Universität Frankfurt am Main
Fachbereich Biologie und Informatik
Institut für Informatik
Lehrstuhl für Theoretische Informatik

Seminar Algorithmisches Lernen, WS 2001/02

Exploiting Random Walks for Learning

Fabian Wleklinski – fabian@wleklinski.de

Abstract

Beim klassischen PAC-Modell wird vorausgesetzt, dass die Beispiele unabhängig aufgrund einer gegebenen Häufigkeitsverteilung gezogen werden. Im Gegensatz dazu wird bei dem vorgestellten Ansatz davon ausgegangen, dass die Beispiele durch einen zeitgesteuerten Prozess erzeugt werden. Zwischen unmittelbar aufeinanderfolgenden Beispielen besteht dann oft ein Zusammenhang, dem zusätzliche Informationen entnommen werden können. Für diesen Anwendungsfall werden deterministische und probabilistische Lernmodelle vorgestellt, anschließend werden die Zusammenhänge zwischen diesen Modellen untersucht. Es werden effiziente Algorithmen vorgestellt, die eine boolesche Schwellwert-Funktion, eine zweitermige RSE oder eine zweitermige DNF exakt identifizieren.

1 Inhaltsverzeichnis

2 Abbildungsverzeichnis

3 Vorwort

Diese Ausarbeitung handelt über das algorithmische Lernen stochastischer Prozesse, und ist im Februar 2002 im Rahmen des Seminars „Algorithmisches Lernen“ des Lehrstuhls für Theoretische Informatik am Fachbereich Biologie und Informatik der Universität Frankfurt am Main entstanden. Der Inhalt dieser Ausarbeitung basiert auf dem Paper „Exploiting Random Walks for Learning“ von Peter L. Bartlett, Paul Fischer und Klaus-Uwe Höffgen.

In der Praxis besitzen viele Prozesse die Eigenschaft, dass aufeinander folgende Beispiele eine Beziehung zueinander haben, z.B. dann, wenn sie von einer Irrfahrt erzeugt werden. Um diese Informationen zunutze zu machen, wird das klassische PAC Lernmodell um erweitert.

Der aktuelle Stand dieser Ausarbeitung, sowie die parallel zu dieser Ausarbeitung entstandenen Präsentationsfolien sind in verschiedenen Dateiformaten unter der folgenden Internetadresse erhältlich:

http://www.stormzone.de/uni/Hauptstudium/seminare/algorithmisches_lernen/fw/list.php3

Aus Platzgründen konnten nicht alle Aspekte, die im zugrundeliegenden Paper zur Sprache kamen, genannt werden.

Zur Lektüre sind lediglich Grundkenntnisse über das algorithmische Lernen erforderlich.

Diese Ausarbeitung wurde mit Hilfe von MS Word angefertigt, die dazugehörige Präsentation mit Hilfe von MS PowerPoint. Kostenlose Betrachter für diese Dateiformate sind unter http://www.microsoft.com/office/000/viewers.htm erhältlich.

Fabian Wleklinski

Frankfurt am Main, im Februar 2002

4 Einleitung

Im klassischen PAC-Modell, wie es durch Valiant in [7] vorgestellt wird, werden Informationen über das Zielkonzept durch die unabhängig gezogenen, „beschrifteten“ Beispiele gewonnen. Diese Voraussetzung der Unabhängigkeit ist unabdingbar für die Analyse der meisten PAC Lernalgorithmen. In der Praxis ist diese Bedingung aber oft verletzt: es gibt zeitgesteuerte Prozesse, welche Beispiele erzeugen, von denen sich jeweils zwei aufeinanderfolgende oftmals nur minimal unterscheiden. Viele physikalischen Abläufe, wie z.B. die Flugbahn eines Flugkörpers, haben diese Eigenschaft. Solche Szenarien können durch stochastische Prozesse modelliert werden.

Die Ersten, die PAC Lernen für solche Zwecke angewendet haben, waren Aldous und Vazirani in [1]. In [4] stellt Freund ein angepasstes Lernmodell für DFAs vor, welches auf diesen inkrementellen Veränderungen basiert. In diesem Modell folgt ein „random walk“ den Zustandswechseln des Automaten. Es wird gezeigt, dass die meisten DFAs auf diese Weise ohne Fragen („membership queries“) lernbar sind.

In diesem Papier wird ein generelles Modell gezeigt, bei welchem die Beispiele nicht notwendigerweise unabhängig sind, und keine Fragen erlaubt sind. Die Beispiele werden von einem stochastischen Prozess erzeugt, und es wird gezeigt, dass das Betrachten der Veränderungen von Beispiel zu Beispiel ähnliche Informationen liefert wie die der „membership queries“.

Weil die Beispiele nacheinander generiert und verarbeitet werden wird ein online „mistake bound model“ benutzt. Wie A.Blum gezeigt hat (siehe [2]) ist dieses Modell im Fall unabhängiger Beispiele strenger als das klassische PAC Modell.

Von diesem Modell werden mehrere Varianten für deterministische und stochastische Szenarien vorgestellt. Diese Modelle werden miteinander verglichen, und mit den klassischen Lernmodellen in Beziehung gesetzt. Weiterhin werden Anwendungen dieser Modelle auf dem booleschen „Hypercube[1]“ demonstriert, für den Fall, dass die Beispiele durch einen „random walk“ erzeugt werden, so dass sich aufeinanderfolgende Beispiele nur jeweils in maximal einem Bit unterscheiden.

Um genau zu sein, werden effiziente Algorithmen entwickelt, um boolesche Schwellwertfunktionen, zweitermige RSE („ring-sum-expansion“) oder zweitermige DNF exakt zu identifizieren. Diese genannten Konzeptklassen sind im PAC Modell nicht exakt lernbar; sie sind es nur, wenn Fragen („membership queries“) erlaubt sind. Daher sind die vorgestellten Algorithmen die ersten, welche die genannten Konzeptklassen mittels eines passiven Lernmodells und einer polynomiellen Fehleranzahl *exakt* identifizieren.

Das Paper ist wie folgt aufgebaut: Nach einer Einleitung im ersten Kapitel folgt eine Vorstellung der Notation und der Lernmodelle im zweiten Kapitel. Daran schließt im dritten Kapitel eine Diskussion der Beziehungen zwischen den Lernmodellen untereinander, zu den klassischen Lernmodellen an. In den Kapiteln 4, 5 und 6 werden Lernalgorithmen für boolesche Schwellwertfunktionen, zweitermige RSE und zweitermige DNF vorgestellt.

[1] Zu Deutsch: Hyperkubus

5 Definitionen und Terminologie

Das algorithmische Lernen baut auf den Begriffen „Beispiel“, „Konzept“ und „Hypothese“ auf. Die Idee dahinter ist es, dass ein unwissender Algorithmus versucht, Informationen über einen Sachverhalt zu erlangen (das Konzept), welchen er anfangs nicht oder nur teilweise kennt. Um zu seinem Ziel zu gelangen werden dem Algorithmus Beispiele präsentiert. Aufgrund dieser Beispiele erstellt der Algorithmus nach und nach eine Hypothese, wie das Konzept aussieht.

Algorithmen zum Lernen sind stets in einer Schwarz-Weiß-Welt definiert, so dass es für Beispiele nur zwei mögliche Kategorisierungen gibt: Ein Beispiel kann entweder in dem Konzept enthalten sein, oder nicht. Ein solcher Algorithmus könnte z.B. zu einer gegebenen Handschriftprobe erkennen, ob es sich um ein „A“ handelt – oder nicht. Ein zweiter Algorithmus, bzw. ein anders parametrisierter Algorithmus könnte dasselbe für ein „B“ erreichen. Komplexere Szenarien, in denen auf den ersten Blick mehr als zwei Kategorisierungen nötig sind (z.B. wenn ein Algorithmus aus einer gegebenen Handschriftprobe den konkreten Buchstaben, oder gar vollständige Wörter ermitteln soll), müssen zunächst auf den vereinfachten Fall abgebildet werden.

Beispiele werden als binäre Folgen, d.h. als Folgen aus Nullen und Einsen beschrieben. Für eine bestimmte Beispiellänge *n (die größer als Null sein sollte) ist* X_n die Menge aller Beispiele mit dieser Länge, d.h. Folgen der Länge *n*:

$$X_n = \{0,1\}^n \quad | n > 0 \tag{5.1}$$

Betrachtet man nicht nur Beispiele einer bestimmten Länge, sondern mit beliebiger Länge, lassen sich diese als Vereinigung aller möglichen X_n darstellen:

$$X = \bigcup_{n=1}^{\infty} X_n = \{(0),(1),(0,0),(0,1),(1,0),(1,1),(0,0,0),\ldots\} \tag{5.2}$$

Sei F_n eine Klasse[2] von Funktionen, die eine solche, n-stellige 0/1-Folge als Eingabe entgegennehmen, und 0 oder 1 zurückgeben. F_n sei also eine Klasse von {0,1}-wertigen Funktionen definiert auf X_n. H_n sei identisch definiert wie F_n:

$$F_n = \{ f \mid f : X_n \to \{0,1\} \} \tag{5.3}$$

$$H_n = \{ h \mid h : X_n \to \{0,1\} \} \tag{5.4}$$

F entspricht der zu lernenden Konzeptklasse, *H* der Hypothesenklasse. Formal ist *F* bzw. *H* die Vereinigung aller Funktionsklassen F_n bzw. H_n:

$$F = \bigcup_{n=1}^{\infty} F_n = F_1 \cup F_2 \cup F_3 \cup \ldots \tag{5.5}$$

$$H = \bigcup_{n=1}^{\infty} H_n = H_1 \cup H_2 \cup H_3 \cup \ldots \tag{5.6}$$

An die Funktionen aus F_n bzw. H_n werden keine weiteren Ansprüche gestellt, außer dem, dass jede dieser Funktionen für jede beliebige Eingabe in polynomieller Zeit ausgewertet werden kann.

