Machine Learning, 6, 145–160 (1991)
© 1991 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

A "*PAC*" Algorithm for Making Feature Maps

PHILIP LAIRD (LAIRD@PLUTO.ARC.NASA.GOV)
NASA Ames Research Center, Mail Stop 244-17, Moffett Field, CA 94035

EVAN GAMBLE
Sterling Federal Systems, Inc. and NASA Ames Research Center

Editor: David Haussler

Abstract. Kohonen and others have devised network algorithms for computing so-called *topological feature maps*. We describe a new algorithm, called the *CDF-Inversion* (CDFI) Algorithm, that can be used to learn feature maps and, in the process, approximate an unknown probability distribution to within any specified accuracy. The primary advantages of the algorithm over previous feature-map algorithms are that it is simple enough to analyze mathematically for correctness and efficiency, and that it distributes the points of the map evenly, in a sense that can be made rigorous. Like other vector-quantization algorithms it is potentially useful for many applications, including monitoring and statistical modeling. While not a network algorithm, the CDFI algorithm is well-suited to implementation on parallel computers.

Keywords. Topological feature-map, PAC-learning, probability distribution.

1. Introduction

1.1. Kohonen's algorithm

Let X be a large vector space, and assume there is a probability distribution $\mathbf{p}$ that assigns a finite probability $\mathbf{p}(x)$ to each vector x in X. A well-known algorithm discovered by Kohonen (1982, 1984) produces what are called *self-organizing topological feature maps* of X. Roughly, a feature map partitions the space X into a predetermined number of subsets (called *regions*), and chooses a representative vector from each region. The "map" (regions plus representatives) should have two properties: (1) the probability density $\mathbf{p}(R)$ of each region R should be approximately the same; and (2) certain topological relationships among the representative vectors should be preserved, regardless of the actual distribution. Aside from this broad characterization, there seems to be no agreement about the precise definition of a *correct* feature map.

A brief overview of Kohonen's algorithm is as follows. The algorithm is given a graph $G = (V, E)$ of nodes $V = \{v_1, \ldots, v_r\}$ and edges $E \subseteq V \times V$. The task is to map each node $v \in V$ to an "appropriate" location (vector) $x_v \in X$ in the vector space. Initially each node is mapped an arbitrary vector in X.

Ignore the edges E for the moment, and consider only the nodes. As input, the algorithm receives vectors from X drawn at random with replacement according to the unknown probability distribution $\mathbf{p}$. Let x_* be the next input vector drawn from this distribution. The node v_c whose current map location x_{v_c} is closest to x_* in X is determined. Then the location x_{v_c} is changed by moving it closer to the input vector:

$$x_{v_c} \leftarrow x_{v_c} + \alpha(x_* - x_{v_c}).$$

The scale factor $1 < \alpha < 0$ is gradually reduced to zero as more input vectors are processed; when it reaches zero, the map is complete.

In the final map, the nodes should be well distributed throughout the vector space. "Well distributed" means that each node v represents a set of vectors in the neighborhood of its location x_v in the vector space, and that the total probability of the vectors associated with v are about the same for each v. For example, if there are r nodes in V then each node governs a neighborhood in X of probability about $1/r$. In Kohonen's feature maps, the neighborhood of X governed by the node v is taken to be the set of vectors for which x_v is the nearest neighbor among the set of r vectors.

Consider now the topology of the graph G as defined by the edge relations E. This topology can be transferred to the vector space X if for each edge (v_1, v_2) in E we construct the corresponding edge (x_{v_1}, x_{v_2}) in the map. In addition to the distribution requirement, we also stipulate that the edge relation on G should be preserved by the mapping to X. For example, if G is a rectangular grid, then the resulting graph in X should likewise have the general form of a grid. To accomplish this, Kohonen's algorithm moves, not only the location of the vector v_c closest to the input, but also the locations of those vectors adjacent to v_c according to E (i.e., joined by an edge to v_c). It is remarkable how effectively this simple procedure transfers the general shape of G to the vector space X, even when the node locations are initially random.

Kohonen's algorithm and its variants have been used successfully in a number of applications, e.g., (Kohonen, 1988; Nasrabadi & Feng, 1988; Ritter & Schulten, 1988). Noteworthy characteristics of his algorithm include the fact that it requires very little storage and that each processor performs only very simple calculations. As a model of how neurons are adaptively mapped in response to sensory information, it is also of interest to biologists.

A major limitation of all previously known feature-map algorithms is that they are very difficult to analyze. Such analysis is important, since applications often depend on both the speed of the algorithm and the accuracy of the resulting map. It happens that both theoretical and experimental studies of Kohonen's algorithm have shown that his procedure is not ideal: there is a persistent tendency for nodes to cluster excessively in regions of lower probability density (Hecht-Nielsen, 1987; Ritter & Schulten, 1986). As a result, applications using Kohonen's algorithm have sometimes been modified in *ad hoc* ways to correct for this deficit, (e.g., DiSieno, 1988).

1.2. Feature maps: A definition

We define a feature map as follows. Let $G = (X, E_G)$ be a graph (directed or undirected) with nodes X and edges $E_G \subseteq X \times X$. Let $M = (Y, E_M)$ be another graph such that $|Y| \leq |X|$. ϵ is a parameter in the range $0 < \epsilon < 1/|Y|$, and $\mathbf{p}$ is a probability distribution on the set of nodes X.

Let ϕ: $X \rightarrow Y$ be a surjective function from X onto Y. Let $\equiv$ be the equivalence relation on X such that $x \equiv x'$ iff $\phi(x) = \phi(x')$. The $\equiv$-equivalence classes on X induce a quotient graph, $G/_{\equiv}$ whose nodes are the $\equiv$-equivalence classes, and whose edges connect classes R_1 and R_2 iff there is an edge in E_G between some X-node in R_1 and one in R_2.

