PAC-learning, Occam Algorithm and Compression

Mengqi Zong < *mz2326@columbia.edu* >

May 4, 2012

1 Introduction

The probably approximately correct learning model (PAC learning), proposed in 1984 by Leslie Valiant, is a framework for mathematical analysis of machine learning and it has been widely used to investigate the phenomenon of learning from examples.

In COMS 6253, we spent most of our time studying the results based on PAC-learning model. So it is worthy for us to take a closer look at this model.

This paper consists of two parts. In the first part, we will talk about PAC-learning, Occam algorithm and the relationship between the two. In the second part, we will talk about the relationship between learning and data compression. Then we attempt to apply PAC-learning model to analyze data compression, particularly, relationship between learning r-junta and lossless compression is investigated.

This entire paper will focus on learning over discrete domains.

2 Part I: PAC-learning and Occam algorithm

2.1 PAC-learning

What is the PAC-learning model? Generally, it is a model for mathematical analysis of machine learning, particularly, for the phenomenon of learning from examples.

In this model, the target concept we want to learn is a subset of a domain of elements, and a concept class is a set of such concepts. A learning algorithm is presented with a

collection of examples, each example is a labeled domain element indicating whether or not it is a member of the target concept in the class. All these examples, the domain elements, are drawn randomly according to a fixed probability distribution. Informally, the concept class is PAC-learnable if the learning algorithm runs in polynomial time and, with high probability, outputs the description of a concept in the class that differs by only a small amount from the unknown concept.

2.1.1 Preliminary notation and definitions

If S and T are sets, then we denote the symmetric difference of S and T by $S \oplus T = (S - T) \cup (T - S)$. The cardinality of a set S is denoted by $|S|$. If Σ is a alphabet, then Σ^* denotes the set of all finite length strings of elements of Σ. If $w \in \Sigma^*$, then the length of w, denoted by $|w|$, is the number of symbols in the string w. Let $\Sigma^{[n]}$ denote the set $\{w \in \Sigma^* : |w| \leq n\}$.

Define a *concept class* to be a pair $C = (C, X)$, where X is a set and $C \subseteq 2^X$. The domain of C is X, and the elements of C are *concepts*.

We describe a context for representing concepts over X. Define a class of representations to be a four-tuple $\mathbf{R} = (R, \Gamma, c, \Sigma)$. Σ and Γ are sets of characters. Strings composed of characters in Σ are used to describe elements of X, and strings of characters in Γ are used to describe concepts. $R \subseteq \Gamma^*$ is the set of strings that are concept descriptions or *representations*. Let $c : R \to 2^{\Sigma^*}$ be a function that maps these representations into concepts over Σ.

Note that if $\mathbf{R} = (R, \Gamma, c, \Sigma)$ is a class of representations, then there is an associated concept class $\mathbf{C}(\mathbf{R}) = (c(R), \Sigma^*)$, where $c(R) = \{c(r) : r \in R\}$.

We write $r(x) = 1$ if $x \in c(r)$, and $r(x) = 0$ if $x \notin c(r)$. An *example* of r is a pair $(x, r(x))$. The length of an example $(x, r(x))$ is $|x|$. A sample of size m of concept r is a multiset of m examples of r.

For a class of representations, the *membership problem* is that of determining, given $r \in R$ and $x \in \Sigma^*$, whether or not $x \in c(r)$. For practical reasons, we only consider classes of representations for which the membership problem is decidable in polynomial time. That is, we only consider representation classes $\mathbf{R} = (R, \Gamma, c, \Sigma)$ for which there exists a polynomial-time algorithm EVAL such that for all $r \in R$, $x \in \Sigma^*$, $\text{EVAL}(r, x) = r(x)$. EVAL runs in time polynomial in $|r|$ and $|x|$.

We let $R^{[s]}$ denote the set $\{r \in R : |r| \leq s\}$; that is, the set of all representations from R of length at most s. If $\mathbf{R} = (R, \Gamma, c, \Sigma)$ is a class of representations, $r \in R$, and $\mathcal{D}$ is a probability distribution on Σ^*, then EXAMPLE$(\mathcal{D}, r)$ is an oracle that, when called, randomly chooses an $x \in \Sigma^*$ according to distribution $\mathcal{D}$ and returns the pair $(x, r(x))$.

A randomized algorithm is an algorithm that behaves like a deterministic one with the additional property that, at one or more steps during its execution, the algorithm can flip a fair two-sided coin and use the result of the coin flip in its ensuing computation.

