a 1-D linear threshold function with bias α on the example $\left(\delta^{-1} \langle \mathbf{w}, \mathbf{x} \rangle, y \operatorname{sign}(\delta)\right)$.

- When $\delta = 0$,

$$g_{\mathbf{w}+\alpha\mathbf{d}}(\mathbf{x}) = \operatorname{sign}(\langle \mathbf{w} + \alpha\mathbf{d}, \mathbf{x} \rangle) = \operatorname{sign}(\langle \mathbf{w}, \mathbf{x} \rangle).$$

  Thus the descent step α will not change the output on the input $\mathbf{x}$.

The 1-D linear threshold function is actually a decision stump, which has a deterministic and efficient learning algorithm that minimizes the training error (Holte, 1993). Hence, we can transform all training examples that have $\delta_i \neq 0$ with the mapping below,

$$(\mathbf{x}_i, y_i) \mapsto \left(\delta_i^{-1} \langle \mathbf{w}, \mathbf{x}_i \rangle, y_i \operatorname{sign}(\delta_i)\right), \tag{2.5}$$

and then apply the decision stump learning algorithm to the transformed data set to decide the optimal descent step α^*.

Since α is not restricted to positive numbers, the direction $\mathbf{d}$ is not required to be strictly descent. As an extreme example, using $-\mathbf{d}$ as the search direction in (2.5) will merely negate the transformed 1-D examples, and $-\alpha^*$ will then be returned by the decision stump learning algorithm.

Note that a decision stump can have positive or negative directions. That is, it can be $\operatorname{sign}(x + \alpha)$ or $-\operatorname{sign}(x + \alpha)$. Although we expect the learning algorithm to return a decision stump with positive direction, it is still possible that a negative-direction one will be found.[2] When this happens, the weight vector should be negated; the examples with $\delta_i = 0$ will also have different errors, and thus they cannot be ignored as what we just described. The full update procedure is described in Algorithm 2.3. The classification error on those examples with $\delta_i = 0$ is essential in deciding the optimal direction, acting as an error bias for the positive direction.

We also use a simplified procedure (Algorithm 2.4), considering only positive-direction decision stumps. Since the emergence of negative-direction decision stumps is really rare and usually happens at the beginning of the optimization, we choose the simplified one for our experiments.

[2]This usually happens when the initial weight vector has a training error larger than $\frac{1}{2}$.

Algorithm 2.3: The update procedure for random coordinate descent

Input: A training set $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$ and its sample weight $\{\varphi_i\}_{i=1}^N$; The current weight $\mathbf{w}$; A descent direction $\mathbf{d}$.

for $i = 1$ to N **do** {generate the 1-D data set}
  $\delta_i \leftarrow \langle \mathbf{d}, \mathbf{x}_i \rangle$
  if $\delta_i \neq 0$ **then**
    $x'_i \leftarrow \delta_i^{-1} \langle \mathbf{w}, \mathbf{x}_i \rangle$, $y'_i \leftarrow y_i \operatorname{sign}(\delta_i)$
  else
    $x'_i \leftarrow \infty$, $y'_i \leftarrow y_i \operatorname{sign}(\langle \mathbf{w}, \mathbf{x}_i \rangle)$ {set $\operatorname{sign}(0) = -1$ only here}
  end if
end for
Find the optimal decision stump for $\{(x'_i, y'_i)\}_{i=1}^N$ and $\{\varphi_i\}_{i=1}^N$,

$$(q^*, \alpha^*) = \arg \min_{q \in \{-1,+1\}, \alpha \in \mathbb{R}} \sum_i \varphi_i \left[y'_i \cdot q \cdot \operatorname{sign}(x'_i + \alpha) \leq 0 \right]$$

$\mathbf{w} \leftarrow \mathbf{w} + \alpha^* \mathbf{d}$
if $q^* = -1$ **then**
  $\mathbf{w} \leftarrow -\mathbf{w}$
end if

The computational complexity of both the update procedures is $O[mN + N \log N]$. The mapping (2.5) takes N inner product operations, which has complexity $O[mN]$. The decision stump learning requires to sort the transformed 1-D data set, and the complexity is $O[N \log N]$. Looking for the optimal bias is just an operation linear in N. Compared with the standard perceptron learning whose complexity is $O[mN]$ for every epoch (to examine the inner product with N examples), our update procedure is still very efficient, especially when the number of examples is comparable to 2^m.

2.3.2 Choosing Descent Directions

There are many ways to choose the descent directions.

Even if the cost function we are minimizing is the 0/1 loss training error, we can still adopt the gradient of the perceptron criterion as the descent direction. Actually, we may use the gradient of any reasonable smooth cost function as our descent direction.

Algorithm 2.4: The update procedure for random coordinate descent using a positive-direction decision stump

Input: A training set $\{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$ and its sample weight $\{\varphi_i\}_{i=1}^{N}$; The current weight $\mathbf{w}$; A search direction $\mathbf{d}$.

for $i = 1$ to N **do** {generate the 1-D data set}
  $\delta_i \leftarrow \langle \mathbf{d}, \mathbf{x}_i \rangle$
  if $\delta_i \neq 0$ **then**
    $x'_i \leftarrow \delta_i^{-1} \langle \mathbf{w}, \mathbf{x}_i \rangle$, $y'_i \leftarrow y_i \operatorname{sign}(\delta_i)$
  end if
end for
Find the optimal decision stump for $\{(x'_i, y'_i)\}$ and $\{\varphi_i\}$, only considering those with $\delta_i \neq 0$,

$$\alpha^* = \arg\min_{\alpha \in \mathbb{R}} \sum_{i:\, \delta_i \neq 0} \varphi_i \left[y'_i \cdot \operatorname{sign}(x'_i + \alpha) \leq 0 \right]$$

$\mathbf{w} \leftarrow \mathbf{w} + \alpha^* \mathbf{d}$

The cyclic coordinate descent (CCD), also known as the iterative coordinate descent, can be used when the cost function is not differentiable. It picks one coordinate at a time and changes the value of that coordinate in the weight vector. In other words, if we denote the i-th basis vector by $\mathbf{e}_i$, e.g., $\mathbf{e}_0 = (1, 0, \ldots, 0)^T$, CCD uses $\mathbf{e}_i$ as the descent direction.

However, except for the possible actual meanings that the original coordinates may have, there is nothing special about the original coordinate system—we can set up another coordinate system and do CCD there. That is, we can pick a random basis, which is a set of pairwise orthogonal vectors, and iteratively use each basis vector as the descent direction. In order to avoid local minima caused by a fixed coordinate system, a different random basis shall be put in use every once in a while. Another more radical and more generalized choice is to every time pick a new random vector as the descent vector, as summarized in Algorithm 2.5, the *random coordinate descent* (RCD) algorithm.

We have investigated two general ways of picking random vectors. The first one, which we refer to as the *uniform random vectors*, picks each component of the vector from a uniform distribution spanned over the corresponding feature range. If the input features of the examples have been normalized to $[-1, 1]$ (see Section 2.4 for

Algorithm 2.5: Random coordinate descent algorithm for perceptrons

Input: A training set $\{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$ and its sample weight $\{\varphi_i\}_{i=1}^{N}$; Number of epochs T.

```
Initialize w
for T epochs do
    Generate a random vector d ∈ ℝ^{m+1} as the descent coordinate
    Do the weight update procedure with {(x_i, y_i)}_{i=1}^N, {φ_i}_{i=1}^N, w, and d
end for
return w as the perceptron weight
```

more details), each component is a random number uniformly in $[-1, 1]$. The other one uses Gaussian distribution instead of the uniform distribution, and is named *Gaussian random vectors*. If the features have been normalized to have zero mean and unit variance, each component is then picked from a unit Gaussian distribution. This approach has the nice property that the angle of the random vectors is uniformly distributed.

2.3.3 Variants of RCD

We can get different variants of RCD by using different schedules of random descent directions. For example, if $\mathbf{e}_i$ $(i = 0, \ldots, m)$ is iteratively picked as the descent direction, we get CCD.

When a random basis of $(m + 1)$ pairwise orthogonal vectors is used for every $(m + 1)$ epochs, we refer to it as RCD-conj. RCD-grad is RCD with the gradient of the perceptron criterion.

One thing we have noticed is that the range of the bias, w_0, can be quite different from those of the other components of $\mathbf{w}$. Thus it might be necessary to have a descent direction devoted to adjusting w_0 only. If the vector $\mathbf{e}_0$ is adopted every $(m + 1)$ epochs in addition to other settings, RCD becomes RCD-bias, and RCD-conj becomes RCD-conj-bias.

2.4 Experiments

We compare our RCD algorithms with several existing perceptron learning algorithms, as both standalone learners and base learners for AdaBoost. Experiments are carried out on nine real-world data sets[3] from the UCI machine learning repository (Hettich et al., 1998), and three artificial data sets[4]. Each real-world data set is randomly shuffled and split with 80% of the data for training and the rest for testing. Each artificial data set has 5000 randomly generated examples, of which 600 are used for training. The perceptron algorithms are allowed to run $T = 2000$ epochs. This is repeated 500 times to get the mean and the standard error of the training and test errors.

Data Preprocessing. Solely based on the feature distribution in the training set, we shift and scale the features in the training set to $[-1, 1]$, and correspondingly normalize the test set.[5] Thus we use the uniform random vectors for RCD algorithms.

Initial Seeding. We initialize the perceptron weight vector with two possible vectors, the zero vector and the Fisher's linear discriminant (FLD, see for example (Bishop, 1995)). For the latter case, when the within-class covariance matrix estimate happens to be singular, we regularize it with a small eigenvalue shrinkage parameter of the value 10^{-10}, just large enough to permit numerically stable inversion (Friedman, 1999).

[3]They are australian (Statlog: Australian Credit Approval), breast (Wisconsin Breast Cancer), cleveland (Heart Disease), german (Statlog: German Credit), heart (Statlog: Heart Disease), ionosphere (Johns Hopkins University Ionosphere), pima (Pima Indians Diabetes), sonar (Sonar, Mines vs. Rocks), and votes84 (Congressional Voting Records), with incomplete records removed.

[4]They are ringnorm and threenorm (Breiman, 1996, 1998), and yinyang (Li et al., 2005, Yin-Yang).

[5]Note that a common practice is to normalize based on all the examples, with the benefit of doing it only once before the data splitting. However, since our RCD algorithms are affected by the range of the random descent directions, even this "tiny" peek into the test set will give our algorithms an unfair edge.

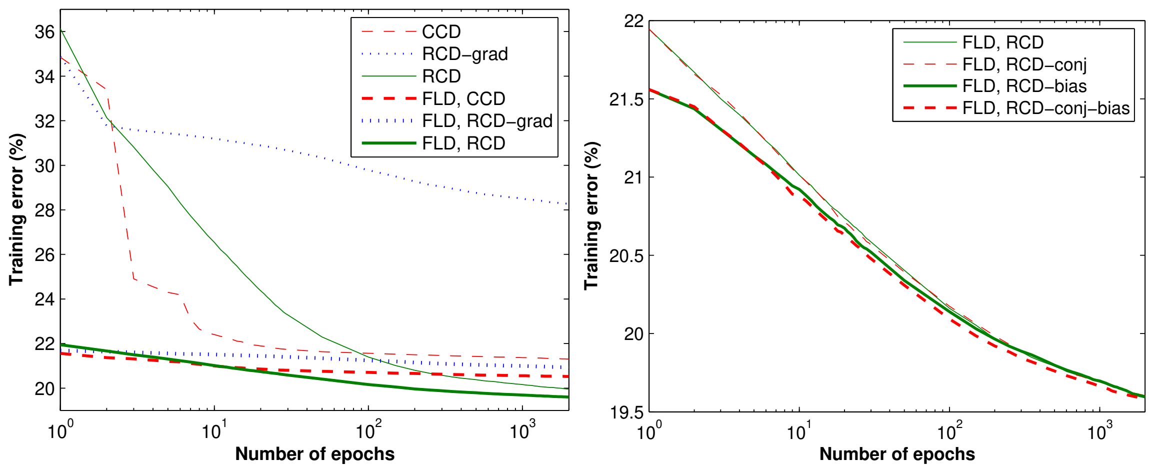


Figure 2.1: Training errors of several RCD algorithms on the pima data set

2.4.1 Comparing Variants of RCD

We first look at the in-sample performance of our RCD algorithms. Figure 2.1 shows, for the pima data set,[6] the training errors for several RCD algorithms. We can see that

- With FLD as a much better initial weight vector, the RCD algorithms achieve final training errors significantly lower than those obtained from the zero starting vector.
- RCD-grad does not work as well as other RCD algorithms. Apparently this is because the descent direction it uses is the gradient of the perceptron criterion, but the optimization is for the training error.
- Randomness in the direction picking is important. Even without FLD, RCD surpasses CCD with FLD in the end.
- Whether to use groups of orthogonal directions seems not affecting the performance significantly.
- The bias direction $\mathbf{e}_0$ does yield a better optimization, especially at the beginning. However, the edge gets smaller with more training epochs.

[6] Most plots in this paper are based on results on the pima data set. However, there is nothing special about pima. It is just a data set picked for illustration purposes.

Thus for clearer comparison with other perceptron learning algorithms, we shall focus on RCD and RCD-bias.

2.4.2 Comparing with Other Algorithms

We compare our RCD algorithms with several other perceptron algorithms, including the pocket algorithm with ratchet (pocket) (Gallant, 1990), averaged-perceptron (ave-perc) (Freund and Schapire, 1999), stochastic gradient descent with a learning rate 0.002 on the SVM hinge loss (SGD-hinge) and that on the modified least-squares (SGD-mls) (Zhang, 2004), and the soft-margin SVM with the linear kernel and parameter selection (soft-SVM) (Chang and Lin, 2001; Hsu et al., 2003).

It should be mentioned that when Freund and Schapire (1999) proposed the voted-perceptron and averaged-perceptron algorithms, they did not pay much attention to how the examples should be presented in multi-epoch runs, since their theoretical result on the error bound is only applicable to one-epoch run of the voted-perceptron. We find that cycling through examples with a fixed order[7] is not optimal for multi-epoch runs of the averaged-perceptron. Randomly permuting the training set at the beginning of each epoch or simply choosing examples at random at each trial can improve both the in-sample and the out-of-sample performance (see Figure 2.2 for a comparison on the pima data set). In our experiments, we use averaged-perceptron with the random sampling (see line 3 of Algorithm 2.6). Figure 2.2 also shows that using FLD only helps for early epochs.

ROMMA and aggressive ROMMA (Li and Long, 2002) perform miserably on most of the data sets we tried. The solution oscillates, especially when random sampling is used, and the training and test errors keep high. They also have numerical problems when running for more than several hundreds of epochs, even with the normalized data. We thus exclude them from further comparisons.