We say that ϕ *is a feature map of* G (with respect to M, $\mathbf{p}$, and ϵ) if

- M is isomorphic to a subgraph of $G/_{\equiv}$; and
- for every $\equiv$-class R over X, $\mathbf{p}(R)$, the total probability of the set of nodes in R, is equal to $1/|Y| \pm \epsilon$.

Intuitively, the mapping ϕ partitions the nodes of the graph G into $|Y|$ approximately equiprobable regions, in such a way that the edge relations in M are preserved in the quotient graph.

A *feature map problem* consists of five parts:

- the graphs G and M;
- the *accuracy parameter* $0 < \epsilon < 1/|Y|$;
- a *confidence parameter* δ such that $0 < \delta < 1$;
- a fixed, but unknown, probability distribution $\mathbf{d}$ over the nodes X of G. This distribution is observed by means of an infinite sequence of independent, random variables, $\{\mathbf{x}_i, i \geq 1\}$, each selected according to $\mathbf{p}$; the value of each $\mathbf{x}_i$ is a node in X.

A general algorithm for the feature map problem will accept as input the graphs G and M, the parameters ϵ and δ, and the stream $\{\mathbf{x}_i, i \geq 1\}$, and with probability at least $1 - \delta$, it will construct a feature map ϕ of G with respect to M and halt. The samples $\mathbf{x}_i$ are obtained by the algorithm in sequence and cost one unit of time each to read.

As a minimum complexity requirement, a feature map algorithm should run in time polynomial in the size of the input and in the values $1/\epsilon$ and $1/\delta$. Since in practice G is often quite large, an algorithm requiring time that is polynomial in the size of G may not be fast enough for applications. Moreover, the general feature map problem has an NP-complete subgraph-isomorphism problem embedded in it. For these reasons, feature map algorithms are usually designed to solve particular cases of the general problem. G, for example, is often limited to a family of graphs that can be encoded in $\mathcal{O}(\text{polylog } |X|)$ bits.

Instances of the feature map problem may, of course, have no solution. For example, this is the case if no partition of the nodes of G simultaneously satisfies the equal-probability and subgraph-isomorphism requirements. A general algorithm will recognize when a given feature map problem has no solution, print *fail*, and halt. In practice, however, algorithms are designed to compute some mapping that may not be a correct feature map but is nonetheless useful for the intended application.

In all learning problems, the choice of representation is an important consideration, because by choosing the class of hypotheses or concepts that a learning algorithm may output, one trades expressiveness (and, with it, accuracy) against the computational requirements for finding the best hypothesis in the class to describe the input information. When learning feature maps, the same choice comes into play. If $|G|$ is large, the number of possible maps ϕ is huge, and some feature maps may require more than polynomial time just to write them down. As a practical matter, therefore, we have to limit the family of mappings ϕ that we consider as potential feature maps. Typically this means admitting only classes of subsets of X that can be represented efficiently with some encoding. Doing so, however, may mean failure to find a correct feature map, even if one exists.

For example, let G be an undirected graph consisting of the four points (1, 1), (1, 2), (2, 1), and (2, 2), connected by edges so as to form a square in the Cartesian plane. Assume **p** assigns these points the probabilities 1/2, 1/6, 1/6, and 1/6, respectively. Let M be the (undirected) graph with two nodes y_1 and y_2 joined by an edge. If we admit only axis-parallel (orthogonal) subsets of the nodes of G, such as $\{(1, 1), (1, 2)\}$ or $\{(1, 2), (2, 2)\}$, then if $\epsilon < 1/6$, no feature map can be found. A correct feature map exists, however, for arbitrary ϵ: map the node (1, 1) to y_1 and the remaining nodes to y_2.

1.3. Summary of the results

The main result of this article is an algorithm (called the *CDFI* aglorithm) that inverts a univariate cumulative probability distribution function in a particular way. We show how this simple algorithm can be used to obtain feature maps efficiently over one-dimensional vector spaces. Unlike previously published feature map algorithms, ours is simple enough to allow formal analysis for correctness and computational complexity. Moreover, while not a "neural-network" algorithm, the CDFI procedure is naturally parallelizable in a straightforward way using a feasibly small number of processors. We then show how one can apply this algorithm to obtain feature maps of Euclidean n-spaces (the ones most often used in applications). Examples of maps obtained in this way are exhibited.

The results of this paper were originally obtained within the context of NASA applications research, while studying different approaches to the *unsupervised learning problem*. "Learning" here refers to the fact that the process must examine a stream of data and formulate a stochastic model reducing the informational complexity contained in that data. The quality factors of an algorithm for this problem include:

- time efficiency: the algorithm should converge quickly.
- space efficiency: the algorithm should require only a small amount of storage. This is especially important for applications intended for space flight.
- accuracy: the resulting regions should differ in probability from one another by no more than some arbitrary specified error.
- confidence: the knowledge that, even though stochastic events may occasionally cause the algorithm to fail to achieve the above goals, the likelihood of such a failure can be made arbitrarily small.
- robustness: the ability to achieve these properties independently of the actual distribution over the space.

Learning algorithms with these properties (sometimes called *PAC* algorithms) have been a topic of active research in recent years, beginning with the work of Valiant (1984).

2. The CDFI algorithm

Let $G = (X, E_G)$ be an undirected graph with N nodes X, labeled for convenience by the integers $\{1, \ldots, N\}$, and E_G, the set of edges $(i, i + 1)$, $1 \leq i \leq N - 1$. As noted earlier, N is often quite large—too large to store and too large to enumerate. The input for our algorithm includes an infinite stream $x_1, x_2, \ldots$ of integers in X drawn independently and randomly according to some unknown multinomial probability distribution on X. The probability that the integer $x \in X$ will be presented next is written $\mathbf{p}(x)$. The cumulative probability function, $\mathbf{P}(x)$, gives the probability that the next integer will be x or less: $\mathbf{P}(x) = \Sigma_{y \le x} \mathbf{p}(y)$.