2.1.2 PAC-learnability

Definition 1. *The representation class* $\mathbf{R} = (R, \Gamma, c, \Sigma)$ *is PAC-*learnable *if there exists a (possibly randomized) algorithm* L *and a polynomial* p_L *such that for all* $s, n \geq 1$*, for all* ϵ *and* δ*,* $0 < \epsilon, \delta < 1$*, for all* $r \in R^{[s]}$*, and for all probability distributions* $\mathcal{D}$ *on* $\Sigma^{[n]}$*, if* L *is given as input the parameters* s, ϵ*, and* δ*, and may access the oracle EXAMPLE*$(\mathcal{D}, r)$*, then* L *halts in time* $p_L(n, s, 1/\epsilon, 1/\delta)$ *and, with probability at least* $1 - \delta$*, outputs a representation* $r' \in R$ *such that* $\mathcal{D}(r' \oplus r) \leq \epsilon$*. Such an algorithm* L *is a polynomial-time learning algorithm for* $\mathbf{R}$*.*

For the rest of the paper, without loss of generality, we will let $\Sigma = \{0, 1\}$.

Here are a few comments about PAC-learning and PAC-learnability:

1. Every concept class is learnable, but not necessarily PAC-learnable.

   For any $f : \{0,1\}^n \to \{0,1\}$, we can represent f using a truth table with 2^n entries. That is, we can learn f by trying every possible examples $(x, f(x))$. In this way, learning f is just a matter of time.

   For PAC-learnability, one important thing is the learning algorithm L must run in polynomial time. Because any algorithm runs in time beyond polynomial time is useless in a practical setting.

2. The learning algorithm must output a representation r' as the hypothesis of r from $\mathbf{R}$. That is, both r and r' must come from the same concept class.

   This is a strong restriction. For a given concept class, there are more than one representation forms. And the PAC-learnability of concept class may depend on the choice of representations. That is, PAC-learnability is in fact a property of classes of representations rather than of classes of concepts.

For example, if RP $\neq$ NP, we can show that the representation class of 3-term DNF formulae is not efficiently PAC learnable. However, the representation class of 3-CNF formulae is efficiently PAC learnable. Both representation classes can represent the concept class of 3-term DNF.

Note that 3-term DNF $\subsetneq$ 3-CNF. Also, the reduction from 3-term DNF to 3-CNF can be done in polynomial-time while the reduction from 3-CNF to 3-term DNF is NP-hard.

We will later talk about some variants of the definition of PAC-learning.

3. The PAC-learning model is a distribution independent model.

   The key part of why it is distribution independent is that the goal of our querying the oracle is to build a training set. How well can we learn about the target concept depends on how many different examples we get, not how many times we query the oracle.

   In this way, distribution matters on how many times do we have to query the oracle in order to get a large enough training set. However, most of the PAC-learning results are about an upper bound of the size of the training set. This makes sense because if the minimum size of the training set in order to learn the concept is beyond polynomial, then this concept class can never be PAC-learnable.

   In a sense, the reason why this model is distribution free is that we mostly care about the time complexity of processing a training set to learn the target concept, not the time complexity of creating the training set.

2.2 Occam's Razor and Occam algorithm

The PAC learning model defines learning directly in terms of the predictive power of the hypothesis output by the learning algorithm. Now we will introduce a rather different definition of learning that makes no assumptions about how the instances in a labeled sample are chosen, Occam learning. Instead of measuring the predictive power of a hypothesis, Occam learning judges the hypothesis by how succinctly it explains the observed data.

2.2.1 Occam learning

Occam's Razor is a principle given by theologian William of Occam on simplicity. This principle can be interpreted into the methodology of experimental science in the

following form: given two explanations of the data, all other things being equal, the simpler explanation is preferable. Today, in the filed of machine learning, this principle is still very alive because the goal of machine learning is often to discover the simplest hypothesis that is consistent with the sample data.

Occam learning is a result of applying the principle of Occam's Razor to computational learning theory. Here is the definition of Occam learning.