Figure 2.3 presents the performance of the selected algorithms on the pima data

[7]This is what was implied in (Freund and Schapire, 1999; Li and Long, 2002) although they did *preprocess* the training examples with a random permutation.

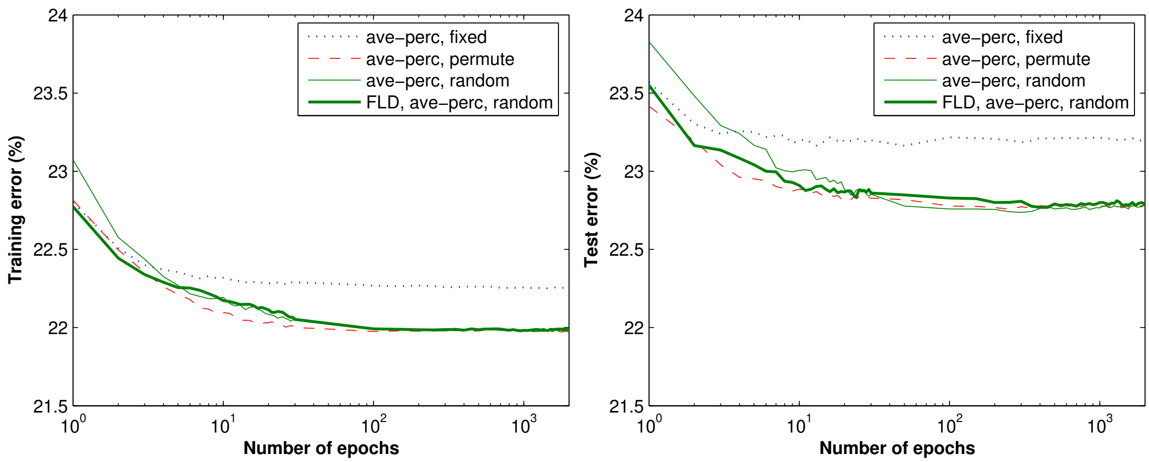


Figure 2.2: Training and test errors of the averaged-perceptron algorithm on the pima data set

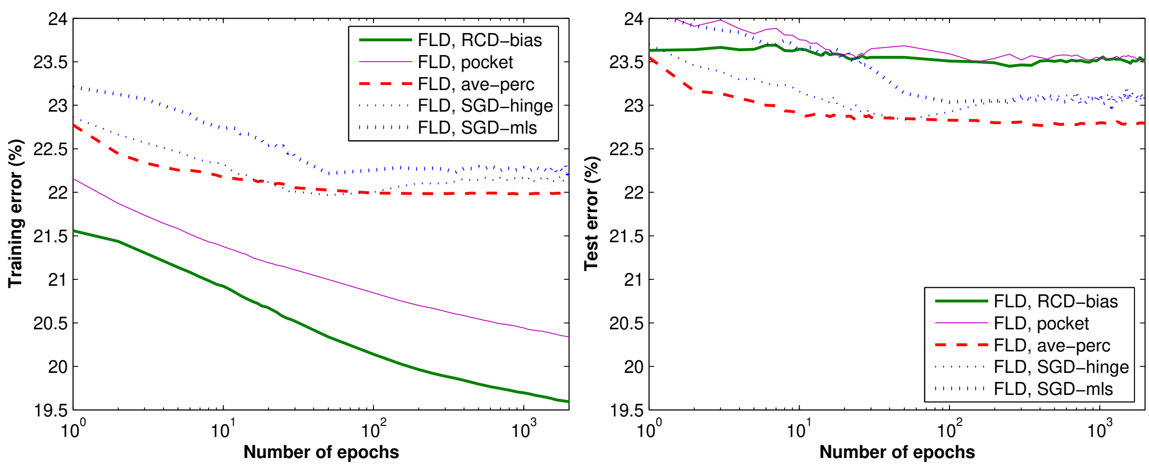


Figure 2.3: Training and test errors of several perceptron learning algorithms on the pima data set

set.[8] In the competition for low training errors, RCD-bias is clearly the best, and pocket follows. However, when the test error is concerned, the other three methods, especially ave-perc, are the winners. Tables 2.2 and 2.3 give the training and test errors on all the data sets at the end of the 2000 epochs. The errors of soft-SVM are also included. Again, we observe that RCD and RCD-bias achieve the lowest training errors for most data sets, but only achieve the lowest test errors for two artificial data sets, ringnorm and yinyang. The soft-SVM and ave-perc, both heavily regularized, overall achieve much better test errors. Since most real-world data sets may be

[8]We did not show the curves for RCD because they are very close to those of RCD-bias.

Table 2.2: Training errors (%) of several perceptron learning algorithms initialized with FLD

data set	RCD	RCD-bias	pocket	ave-perc	SGD-hinge	SGD-mls	soft-SVM
australian	10.12 ± 0.03	$\mathbf{9.98 \pm 0.03}$	10.81 ± 0.03	12.19 ± 0.03	14.11 ± 0.03	12.70 ± 0.04	14.33 ± 0.03
breast	$\mathbf{1.68 \pm 0.01}$	$\mathbf{1.68 \pm 0.01}$	1.86 ± 0.01	2.87 ± 0.02	2.66 ± 0.02	2.77 ± 0.02	2.70 ± 0.02
cleveland	$\mathbf{10.57 \pm 0.05}$	$\mathbf{10.62 \pm 0.05}$	12.07 ± 0.05	14.40 ± 0.06	14.31 ± 0.06	14.48 ± 0.06	14.74 ± 0.05
german	19.16 ± 0.04	$\mathbf{18.80 \pm 0.03}$	21.10 ± 0.03	21.31 ± 0.04	21.54 ± 0.04	22.18 ± 0.05	21.48 ± 0.04
heart	$\mathbf{9.48 \pm 0.05}$	$\mathbf{9.49 \pm 0.05}$	11.22 ± 0.05	13.64 ± 0.06	13.73 ± 0.06	13.82 ± 0.06	14.20 ± 0.06
ionosphere	3.88 ± 0.04	3.97 ± 0.04	$\mathbf{3.41 \pm 0.05}$	4.92 ± 0.06	4.55 ± 0.04	5.14 ± 0.05	6.95 ± 0.10
pima	$\mathbf{19.60 \pm 0.04}$	$\mathbf{19.60 \pm 0.03}$	20.34 ± 0.03	21.99 ± 0.04	22.15 ± 0.04	22.25 ± 0.04	22.09 ± 0.04
ringnorm	27.61 ± 0.07	$\mathbf{27.36 \pm 0.08}$	30.46 ± 0.07	35.49 ± 0.11	31.92 ± 0.09	34.52 ± 0.13	31.82 ± 0.09
sonar	2.56 ± 0.04	2.62 ± 0.04	$\mathbf{0.00 \pm 0.00}$	0.37 ± 0.02	2.23 ± 0.05	1.42 ± 0.06	11.58 ± 0.20
threenorm	$\mathbf{11.41 \pm 0.06}$	$\mathbf{11.39 \pm 0.06}$	13.53 ± 0.06	14.43 ± 0.06	14.23 ± 0.06	14.51 ± 0.06	14.47 ± 0.06
votes84	$\mathbf{1.32 \pm 0.02}$	$\mathbf{1.31 \pm 0.02}$	1.46 ± 0.02	2.42 ± 0.03	1.84 ± 0.03	2.48 ± 0.03	3.02 ± 0.04
yinyang	$\mathbf{15.33 \pm 0.05}$	$\mathbf{15.36 \pm 0.05}$	15.61 ± 0.05	19.10 ± 0.07	18.89 ± 0.08	19.03 ± 0.07	18.89 ± 0.08

(results within one standard error of the best are marked in bold)

Table 2.3: Test errors (%) of several perceptron learning algorithms initialized with FLD

data set	RCD	RCD-bias	pocket	ave-perc	SGD-hinge	SGD-mls	soft-SVM
australian	14.24 ± 0.12	13.92 ± 0.12	14.31 ± 0.12	$\mathbf{13.64 \pm 0.12}$	14.72 ± 0.12	13.87 ± 0.12	14.78 ± 0.12
breast	3.65 ± 0.07	3.61 ± 0.07	3.43 ± 0.06	3.36 ± 0.06	3.34 ± 0.06	$\mathbf{3.28 \pm 0.06}$	$\mathbf{3.22 \pm 0.06}$
cleveland	18.68 ± 0.22	18.57 ± 0.21	18.49 ± 0.21	$\mathbf{16.74 \pm 0.20}$	17.24 ± 0.20	$\mathbf{16.76 \pm 0.20}$	$\mathbf{16.72 \pm 0.20}$
german	24.45 ± 0.12	23.70 ± 0.13	25.24 ± 0.13	$\mathbf{23.24 \pm 0.12}$	23.66 ± 0.13	24.05 ± 0.13	23.64 ± 0.12
heart	18.13 ± 0.21	18.20 ± 0.22	17.63 ± 0.20	$\mathbf{16.51 \pm 0.20}$	16.70 ± 0.20	$\mathbf{16.49 \pm 0.20}$	$\mathbf{16.45 \pm 0.20}$
ionosphere	13.91 ± 0.17	14.72 ± 0.18	12.87 ± 0.18	12.76 ± 0.18	$\mathbf{12.45 \pm 0.17}$	12.63 ± 0.18	$\mathbf{12.57 \pm 0.17}$
pima	23.79 ± 0.14	23.50 ± 0.14	23.50 ± 0.14	$\mathbf{22.79 \pm 0.14}$	23.13 ± 0.13	23.07 ± 0.14	23.19 ± 0.14
ringnorm	35.83 ± 0.04	$\mathbf{35.65 \pm 0.04}$	36.59 ± 0.04	39.27 ± 0.08	36.01 ± 0.05	38.38 ± 0.10	35.70 ± 0.05
sonar	25.98 ± 0.29	26.20 ± 0.29	25.20 ± 0.25	25.09 ± 0.26	24.72 ± 0.28	24.90 ± 0.28	$\mathbf{23.89 \pm 0.27}$
threenorm	16.82 ± 0.03	16.86 ± 0.03	17.65 ± 0.04	16.14 ± 0.02	16.33 ± 0.02	16.18 ± 0.02	$\mathbf{16.08 \pm 0.02}$
votes84	5.21 ± 0.09	5.00 ± 0.10	5.24 ± 0.10	4.52 ± 0.10	5.17 ± 0.09	4.70 ± 0.11	$\mathbf{4.39 \pm 0.09}$
yinyang	$\mathbf{17.71 \pm 0.02}$	17.75 ± 0.02	17.74 ± 0.02	19.25 ± 0.02	19.12 ± 0.02	19.21 ± 0.02	19.21 ± 0.02

(results within one standard error of the best are marked in bold)

noisy or contain errors, overfitting might be the reason for the inferior out-of-sample performance of the RCD algorithms.

The two artificial data sets, ringnorm and yinyang, have quite different nature. The former is 20-dimensional and inherently noisy, and the latter is 2-dimensional and has clean boundaries. However, overfitting seems to be no problems for these two data sets. Figure 2.4 shows that, approximately, the lower the training error, the lower the test error. We are still unclear for what problems the perceptron model will induce no or very little overfitting.

We should also note that pocket is much slower than other algorithms such as ave-perc and RCD. This is because every time a new weight vector is considered

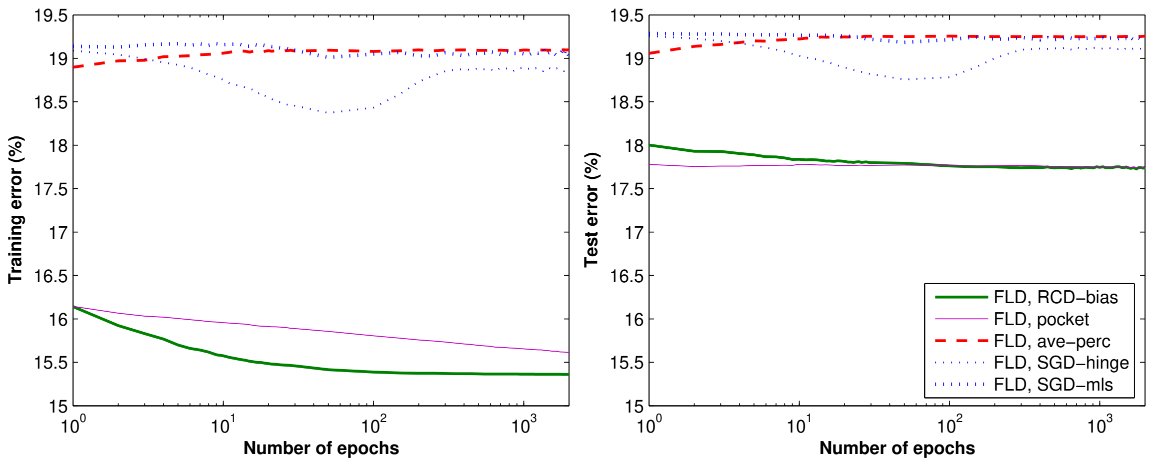


Figure 2.4: Training and test errors of several perceptron learning algorithms on the yinyang data set

for the "pocket," mN multiplications have to be done for computing the training error. Thus pocket may actually go over all the examples many times in one epoch, especially when the initial weight has good quality. For example, for the pima data set, the average number of training error computations is 7463.7 for 2000 epochs if initialized with the zero vector, and 33170.0 if initialized with FLD.

2.4.3 Ensembles of Perceptrons

AdaBoost (Freund and Schapire, 1996) is probably the most popular algorithm among the boosting family that generates a linear combination of base hypotheses. It improves the accuracy of the base learner by gradually focusing on "hard" examples. At each iteration, AdaBoost gives the base learner a set of sample weights, and asks for a hypothesis that has a low weighted training error. Thus in order to work with AdaBoost, a base learner should be able to take care of weighted data.

Our RCD algorithms are ideal for working with AdaBoost, since they are designed to directly minimize the weighted training error. For the other algorithms, small modifications are needed to accommodate weighted data.

Take pocket for example. Given a set of sample weights $\{\varphi_i\}_{i=1}^N$, we may modify line 4 of Algorithm 2.1 to "randomly pick an example $(\mathbf{x}_k, y_k)$ according to the distribution defined by $\{\varphi_i\}$," and replace line 8 with the weighted training error

Algorithm 2.6: The randomized averaged-perceptron algorithm with reweighting

Input: A training set $\{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$ and its sample weight $\{\varphi_i\}_{i=1}^{N}$; Number of epochs T.