The target graph M is, likewise, an undirected graph of h nodes (where $1 < h < N$), joined by edges $(i, i + 1)$ into a line. Forming a feature map, then, consists of partitioning the set of integers from 1 to N into h regions of nearly equal probability. To state it equivalently but in somewhat different terms, let $m_0, m_1, \ldots, m_h$ be integers such that $m_0 = 0$, $m_h = N$, and $m_{i-1} < m_i$ for $1 \le i \le h$. Let R_i be the set of integers belonging to the subinterval $(m_{i-1}, m_i]$. We write $\mathbf{p}(R_i)$ for the probability that the next input belongs to R_i. Then the m_i are to be chosen so that

$$\mathbf{p}(R_i) = \frac{1}{h} \pm \epsilon. \tag{1}$$

In practice, two circumstances may cause our algorithm to fail. One is plain bad luck: if the algorithm bases its calculations on a highly unrepresentative sample of X, then (1) may not be satisfied. But probability theory shows that "most" samples of sufficient size are representative, and that we can quantify the likelihood of an improbable sample. By these techniques the algorithm limits the probability of this type of failure on any random run to an arbitrarily small fraction $\delta > 0$ specified by the user. The other circumstance is poor spread: if a few symbols in X have very high probability, then it may be impossible to satisfy (1). For example, it may happen that a single symbol occurs with probability one. In such situations, some of the regions found by our algorithm may not satisfy (1). As noted above, we consider poor spread as an anomalous problem instance and shall be content to have our algorithm report failure in such cases.

The algorithm is given the two parameters: δ and ϵ, satisfying $0 < \delta < 1$ and $0 < \epsilon < 1/h$. δ is an upper bound on the likelihood that one or more of the intervals constructed by the algorithm is unacceptable (too big or small), and ϵ is the maximum permitted variation in the interval probabilities, as in equation 1.

For $0 \le i \le h$, we define the sequence $m(0), m(1), \ldots, m(h)$ of $h + 1$ integers as follows:

$$m(i) = \min\{x \in X \cup \{0\} \mid \mathbf{P}(x) \ge i/h\}.$$

(Note that $m(0) = 0$.) Intuitively, $m(i)$ is the smallest integer in X for which the likelihood is at least i/h that the next integer we observe will be $m(i)$ or smaller. For example, $m(h/2)$ is the statistical median of X. The output of the CDFI Algorithm is a set of estimates $\hat{m}(i)$ for the values of $m(i)$. Given these values, we can take as intervals $R_i = (m(i-1), m(i)]$ (for $1 \le i \le h$). If α is the maximum probability of any node in X, then there must be some node whose cumulative probability is between i/h and $(i/h) + \alpha$. Hence $\mathbf{P}(m(i)) \le (i/h) + \alpha$. Since $\mathbf{P}(R_i) = \mathbf{P}(m(i)) - \mathbf{P}(m(i-1))$, we have

$$\max\left(\frac{1}{h} - \alpha, 0\right) \le \mathbf{P}(R_i) \le \min\left(\frac{1}{h} + \alpha, 1\right).$$

Hence, unless some node has a probability larger than ϵ, each of the regions R_i have probability within ϵ of $1/h$. An interval R_i is said to be *unacceptable* if $|\mathbf{p}(R_i) - i/h| > \epsilon$, and *acceptable* otherwise. Our objective is to see that *all* intervals R_i computed from the output of the CDFI algorithm are acceptable, with probability at least $1 - \delta$. We shall find that the following condition ensures that a solution is possible:

Mappability Condition: *For no $x \in X$ is $\mathbf{p}(x) \geq \epsilon/2$.*

2.1. *Estimating m(i)*

Consider the problem of estimating $m(i)$ (for $1 \leq i \leq h$) such that condition (1) holds with the specified confidence. If space efficiency were not a consideration, we could use a very simple histogram procedure that obtains a sample $x_1, \ldots, x_\nu$ from the distribution and determines the smallest value $\hat{m}(i)$ in the sample such that a proportion of at least i/h of the sample are no greater than $\hat{m}(i)$. The sample size ν required to satisfy the accuracy and confidence requirements simultaneously for each of the $\hat{m}(i)$ can be determined by statistical techniques based on the uniform convergence of random variables to their expectations (Pollard, 1984; Haussler, 1990). The time required to find the element $\hat{m}(i)$ is then $\Theta(\nu)$. However, the cost of storing all ν sample points is prohibitive for many purposes. We shall, therefore, seek an *incremental* algorithm, one that stores only a small, constant number of sample points and whose space requirements are $\mathcal{O}(\log \nu)$ instead of $\mathcal{O}(\nu)$. Such an algorithm can be used even on a small microprocessor with limited memory, for a wide range of values of the parameters h, ϵ, and δ.

Also, instead of a loop that runs the procedure to estimate $m(i)$ h times (one for each value of i), we envision a procedure of which h copies can all execute in parallel and share the same sample data. With h up to about 1000, today's SIMD architectures can readily handle this amount of parallelism.

Now suppose we could ask an "oracle" for the cumulative probability $\mathbf{P}(x)$ of any $x \in X$. Then a straightforward search for the least x such that $\mathbf{P}(x) > i/h$ would find $m(i)$ easily and quickly. Unhappily, we do not know how to obtain such an oracle. Instead, we shall assume, and later construct, an oracle[1] $\mathcal{P}$ with the following behavior.

The oracle $\mathcal{P}$:

- Input to the oracle: values i, h such that $1 \leq i \leq h$; a symbol $x \in X$; and the parameters ϵ and δ.
- Output from the oracle: indication of one of the following:
  A_1: $\mathbf{P}(x) < i/h - \epsilon/2$ with probability at least $1 - \delta$.
  A_2: $\mathbf{P}(x) \geq i/h$ with probability at least $1 - \delta$.

