Technical Reports on Mathematical and Computing Sciences: TR-C162
title: How Can Computer Science Contribute to Knowledge Discovery?
author: Osamu Watanabe
affiliation:
Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology
Meguro-ku Ookayama, Tokyo 152-8552, Japan
email: watanabe@is.titech.ac.jp
acknowledgments to financial supports:
The author is supported in part by a Grant-in-Aid for Scientific Research on Priority Areas "Discovery Science" from Japanese Goverment.

Abstract. *Knowledge discovery*, that is, to analyze a given massive data set and derive or discover some knowledge from it, has been becoming a quite important subject in several fields including computer science. Good softwares have been demanded for various knowledge discovery tasks. For such softwares, we often need to develop efficient algorithms for handling huge data sets. *Random sampling* is one of the important algorithmic methods for processing huge data sets. In this paper, we explain some random sampling techniques for speeding up learning algorithms and making them applicable to large data sets [15, 16, 4, 3]. We also show some algorithms obtained by using these techniques.

1. Introduction

For knowledge discovery, or more specifically, for a certain kind of data mining task, it would be quite helpful if we can use learning algorithms on a very large data set. In this paper, I explain some random sampling techniques we have developed for scaling up learning algorithms. But before going into a technical discussion, let us see in general what is expected to us computer scientists in knowledge discovery, and locate our problem in there.

For investigating various problems for knowledge discovery, we had in Japan a three-year research project "Discovery Science" (chair *Setsuo Arikawa*, Kyushu Univ.) from April 1998 to March 2001, sponsored by the Ministry of Education, Science, Sports and Culture. This is a quite large project involving many computer scientists and researchers in related fields, from philosophers and statisticians to scientists struggling with actual data. As a member of this project, I have learned many aspects of knowledge discovery. What I will explain below is my personal view on knowledge discovery that I have developed through my experience. Due to the space limit, I cannot explain examples in detail; I cannot even cite and list related papers either. Please refer to a coming book reporting our achievements in the Discovery Science Project that will be published from Springer. (On the other hand, for those explained in the following sections, please refer [15, 16, 4, 3] as their original sources.)

The ultimate goal of computer science is to provide good computer softwares (sometimes, computer systems) that would help activities of human beings. This is also true for knowledge discovery. We want good softwares helping us to analyze "complex data" and discover some "useful rules" hidden in the data. But our approach may differ depending on the type of the "complexity" of data we are given and also the type of the "usefulness" of a rule we are aiming for, and roughly speaking, there are two basic approaches.

The first approach is to develop (semi) automatic discovery systems. This is the case where a given data set consists of a large number of pieces of information of the same type and it is not required to obtain the very best rule explaining the data.

For example, *K. Yada* etal used several data mining tools and analyzed a huge sales data of a big drugstore chain to find out a purchase pattern of a "loyal customer", a customer who will become profitable to the stores. Here it may not be so important to obtain the best rule for this loyal customer prediction. As pointed out by *Yada* etal, a simple rule would be rather useful for business so long as it has a reasonable prediction power. An important and nontrivial task that they did was to design an efficient and robust software that converts actual purchase records to a data set applicable to data mining tools. In general, it is an interesting subject of software engineering to design such a converting software systematically. On the other hand, once we have a nice and clean data set from the actual data, we could use various learning algorithms. Note that these learning algorithms should work very efficiently and should be applicable to a very large data set. Here is the point where we can use random sampling techniques explained in the following sections.

On the other hand, it is sometimes the case that the complexity of data is due to not only its volume. Or we cannot assume any similarity in data. A typical example is a DNA sequence. Suppose that we have a set of complete DNA sequences of some single person. Certainly, the amount of information is huge, but each piece of information does not have the same role. In such a case, we cannot hope for any automatic discovery system. Then we aim for developing discovery systems that work with human intervention. This the second approach.

For example, *R. Honda* etal tried to develop a software that processes satellite images of lunar and finds out craters. They first tried a semi automatic software by using an unsupervised learning algorithm, Kohonen's self organizing map. But it turned out that the level of noise differs a lot depending on images. That is, the data is complex and not of the same type. Thus they changed a strategy and aimed for a software that works in collaboration with a human expert; then a quite successful software was obtained.

In the extreme case, the role of a software is to help human experts to make their discovery. This applies to the case where we want to find the very best rule from a limited number of examples. Note that even if the number of examples is limited, each example is complex, consisting of a large amount of data like the DNA sequence of one person. Thus, an efficient software that filters out and suggests important points would be quite useful. *Y. Ohsawa* introduced the notion of "key graph" to state potential motivations or causes of some events. He developed a software that extracts a key graph from data and applied it for the history of earthquakes to predict some risky areas. *H. Tsukimoto* etal made a software that processes f-MRI brain images and finds out activated areas in brain. By using his algorithm, it is possible to point out some candidate areas of activation from a very fuzzy original image data. Learning algorithms are also useful to filter out unnecessary attributes. *O. Maruyama*, *S. Miyano*, etal proposed a software that, using some learning algorithm, not only filters out irrelevant attributes but also creates a "view", a combination of attributes, that gives a specific way of understanding a given data. Visualization is also useful for human experts.

So far, we have been considering software developments. But techniques or methods developed in computer science themselves could be also useful for scientific discovery. *H. Imai* etal used several data compression algorithms to study the redundancy or incompressibility of DNA sequences, which may lead, in future, some important scientific discovery on DNA sequences. *S.*

Kurohashi etal used a digitalized Japanese dictionary to study the Japanese noun phrase "N_1 *no* N_2". Though "*no*" in Japanese roughly corresponds to "of" in English, its semantic role had not been clearly explained among Japanese language scholars. By computer aided search through some digitalized Japanese dictionary, *S. Kurohashi* etal discovered that the semantic role of "*no*" (of certain types) can be found in the definition statement of the noun N_2 in the dictionary. This may be a type of discovery that researchers other than computer scientists could not think of.

