Technische Universität Darmstadt
Fachbereich Informatik
Fachgebiet Knowledge Engineering

Diplomarbeit

Entwurf und Implementierung einer Entwicklungsumgebung für Regel-Lerner

Simon Siegler

Februar 2007

Betreuer: Prof. Dr. Johannes Fürnkranz

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, 6. März 2007 Simon Siegler

Inhaltsverzeichnis

1 Einleitung

Ein Teilgebiet des maschinellen Lernes ist das Lernen von Klassifikationen. Dabei werden Objekte durch ihre Eigenschaften in Form einer Tabelle beschrieben, so dass jede Spalte für eine Eigenschaft steht und in jeder Zeile für ein Objekt die Ausprägungen der Eigenschaften notiert sind. Ziel des Lerners ist es dabei, eine der Eigenschaften aus den übrigen herleiten zu können.

Ein für Menschen leicht verständlicher Ansatz ist das Lernen von Regeln, also Implikationen von Aussagen über die Eigenschaften der Objekte. Für jede Ausprägung der Klasse wird eine Bedingung formuliert, aus der später auf die Ausprägung geschlossen werden soll. Solche Regeln begegnen uns Menschen täglich in verschiedensten Formen, z.B. Verkehrsregeln oder Verhaltensregeln, weshalb sie uns vertraut erscheinen.

Um solche Regeln aus Beispielen zu lernen existieren verschiedene Methoden. Eine davon ist die Separate&Conquer-Methode, die in (Fürnkranz, 1999) detailliert untersucht wurde. Ihr sind viele Regel-Lerner zuzuordnen, die sich stark unterscheiden können, aber alle dem gleichen prinzipiellen Schema folgen.

Für die Konstruktion von Regel-Lernern der Separate&Conquer-Familie nach dem Baukastensystem existiert das SeCo-Framework, welches in (Thiel, 2005) zum Vergleich verschiedener Lerner dieser Familie entworfen wurde. Zur Kodierung von Daten verwendet das SeCo-Framework die weka-Bibliothek. Als allgemeine Plattform für verschiedenste Lerner bietet diese eine große Menge an Funktionalität an, die für das Lernen von Regeln nicht benötigt wird. Außerdem sind die verwendeten Datenstrukturen nicht auf das Lernen von Regeln zugeschnitten und daher nicht optimal für die Verwendung in einem Regel-Lerner.

Das Ziel dieser Arbeit ist es, eine geeignetere Basis für das SeCo-Framework zu entwickeln, die dennoch offen für die Entwicklung weiterer Regel-Lerner ist. Dazu ist ein Modell für das Lernen von Klassifikationsregeln zu entwerfen und in Form einer Bibliothek zu implementieren. Außerdem soll die Eignung von Bitvektoren bei der Kodierung von Daten und Regeln untersucht werden.

Während die meisten Lerner und Bibliotheken für Lerner die Modellierung und Kodierung der Daten nur informell behandeln und von irgendeiner Darstellung ausgehen, die in der Praxis verbreitet ist, wird in dieser Arbeit ein mathematisches Modell für Daten wie sie im maschinellen Lernen auftreten formalisiert. Aus diesem Modell werden dann Kodierungen für Daten und Regeln abgeleitet.

Neben der Implementierung der eigentlichen Bibliothek ist auch die Untersuchung verschiedener Kodierungen ein Schwerpunkt dieser Arbeit. Um die Eignung der Bitvek-

torkodierung zu untersuchen, wird daher zusätzlich eine Ganzzahlkodierung entwickelt, die sich an den Kodierungen existierender Systeme orientiert.

Diese Arbeit ist in sechs Kapitel unterteilt: Nach dieser Einleitung folgt in Kapitel 2 die Darstellung des mathematischen Modells zur Formulierung von Beispielen und Regeln. Außerdem wird die grundlegende Problematik des Lernens von Regeln vorgestellt. Kapitel 3 leitet daraus verschiedene Varianten zur Kodierung von Daten und Regeln her und vergleicht deren theoretische Eigenschaften. Die Realisierung dieser Konzepte in Form von Schnittstellen und Klassen in Java erörtert Kapitel 4. Insbesondere werden zwei der Kodierungen aus Kapitel 3 implementiert. Diese werden in Kapitel 5 bezüglich ihrer praktischen Eigenschaften verglichen. Schließlich folgt in Kapitel 6 eine Zusammenfassung der Ergebnisse mit einer knappen Diskussion offener Fragen.