Roughly, the oracle takes as inputs x, i, and h and in return tells us, with high probability, whether $\mathbf{P}(x)$ is below i/h or not. Also, the oracle admits that it can make mistakes, but it bounds by δ the probability that its response is erroneous.

```
*Algorithm 1* (Search Algorithm)
*Input*: $N, i, h, \epsilon, \delta$.
*Output*: A symbol in $X$ comprising an estimate of $m(i)$.
*Procedure*:
  1. Initialize: $x := \lfloor N/2 \rfloor$. $L := 0$. $U := N$. $\delta' = \delta/(h\lceil \log N \rceil)$.
  2. While $U - L > 1$:
      2.1 If $\mathcal{P}(x, i, h \mid \epsilon, \delta')$ indicates that $\mathbf{P}(x) < i/h - \epsilon/2$ (outcome $A_1$), then set
          $L := x$ and $x := x + \lceil (U - x)/2 \rceil$.
      2.2 Else set $U := x$ and $x := x - \lfloor (x - L)/2 \rfloor$.
  3. Output $U$ and terminate.
```

Figure 1. The search algorithm.

Based on this oracle, we define the Search Algorithm (Figure 1) that determines $\hat{m}(i)$ for one value of i. As noted above, the h executions (one for each $1 \leq i \leq h$) can occur in parallel, since they do not interact with one another except for the sharing of input examples x_i.

How good are the estimates $\hat{m}(i)$ resulting from this algorithm? The following useful lemma answers this question.

Lemma 1. *When Algorithm 1 halts, with probability at least* $1 - (\delta/h)$ *it emits a value* x *such that*

(i) $\mathbf{P}(x) \geq i/h - \epsilon/2$, *and*
(ii) $\mathbf{P}(x - 1) < i/h$.

Proof. First assume that the oracle never makes a mistake. Note that at termination U is the emitted value x, and $L = U - 1$. If $L = 0$ then condition (ii) clearly holds since $\mathbf{P}(L) = 0$. Otherwise, L is modified at least once by step 2.1. Each such change to L occurs after the oracle indicates that $\mathbf{P}(L) < i/h$; at termination, therefore, (ii) holds.

Similarly, condition (i) holds if $x = N$ (i.e., if U is never changed during the algorithm). Otherwise, U is last assigned in the algorithm (step 2.2) after the oracle has responded with outcome A_2. In either case, $\mathbf{P}(U) \geq i/h - \epsilon/2$.

The oracle may make a mistake on any answer with probability $\delta/h \lceil \log N \rceil$. Because of the binary-search strategy, at most $\lceil \log N \rceil$ calls are made on the oracle. Thus the probability of one or more erroneous responses from the oracle is no greater than δ/h. Hence with probability $\geq 1 - (\delta/h)$ all oracle responses are correct, and the conditions (i) and (ii) both hold. □

For each i the likelihood that the conditions of the lemma fail to hold is at most δ/h. Thus the probability is at most δ that the conditions fail on one or more of the h executions of the algorithm. The confidence, therefore, is $1 - \delta$, that all values of $\hat{m}(i)$ obtained from Algorithm 1 satisfy Lemma 1. Note also that Lemma 1 is valid whether or not the Mappability Condition is true. When the Mappability Condition holds, we also have condition (1) as a corollary.

Corollary 2. *Suppose that the Mappability Condition holds, and that Algorithm 1 is executed for each* $1 \le i \le h$. *Then, with confidence* $1 - \delta$, *for each of the regions* $R_i = (\hat{m}(i-1), \hat{m}(i)]$,

$$\mathbf{p}(R_i) = 1/h \pm \epsilon. \tag{2}$$

Proof. From the lemma, $\mathbf{P}(\hat{m}(i)) \ge i/h - \epsilon/2$, and $\hat{m}(i)$ is the least x such that A_2 holds. Suppose that $\mathbf{P}(\hat{m}(i)) \ge i/h + \epsilon/2$. Then either the Mappability Condition or condition (ii) in the lemma must be violated. Thus $|\mathbf{P}(\hat{m}(i)) - i/h| \le \epsilon/2$ for all $1 \le i \le h$ (with confidence $1 - \delta$), and the result follows directly. □

Consider briefly the time complexity of Algorithm 1. For each of the h values of i, there are $\mathcal{O}(\log N)$ repetitions of the **while** loop, and each repetition requires an oracle call and $\mathcal{O}(\log N)$ time to copy $\mathcal{O}(\log n)$ bits into L or U. Thus the time to compute $\hat{m}(i)$ on a serial machine is $\mathcal{O}((\log N)(\log N + t))$, where t is the cost of each oracle call. In practice we have found that the time t for the oracle call dominates the total run time.

2.2. *Constructing the oracle* $\mathcal{P}$

Given an integer x, the oracle's task is to determine (within the specified confidence) whether $\mathbf{P}(x)$ is $\le i/h - \epsilon/2$ (response A_1) or $\ge i/h$ (response A_2). If $i/h - \epsilon/2 < \mathbf{P}(x) < i/h$, either response will do. The case where $i = h$ is special and treated in the appendix; in this section, we assume $1 \le i < h$.

Consider a process that obtains a sample point x_i from the distribution and responds "Heads" if $x_i \le x$ and "Tails" otherwise. Then the probability of the "Heads" response is $\mathbf{P}(x)$, and the probability of the "Tails" response is $1 - \mathbf{P}(x)$. In effect, this process is a coin flip with a probability of $\mathbf{P}(x)$ of getting heads ("1"). Applying Hoeffding's inequality (Vapnik, 1982), we can show that, by obtaining in sequence $\nu = (8/\epsilon^2)\ln(2/\delta)$ sample points and counting the proportion of Heads (which we can do without storing more than one point at a time), the process can estimate $\mathbf{P}(x)$ that is within $\pm\epsilon/4$ of its true value, with probability at least $1 - \delta$. Then if this proportion of Heads does not exceed $i/h - \epsilon/4$, the oracle responds "A_1," and "A_2" otherwise. Specifics are in Figure 2.

It is hard to imagine a simpler oracle to implement. The sample size ν required for each point is polynomial in all the critical parameters; and the fact that this many sample points must be obtained for each of the long N iterations of the Search Algorithm is mitigated somewhat by the knowledge that all h processes can use the same data.