[2] d.h. eine beliebig grosse, auch unendlich grosse Menge von Funktionen, die durch die Teilung bestimmter Eigenschaften „zusammengehören“.

Sei *S* die Menge aller Kombinationen von je einem Beispiel (Element aus *X*) und einer Kategorisierung (Null oder Eins):

$$\begin{aligned} S &= X \times \{0,1\} = \\ &= \{\{(0),0\},\{(0),1\},\{(1),0\},\{(1),1\},\{(0,0),0\},\{(0,0),1\},\ldots\} \end{aligned} \tag{5.7}$$

Der Begriff des „Beispiels“ wird durch die Funktion *sam()* formalisiert, die eine unendlich lange Folge von Beispielen und Kategorisierungen auf eine *t*-lange Beispielfolge verkürzt, indem sie alle weiteren Beispiele und Kategorisierungen „abschneidet“. Solch eine Folge wird im weiteren Verlauf „*t*-Beispielfolge“ genannt.

$$\begin{aligned} x &= (x_1, x_2, \ldots) \quad \forall i : x_i \in X_n \quad |x| = \infty \\ \mathrm{sam}_t(x, f) &= (x_1, f(x_1), x_2, f(x_2), \ldots, x_t, f(x_t)) \end{aligned} \tag{5.8}$$

Das bereits eingeführte S entspricht der Menge aller kategorisierten Beispiele, und darf insofern auch als Menge aller einelementigen, kategorisierten Beispielfolgen betrachtet werden. Für den Übergang zu kategorisierten Beispielfolgen mit beliebiger Länge wird S^* wie folgt definiert:

$$S^* = \bigcup_{i=1}^{\infty} S^i = S \cup S^2 \cup S^3 \cup \ldots \tag{5.9}$$

Ein Lernalgorithmus für *F* arbeitet wie folgt: Zu jedem Zeitpunkt *t* besitzt er eine Arbeitshypothese $h_t \in H$, die aber noch nicht endgültig ist. Sobald der Algorithmus ein unkategorisiertes Beispiel x_t präsentiert bekommt, schlägt der Algorithmus $h_t(x_t)$ als Kategorisierung vor (entweder Null oder Eins). Erst danach wird dem Algorithmus mitgeteilt, wie die tatsächliche Kategorisierung $f(x_t)$ lautet. Der Algorithmus darf daraufhin seine Arbeitshypothese verändern.

Ein Lernalgorithmus ist also eine Art „Black-Box“, in die eine beliebig lange (aber nicht unendlich lange) Folge von Beispielen zusammen mit ihren Kategorisierungen geworfen wird, und die eine Arbeitshypothese auswirft. Ein deterministischer Lernalgorithmus darf demzufolge auch als eine Abbildung von der Menge aller kategorisierten Beispielfolgen auf die Menge aller Hypothesen betrachtet werden:

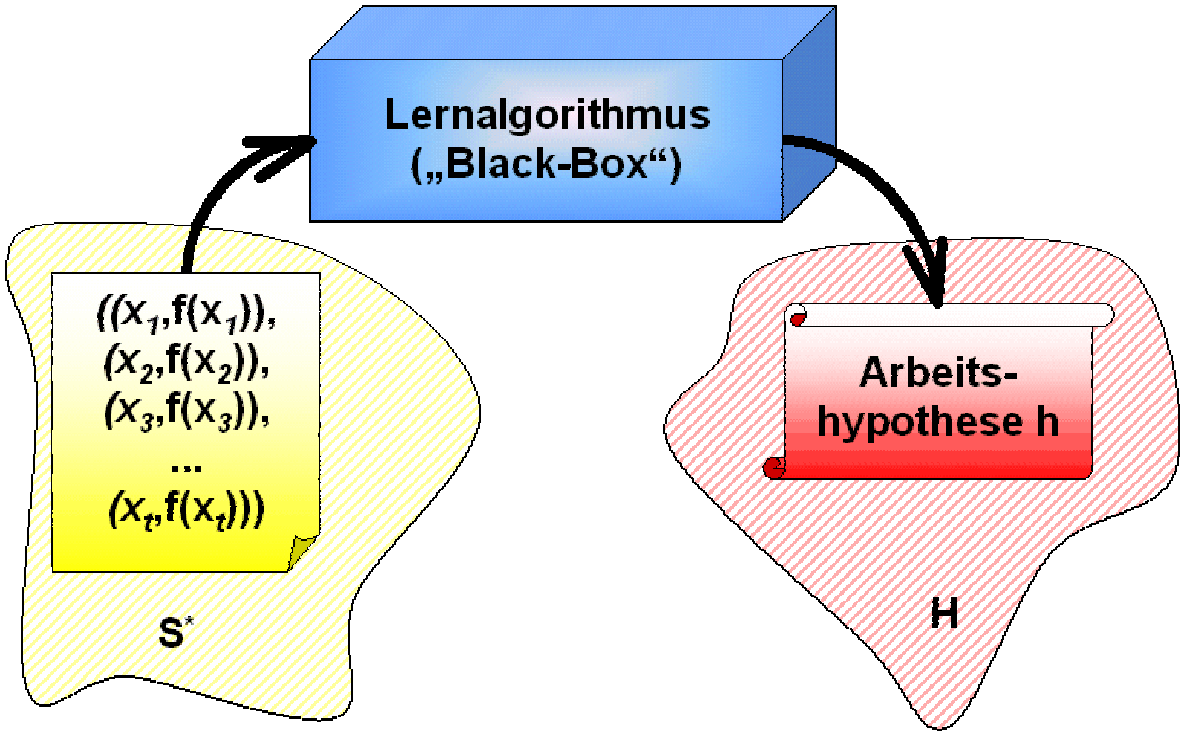

Abbildung 1 – Der Lernalgorithmus als Black-Box

In zwei der vorgestellten Lernmodelle nimmt der Algorithmus noch einen zusätzlichen Performance-Parameter ε entgegen, der einen Wert zwischen Null und Eins annehmen darf. Mittels dieses Parameters kann die gewünschte Güte des Lernergebnisses reguliert werden.

Man bezeichnet einen Lernalgorithmus als *zeitpolynomiell*, wenn die Rechenzeit, die der Algorithmus für eine Vorhersage benötigt, einerseits polynomiell durch die Größe eines Beispieles beschränkt ist, und andererseits polynomiell durch $^1/_\varepsilon$ beschränkt ist.

Sei *x* eine unendliche Folge von Beispielen, die jeweils aus *X* stammen, *f* eine Funktion aus F_n, und *A* ein determistischer Lernalgorithmus für *F*. Die „mistake indicator function“ *M* berechnet dann für einen Performance-Parameter ε und eine Länge *t den „mistake indicator“:*

$$x = (x_1, x_2, \ldots) \in X^{\mathbb{N}} \qquad f \in F_n$$
$$M^t_{A,f}(\varepsilon, x) = \begin{cases} 1 & \text{wenn } A(\varepsilon, \text{sam}_{t-1}(x, f))(x_t) \neq f(x_t) \\ 0 & \text{sonst} \end{cases} \tag{5.10}$$

Der „mistake indicator“ sagt aus, ob oder ob nicht der Lernalgorithmus nach Verarbeitung von den gegebenen *t-1* Beispielen das *t*-te Beispiel richtig klassifizieren wird. Für Lernalgorithmen, die keinen Performanceparameter ε benötigen, ändert man die Deklaration leicht ab:

$$M^t_{A,f}(x) = \begin{cases} 1 & \text{wenn } A(\text{sam}_{t-1}(x, f))(x_t) \neq f(x_t) \\ 0 & \text{sonst} \end{cases} \tag{5.11}$$

Randomisierte Lernalgorithmen nehmen einen zusätzlichen „Zufallsparameter“ entgegen, der ihr Ergebnis beeinflusst. In diesem Fall wird der Erwartungswert über alle „mistake indicators“ gebildet, in dem die Eins oder Null jeweils mit der Wahrscheinlichkeit für diesen Zufallsparameter gewichtet wird. Der so ermittelte Wert ist dann der „mistake indicator“ des randomisierten Lernalgorithmus.

Für die weitere Betrachtung werden nicht alle möglichen Beispielfolgen zugelassen, sondern nur unendliche Beispielfolgen, die bestimmten Bedingungen genügen. Für jede Beispiellänge *n* definieren die Teilmenge der erlaubten Beispiele:

$$\chi_n \subset X_n^{\mathbb{N}} \quad \forall n \tag{5.12}$$

In Anlehnung an (5.2) wird auch für die erlaubten Beispiele die Vereinigung über alle Beispiellängen definiert:

$$\chi = \bigcup_{n=1}^{\infty} \chi_n = \{?\} \tag{5.13}$$

Die Einschränkung könnte z.B. eine bestimmte Beziehung zwischen aufeinanderfolgenden Beispielen erzwingen. Zum Beispiel wird W_n als die Menge aller möglichen „Walks“ auf $X_n = \{0,1\}^n$ definiert. Unter einem „Walk“ versteht man Folgen, bei denen sich zwei aufeinanderfolgende Elemente maximal um ein einziges Bit unterscheiden (d.h. der Hamming-Abstand zweier aufeinanderfolgender Elemente darf nicht größer als Eins sein).

Es werden insgesamt drei Lernmodelle vorgestellt: Das erste ist ein deterministisches Lernmodell, mit einem Lernalgorithmus, der für eine beliebige, erlaubte Eingabe $x \in \chi$ und eine beliebige Funktion *f* aus F_n nur wenige Fehler machen darf.