2 Voraussetzungen

2.1 Grundbegriffe

Zunächst werden einige grundlegende Begriffe benötigt, um über den Aufbau und die Funktionsweise von Regel-Lernern zu sprechen. In Abschnitt 2.1.1 werden die Begriffe zur Beschreibung von Daten definiert. Ausgehend vom Begriff des Attributs werden die Begriffe Instanz und Datensatz eingeführt und erläutert. Abschnitt 2.1.2 stellt darauf aufbauend den Begriff der Regel vor. Dazu wird eine Sprache von Aussagen über Instanzen verwendet.

2.1.1 Daten

Der zentrale Begriff zur Beschreibung von Daten ist der des Attributs. Instanzen, Datensätze und sogar Regeln sind über Mengen von Attributen definiert.

Definition 2.1 (Attribut)**.** Ein *Attribut* ist ein Paar (A, D_A) bestehend aus einem Namen A und einer Menge zulässiger Werte D_A. Diese Menge wird als *Wertebereich* oder *Domäne* des Attributs bezeichnet. Wenn $|D_A| \in \mathbb{N}$, heißt das Attribut (A, D_A) *endlich*, ansonsten *unendlich*. Attribute mit genau zwei zulässigen Werten werden auch *binär* genannt.

Ein Attribut beschreibt damit ein Merkmal oder eine Eigenschaft mit den möglichen Ausprägungen. Beispiele sind $(\text{Anzahl}, \mathbb{N})$, $(\text{rund}, \{\text{ja}, \text{nein}\})$ oder $(\text{Länge}, \mathbb{R})$.

Oft bezeichnet man Attribute durch ihre Namen, wenn dadurch keine Mehrdeutigkeiten entstehen. So schreibt man statt „Attribut $(\text{rund}, \{\text{ja}, \text{nein}\})$ ist binär“ einfach „Attribut rund ist binär“. Da der Name nur der Identifikation eines Attributs dient und darüber hinaus keine Funktion hat, können Mehrdeutigkeiten durch Umbenennung aufgelöst werden. Im Folgenden wird also von Attributen mit verschiedenen Namen ausgegangen. Dies erlaubt es, A statt (A, D_A) zu schreiben.

Die Domäne eines Attributs kann mit einer Ordnungsrelation ausgestattet sein. In solchen Fällen spricht man von geordneten Attributen.

Definition 2.2 (geordnete Attribute)**.** Ist A ein Attribut und $\leq$ eine Ordnungsrelation auf D_A, dann heißt A *geordnet* bezüglich $\leq$. Ist die Relation $\leq$ linear (total), so heißt A *linear (total geordnet)* bezüglich $\leq$.

Mit der üblichen Ordnung $\leq$ auf $\mathbb{N}$ wird *Anzahl* so zu einem geordneten Attribut. Die Ordnungsrelation ist in diesem Fall sogar linear. Ebenfalls linear ist das Attribut

$$(\text{humidity}, \{\text{high}, \text{low}, \text{medium}\})$$

mit der Ordnung

$$<_{humidity} := \{(\text{low}, \text{medium}), (\text{medium}, \text{high})\}^*.$$

Eine partielle Ordnung ist zum Beispiel die Teilbarkeitsordnung $|$ auf $\mathbb{N}$, denn für zwei teilerfremde Zahlen a und b gilt weder $a|b$ noch $b|a$. Häufige Beispiele für partiell geordnete Attribute sind mengenwertige Attribute.

Definition 2.3 (mengenwertige Attribute)**.** Ein Attribut A heißt *mengenwertig*, falls es eine Menge X mit $D_A = 2^X$ gibt. Die Menge X heißt *Basismenge* von A.

Mengenwertige Attribute sind also immer bezüglich der Teilmengenordnung $\subseteq$ geordnet. Außer für triviale Beispiele mit einelementiger oder leerer Basismenge sind diese Ordnungen immer partiell. Abbildung 2.1 zeigt diese Ordnung für das Attribut rund, Abbildung 2.2 für humidity.