Definition 2. *Let $\alpha \geq 0$ and $0 \leq \beta < 1$ be constants. L is an (α, β)-**Occam algorithm for $\mathcal{C}$ using $\mathbf{H}$** if on input a sample S of cardinality m labeled according to $c \in \mathcal{C}_n$, L outputs a hypothesis $h \in \mathcal{H}$ such that:*

- *h is consistent with S.*
- *$size(h) \leq (n \cdot size(c))^{\alpha} m^{\beta}$.*

*We say that L is an **efficient** (α, β)-Occam algorithm if its running time is bounded by a polynomial in n, m and size(c).*

The crucial difference between PAC learning and Occam learning is that PAC learning, the random sample drawn by the learning algorithm is intended only as an aid for reaching an accurate model of some external process (the target concept and distribution), while in Occam learning we are concerned only with the fixed sample before us, and not any external process. As we can see, the goal of PAC learning and Occam learning are different. Later we will show that there is relationship between the two.

Another important property about Occam learning is that Occam learning no longer requires the hypothesis h and concept c must be in the same representation class. This is exactly what Occam's Razor is talking about: always prefer a simple hypothesis that is consistent with the sample.

2.2.2 Occam algorithm

We now give an definition of Occam algorithm that uses the same notation as that of the previous definition of PAC-learning. Also, this definition is more strict than Definition 2. Because this definition requires hypothesis and target concept use the same representations. We will talk about the difference between the two definitions later in detail.

Definition 3. *A randomized polynomial-time (length-based) Occam algorithm for a class of representations* $\mathbf{R} = (R, \Gamma, c, \Sigma)$ *is a (possibly randomized) algorithm* O *such that there exists a constant* $\alpha < 1$ *and a polynomial* p_O*, and such that for all* $m, n, s \geq 1$ *and* $r \in R^{[s]}$*, if* O *is given as input any sample* $M \subseteq S_{m,n,r}$ *and* $1/\gamma$*, and, with probability at least* $1 - \gamma$*, outputs a representation* $r' \in R$ *that is consistent with* M*, and such that* $|r'| \leq p_O(n, s, 1/\gamma)m^\alpha$*.*

Note that for Occam algorithms, there is a restriction about the size of r'. Most importantly, the size of r' can't grow too fast with m. This is a strong restriction. And this restriction becomes very important when talking about the relationship between PAC-learnability and Occam algorithm.

Intuitively, Occam algorithm must be a fast algorithm. On one hand, it runs in polynomial time of n, m, s. On the other hand, the output r' can not be too large. Because if it is too large, like its size is exponential of n, then the algorithm can not be fast. Both requirements are trying to guarantee the Occam algorithm to be a fast algorithm.

2.2.3 Occam's Razor

In this section, we will show that any Occam algorithm is also a PAC learning algorithm.

Theorem 4 (Occam's Razor)**.** *Let* L *be an efficient* (α, β)*-Occam algorithm for* $\mathcal{C}$ *using* $\mathcal{H}$*. Let* $\mathcal{D}$ *be the target distribution over the instance space* X*, let* $c \in \mathcal{C}_n$ *be the target concept, and* $0 < \epsilon, \delta \leq 1$*. Then there is a constant* $a > 0$ *such that if* L *is given as input a random sample* S *of* m *examples drawn from EXAMPLE*$(c, \mathcal{D})$*, where* m *satisfies*

$$m \geq a\left(\frac{1}{\epsilon}\log\frac{1}{\delta} + \left(\frac{(n \cdot size(c))^\alpha}{\epsilon}\right)^{\frac{1}{1-\beta}}\right)$$

then with probability at least $1 - \delta$ *the output* h *of* L *satisfies* $error(h) \leq \epsilon$*. Moreover,* L *runs in time polynomial in* $n, size(c), 1/\epsilon,$ *and* $1/\delta$*.*

Theorem 5 (Occam's Razor, Cardinality Version)**.** *Let* $\mathcal{C}$ *be a concept class and* $\mathcal{H}$ *a representation class. Let* L *be an algorithm such that for any* n *and any* $c \in \mathcal{C}_n$*, if* L *is given as input a sample* S *of* m *labeled examples of* c*, then* L *runs in time polynomial in* n, m *and* $size(c)$*, and outputs an* $h \in \mathcal{H}_{n,m}$ *that is consistent with* S*. Then there is a constant* $b > 0$ *such that for any* n*, any distribution* $\mathcal{D}$ *over* X_n*, and any target concept* $c \in \mathcal{C}_n$*, if* L *is given as input a random sample from EXAMPLE*$(c, \mathcal{D})$ *of* m *examples, where* $|H_{n,m}|$ *satisfies*

$$\log |\mathcal{H}_{n,m}| \leq b\epsilon m - \log \frac{1}{\delta}$$

(or equivalently, where m satisfies $m \geq (1/b\epsilon)(\log |\mathcal{H}_{n,m}| + \log 1/\delta)$)
then L is guaranteed to find a hypothesis $h \in \mathcal{H}_n$ that with probability at least $1 - \delta$ obeys error$(h) \leq \epsilon$.