6 Das Fehlerschranken-Modell

Im vorangegangenen Abschnitt ist der sogenannte „mistake indicator“ eingeführt worden. Er gibt für einen bestimmten Zeitpunkt des Lernens durch den Algorithmus an, ob oder ob nicht der Algorithmus eine falsche Voraussage für die Beispielklassifizierung machen wird.

6.1 Die Fehleranzahl

Betrachtet man nicht nur einen bestimmten Zeitpunkt, sondern den Algorithmus gesamtheitlich, entsteht der Begriff der *Fehleranzahl*. Darunter versteht man für eine Beispielfolge $x \in \chi$ und ein Konzept *f* die Anzahl der Zeitpunkte, zu denen der Lernalgorithmus *A* eine falsche Beispielklassifizierung vornimmt.

$$N_{A,f}(x) = \sum_{t=1}^{\infty} M_{A,f}^{t}(x) \tag{6.1}$$

Da *x* unendlich lang ist, kann die Fehlerschranke *N* nur bestimmt werden, wenn die Summe konvergiert.

6.2 Die Fehlerschranke

Betrachtet man nun nicht nur eine bestimmte Eingabe, und ein bestimmtes Konzept, sondern alle (gültigen) Eingaben und alle Konzepte (aus *F*), führt dies zu dem Begriff der „*Fehlerschranke*“[3]:

$$\hat{N}_{A,F_n,\chi_n} = \max_{f \in F_n} \max_{x \in \chi_n} N_{A,f}(x) \tag{6.2}$$

Die Fehlerschranke ist also für alle erlaubten Eingaben und alle Konzepte die Maximalanzahl an Fehlern, die der Lernalgorithmus während des Lernens eines Konzeptes machen wird. (Für probabilistische Lernalgorithmen muss diese Aussage entsprechend angepasst werden.)

6.3 Fehlerschrankenlernbarkeit

Man bezeichnet eine Konzeptklasse *F* als „*fehlerschrankenlernbar*“[4] durch χ mittels *H*, wenn es irgendeinen Lernalgorithmus *A* gibt, der in Polynomialzeit läuft, und für den die Fehlerschranke $\hat{N}_{A,F_n,\chi_n}$ nur polynomiell mit *n* (Länge der Beispiele) wächst.

Wenn eine Konzeptklasse *F* fehlerschrankenlernbar ist, führt dies zu der Frage, welche Fehlerschranke der „beste“ Lernalgorithmus für diese Konzeptklasse besitzt. Dazu betrachtet man das Minimum der Fehlerschranken über alle (Polynomialzeit-)Lernalgorithmen *A*:

$$\hat{N}_{F_n,\chi_n} = \min_{A} \hat{N}_{A,F_n,\chi_n} \tag{6.3}$$

Fehlerbeschränktes Lernen lässt sich für den Fall, dass $\chi_n = X_n^{\mathbb{N}}$, d.h. dass alle Beispielfolgen gültige Beispielfolgen sind, durch Littlestones Lernmodell realisieren, wie es in [5] vorgestellt wird. Bei den folgenden Beispielanwendungen wird fehlerbeschränktes Lernen für „random walks“ auf dem booleschen Würfel (d.h. $\chi_n = W_n$) benutzt werden.

[3] Englisch: „*mistake bound*“
[4] Englisch: „*mistake bound learnable*”

6.4 Exakte Fehlerschrankenlernbarkeit

Anwendungen sind oftmals darauf angewiesen, die Repräsentation der Hypothese in einer ganz bestimmten Darstellungsform zu erhalten. Zum Beispiel ist es extrem aufwendig, eine Funktion, welche eine Fallunterscheidung darstellt, als boolesches Netzwerk zu implementieren. In einem Szenario, in dem eben solch eine Fallunterscheidung gelernt werden soll, macht es also keinen Sinn, wenn der Lernalgorithmus seine Hypothese in Darstellung eines booleschen Netzwerkes repräsentieren würde. Eine konkrete Anwendung bestimmt also die Repräsentationsform der Hypothesen für den Algorithmus.

Im Allgemeinen beginnt ein Lernalgorithmus mit einer sehr weit gefassten Arbeitshypothese, und konkretisiert diese mit wachsender Anzahl an (Gegen-)Beispielen, bis sie nur noch aus einem Konzept besteht. Es kann aber der Fall eintreten, dass sich die Arbeitshypothese nie einem Konzept annähert, sondern immer aus zwei oder mehren Konzepten bestehen bleibt. (Daran ändert auch die Tatsache nichts, dass die Beispielfolge unendlich lang ist.)

Auch wenn eine Konzeptklasse exakt lernbar ist, kann der Fall eintreten, dass abhängig von der konkreten Beispielfolge $x \in \chi$ dem Lernalgorithmus *A* die Hände gebunden sind, so dass er sich „nicht festlegen“ kann. (Das ist z.B. dann der Fall, wenn dem Lernalgorithmus immer wieder dasselbe Beispiel präsentiert wird.)

Konkrete Anwendungen können aber erfordern, dass ein Lernalgorithmus eine exakte Hypothese ermittelt! Dies führt zu dem Begriff der exakten Fehlerschrankenlernbarkeit:

Man bezeichnet *F* als „exakt fehlerschrankenlernbar“[5] durch χ mittels *H*, wenn *F* bereits fehlerschrankenlernbar ist, und darüber hinaus der Lernalgorithmus *A* zu jedem Zeitpunkt weiß, ob sich die Arbeitshypothese exakt einem Konzept angenähert hat – oder nicht. Sobald der Lernalgorithmus weiß, dass die Arbeitshypothese exakt einem Konzept entspricht, soll der Algorithmus auch in der Lage sein, die gewünschte Repräsentation dieses Konzeptes in polynomieller Zeit zu berechnen. Falls sich die Arbeitshypothese des Lernalgorithmus noch keinem Konzept angenähert hat, soll der Lernalgorithmus dies „zugeben“, und nicht etwa eine unexakte Hypothese zurückgeben.

[5] Englisch: „exactly mistake bound learnable“

7 Irrfahrten auf dem booleschen Würfel

Für die folgenden Beispiele ist der Begriff der Irrfahrt auf den Kanten des booleschen Würfels erforderlich. Darunter ist zu verstehen, dass die Beispiele den Ecken eines angenommenen *n*-dimensionalen Würfels entsprechen, in dem Sinne, dass jede der 2^n Ecken mit einem eindeutigen *n*-Tupel aus Nullen und Einsen beschriftet ist.

Beispielsweise für *n=3* ist der boolesche Würfel wie folgt definiert:

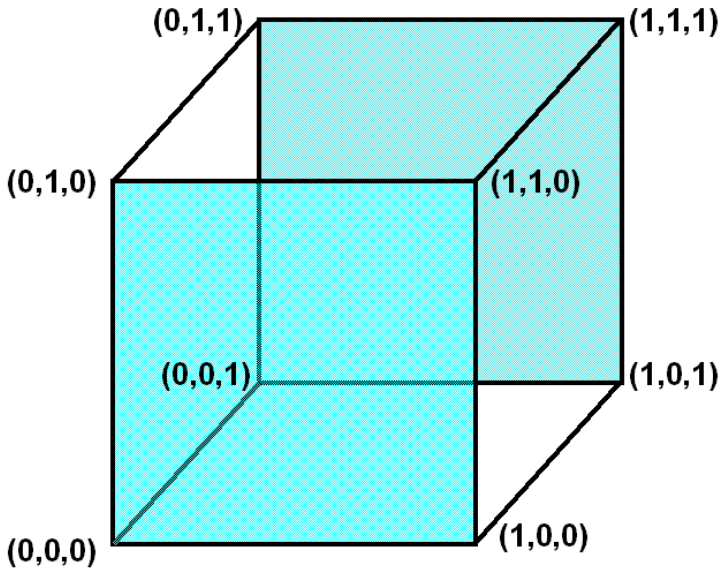

Abbildung 2 – der boolesche Würfel für n=3

Die Menge der gültigen Beispielfolgen χ_n, welche durch die Irrfahrt auf einem booleschen Würfel definiert wird, wird fortan als W_n bezeichnet. Dabei besitzt jedes Beispiel die Länge *n*.

Die Erzeugung der Beispiele unterliegt dabei der Einschränkung, dass von einer Ecke immer nur zu einer der *n* Nachbarecken gewechselt werden kann, oder auf der Stelle verharrt wird. Es stehen also zu jedem Zeitpunkt nur *n+1* Möglichkeiten (statt 2^n Möglichkeiten) für das jeweils folgende Beispiel zur Verfügung.

In Abbildung 2 ist deutlich zu sehen, dass bei einem Sprung von einem Eckpunkt zu einem seiner Nachbarpunkte im jedem Fall immer nur ein Bit (eine Stelle des Dreiertupels) verändert; oder anders formuliert, einen maximalen Hamming-Abstand von Eins besitzen.

Damit ist die Wahrscheinlichkeit des Wechsels von einer Ecke zu einer anderen, bzw. von einem Beispiel zu einem anderen $^1/_{n+1}$, wenn es sich dabei um einen „Nachbarn“ handelt, und Null anderenfalls:

$$Ws\left(x_t = v \mid x_1, \ldots, x_{t-1}\right) = \begin{cases} \dfrac{1}{n+1} & \text{falls } \operatorname{ham}\left(v, x_{t-1}\right) \le 1 \\ 0 & \text{sonst} \end{cases} \tag{7.1}$$

8 Lernen boolescher Schwellwertfunktionen

8.1 Einleitung

Boolesche Schwellwertfunktionen[6] lassen sich mit dem Fehlerschranken-Modell lernen, wie der folgende Algorithmus demonstriert. Die Idee hinter diesem Algorithmus ist es, dass die Vorhersage der Klassifizierung eines Beispiels immer derart berechnet wird, dass eine falsche Vorhersage das „Wissen" des Algorithmus über das Zielkonzept erhöht.