1: $t \leftarrow 1$, $\gamma_t \leftarrow 0$, initialize $\mathbf{w}_t$
2: **for** $T \times N$ trials **do**
3:   Randomly pick an example $(\mathbf{x}_k, y_k)$ with uniform probability
4:   **if** $y_k \langle \mathbf{w}_t, \mathbf{x}_k \rangle > 0$ **then** $\{\mathbf{w}_t$ correctly classifies $(\mathbf{x}_k, y_k)\}$
5:     $\gamma_t \leftarrow \gamma_t + N\varphi_k$
6:   **else** $\{\mathbf{w}_t$ wrongly classifies $(\mathbf{x}_k, y_k)\}$
7:     $t \leftarrow t + 1$
8:     $\mathbf{w}_t \leftarrow \mathbf{w}_{t-1} + N\varphi_k y_k \mathbf{x}_k$ {using the sample weight}
9:     $\gamma_t \leftarrow N\varphi_k$
10:   **end if**
11: **end for**
12: **return** $\mathbf{w} = \sum_{i=1}^{t} \gamma_i \mathbf{w}_i$

$\sum_i \varphi_i \left[y_i \langle \mathbf{w}, \mathbf{x}_i \rangle \leq 0\right]$. We refer to this as *resampling*. Alternatively, we can keep picking examples with uniform probability, but modify quantities related to sample weights in a proper way. Here we change line 6 to "$\gamma \leftarrow \gamma + N\varphi_k$" and line 14 to "$\mathbf{w} \leftarrow \mathbf{w} + N\varphi_k y_k \mathbf{x}_k$." Of course we also modify line 8 as before. We refer to this as *reweighting*. The modified ave-perc with reweighting is shown in Algorithm 2.6. Note that the names reweighting and resampling have slightly different meanings from those by Freund and Schapire (1996).

Our experiments with AdaBoost (see Subsection 2.4.4 for settings) show that there is no significant difference between the resampling and reweighting methods. Since resampling usually requires $O[\log N]$ time to generate a random index according to the sample distribution, we prefer the reweighting method for its low computational overhead.

2.4.4 AdaBoost with Perceptrons

For the 12 data sets we use, 200 epochs seem sufficient for all perceptron learning algorithms to achieve a reasonable solution. Thus our base learners for AdaBoost are the perceptron learning algorithms with 200 epochs. We run AdaBoost up to 200 iterations. Often when the sample distribution becomes far away from the initial

Table 2.4: Test errors (%) and number of iterations (#ite) of AdaBoost (the #ite of the first three algorithms is 200)

data set	RCD	RCD-bias	pocket	ave-perc	#ite	SGD-hinge	#ite	SGD-mls	#ite
australian	15.45 ± 0.12	15.49 ± 0.12	15.75 ± 0.12	$\mathbf{13.61 \pm 0.12}$	6.4	15.97 ± 0.13	12.3	14.00 ± 0.12	8.8
breast	$\mathbf{3.21 \pm 0.06}$	3.34 ± 0.06	3.41 ± 0.07	3.35 ± 0.06	3.2	$\mathbf{3.27 \pm 0.06}$	7.4	$\mathbf{3.24 \pm 0.06}$	4.7
cleveland	18.00 ± 0.21	18.22 ± 0.21	18.95 ± 0.20	$\mathbf{16.81 \pm 0.20}$	3.3	17.16 ± 0.20	10.0	$\mathbf{16.74 \pm 0.20}$	5.9
german	25.17 ± 0.13	25.37 ± 0.12	25.57 ± 0.13	$\mathbf{23.25 \pm 0.12}$	2.9	23.71 ± 0.13	9.2	23.96 ± 0.13	7.5
heart	17.60 ± 0.21	17.58 ± 0.22	18.94 ± 0.21	$\mathbf{16.55 \pm 0.20}$	3.0	16.95 ± 0.21	10.7	$\mathbf{16.54 \pm 0.20}$	5.3
ionosphere	$\mathbf{10.36 \pm 0.16}$	$\mathbf{10.30 \pm 0.16}$	11.65 ± 0.17	13.21 ± 0.17	6.9	11.71 ± 0.17	25.3	12.67 ± 0.17	11.0
pima	24.87 ± 0.14	24.79 ± 0.14	25.15 ± 0.14	$\mathbf{22.77 \pm 0.14}$	3.0	23.18 ± 0.14	4.9	23.01 ± 0.14	4.5
ringnorm	8.60 ± 0.05	12.22 ± 0.07	$\mathbf{7.12 \pm 0.04}$	39.29 ± 0.08	2.3	27.41 ± 0.22	29.2	38.32 ± 0.09	2.1
sonar	16.44 ± 0.25	$\mathbf{16.06 \pm 0.25}$	25.02 ± 0.27	25.77 ± 0.27	199.7	21.23 ± 0.27	195.5	25.37 ± 0.27	146.7
threenorm	$\mathbf{14.51 \pm 0.02}$	15.34 ± 0.03	14.95 ± 0.02	16.14 ± 0.02	2.7	16.27 ± 0.02	5.2	16.17 ± 0.02	3.9
votes84	$\mathbf{4.25 \pm 0.09}$	$\mathbf{4.24 \pm 0.09}$	4.54 ± 0.10	4.74 ± 0.10	7.0	4.78 ± 0.10	32.4	4.68 ± 0.10	7.8
yinyang	$\mathbf{3.95 \pm 0.03}$	$\mathbf{3.98 \pm 0.03}$	4.87 ± 0.02	19.25 ± 0.02	2.5	19.11 ± 0.02	2.6	19.23 ± 0.02	2.7

(results within one standard error of the best are marked in bold)

uniform one, the base learner fails to find a perceptron with a small training error because the cost function it tries to minimize becomes so different from the training error. When this happens, AdaBoost stops at some iteration earlier than 200. We record the training error, test error, as well as the number of iterations, at the end of the AdaBoost run. The numbers are averaged over 500 random splits of the original data set.

We tried resampling and reweighting with pocket, ave-perc, SGD-hinge, and SGD-mls. There was no significant difference in the training error, test error, or the number of AdaBoost iterations. We also tested the two initialization methods for perceptrons, zero vector and FLD, and found that there was no decisive advantage in one or the other. So we only list the results of the simplest setting, reweighting and initialization with the zero vector, in Table 2.4.

First we notice that algorithms not aiming at minimizing the training error, ave-perc, SGD-hinge, and SGD-mls, do not really benefit from working with AdaBoost. Their numbers of iterations are usually small, and the test errors are similar to those listed in Table 2.3.

AdaBoost with our RCD algorithms and pocket never early stops before the specified 200 iterations. The resulted ensembles based on RCD and RCD-bias always achieve the zero training error, and those based on pocket also almost always get the zero training error. For about half of the data sets, they also achieve the lowest test errors.

2.5 Conclusion

We proposed a family of new perceptron learning algorithms that directly optimizes the training error. The main ingredients are random coordinate descent (RCD) and an update procedure to efficiently minimize the training error along the descent direction. We also discussed several possible approaches to initialize the algorithms and to choose the descent directions. Our experimental results showed that RCD algorithms were efficient, and usually achieved the lowest training errors compared with several other perceptron learning algorithms. This property also makes them ideal base learners for AdaBoost.

We discussed the resampling and reweighting approaches to making several other perceptron algorithms work with AdaBoost. However, most of them optimize cost functions other than the training error, and do not benefit from aggregating. In contrast, the test error may be dramatically decreased if RCD algorithms and the pocket-ratchet algorithm are used with AdaBoost.

For noisy and/or high-dimensional data sets, regularized algorithms such as the averaged-perceptron algorithm and the soft-margin SVM may achieve better out-of-sample performance. Future work will be focused on regularizing RCD algorithms.

Acknowledgments

I wish to thank Yaser Abu-Mostafa, Hsuan-Tien Lin, and Amrit Pratap for many valuable discussions. This work was supported by the Caltech SISL Graduate Fellowship.

Chapter 3

CGBoost: Conjugate Gradient in Function Space

The superior out-of-sample performance of AdaBoost has been attributed to the fact that it minimizes a cost function based on the margin, in that it can be viewed as a special case of AnyBoost, an abstract gradient descent algorithm. In this paper, we provide a more sophisticated abstract boosting algorithm, CGBoost, based on conjugate gradient in function space. When the AdaBoost exponential cost function is optimized, CGBoost generally yields much lower cost and training error but higher test error, which implies that the exponential cost is vulnerable to overfitting. With the optimization power of CGBoost, we can adopt more "regularized" cost functions that have better out-of-sample performance but are difficult to optimize. Our experiments demonstrate that CGBoost generally outperforms AnyBoost in cost reduction. With suitable cost functions, CGBoost can have better out-of-sample performance.

3.1 Introduction

AdaBoost (Freund and Schapire, 1996) is probably the most popular algorithm among the boosting family that generates a linear combination of weak hypotheses. Given a weak learner $\mathcal{L}$, AdaBoost iteratively adds hypotheses generated by $\mathcal{L}$ to the linear combination. It emphasizes difficult examples by giving them higher sample weights and favors hypotheses with lower training errors by giving them larger coefficients. AdaBoost can be viewed as a special case of AnyBoost (Mason et al., 2000b), a general gradient descent in function space.

It has been observed experimentally that AdaBoost keeps improving the out-of-sample error even after the training error of the linear combination has reached zero (Breiman, 1996). One explanation to this is that AdaBoost improves the margins of the training examples even after all the examples have positive margins, and larger margins imply better out-of-sample performance (Schapire et al., 1998). However, this explanation was challenged in (Grove and Schuurmans, 1998) where the algorithms achieve larger minimum margins than AdaBoost, but do not have better out-of-sample performance than AdaBoost, mostly worse. Another related explanation is that AdaBoost optimizes a cost function based on example margins (Mason et al., 2000b). Although there is a theoretical bound on the out-of-sample error based on cost, it is still unclear whether minimizing the cost is helpful in practice.

We take a closer look at this question, examining how the cost function, in and of itself, affects the out-of-sample performance. To do so, we apply more sophisticated optimization techniques directly to the cost function. We obtain three sets of results:

1. The introduction of a new abstract boosting algorithm, CGBoost, based on conjugate gradient in function space which has better cost optimization performance than AnyBoost.

2. The conclusion that AdaBoost cost function is much more vulnerable to overfitting when it is directly minimized instead of being minimized within the confines of the AdaBoost algorithm.

3. The identification of more "regularized" cost functions whose direct minimization results in a better out-of-sample performance than that of the AdaBoost cost function.

The paper is organized as follows: The CGBoost algorithm and its implementation with the margin cost functions are introduced in Section 3.2. In Section 3.3, we compare CGBoost and AnyBoost with two different cost functions. One cost function is observed to have better out-of-sample performance but is more difficult to optimize.

CGBoost has superior performance than AnyBoost with that cost function. We then give results on some UCI data sets in Section 3.4.

3.2 CGBoost

We assume the examples $(\mathbf{x}, y)$ are randomly generated according to some unknown probability distribution on $X \times Y$ where X is the input space and Y is the output space. Since this paper focuses on voted combinations of binary classifiers, $Y = \{-1, 1\}$.

The voted combination is $\operatorname{sign}(F(\mathbf{x}))$ where

$$F(\mathbf{x}) = \sum_{t=1}^{T} \alpha_t f_t(\mathbf{x})$$

with weak hypotheses $f_t \colon X \to Y$ from a base learning model $\mathcal{G}$, and hypothesis coefficients $\alpha_t \in \mathbb{R}^+$. Let $\operatorname{lin}(\mathcal{G})$ denote the set of all linear combinations (with nonnegative coefficients) of functions in $\mathcal{G}$, and $C \colon \operatorname{lin}(\mathcal{G}) \to \mathbb{R}^+$ be a cost function. We want to construct a combination $F \in \operatorname{lin}(\mathcal{G})$ to minimize $C(F)$.

3.2.1 AnyBoost: Gradient Descent

AnyBoost (Mason et al., 2000b) is an abstract boosting algorithm that provides a general framework for minimizing C iteratively via gradient descent in function space.

Suppose we have a function $F \in \operatorname{lin}(\mathcal{G})$ and we wish to find a "direction" $f \in \mathcal{G}$ so that the cost $C(F + \epsilon f)$ decreases for some small positive ϵ. The desired direction such that the cost decreases most rapidly (for small ϵ) is the negative functional gradient $-\nabla C(F)$, where

$$\nabla C(F)(\mathbf{x}) = \left. \frac{\partial C(F + \tau 1_{\mathbf{x}})}{\partial \tau} \right|_{\tau=0},$$

where $1_{\mathbf{x}}$ is the indicator function of $\mathbf{x}$. In general, it may not be possible to choose $f = -\nabla C(F)$ since f has to be one of the hypotheses in $\mathcal{G}$. So, instead, AnyBoost searches for f that maximizes $\langle -\nabla C(F), f\rangle$.[1] After f is fixed, a line search can be used to determine the coefficient of f in the new combination of hypotheses.

3.2.2 CGBoost: Conjugate Gradient

If we replace gradient descent in AnyBoost with the more efficient conjugate gradient technique (Nash and Sofer, 1996, §12.4), we obtain a new and more powerful abstract boosting algorithm: CGBoost (Algorithm 3.1). The main difference between conjugate gradient and gradient descent is that the former also utilizes the second-order information of the cost to adjust search directions so that the cost could be decreased faster.

Let d_t denote the search direction at iteration t, and $f_t \in \mathcal{G}$ denote the weak hypothesis approximating the negative functional gradient $-\nabla C(F)$. Instead of letting $d_t = f_t$ directly, we choose the search direction to be

$$d_t = f_t + \beta_t d_{t-1}, \tag{3.1}$$

where $\beta_t \in \mathbb{R}$ and d_{t-1} is the direction from last iteration. With this change, the search direction d_t is no longer limited to a single hypothesis in $\mathcal{G}$. Instead, it is some linear combination of the current and previous f_t's and thus $d_t \in \operatorname{lin}(\mathcal{G})$.

The β_t in equation (3.1) determines how much the previous search direction d_{t-1} affects the current direction d_t. If $\beta_t = 0$, d_t is solely determined by the current gradient f_t, which usually helps conjugate gradient recover from some bad situations (Nash and Sofer, 1996, pp. 408). In Algorithm 3.1, β_1 is effectively forced to be 0 since d_0 is initialized to 0. For reasons that will be explained in Section 3.3, β_t is also clipped to 0 for the first several iterations. For other cases, we can use the

[1] The inner product $\langle\cdot,\cdot\rangle$ is define on $\operatorname{lin}(\mathcal{G})$. Generally, we want $f \in \mathcal{G}$ to maximize the *normalized* inner product $\langle -\nabla C(F), f\rangle / \sqrt{\langle f, f\rangle}$. For the inner product definition (3.4) used in this paper and some other papers (Mason et al., 2000b), $\langle f, f\rangle$ is a constant for binary classifiers. So there is no need for normalization.