Still, we can do better. The problem with this implementation becomes apparent when i/h is close to 1 and we are testing a point x for which $\mathbf{P}(x)$ happens to be close to zero. To an observer it quickly becomes apparent that we are seeing nearly all 0's when we expect

```
Algorithm 2: The Oracle $\mathcal{P}(x, i, h \mid \epsilon, \delta)$ (Basic Version)
Procedure:
  1. (Initialization) $r := 0$. $\nu = (8/\epsilon^2)\ln(2/\delta)$.
  2. For $j := 1$ to $\nu$:
     Let $x_j$ be the next sample point from the distribution; if $x_j \le x$, $r := r + 1$.
  3. If $r/\nu \le (i/h - \epsilon/4)$, return "$A_1$". Else, return "$A_2$".
```

Figure 2. Basic implementation of $\mathcal{P}$.

to see nearly all 1's. So we should be able to reach a decision with a much smaller sample size than would be required if $\mathbf{P}(x)$ and i/h were closer.

What we require is a hypothesis test that adjusts its sample size dynamically, instead of using a fixed sample size determined before the start of the experiment. Such a test is called a *sequential procedure*, and the Sequential Probability Ratio Test (SPRT) due to Wald (Wald, 1947; Mood & Graybill, 1963) is well suited to problems such as the one faced by the oracle.

Details of the sequential version of the oracle are given in Figure 3. Basically, for each new observation x_i, the quantity Z (representing the log-likelihood ratio of the sample) is increased or decreased by a fixed amount based on whether $x_i \le x$. Then Z is compared with the boundary values, *UPPER* and *LOWER*. The oracle continues as long as Z remains between the two boundary values. If it equals or surpasses *UPPER*, then "A_2" is the response; if it is less than or equal to *LOWER*, then "A_1" is the response. The boundary values have been chosen so that the probability of an incorrect response is at most δ.

With a sequential procedure, the most interesting quantity is the expected sample size. Unfortunately, for this problem it is difficult to say anything precise about the expected sample size over the set of log N iterations, because this value depends critically upon the particular distribution $\mathbf{p}$. It has been shown that the SPRT is, in a strong sense, optimal

```
Algorithm 3: The Oracle $\mathcal{P}(x, i, h \mid \epsilon, \delta)$ (Sequential Version)
Procedure:
  1. (Initialization) $Z := 0$. $UPPER = \log(2 - \delta) - \log\delta$. $LOWER = -UPPER$.
  2. While $LOWER < Z < UPPER$
     2.1 Let $x_j$ be the next sample point from the distribution.
     2.2 If $x_j \le x$, $Z := Z - |\log(1 - i/h)| + |\log(1 - i/h + \epsilon/2)|$.
     2.3 Else $Z := Z + |\log(i/h)| - |\log(i/h - \epsilon/2)|$.
  3. If $Z \ge UPPER$, return "$A_2$". Else return "$A_1$".
```

Figure 3. Sequential implementation of $\mathcal{P}$.

in minimizing the expected sample size for all comparable tests with the same confidence (Mood & Graybill, 1963).

Although the SPRT can in principle continue forever without returning a response, the theory shows that this occurs with zero probability. Still, no fixed upper bound on the sample size can be given. A reasonable approach to preventing "runaway" sampling is to combine Algorithms 2 and 3, halting whenever one of the two versions of the oracle halts.

Comparing the worst-case complexities of the two versions, we can see that they both require time $\mathcal{O}(\epsilon^{-2} \log(1/\delta))$ and space of about $\mathcal{O}(\log N + \log \nu)$ (the SPRT requires slightly more space, on the order of a factor of $\log \log h$).

2.3. The complete algorithm

Our description of the CDFI Algorithm is nearly complete. With the above results as subroutines, the algorithm is as follows.

1. The program, given values of N, h, ϵ and δ, creates h parallel processes, each executing the Search Algorithm (Algorithm 1) for a different value of i, $1 \leq i \leq h$.
2. Each of these processes calls on the "oracle" $\mathcal{P}$, which observes the input stream for some number of observations until the termination conditions are satisfied. The oracle assists the process in deciding whether $\mathbf{P}(x) \geq i/h$ for various values of x during the search.
3. The i'th process outputs $\hat{m}(i)$, its estimate for $m(i)$.

The correctness of the CDFI algorithm is a consequence of Lemma 1, Corollary 2, and the correctness of the oracle implementation. Given the complete set of values of $\hat{m}$, we take $\hat{m}(0) = 0$ and change $\hat{m}(h)$ to N, so that the $\hat{m}$ values span the entire range of X.[2] The h regions $R_i = (\hat{m}(i - 1), \hat{m}(i)]$ all satisfy equation 2 above, if the Mappability Condition holds. The expected running time is polynomial in $\log N$, h, $\log 1/\delta$, and $1/\epsilon$. Since no more than one example at a time is stored, the storage requirements are quite small.

Theorem 3. If the Mappability Condition holds, Algorithm 2 partitions X into h regions, each of probability $1/h \pm \epsilon$, with confidence $1 - \delta$. With h CREW processors, it requires (parallel) time of $\mathcal{O}(\log^2 N + \epsilon^{-2} \log N \log \delta^{-1})$ and space of $\mathcal{O}(h(\log N + \log \nu))$. □

The map ϕ that sends each of the nodes in R_i to the i'th node in the graph M gives us a feature map for the graph G.

If the Mappability Condition does not hold, the regions R_i may not satisfy equation 2. To detect whether it is satisfied, we may obtain a sample of points and determine the proportion of those points in each region. This is, again, a problem of estimating the parameter of a binomial distribution to within some given accuracy and confidence, and the same techniques apply as in developing the oracle $\mathcal{P}$. Of course, if $\hat{m}(i) = \hat{m}(i + 1)$ for any i, then we know immediately that the condition does not hold.