Die Klasse der booleschen Schwellwertfunktionen besteht aus Funktionen, deren Rückgabe Null oder Eins ist in Abhängigkeit von einem Gewichtsvektor *w* und dem Eingabevektor *x*. Eine boolesche Schwellwertfunktion liefert Eins zurück, wenn das Produkt aus dem Eingabevektor und dem Gewichtsvektor größer oder gleich einer Konstanten τ ist. Formal ausgedrückt enthält die Klasse der booleschen Schwellwertfunktionen Konzepte $f_{w,\tau} : X_n \to \{0,1\}$ mit $w = (w_1, \ldots, w_n) \in \{0,1\}^n$ und einem ganzzahligen, positiven Wert τ, so dass

$$f_{w,\tau}(x_1, \ldots, x_n) = \begin{cases} 1 & \text{falls } w_1 x_1 + \ldots + w_n x_n \geq \tau \\ 0 & \text{sonst} \end{cases} \tag{8.1}$$

Sei $Y^t = \left(y_1^t, \ldots, y_i^1, \ldots, y_n^t\right)$ das t-te Beispiel der Irrfahrt auf dem booleschen Würfel, welches der Algorithmus erhält. Weiterhin sei $\hat{l}^t$ die Klassifikation des t-ten Beispiels, welche der Algorithmus vorhersagt, und l^t die tatsächliche Klassifikation dieses Beispiels. Der Gewichtsvektor und der Schwellwert für die Funktion seien *w* und τ.

In diesem Zusammenhang wird der Begriff des Vektorraums bedeutsam: Für eine Menge *V* von Vektoren bezeichnet *span(V)* der durch die Vektoren aus *V* aufgespannte Vektorraum.

Algorithmus *A* ist wie folgt definiert:

Die vom Algorithmus vorhergesagte Klassifikation des ersten Beispiels ist $\hat{l}^1 = 0$, das bedeutet, der Algorithmus behauptet, das erste Beispiel ist nicht im zu lernenden Konzept enthalten. Solange der Algorithmus keinen Fehler macht, und die vorhergesagte Klassifikation mit der tatsächlichen Klassifikation der Beispiele übereinstimmt, sagt der Algorithmus für jedes Beispiel die Klassifikation des vorhergehenden Beispiels $\hat{l}^t = l^{t-1}$ voraus. Zu irgendeinem Zeitpunkt macht der Algorithmus einen (ersten) Fehler, und das Produkt aus Gewichtsvektor und Eingabevektor hat den Schwellwert τ entweder „von unten kommend" erreicht, oder „von oben kommend" unterschritten:

$$\begin{aligned} &\left(\left(w \cdot Y^{t-1} = \tau\right) \wedge \left(w \cdot Y^t = \tau - 1\right)\right) \vee \left(\left(w \cdot Y^{t-1} = \tau - 1\right) \wedge \left(w \cdot Y^t = \tau\right)\right) \\ &\Downarrow \quad \text{weil } \hat{l}^{t-1} = l^{t-1} \\ &\left(\left(l^{t-1} = 1\right) \wedge \left(w \cdot Y^t - \tau = -1\right)\right) \vee \left(\left(l^{t-1} = 0\right) \wedge \left(w \cdot Y^t - \tau = 0\right)\right) \\ &\Downarrow \\ &w \cdot Y^t - \tau = -l^{t-1} \end{aligned} \tag{8.2}$$

Weiterhin ist bekannt, dass die Kategorisierung des vorhergehenden Beispiels in Abhängigkeit von dem entsprechenden Eingabevektor ausgefallen sein muss:

[6] Englisch: „boolean treshold functions"

$$\begin{array}{l}\left(\left(w\cdot Y^{t-1}=\tau\right)\wedge\left(l^{t-1}=1\right)\right)\vee\left(\left(w\cdot Y^{t-1}=\tau-1\right)\wedge\left(l^{t-1}=0\right)\right)\\ \Downarrow\\ \left(\left(w\cdot Y^{t-1}-\tau=0\right)\wedge\left(l^{t-1}-1=0\right)\right)\vee\left(\left(w\cdot Y^{t-1}-\tau=-1\right)\wedge\left(l^{t-1}-1=-1\right)\right)\\ \Downarrow\\ w\cdot Y^{t-1}-\tau=l^{t-1}-1\end{array} \quad (8.3)$$

Der Algorithmus soll aus seinen Fehlern lernen, und merkt sich daher alle Fehlentscheidungen, indem er sie als Vektoren „verkleidet“ in einem Array speichert. Dazu normalisiert er die Gleichungen zuerst:

$$\begin{array}{l}w\cdot Y^{t}-\tau+l^{t-1}=0\\ w\cdot Y^{t-1}-\tau+1-l^{t-1}=0\end{array} \quad (8.4)$$

und speichert die Vektoren dann in einem Array *S* ab:

$$S=\left\{\left(Y^{t-1},-1,1-l^{t-1}\right),\left(Y^{t},-1,l^{t-1}\right)\right\} \quad (8.5)$$

Der Algorithmus kann diese in *S* abgespeicherten Vektoren (die Gleichungen entsprechen) für die darauffolgenden Beispiele verwenden, um einmal gemachte Fehler zukünftig zu vermeiden. Jedes mal, wenn der Algorithmus einen weiteren Fehler macht, legt er den Vektor der entsprechenden Gleichung in *S* ab. Für ein beliebiges Beispiel Y^t sagt der Algorithmus die Klassifikation wie folgt voraus:

```
IF (Y^t,-1,l^t-1) INSIDE span(s) THEN
     predict 1-l^t-1
ELSE
     predict l^t-1
     IF l^t-1 != l^t THEN
          add(Y^t,-1,l^t-1) to S
```

Die Vektoren in *S* entsprechen einer Menge linear unabhängiger Gleichungen in *w* und τ. Daher ist der Algorithmus in der Lage, das Zielkonzept $f_{w,\tau}$ zu berechnen, sobald *S n+1* Vektoren enthält. Der Algorithmus hat das Konzept daher nach maximal *n+1* Fehlern gelernt. Der Beweis für die Korrektheit dieses Algorithmus kann dem Paper entnommen werden.

9 Lernen zweitermiger RSE

9.1 Einleitung

Eine zweitermige RSE[7] ist die Parität zweier Monotonmonome, z.B. $(x_1 \wedge x_2) \oplus (x_3 \wedge x_4 \wedge x_5)$. Es ist bekannt, dass diese Klasse im PAC Modell nicht exakt lernbar ist, aber mit einer größeren Hypothesenklasse gelernt werden kann (siehe [3]). Der folgende Algorithmus wird eine Arbeitshypothese verwenden, die selber keine zweitermige RSE ist. Auf eine genauere Beschreibung der Hypothesenklasse, die der Lernalgorithmus verwendet, wird verzichtet. Statt dessen werden die Hypothesen implizit durch den Algorithmus beschrieben. Prinzipiell handelt es sich um ineinander verschachtelte Fallunterscheidungen, die auf den bisher gesammelten Informationen basieren.

Es wird gezeigt, dass mit jedem Fehler, den der Algorithmus macht, die Menge der gesammelten Informationen zunimmt, und gleichzeitig die Anzahl der potentiellen Zielkonzepte so lange abnimmt, bis schlussendlich nur noch ein einziges Zielkonzept übrig ist.

9.2 Idee für einen Algorithmus

Sei $X = \{x_1, \ldots, x_n\}$ eine Menge von Variablen, und m_0 und m_1 die Monotonmonome über X, und sei $c=m_0 \oplus m_1$ das Zielkonzept. Da das Zielkonzept c eine Funktion ist, kann es auch als solche verwendet werden: $c(Y^t)$ ergibt den Wert der RSE für das t-te Beispiel, und damit die Klassifikation des t-ten Beispiels.

Ein Monotonmonom wird durch die Menge der Variablen beschrieben, die in ihm enthalten sind. Variablen die entweder in m_0 oder in m_1 vorkommen, werden als „relevant“ bezeichnet. Um aus jedem Fehler Informationen gewinnen zu können, verwendet der Algorithmus die folgenden Mengen, die er während der Verarbeitung immer wieder aktualisiert:

- *E(i)* wird für jedes *i=1,...,n* mit der Menge aller x_i initialisiert. Dann werden nach und nach Elemente aus *E(i)* entfernt, und zwar so lange die folgenden Bedingungen eingehalten sind:
  - Wenn x_i in exakt einem Monom m_j (aus *c*) enthalten ist, dann sollen auch alle anderen Variablen aus diesem Monom in *E(i)* enthalten sein: $E(i) \supseteq m_j$.
  - Wenn x_i in beiden Monomen (aus *c*) enthalten ist, dann sollen auch alle anderen Variablen, die in beiden Monomen enthalten sind, in *E(i)* enthalten sein: $E(i) \supseteq m_0 \cap m_1$.
  - *E(i)* ist eine Obermenge der Literale, welche in den Monomen sind, in denen x_i ist.
- *H(i)* wird für jedes *i=1,...,n* mit der Menge aller x_i initialisiert. Dann werden nach und nach Elemente aus *H(i)* entfernt, und zwar so lange die folgenden Bedingungen eingehalten sind:
  - Wenn x_i in exakt einem Monom m_j (aus *c*) enthalten ist, dann sollen alle Variablen aus dem jeweils anderen Monom m_{1-j} in *H(i)* enthalten sein: $H(i) \supseteq m_{1-j}$.
  - Wenn x_i in beiden Monomen (aus *c*) enthalten ist, dann sollen auch alle anderen Variablen, die in beiden Monomen enthalten sind, in *H(i)* enthalten sein: $H(i) \supseteq m_0 \cap m_1$.