Note that in Theorem 5, L is not necessarily an (efficient) Occam algorithm. In order to assert that L is a (α, β)-Occam algorithm, every hypothesis has bit length at most $(n \cdot \text{size}(c))^\alpha m^\beta$. Thus implying $|H_{n,m}| \leq 2^{(n \cdot \text{size}(c))^\alpha m^\beta}$. So, if each hypothesis is not that succinct enough as to have bit length at most $(n \cdot \text{size}(c))^\alpha m^\beta$, then L is not a (α, β)-Occam algorithm.

Also, in Theorem 5, L is not necessarily an (efficient) PAC learning algorithm. In order for the theorem to apply, we must pick m large enough so that $b\epsilon m$ dominates $\log |H_{n,m}|$. Moreover, since the running time of L has a polynomial dependence on m, in order to assert that L is an (efficient) PAC algorithm, we also have to bound m by some polynomial in n, size(c), $1/\epsilon$ and $1/\delta$. That is, m must be large, but not too large to the extent that it is beyond polynomial in n, size(c), $1/\epsilon$ and $1/\delta$. This really depends on the bound of $|H_{n,m}|$.

Theorem 4 shows that if L is an Occam algorithm for $\mathcal{C}$, then $\mathcal{C}$ is PAC-learnable. This is because the requirement of Occam algorithm gives us a bound of $|H_{n,m}|$. Using this bound, we can bound m then reach the requirement of PAC learnability. Formally, we can rephrase Theorem 4 into Theorem 6 using a different notation.

Theorem 6. *Let $\mathbf{R} = (R, \Gamma, c, \Sigma)$ be a class of representations, with Γ finite. If there exists a randomized polynomial-time (length-based) Occam algorithm for $\mathbf{R}$, then $\mathbf{R}$ is PAC-learnable.*

Theorem 6 shows that the existence of Occam algorithm is the sufficient condition of PAC-learnability.

2.3 PAC-learnability and Occam algorithm

From Theorem 6, we know that the existence of Occam algorithm is the sufficient condition of PAC-learnability. But it remains an open question that whether PAC-learnability is equivalent to the existence of Occam algorithms; i.e. whether the existence of an Occam algorithm is also a necessary condition for PAC-learnability.

In this section, we will show that for many natural concept classes that the PAC-learnability of the class implies the existence of an Occam algorithm for the class. More generally, the property of closure under exception lists is defined, and it is shown that for any concept class with this property, PAC-learnability of the class is equivalent to the existence of an Occam algorithm for the class.

2.3.1 Exception lists

To let PAC-learnability of the class equivalent to the existence of an Occam algorithm of the class, the class of representations must be closed under the operation of taking the symmetric difference of a representation's underlying concept with a finite set of elements from the domain. Further, there must exists an efficient algorithm that, when given as input such a representation and finite set, outputs the representation of their symmetric difference.

Definition 7. *A class* $\mathbf{R} = (R, \Gamma, c, \Sigma)$ *is* polynomially closed under exception lists *if there exists an algorithm EXLIST and a polynomial* p_{EX} *such that for all* $n \geq 1$*, on input of any* $r \in R$ *and any finite set* $E \subset \Sigma^{[n]}$*, EXLIST halts in time* $p_{EX}(n, |r|, |E|)$ *and outputs a representation* $EXLIST(r, E) = r_E \in R$ *such that* $c(r_E) = c(r) \oplus E$*. Note that the polynomial running time of EXLIST implies that* $|r_E| \leq p_{EX}(n, |r|, |E|)$*. If in addition there exist polynomials* p_1 *and* p_2 *such that tighter bound* $|r_E| \leq p_1(n, |r|, \log|E|) + p_2(n, \log|r|, \log|E|)|E|$ *is satisfied, then we say that* $\mathbf{R}$ *is* strongly polynomially closed under exception lists.