Now as we have seen, there are various types of interesting subjects from knowledge discovery. But in the following, we will focus on the case where we need to handle a large number of examples of the same type, and explain some sampling techniques that would be useful for such a case.

2. Sequential Sampling

First let us consider a very simple task of estimating the proportion of instances with a certain property in a given data set. More specifically, we consider an input data set D containing a huge amount of instances and a Boolean function B defined on instances in D. Then our problem is to estimate the proportion of instances x in D such that $B(x) = 1$ holds. (Let us assume that D is a multiset; that is, D may contain the same object more than once.) Clearly, the value p_B is obtained by counting the number of instances x in D such that $B(x) = 1$. But since we consider the situation where D is *huge*, it is impractical to go through all instances of D for computing p_B. A natural strategy that we can take in such a situation is random sampling. That is, we pick up some elements of D randomly and estimate the average value of B on these selected examples.

If we were asked the precise value of p_B, then trivially we would have to go through the whole data set D. Here consider the situation where we only need to get p_B "approximately". That is, it is enough to compute p_B within a certain error bound required in each context. Also due to the "randomness nature", we cannot always obtain a desired answer. Thus, we must be satisfied if our sampling algorithm yields a good approximation of p_B with "reasonable probability". We discuss this type of estimation problem. That is, we consider the following problem.

Problem 1. For given $\delta > 0$ and ε, $0 < \varepsilon < 1$, obtain an estimation $\widetilde{p_B}$ of p_B satisfying the following. (Here the probability is taken over the randomness of the sampling algorithm being used.)

$$\Pr[\ |\widetilde{p_B} - p_B| \le \varepsilon \cdot p_B\ ]\ >\ 1 - \delta. \tag{1}$$

This problem can be solved by the following simple sampling algorithm: Choose n instances from D uniformly at random, count the number m of instances x with $B(x) = 1$, and output the ratio m/n as $\widetilde{p_B}$. Here *sample size* n is a key factor for sampling, and for determining appropriate sample size, so called concentration bounds or large deviation bounds have been used. (see, e.g., [19]). For example, the Chernoff bound gives the following bound for n.

Theorem 2.1. For any $\delta > 0$ and ε, $0 < \varepsilon < 1$, if the sample size n satisfies the following inequality, then the output of the simple sampling algorithm mentioned above satisfies (1).

$$n\ >\ \frac{3}{\varepsilon^2 p_B} \ln\left(\frac{2}{\delta}\right).$$

```
program SampleAlg-1
  m ← 0;  n ← 0;
  while m < A do
    get x uniformly at random from D;
    m ← m + B(x);   n ← n + 1;
  output m/n as an approximation of p_B;
end.
```

Figure 1: Sequential Sampling Algorithm for Problem 1

Notice, however, that this bound is hard to use in practice, because for estimating an appropriate sample size n, we need to know p_B, which is what we want to estimate! Even if we could guess some appropriate value p for p_B, we need to use p such that $p < p_B$, just to be safe, Then if we underestimate p_B and use much smaller p, we have to use unnecessarily large sample size.

This problem may be avoidable if we perform sampling in an *on-line* or *sequential* fashion. That is, a sampling algorithm obtains examples sequentially one by one, and it determines from these examples whether it has already seen enough number of examples. Intuitively, from the examples seen so far, we can more or less obtain some knowledge on the input data set, and it may be possible to estimate the parameters required by the statistical bounds. Thus, we do not fix sample size in advance. Instead sample size is determined *adaptively.* Then we can use more appropriate sample size for the current input data set.

For Problem 1, such a *sequential sampling algorithm* has been proposed by Lipton etal [28, 29], which is stated in Figure 1. (Here we present a slightly simplified version.)

As we can see, the structure of the algorithm is simple. It runs until it sees more than A examples x with $B(x) = 1$. To complete the algorithm, we have to specify a way to determine A. For example, the Chernoff bound guarantees the following way. (In [28] the bound from the Central Limit Theorem is used, which gives a better sample size.)

Theorem 2.2. In the algorithm of Figure 1, we compute A by

$$A \;=\; \frac{3(1+\varepsilon)}{\varepsilon^2}\ln\left(\frac{2}{\delta}\right).$$

Then for any $\delta > 0$ and ε, $0 < \varepsilon < 1$, the output of the algorithm satisfies (1).

Note that A does not depend on p_B. Thus, we can execute the sampling algorithm without knowing p_B. On the other hand, the following bound on the algorithm's sample size is also provable. Comparing it with the bound of Theorem 2.1, we see that the number of required examples is almost the same as the situation where we *knew* p_B in advance.

Theorem 2.3. Consider the sample size n used by the algorithm of Figure 1. Then with probability $> 1 - \delta/2$, we have

$$\text{sample size } n \;\le\; \frac{3(1+\varepsilon)}{(1-\varepsilon)\varepsilon^2 p_B}\ln\left(\frac{2}{\delta}\right).$$

```
program SampleAlg-2
  m ← 0;  n ← 0;
  u ← 0;  α ← ∞;
  while u < α(1 + 1/ε) do
    get x uniformly at random from D;
    m ← m + B(x);   n ← n + 1;
    u ← m/n − 1/2;
    α ← √((1/2n) ln(n(n + 1)/δ));
  output u as an approximation of u_B;
end.
```

Figure 2: Sequential Adaptive Sampling Algorithm for Problem 2

While this is a typical example of sequential sampling, some sequential sampling algorithms have been developed recently (i) that can be applied to general tasks, and (ii) that have theoretical guarantees of correctness and performance [14, 15, 34]. Let us see here one application of such algorithms.