3. Feature maps using the CDFI algorithm

The basic CDF-Inversion Algorithm maps a totally ordered sequence of nodes onto another totally ordered graph. We now consider how it can be used to construct feature maps of a k-dimensional vector space $X = X_1 \times \ldots \times X_k$, the situation treated in the literature.

We can view the tuple $x = (x_1, \ldots, x_k)$ as an element of the set X of all possible k-tuples. After imposing an arbitrary linear order on X (e.g., lexicographic), we simply run the one-dimensional algorithm and let the regions R_i define the feature map. For vector-quantization applications, one chooses a vector in each region R_i to serve as a representative for that region; in our experiments we have used the mean (center of mass) vector in the region. Note that the algorithm enjoys all the properties of such maps listed in Section 1.3.

Figure 4 shows the result of one run of this algorithm. Points in a triangular region of a two-dimensional vector space were chosen with equal probability, and the one-dimensional algorithm was used to produce a map with 35 points. The vectors were linearly ordered left to right, low to high. In the diagram, circles indicate where the map points were placed by the algorithm, and lines are used to show the topological ordering of the points ($\hat{m}(1) \leq \hat{m}(2) \leq \ldots$). The resulting line "snakes" through the region, much like the feature map from Kohonen's algorithm under similar circumstances (Kohonen, 1984, p. 136). Note that one map point fell outside the region, but this was consistent with the error tolerance of the algorithm.[3]

Consider next how we might map a k-dimensional vector space $X = X_1 \times \ldots \times X_k$ onto a graph isomorphic to a two-dimensional grid of $d_1 = d_2$ points. As noted above, it is often necessary, for efficiency reasons, to limit the family of mappings that we consider. In this very simple case, we shall look only for maps which partition X into orthogonal boxes. Select two of the k dimensions of X. Ignoring all vector components but that of the first chosen dimension, run the one-dimensional CDFI algorithm, replacing h in Algorithm 1 by d_1, δ by $\delta/2$, and N by the size of the range of the component. The result is a partition of X into a number of regions based only on the values of this first component. Normally these regions will be d_1 in number unless a few values of the component occur with exceptionally high probability ($\epsilon/2$). (In fact, the first component may be chosen on

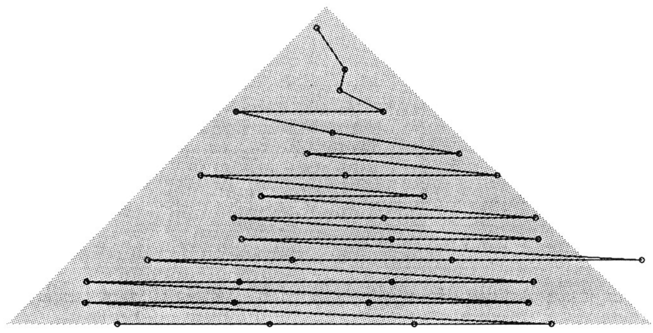


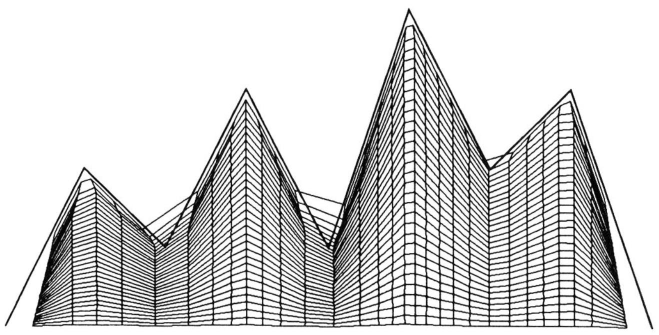
Figure 4. Feature map (linear array) of a polygonal region.

the basis of how well the Mappability Condition applies to projections of the space onto that dimension.) The probability of each region is $1/d_1 \pm \epsilon$, provided that the Mappability Condition holds for the marginal probability distribution of the first component. For each of these regions, we again run the CDFI algorithm to further section it into about d_2 equiprobable regions; for each of these d_1 executions, we specify d_2 for h, $\delta/(2d_1)$ for δ, the size of the range of the second component for N, and $\epsilon(d_2 - 1)/d_2$ for ϵ. Again, assuming mappability, the probability of each of the resulting (sub-)regions will be

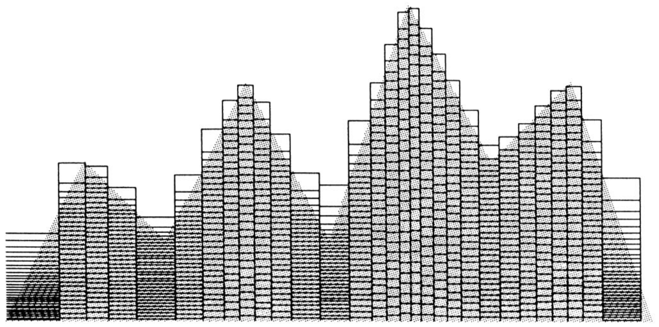
$$\frac{1}{d_2}\left(\frac{1}{d_1} \pm \epsilon\right) \pm \frac{\epsilon(d_2 - 1)}{d_2} = \frac{1}{d_1 d_2} \pm \epsilon.$$

The likelihood that any of the regions deviate from this bound is at most $\delta/2 + d_1(\delta/2d_1) = \delta$.

Figure 5 shows the result of one run mapping a two-dimensional vector space with a 30-by-30 grid. Vectors were chosen uniformly from a polygonal subset of the space. Figure 5(a)



(a) Feature map of a polygonal region.



(b) Corresponding regions.

Figure 5. Two-dimensional map on a grid.

shows a 30-by-30 feature map; each point (x_i, y_i) has been connected by a line to its four neighbors $((x_{i-1}, y_i), (x_{i+1}, y_i), (x_i, y_{i-1})$, and $(x_i, y_{i+1}))$ to show the topological ordering of the points. In Figure 5(b), the regions corresponding to the points are shown; one can clearly see how the map points correspond to regions of comparable probability. The map points shown at the grid intersections in the upper diagram are the center-of-mass points of the corresponding region in the diagram below. (It is interesting to compare this diagram with the corresponding one, Figure 6a, in Ritter & Schulten (1986).)