Note that any representation class that is strongly polynomially closed is also polynomially closed. The definition of polynomial closure can be easily understood - it asserts that the representation r_E that incorporates exceptions E into the representation r has size at most polynomially larger than the size of r and the total size of E, the latter of which is at most $n|E|$.

Intuitively, for polynomially closed, the algorithm EXLIST is trying to build the new concept r_E based on r and E in polynomial time. Note that the running time is polynomial also indicates that the size of the r_E is polynomial. That is, $|r_E| \leq p_{EX}(n, |r|, |E|)$. Now, for strongly polynomially closed, we also want to check the membership in the set E. In this case, we must build a concept E. And the concept E must have a size of $p'(n, \log|r|, \log|E|)|E|$. To sum up, we get $|r_E| \leq p_{EX}(n, |r|, \log|E|) + p'(n, \log|r|, \log|E|)|E|$.

2.3.2 Results for finite representation alphabets

We will show that, over the discrete domains, strong polynomial closure under exception lists guarantees that learnability is equivalent to the existence of Occam algorithms. We already know from Theorem 4 that the existence of Occam algorithm is a sufficient condition of PAC-learnability, so we will focus on strong polynomial closure guarantees that the existence of Occam algorithm is a necessary condition of PAC-learnability.

Theorem 8. *If* $\mathbf{R} = (R, \Gamma, c, \Sigma)$ *is strongly polynomially closed under exception lists and* $\mathbf{R}$ *is PAC-learnable, then there exists a randomized polynomial-time (length-based) Occam algorithm for* $\mathbf{R}$.

Proof. Let L be a learning algorithm for $\mathbf{R} = (R, \Gamma, c, \Sigma)$ with running time bounded by the polynomial p_L. Let EXLIST witness that $\mathbf{R}$ is strongly polynomially closed under exception lists, with polynomials p_1 and p_2 as mentioned in Definition 7. Without loss of generality, we assume that p_1 and p_2 are monotonically nondecreasing in each argument. Recall our convention that $\log x$ denotes $\max\{\log_2 x, 1\}$. Let $a \geq 1$ be a sufficiently large constant so that for all $n, s, t \geq 1$, and for all ϵ and δ such that $0 < \epsilon, \delta < 1$,

$$p_1(n, p_L(n, s, \frac{1}{\epsilon}, \frac{1}{\delta}), \log t) \leq \frac{a}{2}(\frac{ns \log t}{\epsilon\delta})^a.$$

Let $b \geq 1$ be a sufficiently large constant so that for all $n, s, t \geq 1$, and for all ϵ and δ such that $0 < \epsilon, \delta < 1$,

$$p_2(n, \log(p_L(n, s, \frac{1}{\epsilon}, \frac{1}{\delta})), \log t) \leq \frac{b}{2}(\frac{ns \log(t/\epsilon)}{\delta})^b.$$

Let $c_{a,b}$ be a constant such that for all $x \geq 1$, $\log x \leq c_{a,b}(x^{1/(2a+2)(a+b)})$.

We show that algorithm O (Fig. 1) is a randomized polynomial-time (length-based) Occam algorithm for $\mathbf{R}$, with associated polynomial

$$p_O(n, s, \frac{1}{\gamma}) = (c_{a,b})^{a+b} ab(\frac{ns}{\gamma})^{a+b} \text{ and constant } \alpha = \frac{2a+1}{2a+2}$$

Since r' correctly classifies every $x \in \text{strings}(M) - E$ and incorrectly classifies every $x \in E$, r'_E is consistent with M. Since $\mathbf{R}$ is closed under exception lists, $r'_E \in R$.

The time required for the first step of algorithm O is bounded by the running time of L, which is no more than

$$p_L(n, s, \frac{1}{\epsilon}, \frac{1}{\delta}) = p_L(n, s, m^{\frac{1}{\alpha+1}}, \frac{1}{\gamma}),$$

which is polynomial in n, s, m, and $1/\gamma$. Note that this immediately implies that $|r'|$ is bounded by the same polynomial.

For each of the m distinct elements x in strings(M), each of length at most n, the second step executes EVAL(r', x), so the total running time for step 2 is bounded by $(km)p_{eval}(|r'|, n)$, where k is some constant and p_{eval} is the polynomial that bounds the running time of algorithm EVAL. Since $|r'|$ is at most $p_L(n, s, m^{1/(\alpha+1)}, 1/\gamma)$, the running time for the second step is polynomial in n, s, m, and $1/\gamma$.