In some situations, we would like to estimate not $\widetilde{p_B}$ but some other value computed from $\widetilde{p_B}$. In [15], a general algorithm for achieving such a task is proposed. Here, as one example, we consider the problem of estimating $u_B = p_B - 1/2$. (This problem arises when implementing "boosting" techniques, which will be explained in the next section.)

Problem 2. For given $\delta > 0$ and ε, $0 < \varepsilon < 1$, obtain an estimation $\widetilde{u_B}$ of u_B satisfying the following. (For simplifying our discussion, let us assume here that $u_B \geq 0$.)

$$\Pr[\ |\widetilde{u_B} - u_B| \leq \varepsilon \cdot u_B\ ]\ >\ 1 - \delta. \tag{2}$$

Problem 2 is similar to Problem 1, but these two problems have different critical points. That is, while Problem 1 gets harder when p_B gets smaller, Problem 2 gets harder when $u_B = p_B - 1/2$ gets smaller. In other words, the closer p_B is to $1/2$, the more accurate estimation is necessary, and hence the more sample is needed. Thus, Problem 2 should be regarded as a different problem, and a new sampling algorithm is necessary.

For designing a sequential sampling algorithm for estimating u_B, one might want to modify the algorithm of Figure 1. For example, by replacing its while-condition "$m < A$" with "$m - n/2 < B$" and by choosing B appropriately, we may be able to satisfy the new approximation goal. But this naive approach does not work. Fortunately, however, we can deal with this problem by using a slightly more complicated stopping condition. In Figure 2, we state a sequential sampling algorithm for Problem 2. Note that the algorithm does not use any information on u_B; hence, we can use it without knowing u_B at all. On the other hand, as shown in the following theorem, the algorithm estimates u_B with the desired accuracy and confidence. Also the sample size is bounded, with high probability, by $\mathcal{O}(1/(\varepsilon u_B)^2 \log(1/(\delta u_B)))$, which bound is, ignoring the log factor, the same as the situation where u_B were known in advance.

Theorem 2.4. For any $\delta > 0$ and ε, $0 < \varepsilon < 1$, the output of the algorithm of Figure 2 satisfies (2). Furthermore, with probability more than $1 - \delta$, we have

$$\text{sample size } n \lesssim \frac{2(1+2\varepsilon)^2}{(\varepsilon u_B)^2} \ln\left(\frac{1}{\varepsilon\delta u_B}\right).$$

Some Comments on Related Work

Since the idea of "sequential sampling" or "sampling on-line" is quite natural and reasonable, it has been studied in various contexts.

First of all, we should note that statisticians had already made significant accomplishments on sequential sampling during World War II [36]. In fact, from their activities, a research area on sequential sampling — sequential analysis — had been formed in statistics. For example, performing random sampling until the number of "positive observations" exceeds a certain limit, has been studied in depth in statistics. For recent studies on sequential analysis, see, e.g., [24].

In computer science, adaptive sampling techniques have been studied in the database community. The algorithm of Figure 1 was proposed [28, 29] for estimating query size in relational databases. Later Haas and Swami [26] proposed an algorithm that performs better than this in some situations. More recently, Lynch [30] gave a rigorous analysis to the algorithms proposed in [28, 29].

One can see the spirit of adaptive sampling, i.e., to use instances observed so far for reducing a current and future computational task, in some algorithms proposed in the computational learning theory and machine learning community. For example, the Hoeffding race proposed by Maron and Moore [31] attempts to reduce a search space by removing candidates that are determined hopeless from the instances seen so far. A more general sequential local search has been proposed by Greiner [25]. More recently, sequential sampling algorithms have been developed for general data mining tasks and analyzed their performance both theoretically and experimentally; see, e.g., [15, 34].

In the study of randomized algorithms, some sequential sampling algorithms have been also proposed and analyzed [12].

3. An Application of Sequential Sampling to Boosting

Problem 2 discussed in the previous section arises when we want to design a simple learner based on random sampling. Such a simple learner may be too weak for making a reliable hypothesis. But it can be used as a *weak learner* in a boosting algorithm. (Here by a *learner* we mean an algorithm that produces a mechanism or a rule — hypothesis — for making a yes/no classification on a given example. Let us again D to denote a given data set, which contains a huge number of instances, each of which expresses some object called *example* in this paper. Note that each example has a yes/no classification label. That is, we are given the way to classify examples given in D, and our goal is to find a good hypothesis explaining these classifications.)

A *boosting algorithm* is a way to design a strong learner yielding a reliable hypothesis by using a weak learner. Almost all boosting algorithms follow some common outline. A boosting algorithm runs a weak learner several times, say T times, under distributions $\mu_1, ..., \mu_T$ that are slightly modified from the given distribution μ on D and collects *weak hypotheses* $h_1, ..., h_T$. A *final hypothesis* is built by combining these weak hypotheses. Here the key idea is to put more weight, when making a new weak hypothesis, to "problematic instances" for which the previous weak hypotheses perform poorly. That is, at the point when $h_1, ..., h_t$ have been obtained, the boosting algorithm computes a new distribution μ_{t+1} that puts more weight on those instances that have been misclassified by most of $h_1, ..., h_t$. Then a new hypothesis h_{t+1} produced by a weak learner on this distribution μ_{t+1} should be strong on those problematic instances, thereby improving the performance of the combined hypothesis built from $h_1, ..., h_{t+1}$.