[7] Ring Sum Extension

- *T* wird mit der Menge aller x_i initialisiert. Dann werden nach und nach Elemente aus *T* entfernt, und zwar so lange die folgende Bedingung eingehalten ist:
  - § Jedes x_i, welches sowohl in m_0 als auch in m_1 enthalten ist, soll auch in *T* enthalten sein: $T \supseteq m_0 \cap m_1$.
- *R* wird mit der Menge aller x_i initialisiert. Dann werden nach und nach Elemente aus *R* entfernt, und zwar so lange die folgende Bedingung eingehalten ist:
  - § Jedes x_i, welches entweder in m_0 oder in m_1 enthalten ist, soll auch in *R* enthalten sein: $R \supseteq m_0 \cup m_1$.
  - § *R* beinhaltet also zu jedem Zeitpunkt (unter anderem) die Menge der relevanten Variablen.
- *S* wird mit der leeren Menge initialisiert. Dann werden nach und nach Elemente in *S* eingefügt, und zwar so lange die folgende Bedingung eingehalten ist:
  - § Jedes x_i, welches in *S* enthalten ist, soll auch sowohl in m_0 als auch in m_1 enthalten sein: $S \subseteq m_0 \cap m_1$.
- *V* wird mit der leeren Menge initialisiert. Dann werden nach und nach Elemente in *V* eingefügt, und zwar so lange die folgende Bedingung eingehalten ist:
  - § Jedes x_i, welches in *V* enthalten ist, soll auch in m_0 oder in m_1 enthalten sein: $V \subseteq m_0 \cup m_1$.
  - § *V* ist also zu jedem Zeitpunkt eine Teilmenge der relevanten Variablen.

Beachte, dass gilt $0 \leq |E(i)|, |H(i)|, |S|, |T|, |V|, |R| \leq n$. Falls in einem gegebenen Beispiel *Y* alle Variablen y_i, die in *E(i)* (bzw. in *H(i)*) enthalten sind, den Wert Eins haben, so sagt man dass „*E(i)* (bzw. *H(i)*) von dem Beispiel Y^t befriedigt wird“:

$$E(i) \text{ wird durch das Beispiel } Y^t \text{ befriedigt} \Leftrightarrow \forall x_j \in E(i): y_j = 1 \quad (9.1)$$

$$H(i) \text{ wird durch das Beispiel } Y^t \text{ befriedigt} \Leftrightarrow \forall x_j \in H(i): y_j = 1 \quad (9.2)$$

Der Lernalgorithmus testet für jedes Beispiel zunächst, ob das gekippte Bit y_i^{t+1} in *R* enthalten ist, oder nicht.

Falls das Bit y_i^{t+1} nicht in *R* enthalten ist, geht der Lernalgorithmus davon aus, dass es nicht relevant ist, und an der Klassifikation des Beispiels gegenüber dem vorhergehenden Beispiel nichts verändert haben kann: der Algorithmus wiederholt also die Klassifikation des vorhergehenden Beispiels.

Falls das Bit y_i^{t+1} in *R* enthalten ist, weiß der Algorithmus zwar nicht mit Sicherheit, dass das Bit relevant ist, er kann es aber nicht mehr ausschließen, und geht daher davon aus, dass es relevant ist. In diesem Fall unterscheidet der Algorithmus abhängig von der Klassifikation des vorhergehenden Beispiels (0 oder 1) und der Art des Bit-Wechsels (0→1, oder 1→0) vier Fälle, die S1 bis S4 benannt sind:

1. Der Fall S1: $c(Y^t) = 1 \wedge y_i^t = 0 \wedge y_i^{t+1} = 1$

Wenn das Bit *i* in Y^t von Null nach Eins gekippt ist, und $c(Y^t)$ den Wert Eins ergab, wurde für Beispiel *t* offenbar genau eines der Monomer aus *c* zu Eins ausgewertet, und das andere zu

Null. Jenes, welches zu Eins ausgewertet wurde, sei o.B.d.A.[8] das Monom m_1, welches zu Null ausgewertet wurde, das Monom m_0:

$$c\left(Y^t\right)=1 \Rightarrow m_1\left(Y^t\right)=1 \wedge m_0\left(Y^t\right)=0 \quad \text{(o.B.d.A.)} \tag{9.3}$$

Da $y_i^t=0$ kann x_i in m_1 nicht enthalten sein, in m_0 kann es enthalten sein – muss es aber nicht (bzw. ist es genau dann, wenn es relevant ist, was dem Algorithmus aber nicht bekannt ist).

Weil m_1 zu Eins ausgewertet worden ist, wird es offensichtlich von Y^{t+1} befriedigt. Der Algorithmus testet nun, ob auch *E(i)* von Y^{t+1} befriedigt wird.

Wenn *E(i)* von Y^{t+1} befriedigt wird, und x_i eine relevante Variable ist (wovon der Algorithmus zwar ausgeht, aber was er nicht mit Sicherheit weiß), dann muss auch m_0 von Y^{t+1} befriedigt werden. (Die Begründung ist, dass wenn x_i relevant ist, es in m_0 enthalten sein muss – also müssen alle Variablen aus m_0 in *E(i)* enthalten sein ($E(i) \supseteq m_0$), welches wiederum befriedigt wird. Daher muss auch m_0 befriedigt werden.) Wenn aber m_0 und m_1 durch das Beispiel befriedigt werden, nimmt $c(Y^{t+1})$ den Wert 0 an: der Algorithmus sagt 0 vorher.

Falls x_i nicht relevant ist (also auch nicht in m_0 ist), hat der Algorithmus einen Fehler gemacht: die richtige Klassifikation wäre Eins gewesen. Der Algorithmus lernt dann aus diesem Fehler und entfernt x_i aus *R*.

Wenn *E(i)* nicht von Y^{t+1} befriedigt wird, vermutet der Algorithmus, dass auch m_0 nicht von Y^{t+1} befriedigt wird (was er aber nicht sicher weiß). Der Algorithmus sagt dann für das Beispiel die Klassifikation 1 voraus. Sollte m_0 aber doch von dem Beispiel befriedigt werden, so dass der Algorithmus einen Fehler gemacht hat, und die richtige Klassifikation 1 gewesen wäre, lernt der Algorithmus aus seinem Fehler, indem er alle Variablen aus *E(i)* entfernt, die durch Y^{t+1} nicht befriedigt werden:

$$E(i) := E(i) \cap Y^{t+1} \tag{9.4}$$

Wenn x_i in m_0 enthalten ist, ist $E(i) \supseteq m_0$ (nach der Definition von *E(i)*). Falls nun *E(i)* durch Y^{t+1} befriedigt wird, so auch m_0, also ist $m_0(Y^{t+1})=1$, und damit $c(Y^t)=0$. Die Vorhersage ist also korrekt.

2. Der Fall S2: $c\left(Y^t\right)=1 \wedge y_i^t=1 \wedge y_i^{t+1}=0$

Genau wie in Fall S1 ist bekannt, dass Y^t offensichtlich genau eines der beiden Monome befriedigt. Im Gegensatz zu Fall S1 kippt das Bit Y_i^{t+1} aber nicht von 0 nach 1, sondern von 1 nach 0. Analog zu Fall S1 sei das Monom, welches den Wert 1 besitzt, wieder m_1 (siehe (9.3)).

Der Algorithmus macht wieder eine Fallunterscheidung, je nachdem, ob das Beispiel Y^t *E(i)* befriedigt, oder nicht.

Falls das Beispiel Y^t *E(i)* befriedigt, geht der Algorithmus wieder davon aus (da er es nicht besser weiß), dass die Variable x_i relevant ist. Die Variable x_i kann dann entweder nur in m_0, oder nur in m_1, oder in beiden Monomen enthalten sein.

Falls x_i in beiden Monomen enthalten war, müssen beide Monome nun den Wert 0 haben, *c* also den Wert 0. Falls x_i nur in m_1 enthalten ist, würde dieses nun den Wert 0 haben, während m_0 seinen Wert 0 behielte: wieder besäßen beide Monome den Wert 0, und damit auch *c* den Wert 0. Falls x_i nur in m_0 enthalten ist, müssen alle Variablen aus m_0 in *E(i)* enthalten sein, und da *E(i)* durch Y^t befriedigt wird, wird auch m_0 durch Y^t befriedigt: das ist ein Wider-

[8] ohne Beschränkung der Allgemeinheit

spruch zu der Annahme, dass m_0 den Wert 0 besitzt, dieser Fall kann also in S2 nicht eintreten!.

Der Algorithmus sagt also 0 voraus. Der einzige Fehler kann auftreten, wenn die Variable x_i doch nicht relevant ist: in diesem Fall wird x_i aus R entfernt.

Falls das Beispiel Y^t *E(i)* nicht befriedigt, überprüft der Algorithmus, ob Y^{t+1} *H(i)* befriedigt.

Falls Y^{t+1} *H(i)* befriedigt, geht der Algorithmus wieder davon aus, dass x_i eine relevante Variable ist, also mindestens in einem der beiden Monome enthalten ist. Sie kann nicht in beiden Monomen enthalten sein, weil dann Y^{t+1} *H(i)* nicht befriedigen könnte!

(Begründung: Wäre die Variable x_i in beiden Monomen enthalten, wäre sie auch in *H(i)* enthalten. Wenn *H(i)* aber durch Y^{t+1} befriedigt würde, müsste $y_i^{t+1}=1$ sein, und das kann im Fall S2 nicht auftreten.)