Since EXLIST is a polynomial-time algorithm, the time taken by the third step is a polynomial function of $|r'|$ and the length of the representation of E. Again, $|r'|$ is polynomial in n, s, m, and $1/\gamma$, and the length of the representation of E is bounded by some constant times nm, since $|E| \leq m$ and each element $x \in E$ has size at most n. We conclude that O is a polynomial-time algorithm.

To complete the proof, it remains to be shown that with probability at least $1 - \gamma$,

Algorithm O (Inputs: $s, \gamma; M \in S_{m,n,r}$)

1. Run the algorithm L, giving L the input parameters s, $\epsilon = m^{-1/(\alpha+1)}$, and $\delta = \gamma$. Whenever L asks for a randomly generated example, choose an element $x \in$ strings(M) according to the probability distribution $\mathcal{D}(x) = 1/m$ for each of the m (without loss of generality, distinct) elements of M, and supply the example $(x, r(x))$ to L. Let r' be the output of L.
2. Compute the exception list $E = \{x \in \text{strings}(M) : r'(x) \neq r(x)\}$. The list E is computed by running EVAL(r', x) for each $x \in$ strings(M).
3. Output $r'_E = \text{EXLIST}(r', E)$.

Figure 1: Occam algorithm derived from learning algorithm L

$|r'_E| \leq p_O(n, s, 1/\gamma)m^\alpha$. Since $\mathbf{R}$ is strongly polynomially closed under exception lists,

$$\begin{aligned}
|r'_E| &\leq p_1(n, |r|, \log|E|) + p_2(n, \log|r|, \log|E|)|E| \\
&\leq p_1(n, p_L(n, s, \frac{1}{\epsilon}, \frac{1}{\gamma}), \log|E|) + p_2(n, \log(p_L(n, s, \frac{1}{\epsilon}, \frac{1}{\gamma})), \log|E|)|E| \\
&\leq \frac{a}{2}(\frac{ns\log|E|}{\epsilon\gamma})^a + \frac{b}{2}(\frac{ns\log(|E|/\epsilon)}{\gamma})^b|E|
\end{aligned}$$

Since L is a polynomial-time learning algorithm for $\mathbf{R}$, with probability at least $1-\delta$, $D(r \oplus r') \leq \epsilon$. The probability distribution $\mathcal{D}$ is uniform over the elements in strings(M); thus, with probability at least $1-\delta$, there are no more than ϵm elements $x \in$ strings(M) such that $x \in r \oplus r'$. Since $\delta = \gamma$, with probability at least $1-\delta$,

$$|E| \leq \epsilon m = m^{-\frac{1}{a+1}}m = m^{\frac{a}{a+1}}.$$

Substituting the bound on $|E|$ of the last inequality into the previous inequality, and substituting $m^{-1/(a+1)}$ for ϵ, it follows that with probability at least $1-\gamma$,

$$\begin{aligned}
|r'_E| &\leq \frac{a}{2}(\frac{ns\log m^{\frac{a}{a+1}}}{\gamma})^a m^{\frac{a}{a+1}} + \frac{b}{2}(\frac{ns\log m}{\gamma})^b m^{\frac{a}{a+1}} \\
&= \frac{a}{2}(\frac{ns}{\gamma})^a(\frac{a}{a+1}\log m)^a m^{\frac{a}{a+1}} + \frac{b}{2}(\frac{ns}{\gamma})^b(\log m)^b m^{\frac{a}{a+1}} \\
&\leq ab(\frac{ns}{\gamma})^{a+b}(\log m)^{a+b}m^{\frac{a}{a+1}},
\end{aligned}$$

and by choice of $c_{a,b}$,

$$\begin{aligned}
|r'_E| &\leq ab(\frac{ns}{\gamma})^{a+b}(c_{a,b}(m^{\frac{1}{(2a+2)(a+b)}}))^{a+b}m^{\frac{a}{a+1}} \\
&\leq (c_{a,b})^{a+b}ab(\frac{ns}{\gamma})^{a+b}m^{\frac{1}{2a+2}}m^{\frac{a}{a+1}} \\
&\leq p_O(n, s, \frac{1}{\gamma})m^\alpha,
\end{aligned}$$

completing the proof of Theorem 8 □

Corollary 9. *Let Γ be a finite alphabet. If $\mathbf{R} = (R, \Gamma, c, \Sigma)$ is strongly polynomially closed under exception lists, then $\mathbf{R}$ is PAC-learnable if and only if there exists a randomized polynomial-time (length-based) Occam algorithm for $\mathbf{R}$.*

Corollary 9 follows immediately from Theorem 4 and Theorem 8.