Boosting algorithms differ typically on a weighting scheme used to compute modified distributions. In [23] an elegant weighting scheme was introduced by Freund and Schapire, by which they defined a boosting algorithm called AdaBoost. It was proved that AdaBoost has a nice "adaptive" property; that is, it is adaptive to the quality of an obtained weak hypothesis so that the boosting process can converge quickly when better weak hypotheses are obtained. Furthermore, many experimental results have shown that AdaBoost is indeed useful for designing a good learner for sevarl practical applications.

Now we would like to use AdaBoost to design a classification algorithm that can handle a huge data set. For this, we have to solve two algorithmic problems: (i) how to design a weak learner, and (ii) how to implement the boosting process.

Consider the first problem. AdaBoost does not specify its weak learner. Of course, a better learner yields a better hypothesis, which speeds up the boosting process. But a heavy weak learner is not appropriate for handing a large data set; for example, C4.5, a well-known decision tree making software, would be too costly to be used as a weak learner if it were used directly on the original huge data set D. Here random sampling can be used to reduce the data set size. For our discussion, let us take the following simplest approach: We consider a simple weak hypothesis so that the set $\mathcal{H}$ of all weak hypotheses remains relatively small. Then search *exhaustively* for a hypothesis $h \in \mathcal{H}$ that performs well on randomly sampled instances from D. For implementing this approach as a weak learning algorithm, Problem 2 (and its variation) becomes important.

Let us consider our simple approach in more detail. AdaBoost works with any weak learner except for one requirement. That is, for AdaBoost (and in fact any boosting algorithm) to work, it is necessary that a weak learner should (almost) always provide a hypothesis that is better than the random guess. Let $\mathrm{cor}_\mu(h)$ be the probability that a hypothesis h classifies correctly on instance of D that are generated randomly according to distribution μ. Then $\mathrm{adv}_\mu(h) = \mathrm{cor}_\mu(h) - 1/2$ is called the *advantage* of the hypothesis h. Note that the correct probability of the random guess is $1/2$; hence, the advantage of h shows how much h is better than the random guess. In order for AdaBoost to work, it is required that a weak learner should yield a hypothesis h_t (at each tth boosting step) whose advantage $\mathrm{adv}_{\mu_t}(h_t)$ is positive; of course, the larger advantage is the better. Then the task of a weak learner is to search for a hypothesis with (nearly) largest accuracy. We are considering a weak learner that uses random sampling to estimate $\mathrm{adv}_{\mu_t}(h)$ for each $h \in \mathcal{H}$ (and select the one with the best estimated advantage). This estimation is indeed our Problem 2. To guarantee that a selected hypothesis $h \in \mathcal{H}$ is nearly best (with a high probability), the sample size must be determined in terms of the best advantage γ; the more instances are needed if γ is close to 0 (in other words, the correct probability of the best hypothesis is close to $1/2$). That is, what is focused is the advantage of each hypothesis and not the correct probability. Hence, the problem needed to solve is Problem 2 and not Problem 1. Furthermore, the best advantage is not known in advance. Thus, this is the situation where sequential sampling is needed, and the algorithm of Figure 2 satisfies the purpose.

From Theorem 2.4, we can prove that the weak learner designed by using our sequential sampling algorithm needs to see roughly $\mathcal{O}(|\mathcal{H}|/\gamma^2)$ instances (ignoring some logarithmic factor) and runs within time proportional to this sample size. Thus, this weak learner runs with reasonable speed if both $|\mathcal{H}|$ and $1/\gamma$ are bounded within some range; in particular, an important point is that its running time does not depend on the size of the data set D. For example, we conducted some experiments [17] by using the set of *decision stumps*, one node decision trees, for the class $\mathcal{H}$. Our experiments show that the obtained weak learner runs in reasonable amount of time in most cases, independent from data set size.

Next consider the second problem, i.e., an implementation of boosting process. In the boosting process (of AdaBoost and any other boosting algorithms), it is necessary to generate *training examples* under distributions modified from the original distribution[1]. Thus, some implementation of this example generation procedure is necessary. So far, we have two implementation frameworks:- (i) *boosting by sub-sampling* and *boosting by filtering* [20].

In the sub-sampling framework, before starting the boosting process, a set S of a certain number of instances are chosen first from D, and only elements in S are used during the boosting process. More specifically, at each boosting step t, the modified distribution μ_t at this step is defined on S, and the table of probabilities $\mu_t(x)$ for all $x \in S$ is calculated. Then we run a weak learning algorithm, supplying examples chosen from S according to their probabilities. (In many situations, it is not necessary to use any learning algorithm. We can simply find a weak hypothesis h that is best on S under μ_t by calculating the correct probability $\mathrm{cor}_{\mu_t}(h) = \sum_{x \in S(h)} \mu_t(x)$, where $S(h) = \{\, x \in S \,:\, h \text{ is correct on } x \,\}$.) In the filtering framework, on the other hand, we run a weak learning algorithm directly on the original data set D. Whenever an example is requested by a weak learner, it is generated from D by using a "filtering procedure" that yields an instance of D under the modified distribution μ_t.

In the sub-sampling framework, the whole set S is used throughout the boosting process; that is, the sample size for a weak learner is fixed. Thus, the adaptivity of our proposed weak learner loses its meaning. Also in many practical situations, it may not be easy to determine the size of the training set S in advance. On the other hand, the filtering framework fits very well to our weak learner, because the weak learner can control the sample size. Unfortunately, however, the filtering framework cannot be used in AdaBoost; under its weighting scheme, the weight of some examples may get exponentially large, and the filtering procedure cannot produce examples within a reasonable amount of time.