Algorithm 3.1: CGBoost: Conjugate gradient in function space

Require:

- A base learning model $\mathcal{G}$ and an inner product defined on $\mathrm{lin}(\mathcal{G})$
- A differentiable cost function $C\colon\ \mathrm{lin}(\mathcal{G}) \to \mathbb{R}^+$
- A weak learner $\mathcal{L}(F)$ that accepts $F \in \mathrm{lin}(\mathcal{G})$ and returns $f \in \mathcal{G}$ with a large value of $\langle -\nabla C\,(F)\,, f\rangle$

$F_0 \leftarrow 0$, $d_0 \leftarrow 0$
for $t = 1$ to T **do**
  $f_t \leftarrow \mathcal{L}(F_{t-1})$
  $d_t \leftarrow f_t + \beta_t d_{t-1}$ for some $\beta_t \in \mathbb{R}$
  if $\langle -\nabla C(F_{t-1}), d_t\rangle \leq 0$ **then**
    return F_{t-1}
  end if
  $F_t \leftarrow F_{t-1} + \alpha_t d_t$ for some $\alpha_t > 0$
end for
return F_T

Polak-Ribiére formula (Nash and Sofer, 1996, pp. 399)

$$\beta_t = \frac{\langle f_t, f_t - f_{t-1}\rangle}{\langle f_{t-1}, f_{t-1}\rangle}, \tag{3.2}$$

which automates the “restart” mechanism.

Although the search direction of CGBoost is more complicated than that of AnyBoost, the combination F_T is still a linear combination of (at most) T weak hypotheses in $\mathcal{G}$, since all d_t are in the space spanned by $\{f_1, \ldots, f_T\}$. For $i \leq t$, define

$$\beta_{i,t} = \begin{cases} \prod_{j=i+1}^{t} \beta_j, & \text{if } i < t; \\ 1, & \text{if } i = t. \end{cases}$$

We then have $d_t = \sum_{i=1}^{t} \beta_{i,t} f_i$, and

$$F_T = \sum_{t=1}^{T} \alpha_t d_t = \sum_{i=1}^{T} \left(\sum_{t=i}^{T} \alpha_t \beta_{i,t} \right) f_i. \tag{3.3}$$

If the base learning model $\mathcal{G}$ is negation closed, which is trivially true for almost all reasonable binary classification models, it is obvious that $F_T \in \mathrm{lin}(\mathcal{G})$. In the following subsection, we will see a definition for the inner product that also guarantees the coefficients in (3.3) are nonnegative. If it is the size of linear combinations that generally decides the complexity of the combination, these features imply that using conjugate gradient will not increase the complexity.

The search step α_t can be determined by some line search technique. If α_{t-1} ensures that $\nabla C(F_{t-1}) \perp d_{t-1}$ (which could be achieved by an *exact* line search), we have

$$\langle -\nabla C(F_{t-1}), d_t \rangle = \langle -\nabla C(F_{t-1}), f_t \rangle .$$

That is, the adjusted direction d_t is just as close to $-\nabla C(F_{t-1})$ as f_t is, while f_t is guaranteed by the weak learner $\mathcal{L}$ to be a good approximation of the negative gradient.

3.2.3 CGBoost with Margin Cost Functions

Commonly used cost functions are usually defined on example margins. Given a training set $S = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}$ of N examples, the margin cost of F has the form

$$C(F) = \frac{1}{N} \sum_{i=1}^{N} c(y_i F(\mathbf{x}_i)),$$

where $y_i F(\mathbf{x}_i)$ is the margin of example $(\mathbf{x}_i, y_i)$ and $c\colon \mathbb{R} \to \mathbb{R}^+$ is a (decreasing) function of the margin. We may thus use $c(\cdot)$ to refer the whole cost function C.

The inner product between hypotheses f and g is defined as

$$\langle f, g \rangle = \frac{1}{N} \sum_{i=1}^{N} f(\mathbf{x}_i) g(\mathbf{x}_i). \tag{3.4}$$

In this case,

$$\langle -\nabla C(F), f \rangle = \frac{1}{N^2} \sum_{i=1}^{N} y_i f(\mathbf{x}_i) \cdot \left[-c'(y_i F(\mathbf{x}_i)) \right] .$$

Maximizing $\langle -\nabla C(F), f \rangle$ is thus equivalent to minimizing the training error with sample weight $D(i) \propto -c'(y_i F(\mathbf{x}_i))$. This explains why a weak learner $\mathcal{L}$ is used in

Algorithm 3.2: CGBoost with margin cost functions

Require:

- A base learning model $\mathcal{G}$ containing hypotheses $f\colon X \to \{-1, 1\}$
- A differentiable cost function $c\colon \mathbb{R} \to \mathbb{R}^+$
- A weak learner $\mathcal{L}(S, D)$ that accepts a training set S and a sample distribution D and returns $f \in \mathcal{G}$ with small weighted error $\sum_i D(i)\,[f(\mathbf{x}_i) \neq y_i]$

$F_0 \leftarrow 0$, $d_0 \leftarrow 0$
for $t = 1$ to T **do**
  $D_t(i) \leftarrow c'(y_i F_{t-1}(\mathbf{x}_i)) / \sum_{j=1}^{N} c'(y_j F_{t-1}(\mathbf{x}_j))$ for $i = 1, \ldots, N$
  $f_t \leftarrow \mathcal{L}(S, D_t)$
  $d_t \leftarrow f_t + \beta_t d_{t-1}$ for $\beta_t = 1 - \langle f_{t-1}, f_t \rangle$
  if $\sum_{i=1}^{N} D_t(i) y_i d_t(\mathbf{x}_i) \leq 0$ **then**
    return F_{t-1}
  end if
  $F_t \leftarrow F_{t-1} + \alpha_t d_t$ for some $\alpha_t > 0$
end for
return F_T

Algorithm 3.1 to return f that has a large value of $\langle -\nabla C(F), f \rangle$.

For a binary classifier f, the definition in (3.4) gives $\langle f, f \rangle \equiv 1$. Then equation (3.2) becomes

$$\beta_t = 1 - \langle f_t, f_{t-1} \rangle \,,$$

which is always nonnegative.

Algorithm 3.2 summarizes the implementation of CGBoost with margin cost functions.

3.3 Cost Functions

The frameworks of CGBoost and AnyBoost leave the choice of cost functions open to users. However, not all the cost functions are suitable for learning purposes. We will discuss two cost functions in this section as well as the performance of CGBoost and AnyBoost on these cost functions.

3.3.1 AdaBoost Exponential Cost

When margin cost function $c(\rho) = e^{-\rho}$ is used, AnyBoost is equivalent to AdaBoost (Mason et al., 2000b). To have a taste of the performance of CGBoost, we compare CGBoost using this cost function with AdaBoost. Since the same cost function is used, this is a comparison between two optimization methods.

The data set used in this comparison was generated by the Caltech Data Engine.[2] The input space is $X = \mathbb{R}^8$ and output space is $Y = \{-1, 1\}$. We use 400 examples for training and 3000 examples for testing. Our learning model $\mathcal{G}$ contains decision stumps and the weak learner $\mathcal{L}$ returns the decision stump with best weighted training error.

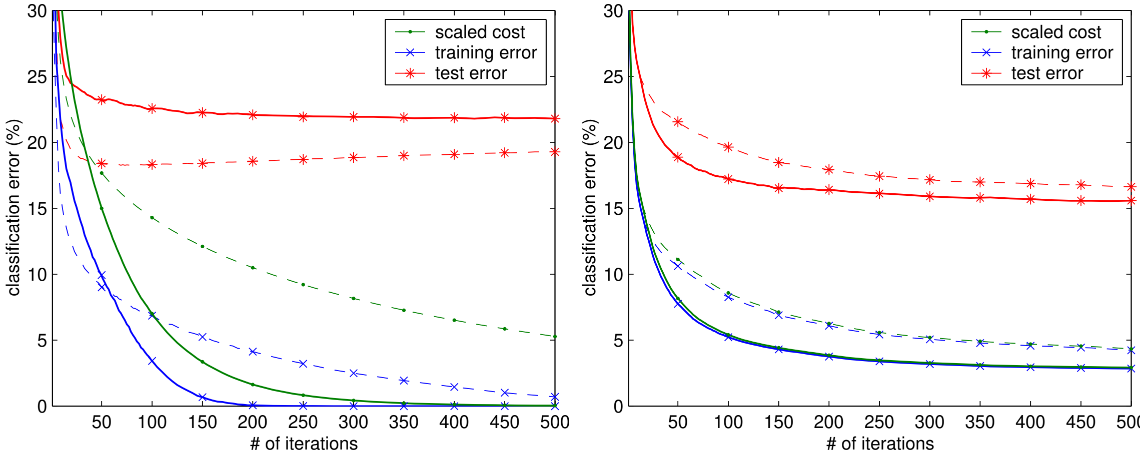
The results averaged over 132 independent trials are shown in Figure 3.1(a). We can see that, though the cost from AdaBoost was lower on average during the first 20–30 iterations, CGBoost overall decreased the cost faster and achieved a significantly lower cost. The training error, with similar trend as the cost, was also decreased much faster by CGBoost.

However, the out-of-sample behavior was the opposite. Noticing that AdaBoost got overfitting after roughly 50 iterations, the deteriorating out-of-sample performance of CGBoost implies that the exponential cost function is more vulnerable to overfitting when optimized directly and more aggressively.

This result is of no surprise since the exponential cost function has a very steep curve for negative margins (see Figure 3.2) and thus emphasizes "difficult" examples too much (Grove and Schuurmans, 1998; Dietterich, 2000; Mason et al., 2000a). While better optimization techniques can help decreasing the cost faster, the out-of-sample performance would be mainly determined by the cost function itself.

Based on the observation of this comparison, we set $\beta_t = 0$ for the first several

[2]The Caltech Data Engine (Pratap, 2003) is a computer program that contains several predefined data models, such as neural networks, support vector machines (SVM), and radial basis functions (RBF). When requested for data, it randomly picks a model, generates (also randomly) parameters for that model, and produces random examples according to the generated model. A complexity factor can be specified which controls the complexity of the generated model. The engine can be prompted repeatedly to generate independent data sets from the same model to achieve *small error bars* in testing and comparing learning algorithms.



(a) Exponential cost (b) Bisigmoid cost

Figure 3.1: Performance of CGBoost (solid curves) and AnyBoost (dashed curves) with two different cost functions. (a) AnyBoost with exponential cost is equivalent to AdaBoost; (b) Bisigmoid cost with $\kappa_+ = 1$ and $\kappa_- = 1.05$ in (3.5). Since $[\rho < 0] \approx \frac{1}{2}c(\rho)$ when $|\rho|$ is large, the training error and cost (scaled by 50) coincide quite well.

iterations in the following experiments to take advantage of the initial efficiency of gradient descent.

3.3.2 Bisigmoid Cost

Because of the power of conjugate gradient as an optimization technique, one can afford to use more "regularized" cost functions that are harder to optimize but have better out-of-sample performance.

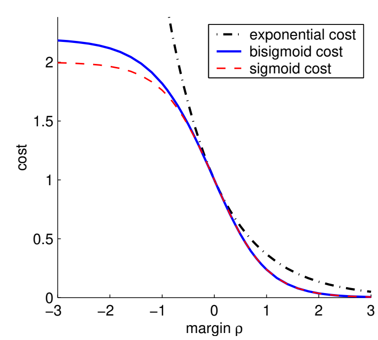


Figure 3.2: Three margin cost functions

The sigmoid margin cost function $c(\rho) = 1 - \tanh(\rho)$ was suggested in (Mason et al., 2000b). Since it has a flatter part for negative margins compared to the exponential cost (see Figure 3.2), it does not punish outliers too much. However, this cost function causes difficulties to AnyBoost and CGBoost. If the weak learner $\mathcal{L}$ finds the optimal hypothesis for the uniform sample distribution, AnyBoost and CGBoost will terminate at the second iteration due to $c'(-\rho) = c'(\rho)$ (Proof is similar to (Mason et al., 2000b, Lemma 12.9)).

A simple technique to avoid $c'(-\rho) = c'(\rho)$ is to concatenate sigmoid functions with different slopes for negative margins and positive margins. Below is what we call the *bisigmoid* function:

$$c(\rho) = \begin{cases} \kappa_+ - \kappa_+ \tanh(\rho/\kappa_+), & \text{for } \rho > 0; \\ \kappa_+ - \kappa_- \tanh(\rho/\kappa_-), & \text{otherwise,} \end{cases} \tag{3.5}$$

where κ_+ and κ_- are positive numbers controlling the slopes for positive and negative margins, respectively. We usually set $\kappa_- > \kappa_+$ so that the negative margins could also be emphasized a bit more. The closer κ_- is to κ_+, the more similar the bisigmoid curve is to a scaled sigmoid.

We applied CGBoost and AnyBoost with the bisigmoid function ($\kappa_+ = 1$ and $\kappa_- = 1.05$) to the problem in Subsection 3.3.1. Again we observed in Figure 3.1(b) that CGBoost optimized the cost and training error faster, though this time with the bisigmoid function, the cost could not be reduced to near zero and the training error was also above zero. However, we observed much better out-of-sample performance of both CGBoost and AnyBoost, compared to test errors in Figure 3.1(a). On average, *CGBoost achieved the lowest test error.*

This result reinforces the idea that the cost functions have great impact on the out-of-sample performance, while the optimization techniques only help to a lesser extent.

3.4 Experimental Results

We compared CGBoost and AnyBoost on six UCI data sets[3] with the exponential cost function and the bisigmoid cost function. In all these experiments κ_+ is fixed to 1. Since the value of κ_- decides how flat the bisigmoid is and thus how difficult

[3]The six UCI data sets are pima-indians-diabetes, sonar, cleveland-heart-disease, voting-records, breast-cancer-wisconsin, and ionosphere. Examples with missing features are removed from the original data sets.

the optimization is, we tried four values of κ_-, namely, 1.05, 1.1, 1.15, and 1.2. We observed that the smaller κ_- is, the more difficult the optimization is.

Each data set was randomly partitioned so that 80%, 10%, and 10% of the examples were used for training, validation, and testing. CGBoost and AnyBoost with different cost functions were allowed to run 300 iterations. Results were averaged over more than 60 trials.

Table 3.1 gives the geometric mean of the cost ratios between two optimization algorithms, CGBoost and AnyBoost, at the final iteration. As we expected, the cost of CGBoost is generally much lower than that of AnyBoost.

Table 3.1: Average final cost ratio of CGBoost to AnyBoost. Numbers less than 0.00005 are shown as 0.0000. To save space, the ratios with $\kappa_- = 1.1$ and $\kappa_- = 1.15$ are omitted.

Cost Function	**pima**	**sonar**	**clevel**	**vote84**	**cancer**	**iono**
exponential	0.5716	0.0000	0.0675	0.1006	0.0656	0.0000
$\kappa_- = 1.05$	0.8067	0.2674	0.6882	0.4997	0.8896	0.9949
$\kappa_- = 1.2$	0.7615	0.0000	0.6374	0.8058	0.9011	0.0000

We also compared the out-of-sample performance of these two algorithms. During one trial, the linear combination with the best validation error was picked. That is, for the exponential cost, validation chose the size of boosting; for the bisigmoid cost, validation also chose the "optimal" κ_- value.