3 Part II: Learning and data compression

In this part, we will talk about the relationship between learning and data compression. And we will try to apply PAC-learning model to analyze data compression problems.

3.1 Learning versus Prediction

There are two common variants on the standard definition of PAC-learning. Under these alternative definitions, the hypothesis output by the learning is not required to be of the same form as the target concept description.

The notion of learning one representation class $\mathbf{R} = (R, \Gamma, c, \Sigma)$ in terms of another representation class $\mathbf{R}' = (R', \Gamma', c', \Sigma')$ was introduced. Under this definition, a learning algorithm for $\mathbf{R}$ is required to output hypotheses in R' rather than in R. (R' maybe a superset of R). Otherwise, the definition is identical to Definition 1. A representation class $\mathbf{R}$ is *polynomially predictable* if there exists a representation class R' with an uniform polynomial-time evaluation procedure (i.e. an algorithm EVAL as defined above) such that $\mathbf{R}$ is PAC-learnable in terms of $\mathbf{R}'$.

For Occam algorithm, we can do the same thing. A length-based Occam algorithm for $\mathbf{R}$ in terms of $\mathbf{R}'$ is required to output a representation from R', rather than from R. As we can see, Definition 2 is the generalized version of Definition 3.

3.2 Occam algorithm and compression

Occam algorithm, in some sense, is a compression algorithm.

The data compression problem is to find a small representation for the data, that is, an hypothesis h that is significantly smaller than the original data set and also is consistent with the sample.

Recall from Definition 2, for a sample S of cardinality m labeled according to $c \in \mathcal{C}_n$, (α, β)-Occam algorithm L outputs a hypothesis $h \in \mathcal{H}$ with $size(h) \leq (n \cdot size(c))^{\alpha} m^{\beta}$ that is consistent with S.

Note that for an (α, β)-Occam algorithm, we only know that $size(h) \leq (n \cdot size(c))^{\alpha} m^{\beta}$. If $size(h) \ll size(c)$, then the Occam algorithm is a compression algorithm.

3.3 Lossless data compression and learning r-junta

Lossless data compression is a class of data compression algorithms that allows the exact original data to be reconstructed from the compressed data. We will show that we can model a simple type of lossless data compression problem to the problem of learning junta.

3.3.1 Preliminary definitions

Definition 10. *Variable i in $f(x_1, ..., x_n)$ is* relevant *if $\exists\ x \in \{-1, 1\}^n$ such that $f(x^{i\leftarrow 1}) \neq f(x^{i\leftarrow -1})$.*

Definition 11. *Function $f : \{-1, 1\}^n \to \{-1, 1\}$ is an r-junata if f has $k \leq r$ relevant variables.*

Example 12. *$x_{17} \oplus (x_{412} \vee (x_{916,774} \oplus x_{17}))$ is a 3-junta. This can be treated as a function of the 3 variables $x_{17}, x_{412}, x_{916,774}$.*

3.3.2 Description length

For an arbitrary function $f : \{-1, 1\}^n \to \{-1, 1\}$, the best description length we could hope for would be $DL(f) = 2^n$ bits. Because there are 2^n entries in the truth table.

For an r-junta, the description length of an r-junta is $r \log n + 2^r$ bits. First, since there are n variables, we need $\log n$ bits to represent each variable. Then for r variables, we need $r \log n$ bits. Second, for r variables, the truth table contains 2^r entries. So it takes 2^r bits to write down the truth table. To sum up, the description length of an r-junta is $r \log n + 2^r$.

So, if we know the f is a junta, then we could use only $r \log n + 2^r$ bits instead of 2^n bits to represent this function. In this case, we successfully compressed the data. And the compression rate is

$$r = 1 - \frac{r \log n + 2^r}{2^n}$$

Example 13. *Assume for function f, there are* 16 *variables, only* 13 *variables are relevant. By converting the representation form from general form to r-junta form, we achieve a compression rate of*

$$r = 1 - \frac{13 \times \log 16 + 2^{13}}{2^{16}} = 0.8742$$

That is, we saved 87.42% of the original space.