In [16] we introduced a new weighting scheme, which is obtained from the one used in AdaBoost by simply limiting the weight of each example by 1. Under this weighting scheme, it is possible to follow the filtering framework. This boosting algorithm — MadaBoost — implemented in the filtering framework is stated in Figure 3. Some explanation may be helpful for understanding some of the statements. First look at the main program of MadaBoost. The statement "$(h_t, \gamma_t) \leftarrow$ WeakLearn(FiltEx(t))" means that (i) some weak learning algorithm is executed by supplying examples by using FiltEx(t), and then (ii) a hypothesis h_t with advantage

[1]Since we are assuming that the given data set D is huge, we may be able to assume that instances there already reflect the underlying distribution; thus, we may regard the uniform distribution on D as the original distribution.

program MadaBoost
  for $t \leftarrow 1$ **to** ∞ (until FiltEx terminates the iteration) **do**
    $(h_t, \gamma_t) \leftarrow$ WeakLearn(FiltEx(t))
    $\beta_t \leftarrow \sqrt{1 - 2\gamma_t/(1 + 2\gamma_t)}$;
  end-for
  output the following f_t:
  $f_t(x) = \mathrm{argmax}_{y \in Y} \sum_{i:h_i(x)=y} \log \frac{1}{\beta_i}$
end.

procedure FiltEx(t)
  loop-forever
    generate x uniformly at random from D;
    $w_t(x) \leftarrow \min\{1, \prod_{i=1}^{t} \beta^{\mathrm{cons}(h_i,x)}\}$;
    % $\mathrm{cons}(h_i, x) = 1$ if $h_i(x)$ is correct and -1 otherwise.
    with probability $w_t(x)$, output x and exit;
    if # of iterations exceeds some limit **then**
      exit and terminate the iteration of MadaBoost;
  end-loop
end.

Figure 3: MadaBoost (in the Filtering Framework)

γ_t ($= \mathrm{adv}_{\mu_t}(h_t)$) is obtained. For example, our simple weak learner based on the sequential sampling algorithm can be used here, which also gives a reasonable estimation of the advantage γ_t. Using this advantage γ_t, the parameter β_t is defined next, which is used for defining the weight (i.e., probability) of each example, as well as for determining the importance of the obtained hypothesis h_t. A function f_t defined at the termination of the main iteration is the final hypothesis; intuitively, this function makes a prediction for a given example x by taking the weighted majority vote of the classifications made by weak hypotheses $h_1, ..., h_t$. MadaBoost is exactly the same as AdaBoost up to this point. The difference is the way to compute the weight $w_t(x)$ for each example $x \in D$, which is defined and used in the procedure FiltEX. When FiltEX(t) is called, it draws examples x randomly from D and filter them according to their current weights $w_t(x)$. In AdaBoost, the weight $w_t(x)$ is computed as $\prod_{i=1}^{t} \beta^{\mathrm{cons}(h_i,x)}$; here we simply limit it by 1. Then it is much easier to get one example. Furthermore, if it gets hard to generate an example, then the boosting process (and the whole algorithm) can be terminated. This is because it is provable (under the new weighting scheme) that the obtained hypothesis is accurate enough when it gets hard to generate an example by this filtering procedure.

In [16] it is proved that the weighting scheme of MadaBoost still guarantees polynomial-time convergence. Unfortunately, MadaBoost's convergence speed that can be proved in [16] is exponentially slower than AdaBoost. However, our experiments [17] show that there is no significant difference on the convergence speed between AdaBoost and MadaBoost; that is, more or less the same number of boosting steps is sufficient. On the other hand, our experiments show that MadaBoost with our weak learner performs quite well on large data sets.

Some Comments on Related Work

The first boosting algorithm has been introduced by Schapire [33], for investigating a rather theoretical question asking the equivalence between the strong and weak PAC-learnability notions. But due to the success of AdaBoost for practical applications, several boosting techniques have been proposed since AdaBoost; see, e.g., the Proceedings of *the 13th Annual Conf. on Computational Learning Theory* (COLT'00).

Heuristics similar to boosting or related boosting have been also proposed and investigated experimentally; see, e.g., [7, 13, 1].

It has been said that boosting does not work well when error or noise exists. That is, the case when some of the instances in D have a wrong classification label. In particular, this problem seems serious in AdaBoost. As we mentioned above, the weighting scheme of AdaBoost changes weights rapidly, which makes AdaBoost too sensitive to erroneous examples. Again this problem is reduced by using a more moderate weighting scheme like MadaBoost. In fact, it is shown [16] that MadaBoost is robust to a standard statistical errors, errors that could be corrected by seeing sufficient number of instances. (Recall that D may be a multiset and one example may appear in D more than once. Then by seeing enough number of instances of D for the same example, we may be able to fix it even if some instance is labeled wrongly. See, e.g., [27] for the formal argument.) For solving this problem of AdaBoost, Freund [21] introduced a new weighting scheme and proposed a new boosting algorithm — BrownBoost —- which is also robust to the statistical errors of the above type. Unfortunately, however, even these algorithms may not be robust against errors that are not fixed by just looking many instances. One typical example is the case where the data set contains some exceptions, examples that should be treated as exceptions.

4. Random Sampling for Support Vector Machines

Boosting provide us with a powerful methodology for designing better learning algorithms. Unfortunately, however, boosting algorithms may not work sufficiently if a given data set contains many errors or exceptions. On the other hand, there is a popular classification mechanism — support vector machine (in short SVM) — that works well even if the data set contains some "outliers", i.e., erroneous examples or exceptions. Here again we show that random sampling can be used for designing an efficient SVM training algorithm. (*Remark.* One important point of the SVM approach is so called the "kernel method" that enables us to obtain hyperplane separation in a much higher dimension space than the original space. In this paper, this point is omitted, and we consider a simple hyperplane separation. See our original paper [3] for an extension to the kernel method. See also [4] that proposes another algorithm derived from a similar approach.)