The average test errors are listed in Table 3.2. Although it seems that CGBoost did not yield better test errors, the results from these algorithms are similar, and the relatively high error bars prevent us from drawing statistically significant conclusions for these limited data sets.

3.5 Conclusions

AdaBoost can be viewed as gradient descent of the exponential cost function in function space. In this paper, we introduced a new boosting algorithm, CGBoost, based on conjugate gradient in function space. We demonstrated with Caltech Data Engine

Table 3.2: Average test errors of CGBoost and AnyBoost on six UCI data sets. The lowest error in each column is in bold font. The error bars are much higher than the differences.

Cost	Method	pima	sonar	clevel	vote84	cancer	iono
exp.	AnyBoost	**25.10%**	**19.74%**	16.56%	**4.38%**	5.38%	**11.42%**
exp.	CGBoost	25.72%	22.02%	17.83%	4.46%	4.99%	12.62%
bisigmoid	AnyBoost	25.83%	23.97%	**16.45%**	4.59%	**4.14%**	11.49%
bisigmoid	CGBoost	26.25%	24.07%	17.77%	4.67%	4.78%	11.77%
roughly error bar		4.85%	9.22%	7.45%	3.04%	2.60%	5.59%

data and UCI data that CGBoost generally optimized the cost faster and achieved much lower training error.

We also observed that the exponential cost of AdaBoost was much more vulnerable to overfitting when it was minimized by the more aggressive CGBoost. The bisigmoid cost function, which has a flatter part for negative margins, was introduced to alleviate the overfitting problem. It also avoids the optimization difficulties that normal sigmoid function might have. Our experiments showed that, though it is harder to optimize, it generally leads to better out-of-sample performance. CGBoost with the bisigmoid cost yielded the lowest test error with Data Engine data.

However, the impact of cost functions on the out-of-sample performance still remains unclear, partly due to the statistically insignificant results on the UCI data sets.

Acknowledgments

This work was supported by the Caltech Center for Neuromorphic Systems Engineering under NSF Cooperative Agreement EEC-9402726.

Chapter 4

Multiclass Boosting with Repartitioning

A multiclass classification problem can be reduced to a collection of binary problems with the aid of a coding matrix. The quality of the final solution, which is an ensemble of base classifiers learned on the binary problems, is affected by both the performance of the base learner and the error-correcting ability of the coding matrix. A coding matrix with strong error-correcting ability may not be overall optimal if the binary problems are too hard for the base learner. Thus a trade-off between error-correcting and base learning should be sought. In this paper, we propose a new multiclass boosting algorithm that modifies the coding matrix according to the learning ability of the base learner. We show experimentally that our algorithm is very efficient in optimizing the multiclass margin cost, and outperforms existing multiclass algorithms such as AdaBoost.ECC and one-vs-one. The improvement is especially significant when the base learner is not very powerful.

4.1 Introduction

Many efforts of the machine learning research have been focused on binary classification problems. For a multiclass classification problem with more than two different class labels, it is possible to reformulate it as a collection of binary problems. The most popular approaches are one-vs-all where each class is compared against all others, and one-vs-one where all pairs of classes are compared (Allwein et al., 2000).

Dietterich and Bakiri (1995) and Allwein et al. (2000) unified and generalized most such approaches with error-correcting codes. In their framework, an error-correcting coding matrix is first given, with each row associated with a class from the multiclass problem. Binary classifiers (also called base classifiers) are then learned, one for each column of the matrix, on training examples that are relabeled according to the column. Given an unseen input, the vector formed by the outputs of the base classifiers is compared with every row of the coding matrix, and the class associated with the “closest” row is predicted as the class of the input.

The coding matrix is usually chosen for strong error-correcting ability (Dietterich and Bakiri, 1995). However, strong error-correcting ability alone does not guarantee good learning performance—one important assumption for normal error-correcting codes that errors are uncorrelated may not hold for the base classifiers (Guruswami and Sahai, 1999). Thus the choice of the coding matrix has to balance the needs of strong error-correction and uncorrelated classifier errors, and is usually problem-dependent (Allwein et al., 2000).

Multiclass boosting algorithms based on error-correcting codes (Schapire, 1997; Guruswami and Sahai, 1999) tackle the error correlation among the base classifiers by deliberately reweighting the training examples. They usually start off with an empty coding matrix and all classes indistinguishable from others, and then iteratively append columns to the matrix and train base classifiers so that the confusion between classes can be gradually reduced. The examples are reweighted in a fashion similar to the weighting scheme in the binary AdaBoost (Freund and Schapire, 1996), aiming at uncorrelated errors. In order to reduce the confusion between classes as fast as possible, in each iteration, a max-cut problem can be solved so that the “optimal” matrix column is obtained.

It is however common that researchers usually do not pursue the “optimal” coding matrix when applying the multiclass boosting algorithms. Instead, some choose the matrix columns at random (Schapire, 1997; Guruswami and Sahai, 1999).[1] Although the fact of max-cut being NP-complete prevents an efficient solution, this is not

[1] We actually did not find out how Guruswami and Sahai (1999) chose the columns.

exactly the reason for researchers not using it; after all, many multiclass classification problems have less than ten classes, and even some simple heuristic methods can do better in reducing the confusion than a random method. It is mostly because that, combined with the boosting algorithm, a max-cut or heuristic method does not improve over a random one (Schapire, 1997).

In this paper, we discuss why max-cut does not work well with existing multiclass boosting algorithms, and propose a general remedy which leads to a new boosting algorithm. We first discuss in Section 4.2 how AdaBoost.ECC, a typical multiclass boosting algorithm, can be explained as gradient descent on a margin cost function (Sun et al., 2005). The trade-off between the error-correcting ability and the base learning performance is then explained. We propose in Section 4.3 the new algorithm to achieve a better trade-off by modifying the coding matrix according to the learning ability of the base learner. In Section 4.4, our algorithm is tested on real-world data sets with four base learners of various degrees of complexity, and the results are quite promising. Finally we conclude in Section 4.5.

4.2 AdaBoost.ECC and Multiclass Cost

Consider a K-class classification problem where the class labels are 1, 2, …, K. The training set contains N examples, $\{(\mathbf{x}_n, y_n)\}_{n=1}^{N}$, where $\mathbf{x}_n$ is the input and $y_n \in \{1, 2, \ldots, K\}$. To reduce the multiclass problem to a collection of T binary problems, we use a coding matrix $\mathbf{M} \in \{-1, 0, +1\}^{K \times T}$ (Allwein et al., 2000). A base classifier f_t is learned on the relabeled examples $\{(\mathbf{x}_n, \mathbf{M}(y_n, t)) : \mathbf{M}(y_n, t) \neq 0\}$ based on the t-th column of $\mathbf{M}$, and classes that are relabeled as 0 are omitted. The columns of $\mathbf{M}$ are also called partitions (or partial partitions if there are 0's) since they define the way the original examples are split.

Given an input $\mathbf{x}$, the ensemble output $\mathbf{F}(\mathbf{x}) = (f_1(\mathbf{x}), \ldots, f_T(\mathbf{x}))$ is computed, and the Hamming decoding[2] (Allwein et al., 2000) is used to predict the label of $\mathbf{x}$.

[2]We consider base classifiers with outputs in $\{-1, +1\}$ (see experiment settings in Section 4.4). Thus a loss-based decoding is equivalent to the Hamming decoding.

Algorithm 4.1: AdaBoost.ECC (Guruswami and Sahai, 1999)

Input: A training set $\{(\mathbf{x}_n, y_n)\}_{n=1}^{N}$; number of epochs T
Initialize $\tilde{D}_1(n,k) = 1$; $\mathbf{F} = (0, 0, \ldots, 0)$, i.e., $f_t = 0$
for $t = 1$ to T **do**
  Choose the t-th column $\mathbf{M}(\cdot, t) \in \{-1, +1\}^K$
  $U_t = \sum_{n=1}^{N} \sum_{k=1}^{K} \tilde{D}_t(n,k) [\![\mathbf{M}(k,t) \neq \mathbf{M}(y_n, t)]\!]$
  $D_t(n) = U_t^{-1} \cdot \sum_{k=1}^{K} \tilde{D}_t(n,k) [\![\mathbf{M}(k,t) \neq \mathbf{M}(y_n, t)]\!]$
  Train f_t on $\{(\mathbf{x}_n, \mathbf{M}(y_n, t))\}$ with distribution D_t
  $\varepsilon_t = \sum_{n=1}^{N} D_t(n) [\![f_t(\mathbf{x}_n) \neq \mathbf{M}(y_n, t)]\!]$
  $\alpha_t = \frac{1}{2} \ln(\varepsilon_t^{-1} - 1)$
  $\tilde{D}_{t+1}(n,k) = \tilde{D}_t(n,k) \cdot e^{-\frac{\alpha_t}{2}[\mathbf{M}(y_n,t) - \mathbf{M}(k,t)] f_t(\mathbf{x}_n)}$
end for
return the coding matrix $\mathbf{M}$, the ensemble $\mathbf{F}$ and α_t

In the most general settings, there is a coefficient α_t for every base classifier f_t. The Hamming distance between $\mathbf{F}(\mathbf{x})$ and the k-th row $\mathbf{M}(k)$ is

$$\Delta\left(\mathbf{M}(k), \mathbf{F}(\mathbf{x})\right) = \sum_{t=1}^{T} \alpha_t \frac{1 - \mathbf{M}(k,t) f_t(\mathbf{x})}{2}.$$

Label y is predicted if $\mathbf{M}(y)$ has the smallest Hamming distance to $\mathbf{F}(\mathbf{x})$.

To correctly classify an example $(\mathbf{x}, y)$, we want $\Delta\left(\mathbf{M}(y), \mathbf{F}(\mathbf{x})\right)$ to be smaller than $\Delta\left(\mathbf{M}(k), \mathbf{F}(\mathbf{x})\right)$ for any $k \neq y$. Naturally, we may define the margin of the example $(\mathbf{x}, y)$ for class k as the difference between these two distances,

$$\rho_k\left(\mathbf{x}, y\right) = \Delta\left(\mathbf{M}(k), \mathbf{F}(\mathbf{x})\right) - \Delta\left(\mathbf{M}(y), \mathbf{F}(\mathbf{x})\right). \tag{4.1}$$

A learning algorithm should pick a coding matrix $\mathbf{M}$, T base classifiers f_t's, and their coefficients α_t's, such that the margins of the training examples are as large as possible.

AdaBoost.ECC (Guruswami and Sahai, 1999) is one such algorithm with a boosting style (Algorithm 4.1).[3] It starts from an empty coding matrix, and iteratively generates columns and base classifiers. Just as AdaBoost (Freund and Schapire, 1996)

[3]We only discuss the symmetric AdaBoost.ECC in this paper; nevertheless, our improvement can also be against the asymmetric AdaBoost.ECC.

optimizes some cost as gradient descent in the function space (Mason et al., 2000b), AdaBoost.ECC optimizes an exponential cost function based on the margins (Sun et al., 2005)[4]

$$C(\mathbf{F}) = \sum_{n=1}^{N} \sum_{k \neq y_n} e^{-\rho_k(\mathbf{x}_n, y_n)}. \tag{4.2}$$

We will briefly show how AdaBoost.ECC optimizes this cost in the t-th iteration. Using the definitions in Algorithm 4.1, we notice that by induction, $\mathbf{F} = (f_1, \ldots, f_t, 0, \ldots)$ and $\tilde{D}_{t+1}(n,k) = e^{-\rho_k(\mathbf{x}_n, y_n)}$. So $C(\mathbf{F}) = \sum_{n=1}^{N} \sum_{k=1}^{K} \tilde{D}_{t+1}(n,k) - N$ for $\tilde{D}_{t+1}(n, y_n)$ is always 1. The negative gradient at $\alpha_t = 0$ is thus

$$\begin{aligned} -\left.\frac{\partial C\left(\mathbf{F}\right)}{\partial \alpha_t}\right|_{\alpha_t=0} &= -\sum_{n=1}^{N} \sum_{k=1}^{K} \left.\frac{\partial \tilde{D}_{t+1}(n,k)}{\partial \alpha_t}\right|_{\alpha_t=0} \\ &= \sum_{n=1}^{N} \sum_{k=1}^{K} \tilde{D}_t(n,k) \left[\frac{\mathbf{M}(y_n,t) - \mathbf{M}(k,t)}{2}\right] f_t(\mathbf{x}_n). \end{aligned} \tag{4.3}$$

The last equality is due to step 9 in Algorithm 4.1. Since $\mathbf{M}(k,t)$ in AdaBoost.ECC can only be -1 or $+1$, the negative gradient can further be reduced to

$$U_t \sum_{n=1}^{N} D_t(n) \mathbf{M}(y_n, t) f_t(\mathbf{x}_n) = U_t(1 - 2\varepsilon_t). \tag{4.4}$$

AdaBoost.ECC tries to maximize this negative gradient and then picks α_t to exactly minimize the cost along the negative gradient.

Two steps in Algorithm 4.1 directly affect the maximization of the negative gradient (4.4). One is step 3 where the t-th column is picked. The t-th column decides the value of U_t, which indicates the error-correcting ability of the column. Roughly speaking, the larger U_t is, the stronger the error-correcting ability is, the faster the cost is reduced, and the smaller the training error bound is (Guruswami and Sahai, 1999, Theorem 2). The other is step 6, where the base classifier is learned. It is also obvious that both the cost and the training error bound can be smaller if the base

[4]Although Sun et al. (2005) used different definitions for the ensemble output and the distance measure, their cost function is equivalent to ours.

learner can achieve a smaller ε_t. It seems that in order for a better cost optimization, we should both maximize U_t and minimize ε_t.

A max-cut method has been proposed to obtain the "optimal" partition that maximizes U_t (Schapire, 1997). However, it appears that researchers prefer a somewhat random method for picking the partitions, e.g., rand-half that randomly picks half of the classes for label -1 and the other half for $+1$ (Schapire, 1997; Sun et al., 2005). This is actually with a reason: in long run, using the "optimal" partitions from max-cut is usually worse than using the random partitions, in both training and testing.

Let's look at a toy problem where points in a rectangle are assorted into seven tangram pieces (Figure 4.1). To compare the two column-picking methods, rand-half and max-cut, we ran AdaBoost.ECC on 500 random examples. Our base classifiers are perceptrons, which separate points with a straight line. It turned out that rand-half was more efficient in reducing the cost (Figure 4.2). And as a matter of fact, the test error in this experiment was also smaller with rand-half.