3.3.3 A model for a simple lossless compression problem

Here is a simple lossless compression problem: Suppose we have a function $f : \{-1, 1\}^n \to \{-1, 1\}$. We know there are only r of the n variables are relevant. But we don't know which r variables are relevant. We want to convert f from general representation form to the r-junta representation form to achieve the data compression. How do we design the algorithm?

This problem is equivalent to the problem of learning r-junta with an membership oracle. And the problem can be solved in $\text{poly}(2^r, n)$ time.

Theorem 14. *Any r-junta can be learned in time* $\text{poly}(2^r, n)$ *and sample complexity* $\text{poly}(2^r, \log n)$ *with an membership oracle.*

Proof. Here is the algorithm of learning r-junta with an membership oracle:

1. Pick two examples $x, y \in \{-1, 1\}^n$ which differ in at least one bit position, such that $f(x) \neq f(y)$.

2. Construct a new example z such that z matches x and y in the bit positions where x and y match, and z differs from x in at least one bit position and also differs from y in at least one other bit position.

3. If $f(z) = f(x)$, set $x = z$, otherwise set $y = z$. Repeat steps 2 and 3 until x and y differ in one position and $f(x) \neq f(y)$. This bit position is a relevant variable.

4. Set the bit position learned in step 3 equal to 0, and (recursively) repeat steps 1-4 to find the remaining relevant variables. Also set the bit position learned in step 3 equal to 1, and (recursively) repeat steps 1-4 to find the remaining relevant variables.

In Step 1, it may be nontrivial to find a pair of examples that yield different function values. For example, the function f may be a constant (no relevant variables) or may

be well-approximated by a constant if most examples yield the same function value.

Note that if there are r relevant variables there are 2^r different ways in which the values of the relevant variables may be set. All settings cannot yield the same function value, so at least one of the 2^r settings has a function value different than the others. Therefore, if we randomly and uniformly choose examples (and therefore randomly and uniformly choose values for the r relevant variables), there is at least a $1/2^r$ chance that we pick an example with a function value different than the most common function value.

Consequently, choosing poly(2^r) random examples provides a high probability that we will select examples with both function values. If all poly(2^r) function values are the same, we conclude with high probability that the function f is constant. Since each example is length n, selecting a random example is performed in time $O(n)$ so determining whether f is constant (or finding a pair of example with different function values) takes time poly$(2^r, n)$.

In Step 2, we construct a z that is along the path on the Boolean hypercube from x to y. Let P be the set of bit positions in which x and y differ. The construction guarantees that z matches x in some positions in P and matches y in the remaining bit positions in P, and therefore differs from by the other half of the bit positions. This places z at the midpoint along the path from x to y, and so this step is in general repeated $\log n$ times.

In Step 3, we are effectively moving x or y to shorten the distance between those points on the Boolean hypercube. We repeat until that distance is one, so that the two points are adjacent and thus reveal the relevant variable.

In Step 4, note that in fixing the value of a relevant variable in an $rjunta$, we transform the problem to that of learning a (r-1)-junta. Since we perform two recursions for each relevant variable, there are at most 2^r recursive iterations. Therefore the entire algorithm requires time poly$(2^r, n)$ and sample complexity poly$(2^r, \log n)$.

□

Note that all the r variables in an r-junta are relevant, this means that 2^r is the minimum number of examples we could hope for. So, the running time of the learning algorithm for r-juntas is associated with n and 2^r. This indicates poly$(2^r, n)$ is the best we could hope for.

As we can see, this lossless compression problem can be solved in time poly$(2^r, n)$.

3.4 Summary on learning and compression

There are decent relations between learning theory and data compression. It is possible for us to apply results of learning theory to analyze data compression problems. However, this seems to be not easy.

Though having lots in common, the frame work of the two fields are still quite different. As to the work in this paper, all the compressions we talking about are just changing representation class in the learning, not applying results of learning theory to analyze the data compression problems.

More efforts are needed to investigate the relationship between computational learning theory and data compression.

4 References

- Raymond Board, Leonard Pitt (1990). *On the necessity of Occam algorithms.*
- Robert E. Schapire (1990). *The Strength of Weak Learnability.*
- Micheal J. Kearns, Umesh V. Vazirani (1994). *An introduction to computational learning theory.*
- Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, Manfred K (1986). Warmuth. *Occam's Razor.*
- Elchanan Mossel, Ryan O'Donnell, Rocco A. Servedio (2003). *Learning Juntas.*
- L.G. Valiant (1984). A Theory of the Learnable.