Falls x_i also nur in einem Monom enthalten ist, so wird dieses Monom durch das Beispiel y^{t+1} auf jeden Fall zu 0 ausgewertet. Das jeweils andere Monom wird zu 1 ausgewertet, da dessen Variablen vollständig in *H(i)* enthalten sind, welches durch y^{t+1} ja befriedigt wird. Wenn eines der Monome 1 ist, und das andere 0, ist c 0 – der Algorithmus sagt also für die Klassifikation 0 voraus.

Ein Fehler kann auftreten, wenn x_i nicht relevant ist, also in keinem Monom enthalten ist. In diesem Fall bleibt die Klassifizierung gegenüber dem vorhergehenden Beispiel unverändert, also 1, und der Algorithmus hat sich geirrt. Er reagiert darauf, indem er x^j aus R entfernt.

Falls Y^{t+1} *H(i)* nicht befriedigt, kann x_i wieder in m_0, oder in m_1, oder in beiden Monomen enthalten sein. I) Falls x_i nur in m_0 enthalten ist, behält dieses seinen Wert von 0 bei, während m_1 nun ebenfalls einen Wert von 0 besitzt. (Begründung: Alle Variablen aus m_1 sind dann in *H(i)* enthalten, welches von Y^{t+1} nicht befriedigt wird, also wird mit gewisser Sicherheit m^1 ebenfalls nicht von Y^{t+1} befriedigt.) II) Falls x_i nur in m_1 enthalten ist, erhält dieses einen Wert von 0, da x_i auf 0 kippt. Dann behält m_0 unverändert den Wert 0. Die RSE wird also auf jeden Fall *c=0*. III) Sollte x_i sowohl in m_0 als auch in m_1 enthalten sein, nimmt c auf jeden Fall den Wert 0 an. Ein Fehler kann nur in I) aufgetreten sein, sofern H(i) (fälschlicherweise) Variablen enthält, die nicht in m_1 enthalten sind, und deren Werte in Y^{t+1} 0 sind: der Algorithmus lernt aus diesem Fehler, indem er *H(i)* derart verkleinert, dass er es mit Y^{t+1} schneidet.

Die Fälle S3 ($c(Y^t)=0 \wedge y_i^t=0 \wedge y_i^{t+1}=1$) und S4 ($c(Y^t)=0 \wedge y_i^t=1 \wedge y_i^{t+1}=0$) werden in ähnlicher Weise behandelt. Der interessierte Leser kann sie entweder im Originaldokument nachschlagen, oder dem folgenden Algorithmus entnehmen.

Es ist zu sehen, dass die Bedingungen an *E(i)*, *H(i)*, etc. stets eingehalten werden. Weiterhin kann man sehen, dass jeder Fehler des Algorithmus unmittelbar zu einer Veränderung, also zu einer Zunahme des Wissens führt. Jede diese Veränderungen entfernt mindestens ein Element aus mindestens einer der Mengen R, T, *E(i)*, *H(i)*, oder fügt mindestens ein Element zu V oder S hinzu. Weil alle diese Mengen eine Größe zwischen 0 und n besitzen, und es nur *O(n)* Mengen gibt, wird es nur $O(n^2)$ Aktualisierungen von Mengen geben. Demzufolge kann der Algorithmus nur $O(n^2)$ Fehler machen.

Unter der Vorbedingung der Irrfahrt auf einem booleschen Würfel sind zweitermige RSE mit $O(n^2)$ Fehlern lernbar.

9.3 Der Algorithmus

```
S1: c(Y^t)=1 ∧ y_i^t=0 ∧ y_i^t+1=1
WENN Y^t+1 befriedigt E(i)
     DANN sage_vorher 0; FALLS fehler: entferne x_i aus R
     SONST sage_vorher 1; FALLS fehler: E(i):=E(i)∩Y^t+1
```

S2: $c(Y^t)=1 \wedge y_i^t=1 \wedge y_i^{t+1}=0$
WENN Y^t befriedigt E(i)
  DANN sage_vorher 0; **FALLS** fehler: entferne x_i aus R
  SONST WENN Y^{t+1} befriedigt H(i)
    DANN sage_vorher 1; **FALLS** fehler: entferne x_i aus R
    SONST sage_vorher 0; **FALLS** fehler: $H(i):=H(i)\cap Y^{t+1}$

S3: $c(Y^t)=0 \wedge y_i^t=0 \wedge y_i^{t+1}=1$
WENN es ein $j\neq i$ gibt, so dass $Y_j^t=0$ und $x_j\in R$
  DANN WENN $x_j\in T$
    DANN sage_vorher 0; **FALLS** fehler: entferne x_j aus T
    SONST WENN Y^{t+1} befriedigt H(j)
      DANN sage_vorher 1; **FALLS** fehler:
        entferne x_j aus R
      SONST sage_vorher 0; **FALLS** fehler:
        $H(j):=H(j)\cap Y^{t+1}$
  SONST sage_vorher 0;

S4: $c(Y^t)=0 \wedge y_i^t=1 \wedge y_i^{t+1}=0$
WENN $x_i\in S$
  DANN sage_vorher 0;
  SONST WENN $x_i\in V$
    DANN WENN Y^t befriedigt E(i)
      DANN sage_vorher 1; **FALLS** fehler:
        füge x_i zu S hinzu
      SONST sage_vorher 0; **FALLS** fehler: $E(i):=E(i)\cap Y^t$
    SONST sage_vorher 0; **FALLS** fehler: füge x_i zu V hinzu

10 Das probabilistische Fehlerschranken-Modell

Bei diesem Lernmodell, und auch bei dem folgenden Modell sind die Beispielfolgen *x* die Pfade von zeitdiskreten, stochastischen Prozessen. Eine Klasse von stochastischen Prozessen, die allesamt Beispiele mit einer Länge *n* erzeugen, wird als ρ_n bezeichnet. Die von diesen Prozessen erzeugten Beispiele werden wieder χ_n bezeichnet. (Daraus folgt, dass die Beispielfolgen, wie sie von den stochastischen Prozessen erzeugt werden, gültig sein sollen!) Wieder gilt:

$$\rho = \bigcup_{n=1}^{\infty} \rho_n = \rho_1 \cup \rho_2 \cup \rho_3 \cup \ldots \tag{10.1}$$

Bislang wurde von dem Lernalgorithmus gefordert, dass er für alle Beispielfolgen und alle Konzepte nur wenige Fehler macht. Dies ändert sich bei dem probabilistischen Fehlerschranken-Modell: es wird statt dessen gefordert, dass ein Lernalgorithmus *für die meisten Beispielfolgen* nur wenige Fehler macht.

Ein deterministischer Lernalgorithmus A nimmt nun zusätzlich einen Vertrauens-Parameter[9] δ entgegen, der zusammen mit der gegebenen Beispielfolge beeinflusst, welche Arbeitshypothese der Algorithmus nach einer bestimmten Anzahl verarbeiteter Beispiele erstellt hat.

10.1 Die *Fehleranzahl*

Für den Begriff der *Fehleranzahl betrachtet man i*n Anlehnung an (6.1) wieder für ein bestimmtes Konzept *f* die Summe der Fehler, welche der Lernalgorithmus beim Verarbeiten einer unendlich langen Beispielfolge $x \in \chi$ macht – vorausgesetzt, diese Summe konvergiert:

$$N_{A,f}(\delta, x) = \sum_{t=1}^{\infty} M_{A,f}^{t}(\delta, x) \tag{10.2}$$

10.2 Die δ-Vertrauens-Fehlerschranke

Anders als beim „Fehlerschranken-Modell“, bei welchem die Fehleranzahl eines Lernalgorithmus über alle Konzepte und alle Eingaben maximiert wurde, werden bei dem probabilistischen Fehlerschranken-Modell eine bestimmte Anzahl von Fehlern unter den Tisch fallen gelassen. Genau genommen ist man bereit, mit einer Wahrscheinlichkeit von (maximal) δ zuzulassen, dass der Algorithmus die Fehlerschranke überschreitet:

$$\hat{N}_{A,F_n,\rho_n,\delta} = \min\left\{ m \in \mathbb{N} : \forall \rho \in \rho_n \forall f \in F_n : \underbrace{P\{x : N_{A,f}(\delta, x) > m\}}_{\substack{\text{Wahrscheinlichkeit, dass Fehleranzahl} \\ \text{des Lernalgorithmus größer als die} \\ \text{Fehlerschranke ist}}} < \delta \right\} \tag{10.3}$$

Die Aussage von (10.3) ist also von der Art „Der Lernalgorithmus *A* macht nur $\hat{N}_{A,F_n,\rho_n,\delta}$ Fehler. Allerdings gibt es mit Wahrscheinlichkeit δ einige Ausrutscher.“.

[9] Englisch: „confidence-parameter“

10.3 Probabilistische Fehlerschrankenlernbarkeit

In Anlehnung an die Fehlerschrankenlernbarkeit bezeichnet man eine Konzeptklasse *F* als „*probabilistisch fehlerschrankenlernbar*“[10], wenn es irgendeinen Lernalgorithmus *A* gibt, der in Polynomialzeit läuft, und für den die Fehlerschranke $\hat{N}_{A,F_n,\rho_n,\delta}$ nur polynomiell mit *n* (Länge der Beispiele) und $^1/_\delta$ wächst.

Man gesteht den potentiellen Lernalgorithmen für eine Konzeptklasse *F* also zu, für längere Beispiele und für weniger Fehlerschranken-„Ausrutscher“ auch entsprechend mehr Rechenzeit zu benötigen.