Why did max-cut, which maximized U_t in every iteration, have a worse performance in optimizing the cost? One probable reason is that the binary problems from max-cut are usually much "harder" for the base learner. To see this in the tangram experiment, we counted how many times a partition was picked during the Ada-

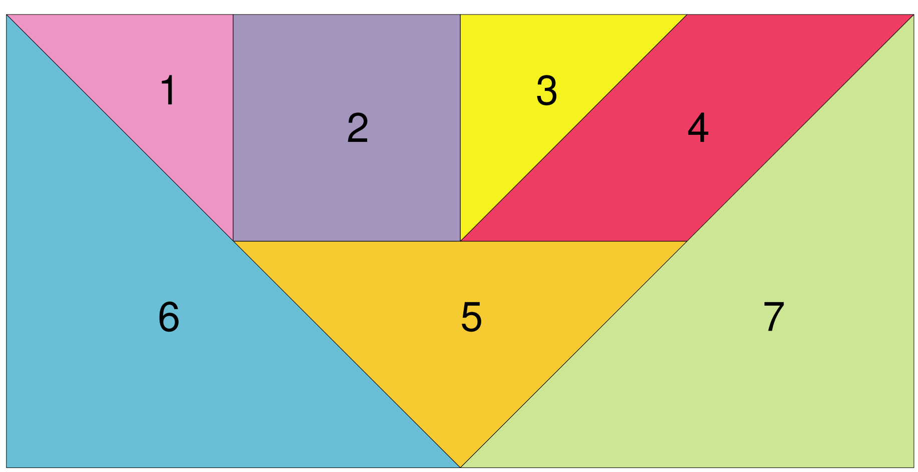


Figure 4.1: The tangram with seven pieces

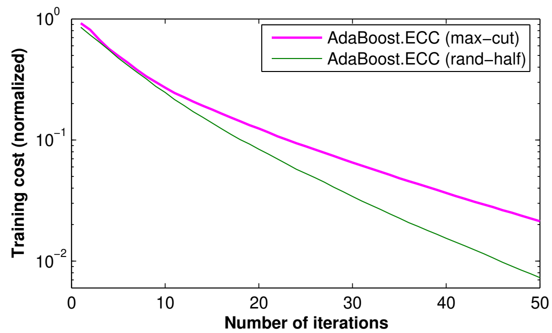
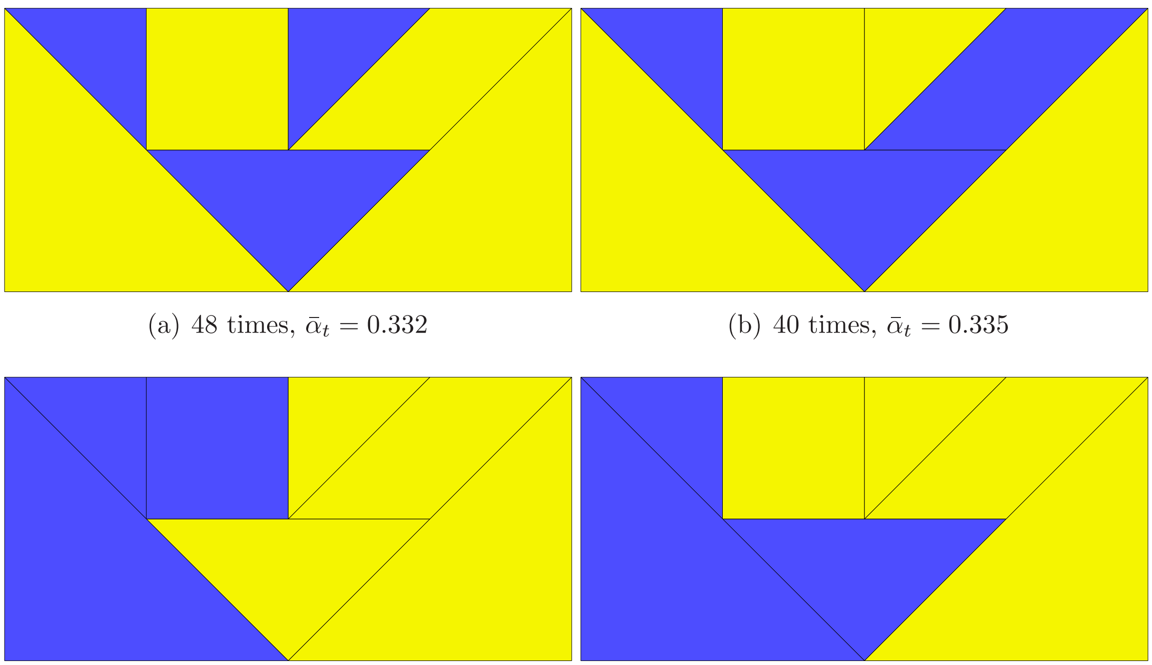


Figure 4.2: AdaBoost.ECC cost in the tangram experiment (normalized by $N(K-1)$)



(a) 48 times, $\bar{\alpha}_t = 0.332$

(b) 40 times, $\bar{\alpha}_t = 0.335$

(c) 9 times, $\bar{\alpha}_t = 0.813$

(d) 11 times, $\bar{\alpha}_t = 0.619$

Figure 4.3: Dominating partitions in the tangram experiment: (a,b) with max-cut; (c,d) with rand-half

Boost.ECC runs, and summed up for this partition the coefficients α_t, which were decided from the weighted error ε_t of the base classifiers trained on the partition. The sum indicates how much the partition influences the ensemble, and the average coefficient (denoted as $\bar{\alpha}$) implies how hard the binary problems are to the base learner. Figure 4.3 gives the two dominating partitions with the largest coefficient sums out of the 200 AdaBoost.ECC iterations. Obviously AdaBoost.ECC with max-cut focused on harder binary problems, while AdaBoost.ECC with rand-half was happy with easier problems. Since harder problems deteriorated the learning of base classifiers, the overall cost reduction was worse for `max-cut`. Note that this situation might be more prominent for later iterations since the boosting nature of AdaBoost.ECC keeps increasing the hardness of the binary problems.

It is thus important to find a good trade-off between maximizing U_t and minimizing ε_t. In next section, we will discuss a remedy based on repartitioning.

4.3 AdaBoost.ECC with Repartitioning

We have seen from the tangram experiment that different partitioning methods may generate binary problems of various hardness levels. How hard a problem is depends on how the relabeled examples distribute in the feature space and how well the base learner can handle such a distribution. For example, with perceptrons as the base classifiers, discriminating tangram classes 1 and 3 from 2 and 4 (Figure 4.3(a)) is much harder than discriminating 1 and 2 from 3 and 4 (Figure 4.3(c)). Thus in order to achieve a good trade-off between maximizing U_t and minimizing ε_t, we should also consider the discriminating ability of the base learner when picking the partitions.

How do we know whether a partition can be well handled by the base learner? We usually do not know unless the base learner has been tried on the partition. The learned classifier has its own preference on how the examples should be relabeled, and thus hints on what partitions better suit the base learner. We can then repartition the examples based on such information so as to reduce the cost even more.

Assume in the t-th iteration, a base classifier f_t has been learned. To find a new and better partition for this f_t, we try to maximize the negative gradient (4.3),

$$\max_{\mathbf{M}(\cdot,t)} \sum_{n=1}^{N} \sum_{k=1}^{K} \tilde{D}_t(n,k) \left[\mathbf{M}(y_n,t) - \mathbf{M}(k,t)\right] f_t(\mathbf{x}_n),$$

which can be reorganized as

$$\max_{\mathbf{M}(\cdot,t)} \sum_{k=1}^{K} \boldsymbol{\mu}(k,t)\mathbf{M}(k,t),$$

with $\boldsymbol{\mu}(k,t)$ defined as $\boldsymbol{\mu}(k,t) =$

$$\sum_{n\colon y_n=k} \sum_{\ell=1}^{K} \tilde{D}_t(n,\ell) f_t(\mathbf{x}_n) - \sum_{n=1}^{N} \tilde{D}_t(n,k) f_t(\mathbf{x}_n). \tag{4.5}$$

Since $\mathbf{M}(k,t) \in \{-1,+1\}$, it is clear that the negative gradient is maximized when $\mathbf{M}(k,t) = \operatorname{sign}\left[\boldsymbol{\mu}(k,t)\right]$.

The repartitioning can also be justified intuitively from a single example point of view. On one side, the contribution of example $(\mathbf{x}_n, y_n)$ to $\mathbf{M}(y_n,t) = \operatorname{sign}\left[\boldsymbol{\mu}(y_n,t)\right]$ is

$$\left[\sum_{\ell=1}^{K} \tilde{D}_t(n,\ell) - \tilde{D}_t(n,y_n)\right] f_t(\mathbf{x}_n).$$

Note that with $\mathbf{F} = (f_1, \ldots, f_{t-1}, 0, \ldots)$, $\tilde{D}_t(i,k)$ is $e^{-\rho_k(\mathbf{x}_n,y_n)}$. So the summation $\sum_{\ell \neq y_n} \tilde{D}_t(n,\ell)$ actually tells, without the current f_t, how close the example is to classes other than its own class y_n. The closer it is to other classes, the larger the summation is, and thus the more likely $\mathbf{M}(y_n,t)$ would be to have the same sign as $f_t(\mathbf{x}_i)$, which would in consequence increase some of the margins of this example after f_t is included. On the other side, the contribution of the example to $\mathbf{M}(k,t)$ where $k \neq y_n$ is $-\tilde{D}_t(n,k) f_t(\mathbf{x}_n)$. With similar reasoning, this implies that if the example is close to class k, $\mathbf{M}(k,t)$ would be requested to have the opposite sign as $f_t(\mathbf{x}_n)$, which also would increase the margin ρ_k.

The repartitioning of $\mathbf{M}(\cdot,t)$ and the learning of f_t can be carried out alternatively. For example, we can start from a partition, train a base classifier on it, repartition the classes, and then train a new base classifier on the new partition. If the base learner always minimizes the weighted training error, the negative gradient would always increase until convergence. In practice, when the base learning is expensive, we may only repeat the repartitioning and learning cycle for several fixed steps.

Algorithm 4.2: AdaBoost.ERP (ECC with repartitioning)

Input: A training set $\{(\mathbf{x}_n, y_n)\}_{n=1}^{N}$; number of epochs T

1: Initialize $\tilde{D}_1(n,k) = 1$; $\mathbf{F} = (0, 0, \ldots, 0)$, i.e., $f_t = 0$
2: **for** $t = 1$ to T **do**
3: Choose an initial column $\mathbf{M}(\cdot, t) \in \{\underline{-1, 0, +1}\}^K$
4: **repeat** {Alternate learning and re-partitioning}
5: $U_t = \sum_{n=1}^{N} \sum_{k=1}^{K} \tilde{D}_t(n,k) [\![\underline{\mathbf{M}(k,t)\mathbf{M}(y_n,t) < 0}]\!]$
6: $D_t(n) = U_t^{-1} \cdot \sum_k \tilde{D}_t(n,k) [\![\underline{\mathbf{M}(k,t)\mathbf{M}(y_n,t) < 0}]\!]$
7: Train f_t on $\{(\mathbf{x}_n, \mathbf{M}(y_n, t))\}$ with distribution D_t
8: $\underline{\mathbf{M}(k,t) = \operatorname{sign}[\boldsymbol{\mu}(k,t)]}$ {See (4.5) for details}
9: **until** convergence or some specified steps
10: Update U_t and D_t with the current $\mathbf{M}(\cdot, t)$, as above
11: $\varepsilon_t = \sum_{n=1}^{N} D_t(n) [\![f_t(\mathbf{x}_n) \neq \mathbf{M}(y_n, t)]\!]$
12: $\alpha_t = \frac{1}{2} \ln(\varepsilon_t^{-1} - 1)$
13: $\tilde{D}_{t+1}(n,k) = \tilde{D}_t(n,k) \cdot e^{-\frac{\alpha_t}{2}[\mathbf{M}(y_n,t) - \mathbf{M}(k,t)] f_t(\mathbf{x}_n)}$
14: **end for**
15: **return** the coding matrix $\mathbf{M}$, the ensemble $\mathbf{F}$ and α_t

Algorithm 4.2 depicts the new multiclass boosting algorithm, AdaBoost.ERP, i.e., AdaBoost.ECC with repartitioning. The changes from AdaBoost.ECC are underlined for better reading. Note that we also allow the initial column $\mathbf{M}(\cdot, t)$ to have 0's (step 3). Since a partial partition will be adjusted in the repartitioning step to a full one, the coefficient α_t can still be decided exactly. The benefit of having a partial partition is that only part of the examples are used for the initial base learning (step 7). This allows, for example, to first focus the learning on local structures of just a pair of classes and then extend to the full partition based on the knowledge learned from the local structures. Besides, the base learning is also faster with less examples.

The repartitioning takes $2NK$ arithmetic operations, which is usually much cheaper than the base learning.

4.4 Experiments

We tested AdaBoost.ERP experimentally on ten multiclass benchmark problems (Table 4.1) from the UCI machine learning repository (Hettich et al., 1998) and the

Table 4.1: Multiclass problems

data set	#train	#test	K	#attribute
dna	2000	1186	3	180
glass	214	-	6	9
iris	150	-	3	4
letter	16000	4000	26	16
pendigits	7494	3498	10	16
satimage	4435	2000	6	36
segment	2310	-	7	18
vehicle	846	-	4	18
vowel	528	462	11	10
wine	178	-	3	13

StatLog project (Michie et al., 1994). For problems with both training and test sets, experiments were run 100 times and the results were averaged. Otherwise, a 10-fold cross-validation was repeated 10 times for a total of 100 runs. When there was randomness in the learning algorithm and/or cross-validation was used, the standard error over 100 runs was also computed. For each run, the training part of the examples were linearly scaled to $[-1, 1]$, and then the test examples were adjusted accordingly.

We tried different ways to set the initial partitions and different schedules to repartition. In the results reported here, the initial partial partitions always contained two classes selected from all the K classes, randomly (denoted by rand-2) or to maximize the corresponding U_t (denoted by max-2). We use a string of "L" and "R" to represent the schedule of base learning and repartitioning in a boosting iteration. For example, "LRL" means that a base classifier was first learned on the two classes in the partial partition, then the partition was adjusted, and finally a new base classifier was trained on the adjusted full partition.