In general, the CDFI algorithm can be used to construct a particular family of feature maps from a k-dimensional vector space X to a k'-dimensional space M by imposing topological orderings on the two spaces and iterating the CDFI algorithm k' times. Different maps can be obtained, for example, by mapping onto a Cartesian grid, a polar grid, or any of a variety of other coordinate systems. This approach is useful only for small k' since the cost of the algorithm increases exponentially with k' and the accuracy is likely to decline with each iteration. It is interesting to note that the resulting feature map $\phi: X \rightarrow M$ is order-preserving: if $x_1 \leq x_2$, then $\phi(x_1) \leq \phi(x_2)$, where $\leq$ denotes the respective partial topological orderings of X and M. Hence order-invariant properties of X are preserved by the mapping ϕ.

4. About the implementation

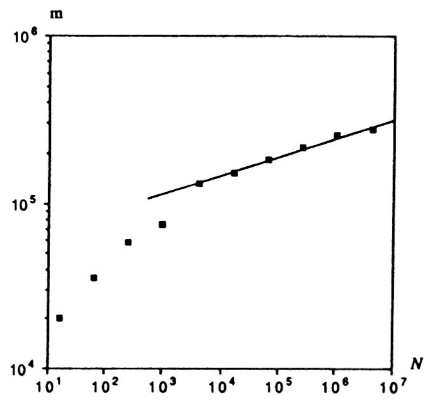
The algorithm has been implemented in C and on a Sun 3/60 Workstation with an MC68881 floating-point accelerator. No concurrency or parallelism was introduced, but input data points $\mathbf{x}_i$ were shared among the h independent routines computing the $\hat{m}(i)$. The program is compact: about 16 kbytes suffice for code and working storage when $h \leq 1000$. The oracle $\mathcal{P}$ was a sequential oracle similar to, but differing in significant details from, that of Algorithm 3.

No heuristics were introduced in order to speed up the algorithm (although many good heuristics suggested themselves). The only optimizations, aside from those provided automatically by the compiler, came as a result of reordering some calculations and making in-line calls to frequently used subroutines. These optimizations were made in about two days' worth of experimentation, and as a result the speed of the program roughly doubled.

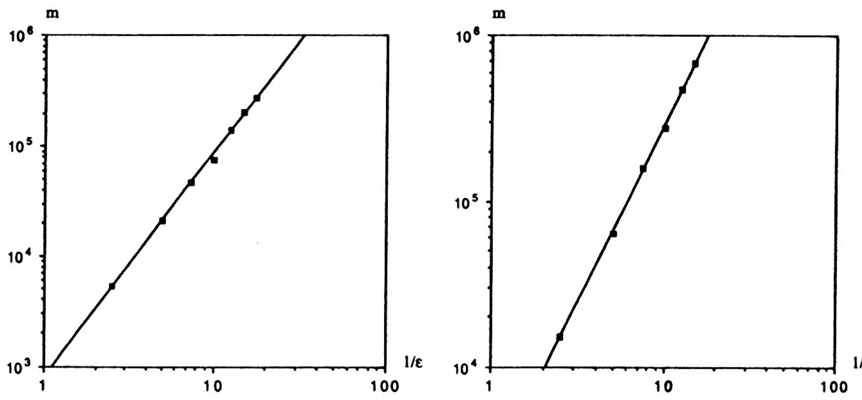
Another result of this work was the observation that the oracle consumes the largest proportion of the computation time (at times, over 70%). Note that the use of parallel computation would not remove this bottleneck, since it is part of all h parallel processes. Evidently the stringent accuracy and confidence conditions entail rather large sample sizes.

To cite a typical run time for this program, we programmed a uniform distribution over $N = 2^{14}$ symbols, with $\epsilon = 0.05$, $h = 10$ and $\delta = 0.2$. The total sample size ν used by the algorithm for this problem was about 150,000 points, with a fluctuation of about 16% from run to run. The time for this run to complete was about 4.5 minutes.

Using our implementation we were able to test the dependence of the sample size ν required by the algorithm as a function of the size N of the vector space and the accuracy ϵ of the result. Some typical results are shown in Figure 6. As expected, we found that ν grew in proportion to $\log N$, except for very small N when fixed computational overhead was a major portion of the run time. We were more interested, however, in the dependence of ν upon $1/\epsilon$. The results were convincing that ν grows as about $1/\epsilon^2$ (with N and δ fixed).



(a) Total sample size m versus vector-space size N. The line was fitted visually to show that m has roughly $\log N$ behavior for large N. (For this run, ϵ was 0.1).



(b) Sample size m versus accuracy $1/\epsilon$ for two different N values: 2^{10} (left) and 2^{22} (right). The lines show least-squares fits, with $\epsilon^{-2.14}$ and $\epsilon^{-2.03}$ dependence, respectively.

Figure 6. Measured sample sizes m versus N and $1/\epsilon$.

5. Conclusions

We have presented a simple algorithm that divides a finite set into a fixed number of approximately equiprobable regions, with statistical guarantees about the accuracy and confidence of the result. We have then applied the algorithm to obtain feature maps of vector spaces, and determined clearly the properties of those maps. Our implementation has shown that the algorithm is feasibly efficient, and simultaneously has identified the main bottleneck as the oracle $\mathcal{P}$.