First we explain basics concerning SVM. Since we only explain those necessary for our discussion, see, e.g., a good textbook [11] for more systematic and comprehensive explanation. A *support vector machine* (discussed here) is a classification mechanism that classifies examples by using a hyperplane. This time our data set D is a set of m instances in some n dimension space. Here again we consider the binary classification, and we assume that instances in D has a binary label $+1$ for *postive* and -1 for *negative* examples. Then SVM training is to compute a hyperplane separating positive and negative examples with the largest margin. Precisely, our goal is to solve the following optimization problem[2]. (Let us assume for a while that D has no erroneous instances and positive and negative examples can be separated by some hyperplane in the n dimension space. To simplify our discussion, we assume here that D is an ordinary set. That is, an example appears at most once as an instance in D; hence, there is no distinction between example and instance. Also we assume that examples in D are indexed as $\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_m$, where each $\boldsymbol{x}_i$ has a label y_i.)

Max Margin (P1)

$$\begin{array}{ll} \text{min.} & \frac{1}{2}\|\boldsymbol{w}\|^2 - (\theta_+ - \theta_-) \quad \text{w.r.t. } \boldsymbol{w} = (w_1, ..., w_n),\ \theta_+, \text{ and } \theta_-, \\ \text{s.t.} & \boldsymbol{w} \cdot \boldsymbol{x}_i \ \geq \ \theta_+ \quad \text{if } y_i = 1, \quad \text{and} \quad \boldsymbol{w} \cdot \boldsymbol{x}_i \ \leq \ \theta_- \quad \text{if } y_i = -1. \end{array}$$

This optimization problem is a quadratic programming (QP in short) problem. Though QP is in general polynomial-time solvable, existing (general) QP solvers are not efficient enough to solve a large QP problems. Here random sampling can be used. In fact, some random sampling techniques have been developed and used for such combinatorial optimization problems (see, e.g., [18]), and we simply use one of them. This technique yields a training algorithm that works particularly well when the dimensionality n (i.e., the number of attributes) is moderate but the size m of D is huge.

The idea of the random sampling technique is in fact similar to boosting. Pick up a certain number of examples from D and solve (P1) under the set of constraints corresponding to these examples. We choose examples randomly according to their "weights", where initially all examples are given the same weight. Clearly, the obtained local solution is, in general, not the global solution, and it does not satisfy some constraints; in other words, some examples are misclassified by the local solution. Then double the "weight" of such misclassified examples, and then pick up some examples again randomly according to their weights. If we iterate this process several rounds, the weight of "important examples", examples defining the solution hyperplane (which are called *support vectors*), would get increased, and hence, they are likely to be chosen. Note that once all support vectors are chosen at some round, then the local solution of this round is the real one, and the algorithm terminates at this point.

More specifically, we can in this way design a SVM training algorithm stated in Figure 4. Let us see the algorithm a bit more in detail. In the algorithm, $w(\boldsymbol{x})$ denotes the current weight of an example $\boldsymbol{x}$. Examples in D are drawn randomly from D so that the probability that each example $\boldsymbol{x}$ is chosen is proportional to its current weight $w(\boldsymbol{x})$. Let R be the set of examples chosen in this way, and we solve (P1) on R to obtain a local solution $(\boldsymbol{w}^*, \theta_+^*, \theta_-^*)$. A *violator* is simply an example that is misclassified by the obtained local solution. The parameter δ denotes a *combinatorial dimension*, a quantity expressing the complexity of a combinatorial optimization problem to be solved; for the problem (P1), we have $\delta \leq n+1$, which is from the fact that $n+1$ support vectors are enough to define the solution hyperplane. Note that the number of examples chosen for R is computed from n and δ $(= n+1)$ and independent from m, the size of D. Thus, our algorithm is appropriate if $m >> n$, i.e., the number of examples is much larger than the number of attributes.

[2]Here we follow [5] and use their formulation. But the above problem can be restated by using a single threshold parameter as given originally in [10].

program RandomSamplingSolver
  set $w(\boldsymbol{x}) \leftarrow 1$ for each $\boldsymbol{x} \in D$;
  $r \leftarrow 6\delta^2$;
  repeat
    $R \leftarrow$ choose r examples $\boldsymbol{x}$ from D randomly according to their weights $w(\boldsymbol{x})$;
    Solve (P1) for R and let $(\boldsymbol{w}^*, \theta_+^*, \theta_-^*)$ be its solution;
    $V \leftarrow$ the set of violators to $(\boldsymbol{w}^*, \theta_+^*, \theta_-^*)$;
    if $w(V) \leq w(D)/3\delta$ **then**
      double the weight $w(\boldsymbol{x})$ of $\boldsymbol{x} \in V$;
  until $V = \emptyset$;
  output the current solution;
end.

Figure 4: SVM Training Algorithm Based on Random Sampling

Note that the weight of violators get increased only when their total weight is at most one third of the total weight, that is, not so many violators exist. But we can prove that this situation does not occur often by using "Sampling Lemma" [8, 18]. Then the following bound follows from this fact.

Theorem 4.1. The average number of iterations executed in the algorithm of Figure 4 is bounded by $6\delta \ln m$.