We used four base learners of various degrees of complexity. The first one is the decision stump, also known as FindAttrTest (Schapire, 1997). The second one is the perceptron with a learning algorithm suitable for boosting (Li, 2005). The third one is a binary AdaBoost (Freund and Schapire, 1996) that aggregates up to 50 decision stumps. The last one is the soft-margin support vector machine with the perceptron kernel (SVM-perceptron) (Lin and Li, 2005b).[5]

[5]For the perceptron kernel, only the regularization parameter C needs to be tuned. For problems

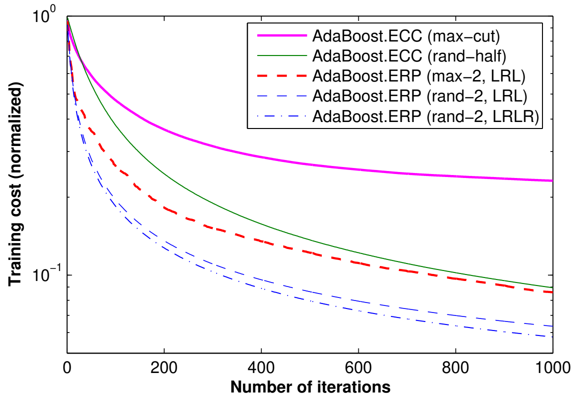
Table 4.2: Test errors (%) of multiclass algorithms with the decision stump as the base learner

	one-vs-one	AdaBoost.ECC		AdaBoost.ERP (max-2)		AdaBoost.ERP (rand-2)	
data set	one-vs-all	max-cut	rand-half	LRL	LRLR	LRL	LRLR
dna	30.61	5.90	5.92 ± 0.02	6.41	**5.56**	5.78 ± 0.03	5.88 ± 0.03
glass	34.10 ± 1.11	27.43 ± 0.95	26.67 ± 0.92	26.05 ± 0.85	25.57 ± 0.89	$\mathbf{25.29 \pm 0.85}$	25.62 ± 0.85
iris	7.60 ± 0.55	7.67 ± 0.61	$\mathbf{6.60 \pm 0.60}$	6.73 ± 0.59	6.80 ± 0.59	7.53 ± 1.10	$\mathbf{6.60 \pm 0.59}$
letter	39.42	32.79 ± 0.19	22.00 ± 0.04	21.05	**17.73**	18.52 ± 0.03	17.84 ± 0.02
pendigits	23.67	9.06	5.94 ± 0.02	6.03	5.80	5.65 ± 0.03	$\mathbf{5.55 \pm 0.02}$
satimage	19.15	14.50	12.57 ± 0.04	**12.10**	12.45	12.59 ± 0.04	12.58 ± 0.04
segment	12.24 ± 0.21	3.28 ± 0.12	1.94 ± 0.09	2.07 ± 0.09	1.97 ± 0.09	$\mathbf{1.90 \pm 0.09}$	1.95 ± 0.09
vehicle	43.31 ± 0.48	26.93 ± 0.40	22.13 ± 0.38	23.28 ± 0.38	22.85 ± 0.39	$\mathbf{22.08 \pm 0.39}$	22.40 ± 0.41
vowel	57.14	59.74	57.98 ± 0.16	**55.63**	59.09	57.40 ± 0.15	57.65 ± 0.13
wine	15.33 ± 0.71	$\mathbf{2.00 \pm 0.32}$	3.17 ± 0.39	2.33 ± 0.36	2.72 ± 0.39	2.83 ± 0.37	2.78 ± 0.37

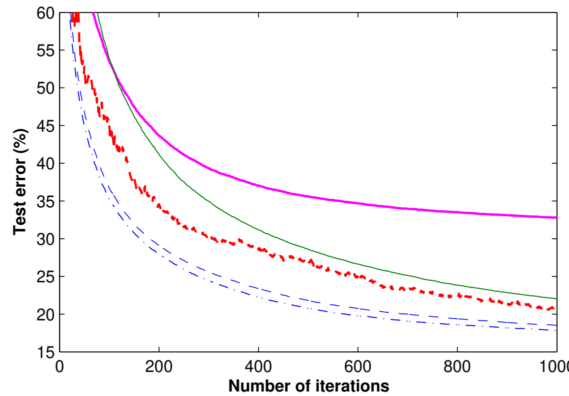
We compared our algorithm with AdaBoost.ECC with max-cut or rand-half. When the decision stump was used as the base learner, each algorithm was run for 500 iterations; for other more powerful base learners, the number of iterations was 200. However, for one exception, the letter data with 26 classes, we ran 1000 iterations with the decision stump and 500 iterations with other base learners. Note also that the exact max-cut for 26 classes is time-consuming so instead we used a simple greedy approximation for the letter data to approximately maximize U_t for AdaBoost.ECC. We also compared with one-vs-one and one-vs-all using the same base learners. For space consideration, we only list the lower test errors of these two algorithms.

Table 4.2 presents the test errors with the decision stump as the base learner, the lowest errors in bold. With this simple base learner, one-vs-one and one-vs-all got quite large errors since they are limited in the number of base classifiers. We can also see that most of the time AdaBoost.ECC with max-cut was worse than AdaBoost.ECC with rand-half. This verified our analysis that, when the base learner is not powerful enough, problems from max-cut would be too hard and the overall learning performance would instead be deteriorated (see also Figure 4.4). With the help of repartitioning, AdaBoost.ERP achieved better test errors for most of the data sets, and for some cases it was substantially better. For better illustration, we also

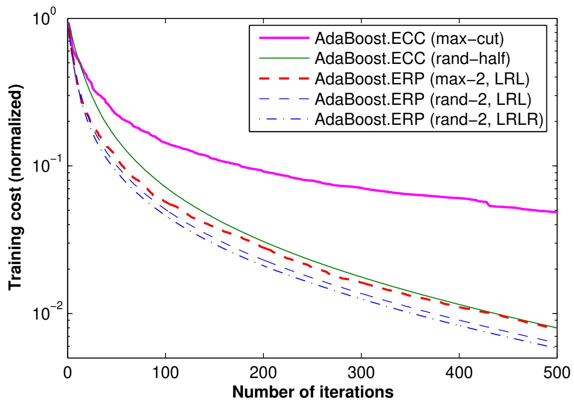
with both training and test sets, a cross-validation with 30% of the training set kept for validation was repeated 10 times. The best $C \in \left\{2^{-3}, 1, 2^3, 2^6, 2^9\right\}$ was then used in the full training and testing. The whole process was repeated 20 time. For problems with no test sets, the best results of the 10-fold cross-validation averaged over 10 times were reported. To support the weighted data, we scale C for each example proportional to its sample weight (Vapnik, 1999).



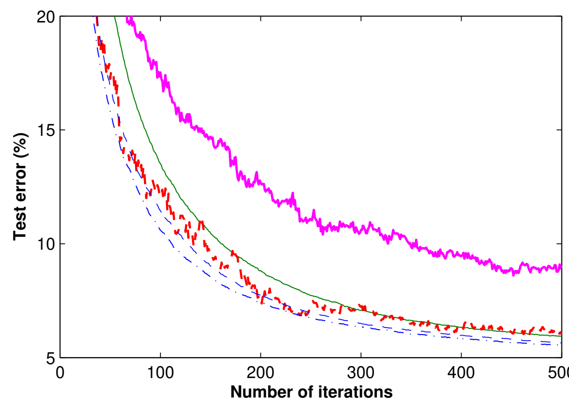
(a) cost on the letter data



(b) test error on the letter data



(c) cost on the pendigits data



(d) test error on the pendigits data

Figure 4.4: Multiclass boosting with the decision stump (AdaBoost.ERP (max-2, LRLR) is very close to that with rand-2)

show in Figure 4.4 the training cost and the test error curves for two large data sets, letter and pendigits. With the same number of base classifiers, AdaBoost.ERP almost always achieved a much lower training cost and a lower test error. More steps of the repartitioning and base learning further improved the learning, although the marginal improvement was small.

With the perceptron as a more powerful base learner, test errors on some data sets were greatly reduced (Tables 4.3, with less number of iterations compared to that with the decision stump). Again repartitioning improved the learning performance on most of the data sets. Figure 4.5 shows the training cost and the test error curves for the letter data set. Observations are similar to those of Figure 4.4, but the improvement

Table 4.3: Test errors (%) of multiclass algorithms with the perceptron as the base learner

data set	one-vs-one one-vs-all	AdaBoost.ECC max-cut	AdaBoost.ECC rand-half	AdaBoost.ERP (max-2) LRL	AdaBoost.ERP (max-2) LRLR	AdaBoost.ERP (rand-2) LRL	AdaBoost.ERP (rand-2) LRLR
dna	25.97 ± 0.26	$\mathbf{8.08 \pm 0.05}$	8.21 ± 0.06	8.18 ± 0.06	8.17 ± 0.06	8.09 ± 0.07	8.11 ± 0.06
glass	35.57 ± 1.16	29.48 ± 0.94	30.00 ± 1.02	$\mathbf{28.38 \pm 0.87}$	29.43 ± 1.04	30.57 ± 1.05	29.81 ± 0.94
iris	4.93 ± 0.57	5.20 ± 0.56	4.40 ± 0.53	4.67 ± 0.51	4.47 ± 0.51	$\mathbf{4.33 \pm 0.53}$	4.60 ± 0.52
letter	22.17 ± 0.07	15.88 ± 0.05	14.66 ± 0.05	13.64 ± 0.05	11.61 ± 0.04	13.65 ± 0.05	$\mathbf{11.59 \pm 0.05}$
pendigits	7.09 ± 0.09	3.71 ± 0.03	3.72 ± 0.03	3.72 ± 0.02	$\mathbf{3.64 \pm 0.03}$	3.71 ± 0.02	3.68 ± 0.03
satimage	15.14 ± 0.06	12.84 ± 0.05	12.60 ± 0.05	$\mathbf{12.37 \pm 0.05}$	12.42 ± 0.05	12.58 ± 0.05	12.57 ± 0.05
segment	7.53 ± 0.18	2.80 ± 0.12	2.74 ± 0.11	2.81 ± 0.11	2.83 ± 0.11	2.74 ± 0.10	$\mathbf{2.60 \pm 0.11}$
vehicle	31.58 ± 0.49	22.22 ± 0.45	20.47 ± 0.42	20.39 ± 0.42	20.86 ± 0.43	20.88 ± 0.44	$\mathbf{20.34 \pm 0.43}$
vowel	56.19 ± 0.29	56.26 ± 0.28	51.61 ± 0.26	50.61 ± 0.22	50.97 ± 0.26	50.40 ± 0.26	$\mathbf{50.31 \pm 0.25}$
wine	3.22 ± 0.41	$\mathbf{2.06 \pm 0.32}$	2.67 ± 0.37	2.39 ± 0.37	2.33 ± 0.37	2.39 ± 0.36	2.56 ± 0.39

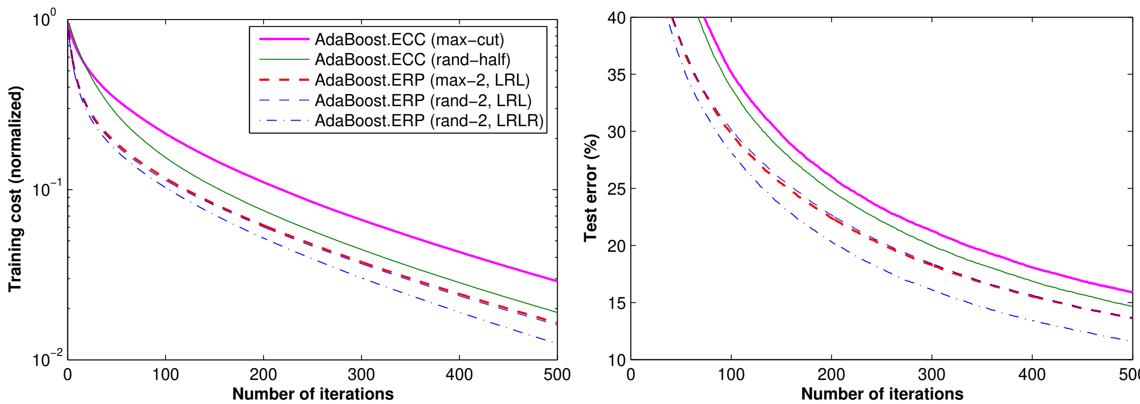


Figure 4.5: Multiclass boosting with the perceptron on the letter data

was not as dramatic as with the decision stump.

The binary AdaBoost was the only weak learner with which one-vs-one actually had comparable or even better performance compared to the boosting algorithms. So we mark in Table 4.4 both the lowest errors among the boosting algorithms and the lowest errors among all the algorithms. Note that with this base learner, AdaBoost.ERP with only one base learning and one repartitioning ("LR") was already comparable to AdaBoost.ECC with base learning on the full training set.

SVM-perceptron brought us the overall lowest test errors for most of the data sets (Table 4.5). Note that we do not have all the results for dna and letter since parameter selection on an ensemble of SVMs is time-consuming. With this powerful base learner, all the multiclass algorithms performed comparably well, although AdaBoost.ERP was still better for some data sets. AdaBoost.ERP was also much faster

Table 4.4: Test errors (%) with the AdaBoost that aggregates 50 decision stumps as the base learner

	one-vs-one	AdaBoost.ECC		AdaBoost.ERP (max-2)		AdaBoost.ERP (rand-2)	
data set	one-vs-all	max-cut	rand-half	LR	LRLR	LR	LRLR
dna	**6.32**	7.93	7.30 ± 0.03	7.50	**6.75**	7.48 ± 0.05	7.17 ± 0.04
glass	26.57 ± 0.87	27.29 ± 0.96	26.52 ± 0.91	26.48 ± 0.92	26.48 ± 0.94	26.62 ± 0.95	$\mathbf{25.57 \pm 0.90}$
iris	6.00 ± 0.60	6.40 ± 0.58	7.00 ± 0.57	$\mathbf{5.67 \pm 0.59}$	6.20 ± 0.59	79.33 ± 0.75	9.13 ± 1.47
letter	**12.12**	40.12 ± 0.24	20.82 ± 0.05	27.88	**16.05**	16.89 ± 0.05	16.30 ± 0.04
pendigits	4.92	8.81	5.37 ± 0.02	$\mathbf{4.00 \pm 0.01}$	5.69	5.55 ± 0.08	5.12 ± 0.02
satimage	**12.55**	14.60	13.87 ± 0.04	14.75	13.65	$\mathbf{12.74 \pm 0.05}$	13.67 ± 0.04
segment	2.66 ± 0.11	2.44 ± 0.10	$\mathbf{1.89 \pm 0.08}$	1.94 ± 0.10	2.15 ± 0.09	2.18 ± 0.10	1.96 ± 0.09
vehicle	24.66 ± 0.41	24.60 ± 0.47	22.82 ± 0.45	26.01 ± 0.43	23.32 ± 0.44	$\mathbf{22.80 \pm 0.43}$	23.05 ± 0.44
vowel	**46.10**	56.93	56.64 ± 0.14	$\mathbf{50.33 \pm 0.03}$	57.14	51.35 ± 0.23	56.21 ± 0.14
wine	$\mathbf{2.61 \pm 0.34}$	4.72 ± 0.51	$\mathbf{2.94 \pm 0.40}$	3.50 ± 0.42	4.72 ± 0.48	3.17 ± 0.37	3.22 ± 0.42

Table 4.5: Test errors (%) of multiclass algorithms with the SVM-perceptron as the base learner

	one-vs-one	AdaBoost.ECC		AdaBoost.ERP (max-2)		AdaBoost.ERP (rand-2)	
data set	one-vs-all	max-cut	rand-half	LR	LRLR	LR	LRLR
glass	28.71 ± 0.96	28.52 ± 0.90	28.14 ± 1.01	29.00 ± 0.98	$\mathbf{28.05 \pm 0.88}$	28.24 ± 0.92	28.19 ± 0.87
iris	4.00 ± 0.47	3.87 ± 0.52	$\mathbf{3.73 \pm 0.49}$	$\mathbf{3.73 \pm 0.48}$	3.93 ± 0.49	3.93 ± 0.50	$\mathbf{3.73 \pm 0.49}$
pendigits	1.71 ± 0.00	1.80 ± 0.01	1.81 ± 0.03	2.34 ± 0.04	$\mathbf{1.64 \pm 0.02}$	3.71 ± 0.14	1.81 ± 0.04
satimage	7.70	7.66 ± 0.02	7.70 ± 0.07	7.71 ± 0.02	7.72 ± 0.05	7.76 ± 0.02	$\mathbf{7.63 \pm 0.06}$
segment	2.09 ± 0.09	2.21 ± 0.10	$\mathbf{2.08 \pm 0.09}$	2.09 ± 0.09	2.16 ± 0.09	2.10 ± 0.09	2.14 ± 0.09
vehicle	17.89 ± 0.37	19.08 ± 0.39	18.65 ± 0.37	17.93 ± 0.38	$\mathbf{17.67 \pm 0.35}$	17.96 ± 0.37	17.89 ± 0.37
vowel	37.45	39.49 ± 0.14	39.42 ± 0.29	$\mathbf{36.44 \pm 0.02}$	39.95 ± 0.19	38.16 ± 0.37	40.03 ± 0.25
wine	0.94 ± 0.22	1.22 ± 0.26	0.94 ± 0.25	$\mathbf{0.89 \pm 0.23}$	1.06 ± 0.23	1.17 ± 0.25	0.94 ± 0.21

compared to AdaBoost.ECC even though two SVMs may be learned in one iteration of AdaBoost.ERP, since the binary problems were usually easier.