Wenn eine Konzeptklasse *F* probabilistisch fehlerschrankenlernbar ist, führt dies zu der Frage, welche Fehlerschranke der „beste“ Lernalgorithmus für diese Konzeptklasse besitzt. Dazu betrachtet man in Anlehnung (6.3) das Minimum der probabilistischen Fehlerschranken über alle (Polynomialzeit-)Lernalgorithmen *A*:

$$\hat{N}_{F_n,\rho_n,\delta} = \min_A \hat{N}_{A,F_n,\rho_n,\delta} \tag{10.4}$$

10.4 Exakte probabilistische Fehlerschrankenlernbarkeit

Analog zu der exakten Fehlerschrankenlernbarkeit bezeichnet man *F* als „exakt probabilistisch fehlerschrankenlernbar“[11] durch ρ mittels *H*, wenn *F* bereits probabilistisch fehlerschrankenlernbar ist, und darüber hinaus der Lernalgorithmus *A* zu jedem Zeitpunkt weiß, ob sich die Arbeitshypothese exakt einem Konzept angenähert hat – oder nicht.

Sobald der Lernalgorithmus weiß, dass die Arbeitshypothese exakt einem Konzept entspricht, soll der Algorithmus auch in der Lage sein, die gewünschte Repräsentation dieses Konzeptes in polynomieller Zeit zu berechnen. Falls sich die Arbeitshypothese des Lernalgorithmus noch keinem Konzept angenähert hat, soll der Lernalgorithmus dies „zugeben“, und nicht etwa eine unexakte Hypothese zurückgeben.

[10] Englisch: „probably mistake bound learnable“
[11] Englisch: „exactly probably mistake bound learnable“

11 Lernen zweitermiger DNF

11.1 Einleitung

In diesem Abschnitt geht es um das Lernen zweitermiger DNF mittels probabilistischem Fehlerschrankenlernen (siehe 10 Das probabilistische Fehlerschranken-Modell). Wieder werden die Beispiele durch eine Irrfahrt auf dem booleschen Würfel erzeugt (siehe 7 Irrfahrten auf dem booleschen Würfel). Es wird gezeigt, dass zweitermige DNF in diesem Fall lernbar sind. Es sei darauf hingewiesen, dass DNF allgemein im PAC Modell nicht exakt lernbar sind ([6]), sondern nur, wenn Fragen („membership queries“) erlaubt sind.

Wie bei dem Algorithmus zum Lernen zweitermiger RSE wird auch hier nicht auf die exakte Ausprägung der Hypothesenklasse eingegangen, sie ergibt sich vielmehr implizit aus den Handlungen des Algorithmus.

11.2 Idee für einen Algorithmus

Für die Beschreibung des Algorithmus wird auf Begriffe des RSE-Algorithmus zurückgegriffen. Wenn $c = m_0 \vee m_1$ das Zielkonzept ist, mit m_0 und m_1 Monomen über $X = \{x_1, \overline{x}_1, \ldots, x_n, \overline{x}_n\}$, wobei $\overline{x}_i$ der Negation von x_i entspricht, dann seien wieder die folgenden Mengen definiert:

- *E(i)* wird für jedes *i=1,...,2n* mit *X* initialisiert. Dann werden nach und nach Elemente aus *E(i)* entfernt, und zwar so lange die folgenden Bedingungen eingehalten sind:
  - Wenn x_i in exakt einem Monom m_j (aus *c*) enthalten ist, dann sollen auch alle anderen Variablen aus diesem Monom in *E(i)* enthalten sein: $E(i) \supseteq m_j$.
  - Wenn x_i in beiden Monomen (aus *c*) enthalten ist, dann sollen auch alle anderen Variablen, die in beiden Monomen enthalten sind, in *E(i)* enthalten sein: $E(i) \supseteq m_0 \cap m_1$.
  - *E(i)* ist eine Obermenge der Literale, welche in den Monomen sind, in denen x_i ist.

  Die obigen drei Punkte gelten sowohl für jedes x_i als auch für jedes $\overline{x}_i$, wobei für die nicht-negierten Variablen der Bereich von 1 bis *n* verwendet wird, und für die negierten Versionen der Bereich von *n+1* bis *2n*.
- *S* wird mit der leeren Menge initialisiert. Dann werden nach und nach Elemente in *S* eingefügt, und zwar so lange die folgende Bedingung eingehalten ist:
  - Jedes x_i, welches in *S* enthalten ist, soll auch sowohl in m_0 als auch in m_1 enthalten sein: $S \subseteq m_0 \cap m_1$.

Die Idee der Vorhersagestrategie ist sehr ähnlich zu der Strategie für die zweitermigen RSE (siehe 9 Lernen zweitermiger RSE), und benutzt an vielen Stellen die selben Argumente. Daher wird die Idee an dieser Stelle nur grob skizziert, tiefergehende Einzelheiten können entweder dem folgenden Algorithmus, oder der Interpretation im Rahmen der RSE entnommen werden.

Die Hauptidee ist es, dass in jedem Monom von *c* ein Literal l_i enthalten ist, z.B. x_i oder $\overline{x}_i$, welches zwar in diesem, aber nicht in dem jeweils anderen Monom von *c* enthalten ist. Diese Literale werden als „*private Literale*“ von m_0 bzw. von m_1 bezeichnet. Weil die kontinuierlichen Veränderungen von *E(i)* für ein privates Literal l_i stets die Eigenschaft $E(i) \supseteq m_j$ erhal-

ten, hat sich die Menge *E(i)* irgendwann dem zugehörigen Monom m_j angenähert. Außerdem finden Veränderungen von *E(i)* nur dann statt, wenn das zugehörige Literal relevant ist, d.h. mindestens in einem der beiden Monome enthalten ist. Daher wird für alle nicht-relevanten Literale, die also in keinem Monom enthalten sind, das zugehörige *E(i)* niemals verändert, und bleibt daher konstant bei seinem Initialwert *X*. Insbesondere wird keines dieser Literale jemals in *S* eingefügt werden. *S* bleibt also immer eine Teilmenge der Schnittmenge der Monome, und daher wird der Fall S4 auch niemals einen Fehler produzieren.

Um zu zeigen, dass dieser Algorithmus im probabilistischen Fehlerschranken-Modell liegt, ist lediglich zu beeisen, dass seine Fehleranzahl durch ein Polynom *p(n,δ)* beschränkt ist, jedenfalls mit einer Wahrscheinlichkeit von *1*-δ. Im Gegensatz zum vorhergehenden Lernen zweitermiger RSE wird bei den DNF nicht jedes Mal eine Veränderung vorgenommen, wenn ein Fehler auftritt. Das ist in den Fällen S2 und S3 je einmal der Fall: „tue nichts“. Aber es kann ein probabilistisches Argument zu Rate gezogen werden: wenn z.B. in Fall S2 ein Fehler auftritt, und keine Veränderung vorgenommen wird, dann gibt es eine Wahrscheinlichkeit von mindestens $1/(n+1)^3$, dass die folgenden Beispiele mehr Informationen liefern werden. Das folgt aus der Tatsache, dass die Zielfunktion in diesem Fall den Wert Eins annimmt. Daher sind höchstens zwei Veränderungen von privaten Literalen nötig (eine pro Monom), damit ein Beispiel wieder den Wert Null besitzt. (Begründung: Wenn die Zielfunktion den Wert Eins annimmt, nimmt auch mindestens eines der Monome den Wert Eins an. Dazu ist es nötig, dass alle Literale dieses Monoms den Wert Eins annehmen: wenn aber eines der Literale kippt, nimmt das Monom den Wert Null an. Nehmen beide Monome den Wert Null an, nimmt auch die Zielfunktion den Wert Null an.)

Im nächsten Schritt kippt ein Bit, so dass die Menge *E(i)* oder *S* aktualisiert wird. Daraus folgt, dass wenn in Schritt 2 ein Fehler auftritt, ohne dass eine Veränderung vorgenommen wird, mit großer Wahrscheinlichkeit $O(n^3)$ Fehler nötig sind, bis es zur nächsten Veränderung kommt! (Selbiges gilt auch für Fehler ohne folgende Veränderungen in Schritt S3.)

Nachdem die Mengen *E(i)* ausreichend oft verändert worden sind, enthalten einige dieser Mengen die Monome des Zielkonzeptes, einige andere Mengen nach wie vor *X*, und die restlichen Mengen enthalten Variablen, von denen bekannt ist, dass sie in $S = m_1 \cap m_0$ enthalten sind.

Also handelt es sich hier um einen Algorithmus, der zweitermige DNF im probabilistischem Fehlerschranken-Modell lernt, d.h. mit der Wahrscheinlichkeit *1*-δ höchstens p(n,δ) Fehler macht, und die Zielfunktion exakt identifiziert. (*p()* ist ein entsprechendes Polynom.) Voraussetzung ist unverändert, dass die Beispiele durch eine gleichförmige Irrfahrt auf dem booleschen Würfel erzeugt werden.