Recall that $\delta \leq n + 1$; thus, this theorem shows that the algorithm needs to solve (P1) roughly $O(n \ln m)$ times. On the other hand, the size of R and hence the time needed to solve (P1) at each iteration is bounded by some polynomial in n.

Next consider the case that a given data set D contains "outliers", and no hyperplane can separate positive/negative examples given in D. In the SVM approach, this situation can be handled by considering "soft margin", that is, by relaxing constraints with slack variables. Precisely, we consider the following generalization of (P1).

$$
\begin{array}{ll}
\multicolumn{2}{l}{\underline{\text{Max Soft Margin (P2)}}} \\
\text{min.} & \dfrac{1}{2}\|\boldsymbol{w}\|^2 - (\theta_+ - \theta_-) + K \cdot \sum_i \xi_i \\
 & \text{w.r.t. } \boldsymbol{w} = (w_1, ..., w_n),\ \theta_+,\ \theta_-, \text{ and } \xi_1, ..., \xi_m \\
\text{s.t.} & \boldsymbol{w} \cdot \boldsymbol{x}_i \ \geq\ \theta_+ - \xi_i \quad \text{if } y_i = 1, \quad \boldsymbol{w} \cdot \boldsymbol{x}_i \ \leq\ \theta_- + \xi_i \quad \text{if } y_i = -1, \\
 & \text{and } \ \xi_i \ \geq\ 0.
\end{array}
$$

Here variables $\xi_i \geq 0$ are newly introduced, which express the penalty of $\boldsymbol{x}_i$ being misclassified by a hyperplane. That is, examples can be misclassified by a solution in this formulation, but the penalty $\boldsymbol{x}_i$ is added to the cost if $\boldsymbol{x}_i$ is misclassified. Intuitively, both the margin from the separating hyperplane and the cost of misclassifications are taken into account in this problem (P2). At this point, we can formally define our notion of "outliers". An *outlier* is an example $\boldsymbol{x}_i$ that is misclassified (in other words, with $\xi_i > 0$) by the solution of (P2). Note here that we use a parameter $K < 1$ for adjusting the degree of influence from misclassified examples, and that the solution of (P2) and the set of outliers depend on the choice of K.

Now our problem is to solve (P2). We assume that the parameter K is appropriately chosen [6]. We may still use the algorithm of Figure 4. The point we need to consider is only the choice of δ, the combinatorial dimension of (P2). Since (P2) is defined with $n+m+2$ variables, δ is bounded by $n+m+1$ by the straightforward generalization of the argument for (P1). But this bound is too large; in fact, the number of examples required for R becomes much larger than the original number m of examples! Fortunately, however, we can show [3] that δ is indeed bounded by $n+\ell+1$, where ℓ is the number of outliers. (The proof uses the geometrical interpretation of (P2) given by Bennett and Bredensteiner [5].) Then by using the bound of Theorem 4.1, we can conclude the following: the algorithm needs to solve (P2) roughly $O((n+\ell)\ln m)$ times, and the time needed to solve (P2) at each time is bounded by some polynomial in $n+\ell$.

Comments on Related Work

The present form of SVM was first proposed by Cortes and Vapnik [10]. Many algorithms and implementation techniques have been developed for training SVMs efficiently; see, e.g., [35, 6]. This is because solving QP for training SVMs is costly. Among speed-up techniques, those called "subset selection" [35] have been used as effective heuristics from the early stage of the SVM research. Roughly speaking, a *subset selection* is to divide the original QP problem into small pieces, thereby reducing the size of each QP problem. Well known subset selection techniques are chunking, decomposition, and sequential minimal optimization (SMO in short). In particular, SMO has become popular because it outperforms the others in several experiments. (See, e.g., [10, 32, 11] for the detail.) Though the performance of these subset selection techniques has been extensively examined, no theoretical guarantee has been given on the efficiency of algorithms based on these techniques. On the other hand, we can theoretically guarantee the efficiency of our algorithm based on random sampling. Compared with existing heuristics, our algorithm may not be efficient as it, but it can be used with the other heuristics to obtain an yet faster algorithm. Furthermore, using our weighting scheme, it may be possible to identify some of the outliers in earlier stages, thereby making the problem easier to solve by removing them.

The idea of using random sampling to design efficient randomized algorithms was first introduced by Clarkson [8]. This approach has been used for solving various combinatorial optimization problems. Indeed a similar idea has been used [2] to design an efficient randomized algorithm for QP. More recently, Gärtner and Welzl [18] introduced a general framework for discussing such randomized sampling techniques. Our algorithm of Figurre 4 and its analysis for (P1) are immediate from their general argument.

References

[1] N. Abe and H. Mamitsuka, Query learning strategies using boosting and bagging, in *Proc. the 15th Int'l Conf. on Machine Learning* (ICML'00), 1–9, 1998.

[2] I. Adler and R. Shamir, A randomized scheme for speeding up algorithms for linear and convex programming with high constraints-to-variable ratio, *Math. Programming* 61, 39—52, 1993.

[3] J. Balcázar, Y. Dai, and O. Watanabe, Provably fast training algorithms for support vector machines, in *Proc. the first IEEE Int'l Conf. on Data Mining*, to appear.

[4] J. Balcázar, Y. Dai, and O. Watanabe, Random sampling techniques for training support vector machines: For primal-form maximal-margin classifiers, in *Proc. the 12th Int'l Conf. on Algorithmic Learning Theory* (ALT'01), to appear.

[5] K.P. Bennett and E.J. Bredensteiner, Duality and geometry in SVM classifiers, in *Proc. the 17th Int'l Conf. on Machine Learning* (ICML'2000), 57–64, 2000.