4.5 Conclusion

We have proposed and tested AdaBoost.ERP, a new multiclass boosting algorithm with error-correcting codes and repartitioning. The repartitioning is meant to find a better coding matrix according to the learning ability of the base learner. Our experimental results have shown that, compared with AdaBoost.ECC, one-vs-one, and one-vs-all, AdaBoost.ERP achieved the lowest training cost and test error on most of the real-world data sets we used. The improvement can be especially significant when the base learner is not very powerful. AdaBoost.ERP was also faster than AdaBoost.ECC when working with SVM-perceptron.

Simple algorithms like one-vs-one have their advantages. Compared to boosting algorithms, their training time is usually much less, and the test error can be comparable or even lower when powerful base learners are used. The test time can also be substantially reduced (Platt et al., 2000). Thus it is interesting to see how boosting algorithms can be further improved in these aspects.

Acknowledgments

The author thanks Yaser Abu-Mostafa, Alex Holub, and the anonymous reviewers for helpful comments. This work was supported by the Caltech SISL Graduate Fellowship.

Bibliography

Abu-Mostafa, Y. S. (1988a). Complexity of random problems. In Abu-Mostafa, Y. S., editor, *Complexity in Information Theory*, pages 115–131. Springer-Verlag.

Abu-Mostafa, Y. S. (1988b). Random problems. *Journal of Complexity*, 4(4):277–284.

Abu-Mostafa, Y. S. (1995). Hints. *Neural Computation*, 7(4):639–671.

Aizerman, M., Braverman, E., and Rozonoer, L. (1964). Theoretical foundations of the potential function method in pattern recognition learning. *Automation and Remote Control*, 25(6):821–837.

Allwein, E. L., Schapire, R. E., and Singer, Y. (2000). Reducing multiclass to binary: A unifying approach for margin classifiers. *Journal of Machine Learning Research*, 1:113–141.

Angelova, A., Abu-Mostafa, Y., and Perona, P. (2005). Pruning training sets for learning of object categories. In Schmid, C., Soatto, S., and Tomasi, C., editors, *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2005)*, volume 1, pages 494–501.

Angiulli, F., Greco, G., and Palopoli, L. (2004). Detecting outliers via logical theories and its data complexity. In Suzuki, E. and Arikawa, S., editors, *Discovery Science: 7th International Conference, DS 2004*, volume 3245 of *Lecture Notes in Artificial Intelligence*, pages 101–113. Springer-Verlag.

Arning, A., Agrawal, R., and Raghavan, P. (1996). A linear method for deviation detection in large databases. In Simoudis, E., Han, J., and Fayyad, U., editors, *Pro-*

ceedings of the Second International Conference on Knowledge Discovery & Data Mining, pages 164–169. AAAI Press.

Barnett, V. and Lewis, T. (1994). *Outliers in Statistical Data*. John Wiley & Sons, 3rd edition.

Bishop, C. M. (1995). *Neural Networks for Pattern Recognition*. Oxford University Press.

Blumer, A., Ehrenfeucht, A., Haussler, D., and Warmuth, M. K. (1987). Occam's razor. *Information Processing Letters*, 24(6):377–380.

Breiman, L. (1996). Bias, variance, and arcing classifiers. Technical Report 460, Department of Statistics, University of California at Berkeley.

Breiman, L. (1998). Arcing classifiers. *The Annals of Statistics*, 26(3):801–824.

Brodley, C. E. and Friedl, M. A. (1999). Identifying mislabeled training data. *Journal of Artificial Intelligence Research*, 11:131–167.

Cauwenberghs, G. and Poggio, T. (2001). Incremental and decremental support vector machine learning. In Leen, T. K., Dietterich, T. G., and Tresp, V., editors, *Advances in Neural Information Processing Systems 13*, pages 409–415. MIT Press.

Chang, C.-C. and Lin, C.-J. (2001). *LIBSVM: A library for support vector machines*. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

Dietterich, T. G. (2000). An experimental comparison of three methods for constructing ensembles of decision trees: Bagging, boosting, and randomization. *Machine Learning*, 40(2):139–157.

Dietterich, T. G. and Bakiri, G. (1995). Solving multiclass learning problems via error-correcting output codes. *Journal of Artificial Intelligence Research*, 2:263–286.

Freund, Y. and Schapire, R. E. (1996). Experiments with a new boosting algorithm. In *Machine Learning: Proceedings of the Thirteenth International Conference (ICML '96)*, pages 148–156. Morgan Kaufmann.

Freund, Y. and Schapire, R. E. (1999). Large margin classification using the perceptron algorithm. *Machine Learning*, 37(3):277–296.

Friedman, J. H. (1999). Regularized discriminant analysis. *Journal of the American Statistical Association*, 84(405):165–175.

Gallant, S. I. (1990). Perceptron-based learning algorithms. *IEEE Transactions on Neural Networks*, 1(2):179–191.

Gamberger, D. and Lavrač, N. (1997). Conditions for Occam's razor applicability and noise elimination. In van Someren, M. and Widmer, G., editors, *Machine Learning: ECML-97*, volume 1224 of *Lecture Notes in Artificial Intelligence*, pages 108–123. Springer-Verlag.

Grove, A. J. and Schuurmans, D. (1998). Boosting in the limit: Maximizing the margin of learned ensembles. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 692–699. AAAI Press/MIT Press.

Grünwald, P. (2005). Minimum description length tutorial. In Grünwald, P. D., Myung, I. J., and Pitt, M. A., editors, *Advances in Minimum Description Length: Theory and Applications*, chapter 2, pages 23–80. MIT Press.

Guruswami, V. and Sahai, A. (1999). Multiclass learning, boosting, and error-correcting codes. In Ben-David, S. and Long, P., editors, *Proceedings of the Twelfth Annual Conference on Computational Learning Theory*, pages 145–155. ACM Press.

Guyon, I., Matić, N., and Vapnik, V. (1996). Discovering informative patterns and data cleaning. In Fayyad, U. M., Piatetsky-Shapiro, G., Smyth, P., and Uthurusamy, R., editors, *Advances in Knowledge Discovery and Data Mining*, pages 181–203. AAAI Press/MIT Press.

Hammer, P. L., Kogan, A., Simeone, B., and Szedmák, S. (2004). Pareto-optimal patterns in logical analysis of data. *Discrete Applied Mathematics*, 144(1–2):79–102.

Hettich, S., Blake, C. L., and Merz, C. J. (1998). UCI repository of machine learning databases. Available at `http://www.ics.uci.edu/~mlearn/MLRepository.html`.

Hodge, V. J. and Austin, J. (2004). A survey of outlier detection methodologies. *Artificial Intelligence Review*, 22(2):85–126.

Holte, R. C. (1993). Very simple classification rules perform well on most commonly used datasets. *Machine Learning*, 11(1):63–91.

Hsu, C.-W., Chang, C.-C., and Lin, C.-J. (2003). A practical guide to support vector classification. Technical report, National Taiwan University.

Knorr, E. M. and Ng, R. T. (1997). A unified notion of outliers: Properties and computation. In Heckerman, D., Mannila, H., Pregibon, D., and Uthurusamy, R., editors, *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining*, pages 219–222. AAAI Press.

Levin, L. A. (1973). Universal sequential search problems. *Problems of Information Transmission*, 9(3):265–266.

Li, L. (2005). Perceptron learning with random coordinate descent. Computer Science Technical Report CaltechCSTR:2005.006, California Institute of Technology, Pasadena, CA.

Li, L., Pratap, A., Lin, H.-T., and Abu-Mostafa, Y. S. (2005). Improving generalization by data categorization. In Jorge, A., Torgo, L., Brazdil, P., Camacho, R., and Gama, J., editors, *Knowledge Discovery in Databases: PKDD 2005*, volume 3721 of *Lecture Notes in Artificial Intelligence*, pages 157–168. Springer-Verlag.

Li, M. and Vitányi, P. (1997). *An Introduction to Kolmogorov Complexity and Its Applications*. Springer-Verlag, 2nd edition.

Li, Y. and Long, P. M. (2002). The relaxed online maximum margin algorithm. *Machine Learning*, 46(1–3):361–387.

Lin, H.-T. and Li, L. (2005a). Infinite ensemble learning with support vector machines. In Gama, J., Camacho, R., Brazdil, P., Jorge, A., and Torgo, L., editors, *Machine Learning: ECML 2005*, volume 3720 of *Lecture Notes in Artificial Intelligence*, pages 242–254. Springer-Verlag.

Lin, H.-T. and Li, L. (2005b). Novel distance-based SVM kernels for infinite ensemble learning. In *Proceedings of the 12th International Conference on Neural Information Processing*, pages 761–766.

Mason, L., Bartlett, P. L., and Baxter, J. (2000a). Improved generalization through explicit optimization of margins. *Machine Learning*, 38(3):243–255.

Mason, L., Baxter, J., Bartlett, P., and Frean, M. (2000b). Functional gradient techniques for combining hypotheses. In Smola, A. J., Bartlett, P. L., Schölkopf, B., and Schuurmans, D., editors, *Advances in Large Margin Classifiers*, chapter 12, pages 221–246. MIT Press.

Merler, S., Caprile, B., and Furlanello, C. (2004). Bias-variance control via hard points shaving. *International Journal of Pattern Recognition and Artificial Intelligence*, 18(5):891–903.

Michie, D., Spiegelhalter, D. J., and Taylor, C. C., editors (1994). *Machine Learning, Neural and Statistical Classification*. Ellis Horwood.

Nash, S. G. and Sofer, A. (1996). *Linear and Nonlinear Programming*. McGraw-Hill series in industrial engineering and management science. McGraw-Hill.

Nicholson, A. (2002). *Generalization Error Estimates and Training Data Valuation*. PhD thesis, California Institute of Technology.

Platt, J. C., Cristianini, N., and Shawe-Taylor, J. (2000). Large margin DAGs for multiclass classification. In Solla, S. A., Leen, T. K., and Müller, K.-R., editors, *Advances in Neural Information Processing Systems 12*, pages 547–553. MIT Press.

Pratap, A. (2003). Data engine for machine learning research. Available at `http://www.work.caltech.edu/dengin/`.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, 1(1):81–106.

Rissanen, J. (1978). Modeling by shortest data description. *Automatica*, 14(5):465–471.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408.

Rosenblatt, F. (1962). *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan.

Schapire, R. E. (1997). Using output codes to boost multiclass learning problems. In Douglas H. Fisher, J., editor, *Machine Learning: Proceedings of the Fourteenth International Conference (ICML '97)*, pages 313–321. Morgan Kaufmann.

Schapire, R. E., Freund, Y., Bartlett, P., and Lee, W. S. (1998). Boosting the margin: A new explanation for the effectiveness of voting methods. *The Annals of Statistics*, 26(5):1651–1686.

Schmidhuber, J. (1997). Discovering neural nets with low Kolmogorov complexity and high generalization capability. *Neural Networks*, 10(5):857–873.

Shavlik, J. W., Mooney, R. J., and Towell, G. G. (1991). Symbolic and neural learning algorithms: An experimental comparison. *Machine Learning*, 6(2):111–143.

Solomonoff, R. J. (2003). The universal distribution and machine learning. *The Computer Journal*, 46(6):598–601.

Sun, Y., Todorovic, S., Li, J., and Wu, D. (2005). Unifying the error-correcting and output-code AdaBoost within the margin framework. In Raedt, L. D. and

Wrobel, S., editors, *ICML 2005: Proceedings of the 22nd International Conference on Machine Learning*, pages 872–879. Omnipress.

Vapnik, V. N. (1998). *Statistical Learning Theory*. Adaptive and Learning Systems for Signal Processing, Communications, and Control. John Wiley & Sons.

Vapnik, V. N. (1999). *The Nature of Statistical Learning Theory*. Springer-Verlag, 2nd edition.

Webb, G. I. (1996). Further experimental evidence against the utility of Occam's razor. *Journal of Artificial Intelligence Research*, 4:397–417.

Wolpert, D. H. and Macready, W. G. (1999). Self-dissimilarity: An empirically observable complexity measure. In Bar-Yam, Y., editor, *Unifying Themes in Complex Systems*, pages 626–643. Perseus Books.

Zhang, T. (2004). Solving large scale linear prediction problems using stochastic gradient descent algorithms. In Brodley, C. E., editor, *ICML 2004: Proceedings of the Twenty-First International Conference on Machine Learning*. Omnipress.

Zupan, B., Bohanec, M., Bratko, I., and Demšar, J. (1997). Machine learning by function decomposition. In Douglas H. Fisher, J., editor, *Machine Learning: Proceedings of the Fourteenth International Conference (ICML '97)*, pages 421–429. Morgan Kaufmann.

Zupan, B., Bratko, I., Bohanec, M., and Demšar, J. (2001). Function decomposition in machine learning. In Paliouras, G., Karkaletsis, V., and Spyropoulos, C. D., editors, *Machine Learning and Its Applications: Advanced Lectures*, volume 2049 of *Lecture Notes in Artificial Intelligence*, pages 71–101. Springer-Verlag.