11.3 Der Algorithmus

Weil die Variablen x_i und ihre negierten Pendants aus Sicht des Algorithmus als eigenständige Variablen betrachtet werden, muss der Algorithmus nicht nur auf das „Kippen“ von Variablen, sondern auch auf das „Kippen“ der negierten Variablen reagieren. In dem folgenden Algorithmus wird nur auf das Kippen der nicht-negierten Variablen reagiert:

S1: $c(Y^t)=0 \wedge y_i^t=0 \wedge y_i^{t+1}=1 \wedge x_i \notin S$
WENN Y^{t+1} befriedigt E(i)
    DANN sage_vorher 1; **FALLS** fehler: $S:=S \cup x_i$
    SONST sage_vorher 0; **FALLS** fehler: $E(i):=E(i) \cap Y^{t+1}$

S2: $c(Y^t)=0 \wedge y_i^t=0 \wedge y_i^{t+1}=1 \wedge x_i \in S$
WENN Y^{t+1} befriedigt E(j) für ein j mit $x_j \notin S \vee \overline{x}_{j-n} \notin S$
    DANN sage_vorher 1; **FALLS** fehler: $S:=S \cup (x_j$ bzw. $\overline{x}_{j-n})$

SONST sage_vorher 0; **FALLS** fehler: tue nichts;

S3: $c(Y^t)=1 \wedge y_i^t=1 \wedge y_i^{t+1}=0 \wedge x_i \notin S$
WENN es ein j gibt, so dass Y^{t+1} E(j) befriedigt
DANN sage_vorher 1; **FALLS** fehler: $S:=S \cup x_j$
SONST sage_vorher 0; **FALLS** fehler: tue nichts;

S4: $c(Y^t)=1 \wedge y_i^t=1 \wedge y_i^{t+1}=0 \wedge x_i \in S$
sage_vorher 0;

12 Das beschränkte Fehlerraten-Modell:

Bei dem beschränkten Fehlerraten-Modell[12] wird von einem Lernalgorithmus *A* gefordert, nur bei einem kleinen Anteil der Beispiele aus einer endlichen Beispielfolge Fehler zu machen. Dieser Lernalgorithmus nimmt einen Parameter ε entgegen, der sein Verhalten beeinflusst und von der Fehlerindikator-Funktion $M^{t}_{A,f}(\varepsilon, x)$ nur „durchgeschliffen" wird:

$$\hat{M}_{A,F_n,\rho_n}(\varepsilon,t) = \underbrace{\sup}_{f\in F_n}\underbrace{\sup}_{P\in\rho_n} E_{x\in P}\left(M^{t}_{A,f}(\varepsilon,x)\right) \tag{12.1}$$

12.1 Probabilistische Fehlerschrankenlernbarkeit

In Anlehnung an die Fehlerschrankenlernbarkeit bezeichnet man eine Konzeptklasse *F* als „lernbar im probabilistischen *Fehlerschrankenmodell*"[13], wenn es irgendeinen Lernalgorithmus *A* gibt, der in Polynomialzeit läuft, so dass für jedes ε>0 und alle $n\in\mathbb{N}$ ein $t_0\in\mathbb{N}$ existiert, so dass für alle $t\geq t_0$ $\hat{M}_{F_n,\rho_n}(\varepsilon,t)<\varepsilon$ ist, und *t* nur polynomiell mit *n* (Länge der Beispiele) und $^1/_\varepsilon$ wächst. Man definiert:

$$\hat{M}_{F_n,\rho_n}(t) = \underbrace{\inf}_{A}\inf\left\{\varepsilon\in(0,1):\hat{M}_{\hat{A},F_n,\rho_n}(\varepsilon,t)<\varepsilon\right\} \tag{12.2}$$

Man gesteht den potentiellen Lernalgorithmen für eine Konzeptklasse *F* also zu, für längere Beispiele (*n*) und für eine bessere Leistung ($^1/_\varepsilon$) auch entsprechend mehr Rechenzeit zu benötigen.

[12] Englisch: „bounded mistake rate model"
[13] Englisch: „learnable in the bounded mistake rate model"

13 Vergleich der vorgestellten Modelle

An dieser Stelle sollen die drei vorgestellten Lernmodelle miteinander verglichen werden, um Gemeinsamkeiten aufzuzeigen und Unterschiede zu betonen:

- Wie es der Name sagt, besteht ein starker Zusammenhang zwischen dem Fehlerschranken-Modell (siehe 6 Das Fehlerschranken-Modell) und dem probabilistischen Fehlerschranken-Modell (siehe 10 Das probabilistische Fehlerschranken-Modell). Wenn ρ eine Klasse von stochastischen Prozessen ist, deren Beispielpfade in χ liegen, dann lernt jeder Algorithmus, der aus χ im Fehlerschranken-Modell lernt, auch aus ρ im probabilistischen Fehlerschranken-Modell:

  $$\hat{N}_{A,F_n,\rho_n,\delta} \leq \hat{N}_{A,F_n,\chi_n} \tag{13.1}$$

  Dieser Zusammenhang verwundert nicht weiter, denn die Definitionen der beiden Ausdrücke (siehe (10.3) und (6.2)) stehen in engem Zusammenhang:

  Setzt man den Vertrauensparameter δ in (10.3) auf Null, so „befragt“ man damit den Algorithmus, ab welcher Fehleranzahl die Wahrscheinlichkeit dafür, dass diese Fehleranzahl eintritt, kleiner oder gleich Null ist. Man fordert also vom Algorithmus die kleinste Fehleranzahl, bei welcher die Wahrscheinlichkeit für das Auftreten derselben erstmals Null ist.

  Dies entspricht der Aussage aus (6.2), wo der Algorithmus „befragt“ wird, wie viele Fehler er im schlimmsten Fall machen wird.

  Zum Beispiel hat ein Algorithmus, der im schlimmsten Fall *z* Fehler macht, eine Wahrscheinlichkeit von Null dafür, dass er *z+1* Fehler macht, aber eine Wahrscheinlichkeit von größer als Null dafür, dass er *z* Fehler macht.

  Für beliebige Werte von δ wird die Fehlerschranke im probabilistischen Fehlerschranken-Modell also stets niedriger sein als die des Fehlerschranken-Modells.

- Ist ein (polynomialzeit-) Lernalgorithmus gegeben, der im probabilistischen Fehlerschranken-Modell arbeitet, kann dieser verwendet werden, um einen zweiten Lernalgorithmus zu konstruieren, der im beschränkten Fehlerraten-Modell arbeitet, und ebenfalls in Polynomialzeit läuft. Voraussetzung hierfür ist, dass die stochastischen Prozesse in ρ stationär sind.

- Ist eine Konzeptklasse *F* im probabilistischen Fehlerschranken-Modell unter ρ lernbar, wobei ρ eine Klasse stochastischer Prozesse ist, so ist *F* auch unter ρ im beschränkten Fehlerraten-Modell lernbar. Weiterhin gilt für die Fehlerschranke $\hat{M}_{F_n,\rho_n}(t)$

  $$t \geq \left\lceil \frac{\hat{N}_{F_n,\rho_n,\varepsilon/2}}{\varepsilon} \right\rceil \quad \Rightarrow \quad \hat{M}_{F_n,\rho_n}(t) < \varepsilon \tag{13.2}$$

14 Ergebnis und Ausblick

Es ist eine Erweiterung des klassischen PAC Lernmodells in drei Varianten vorgestellt worden. Diese Erweiterungen passen das PAC Modell an die Erfordernisse des Lernens stochastischer Prozesse an. Diese neuerworbenen Fähigkeiten sind nach Angaben der Autoren von großer, praktischer Bedeutung, weil die aufeinander folgenden Beispiele in vielen Anwendungszwecken eine Beziehung zueinander haben.

In den drei beispielhaften Anwendungszwecken, die vorgestellt worden sind, wurde gezeigt, wie sich Informationen aus der Vorgänger-Nachfolger-Beziehung der Beispiele extrahieren lassen, die mit denen von Fragen („membership queries“) vergleichbar sind. Weiterhin wurde gezeigt, sogar eine ganz bestimmte Repräsentation des Zielkonzeptes finden lässt. Diese Möglichkeit ist überall dort von Bedeutung, wo das ermittelte Zielkonzept auf irgendeine Art und Weise „implementiert“ werden muss.

Es sind einige vertiefende Forschungen denkbar, welche an die Ergebnisse der Autoren anschließen. Z.B. könnte es interessant sein, die Machbarkeit von geometrischem Lernen mit den neuen Modellen zu ermitteln.

Weiterhin könnte es interessant sein, zu erforschen, welche Rolle es spielt, wenn man die Irrfahrt auf dem booleschen Würfel durch einen anderen Pfad ersetzt. Hier wären z.B. Irrfahrten in der Art denkbar, dass immer eine beliebige, aber konstante Anzahl von Bit gekippt wird. Eine weitere Art von Irrfahrt wäre z.B., dass stets paarweise Bits gekippt werden.

Überhaupt könnte es mit den neuen Erkenntnissen interessant sein, die Beziehungen zwischen dem PAC Lernmodell einerseits, und dem Lernen mit Fragen („memberships queries“) andererseits zu erforschen.

15 Quellenverzeichnis

[1] D. Aldous und U. Vazirani: A Markovian extension of Valiant's learning model. In *Proc. of the 31th Symposium on the Foundations of Comp. Sci.*, pages 392-396. IEEE Computer Society Press, Los Alamitos, CA, 1990.

[2] A. Blum: Separating PAC and mistake-bound learning models over the boolean domain. In Proc*. of the 31th Annu. IEEE Symposium Foundations of Comp. Sci.*, 1990.

[3] P. Fischer und H. Simon: On learning ring-sum-expansions. *SIAM J. Comput.*, 21:181-192, 1992.

[4] Y. Freund, M. Kearns, D. Ron, R. Rubinfeld, R. Schapire und L. Sellie: Efficient learning of typical finite automata from random walks. In *Proc. 25th Anni. ACM Sypos. Theory Comput.*, pages 315-324. ACM Press, New York, NY, 1993.

[5] N. Littlestone. Learning when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, 2:285-318, 1988.

[6] L. Pitt und L. Valiant: Computational limitations on learning from examples. *J. ACM*, 35:965-984, 1988.

[7] L. G. Valiant: A theory of the learnable. *Commun. ACM*, 27(11):1134-1142, Nov. 1984.

[8] P. L. Bartlett, P. Fischer, K. U. Hoeffgen: Exploiting Random Walks for Learning, *ACM*, 1994

[9] G. Schnitger: Algorithmisches Lernen, *2001*

16 Index