A brief comparison with other feature-map algorithms is useful. As noted, our algorithm comes with rigorous definitions of the maps it produces and their accuracy. Kohonen's algorithm appears to be faster, although direct comparison is difficult since crucial parameters controlling the convergence of the algorithm are left unspecified in the literature and in practice are chosen heuristically. In both algorithms, processors perform only local computations, and computations are highly storage efficient. For multidimensional maps our algorithm runs in stages (one for each map coordinate), whereas the network algorithms have the advantage of computing the entire map in one stage. Finally if the algorithms are prematurely terminated prior to convergence, the usefulness of the intermediate state of the programs is different. Kohonen's procedure, for example, produces a map, but one whose topology and distribution will be less correct, the more prematurely the algorithm is interrupted; the CDFI algorithm provides a set of non-disjoint regions (each defined by the U and L values in Algorithm 1) whose probabilities can be estimated and whose topological relationship is correct.

It is likely that better feature map algorithms can be developed than the ones here, based on the CDFI procedure. Instead of imposing an artificial total ordering on the input graph, or mapping the space in stages by component, we suspect it is possible to map the entire space in a single stage like Kohonen's algorithm, while satisfying the statistical accuracy and confidence requirements.

A fundamental assumption of our feature-map algorithm—indeed, of all existing feature-map algorithms—is that the individual observations of vectors in X are statistically independent. In practice this is seldom the case; nevertheless the results of the algorithm can be useful provided the time scale of statistical dependencies is small compared to the time between observations. Extending this and other feature-map algorithms to handle time series is an interesting and potentially useful problem. Recently Paredis (1989) has suggested one approach.

6. Acknowledgments

Will Taylor provided technical support for the graphics. Coe Miles-Schlichting kindly loaned us his program for computing Kohonen feature maps. We benefitted from discussions with Albert Ahumada, who brought several relevant references to our attention. Silvano Colombano and Ron Yager reviewed the early drafts, and Bruce Trumbo offered suggestions for improving the statistical mechanisms in the program. Peter Cheeseman, Peter Friedland, Matthew Self, and our other colleagues at NASA Ames have also contributed to this research. Finally, the anonymous referees made generous suggestions that substantively improved the content, accuracy, and clarity of the paper.

Notes

Portions of this work were presented at the Fourth International Symposium on Methodologies for Intelligent Systems, October, 1989 and at the Fujitsu IIAS-SIS Workshop on Computational Learning Theory, November, 1989.

1. In this article an oracle is merely a subroutine whose existence we assume for now and for which we later provide an algorithm.
2. If the regions R_i are not required to span the entire set X, then $\hat{m}(h)$ can be left unchanged. Otherwise the process that computes $\hat{m}(h)$ (given in the Appendix) can be eliminated and $\hat{m}(h)$ set directly to N.
3. Other feature map algorithms also may place points in zero-probability regions, a fact that has not been acknowledged.

References

Anderson, R., & Friedman, J. (1962). A limitation of the optimum property of the sequential probability ratio test. In *Contributions to probability and statistics*, *6*, Palo Alto, CA: Stanford University Press.

DeSieno, D. (1988). Adding a conscience to competitive learning. *Proceedings of the Second International Conference on Neural Networks* (pp. I-117–I-124).

Haussler, D. (1990). *Generalizing the PAC model for neural net and other learning applications*. (Technical Report UCSC-CRL-89-30). Santa Cruz, CA.: University of California.

Hecht-Nielsen, R. (1987). Counterpropagation networks. *Proceedings of the First International Conference on Neural Networks.*

Kohonen, T. (1982). Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, *43*, 59–69.

Kohonen, T. (1984). *Self-organization and associative memory*. Springer-Verlag.

Kohonen, T. (1988). A "neural" phonetic typewriter. *IEEE Computer, March*, 11–22.

Mood, J., & Graybill, F. (1963). *Introduction to the theory of statistics (2nd edn.)*. New York: McGraw Hill.

Nasrabadi, N., & Feng, Y. (1988). Vector quantization of images based upon the Kohonen self-organizing feature maps. *Proceedings of the First International Conference on Neural Networks* (pp. I-101–I-106).

Paredis, J. (1989). Learning the behavior of dynamical systems from examples. *Proceedings of the Sixth International Workshop on Machine Learning* (pp. 137–139). Morgan-Kaufmann.

Pollard, D. (1984). *Convergence of stochastic processes*. Springer-Verlag.

Ritter, H., & Schulten, K. (1986). On the stationary state of Kohonen's self-organizing sensory mapping. *Biological Cybernetics*, *54*, 99–106.

Ritter, H., & Schulten, K. (1988). Kohonen's self-organizing maps: Exploring their computational capabilities. *Proceedings of the Second International Conference on Neural Networks* (pp. 109–116).

Valiant, L.G. (1984). A theory of the learnable. *C. ACM*, *27*, 1134–1142.

Wald, A. (1947). *Sequential analysis*. New York: John Wiley.

Appendix: Determining $\hat{m}(h)$

Estimating $m(i)$ when $i = h$ is a special case for both the oracles given in Section 2. The hypothesis test is to decide whether $\mathbf{P}(x) \leq 1 - \epsilon/2$ for a given value of x. If it is, then the likelihood of observing no values $> x$ in a sample of size ν is $(1 - \epsilon/2)^{\nu}$. When $\nu = (2/\epsilon) \log \delta^{-1}$, this value is at most δ. Thus a simple oracle similar to the one in Figure 2 handles the case $i = h$ as follows:

> Observe $\nu = (2/\epsilon) \log \delta^{-1}$ sample points from the distribution. If all of them are $\leq x$, reply "A_2." Else reply "A_1."

A sequential version of this algorithm (complementing the oracle in Figure 3) does the obvious thing:

> Stop obtaining sample points when either (a) a point $> x$ is observed, or (2) the sample size reaches ν', where ν' is the least integer such that $(1 - \epsilon/2)^{\nu'} \leq \delta$. In the former case, reply "A_1," and in the latter, "A_2."

This so-called *curtailed-sampling* procedure has been analyzed by Anderson and Friedman (1962), where they prove the following strong optimality property (expressed using our terminology): *any* oracle algorithm that always replies "A_2," in case $\mathbf{P}(x) = 1$ and replies "A_2," with probability at most δ when $\mathbf{P}(x) > 1 - \epsilon/2$ requires a sample size at least as great as ν'.