[6] P.S. Bradley, O.L. Mangasarian, and D.R. Musicant, Optimization methods in massive datasets, in *Handbook of Massive Datasets* (J. Abello , P.M. Pardalos, and M.G.C. Resende, eds.), Kluwer Academic Pub., to appear.

[7] L. Breiman, Pasting small votes for classification in large databases and on-line, *Machine Learning* 36, 85–103, 1999.

[8] K.L. Clarkson, Las Vegas algorithms for linear and integer programming, *J.ACM* 42, 488—499, 1995.

[9] M. Collins, R.E. Schapire, and Y. Singer, Logistic regression, AdaBoost and Bregman Distance, in *Proc. the 13th Annual Conf. on Comput. Learning Theory* (COLT'00), 158–169, 2000.

[10] C. Cortes and V. Vapnik, Support-vector networks, *Machine Learning* 20, 273–297, 1995.

[11] N. Cristianini and J. Shawe-Taylor, *An Introduction to Support Vector Machines*, Cambridge Univ. Press, 2000.

[12] P. Dagum, R. Karp, M. Luby, and S. Ross, An optimal algorithm for monte carlo estimation, *SIAM J. Comput.* 29(5), 1484–1496, 2000.

[13] T.G. Dietterich, An experimental comparison of three methods for constructing ensembles of decision trees: bagging, boosting and randomization, *Machine Learning* 32, 1–22, 1998.

[14] C. Domingo, R. Gavaldà, and O. Watanabe, Practical algorithms for on-line selection, in *Proc. the first Intl. Conf. on Discovery Science* (DS'98), Lecture Notes in AI 1532, 150–161, 1998.

[15] C. Domingo, R. Gavaldà, and O. Watanabe, Adaptive sampling methods for scaling up knowledge discovery algorithms, in *Proc. the 2nd Intl. Conf. on Discovery Science* (DS'99), Lecture Notes in AI, 172–183, 1999. (The final version will appear in *J. Knowledge Discovery and Data Mining*.)

[16] C. Domingo and O. Watanabe, MadaBoost: A modification of AdaBoost, in *Proc. the 13th Annual Conf. on Comput. Learning Theory* (COLT'00), 180–189, 2000.

[17] C. Domingo and O. Watanabe, Scaling up a boosting-based learner via adaptive sampling, in *Proc. of Knowledge Discovery and Data Mining* (PAKDD'00), Lecture Notes in AI 1805, 317–328, 2000.

[18] B. Gärtner and E. Welzl, A simple sampling lemma: Analysis and applications in geometric optimization, *Discr. Comput. Geometry*, to appear. (Also available from `http://www.inf.ethz.ch/personal/gaertner/publications.html`.)

[19] W. Feller, *An Introduction to Probability Theory and its Applications* (Third Edition), John Wiley & Sons, 1968.

[20] Y. Freund, Boosting a weak learning algorithm by majority, *Information and Computation* 121(2), 256–285, 1995.

[21] Y. Freund, An adaptive version of the boost by majority algorithm, in *Proc. the 12th Annual Conf. on Comput. Learning Theory* (COLT'99), 102–113, 1999.

[22] J. Friedman, T. Hastie, and R. Tibshirani, Additive logistic regression: a statistical view of boosting, Technical Report, 1998.

[23] Y. Freund and R.E. Schapire, A decision-theoretic generalization of on-line learning and an application to boosting, *J. Comput. Syst. Sci.* 55(1), 119–139, 1997.

[24] B.K. Ghosh and P.K. Sen eds., *Handbook of Sequential Analysis*, Marcel Dekker, 1991.

[25] R. Greiner, PALO: a probabilistic hill-climbing algorithm, *Artificial Intelligence* 84, 177–204, 1996.

[26] P. Haas and A. Swami, Sequential sampling, procedures for query size estimation, *IBM Research Report* RJ 9101(80915), 1992.

[27] M. Kearns, Efficient noise-tolerant learning from statistical queries, in *Proc. the 25th Annual ACM Sympos. on Theory of Comput.* (STOC'93), 392–401, 1993.

[28] R.J. Lipton, J.F. Naughton, D.A. Schneider, and S. Seshadri, Efficient sampling strategies for relational database operations, *Theoret. Comput. Sci.* 116, pp.195–226, 1993.

[29] R.J. Lipton and J.F. Naughton, Query size estimation by adaptive sampling, *J. Comput. and Syst. Sci.* 51, 18–25, 1995.

[30] J.F. Lynch, Analysis and application of adaptive sampling, in *Proc. the 19th ACM Sympos. on Principles of Database Systems* (PODS'99), 260–267, 1999.

[31] O. Maron and A. Moore, Hoeffding races: accelerating model selection search for classification and function approximation, in *Proc. Advances in Neural Information Process. Systems* (NIPS'94), 59–66, 1994.

[32] J. Platt, Fast training of support vector machines using sequential minimal optimization, in *Advances in Kernel Methods – Support Vector Learning* (B. Scholkopf, C.J.C. Burges, and A.J. Smola, eds.), MIT Press, 185–208, 1999.

[33] R.E. Schapire, The strength of weak learnability, *Machine Learning* 5(2), 197–227, 1990.

[34] T. Scheffer and S. Wrobel, A sequential sampling algorithm for a general class of utility criteria, in *Proc. the 6th ACM Intl. Conf. on Knowledge Discovery and Data Mining* (KDD'00), 2000.

[35] A.J. Smola and B. Scholkopf, A tutorial on support vector regression, NeuroCOLT Technical Report NC-TR-98-030, Royal Holloway College, Univ. London, 1998.

[36] A. Wald, *Sequential Analysis*, John Wiley & Sons, 1947.