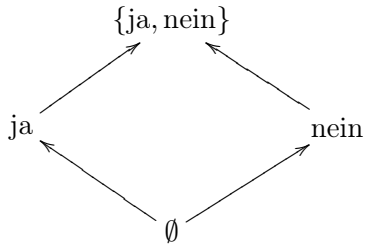


Abbildung 2.1: mengenwertige Attribute als partielle Ordnungen: rund

Definition 2.4 (Instanz)**.** Eine *Instanz* einer Attributmenge $\mathcal{A}$ ist ein Element i aus $\prod_{A\in\mathcal{A}} D_A$, also eine Funktion $i : \mathcal{A} \mapsto \bigcup_{A\in\mathcal{A}} D_A$ mit $\forall A \in \mathcal{A}.i(A) \in D_A$. Daher wird $\mathcal{A}$ als *Definitionsbereich* der Instanz i bezeichnet.

Definition 2.5 (Einschränkung von Instanzen)**.** Für eine Instanz i einer Attributmenge $\mathcal{A}$ und eine Attributmenge $\mathcal{B} \subseteq \mathcal{A}$ bezeichnet $i|_\mathcal{B}$ die Instanz von $\mathcal{B}$ mit $\forall B \in \mathcal{B}.i|_\mathcal{B}(B) = i(B)$. $i|_\mathcal{B}$ heißt *Einschränkung* von i auf $\mathcal{B}$.

Eine Instanz einer Attributmenge $\mathcal{A}$ ordnet also jedem Attribut aus $\mathcal{A}$ genau einen Wert aus dessen Domäne zu. Für die Attributmenge aus den oben genannten Attributen Anzahl, rund und Länge sind i_1 und i_2 mit $i_1(\text{Länge}) = \pi$, $i_1(\text{rund}) = \text{ja}$ und

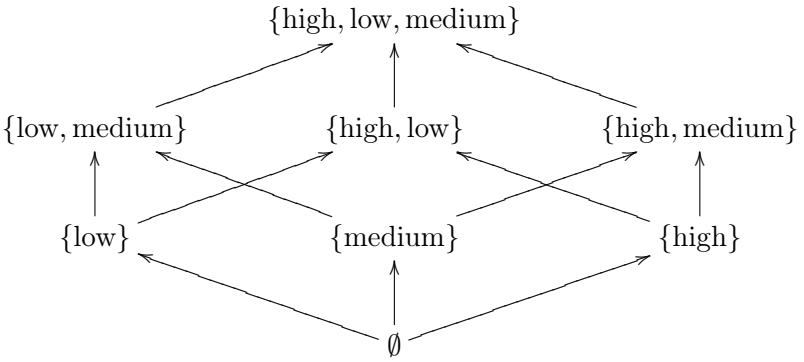


Abbildung 2.2: mengenwertige Attribute als partielle Ordnungen: humidity

	Attribut $A \in \mathcal{A}$		
	Anzahl	rund	Länge
$i_1(A)$	1	ja	π
$i_2(A)$	0	nein	0

Tabelle 2.1: Beispiele für Instanzen

$i_1(\text{Anzahl}) = 1$ sowie $i_2(\text{Anzahl}) = 0$, $i_2(\text{rund}) = \text{nein}$ und $i_2(\text{Länge}) = 0$ zwei Instanzen. Tabelle 2.1 zeigt die beiden Funktionen.

Für endliche Attributmengen lassen sich Instanzen als Tupel des kartesischen Produktes schreiben, indem man eine Reihenfolge der Attribute festlegt. So gibt man zum Beispiel die Reihenfolge $(\text{rund}, \text{Anzahl}, \text{Länge})$ vor und schreibt die Instanzen $i_1 = (\text{ja}, 1, \pi)$ und $i_2 = (\text{nein}, 0, 0)$.

In der Praxis des maschinellen Lernens tauchen ausschließlich endliche Attributmengen auf, so dass die Vorstellung von Instanzen als Tupel sehr verbreitet und völlig ausreichend ist. Für einige der folgenden Definitionen und Sätze ist es jedoch hilfreich, die äquivalente und allgemeinere Definition als Funktion zu verwenden.

Definition 2.6 (Datensatz)**.** Eine Folge von Instanzen über einer Attributmenge $\mathcal{A}$ heißt *Datensatz* über $\mathcal{A}$. Die Folgeglieder bezeichnet man als *Beispiele* des Datensatzes.

Im Gegensatz zu einer Menge von Instanzen, also einer Teilmenge von $\prod_{A \in \mathcal{A}} D_A$, kann ein Datensatz einzelne Instanzen auch mehrfach enthalten. In der Praxis ist ein Datensatz eine endliche Folge über einer endlichen Attributmenge.

Ein Datensatz kann als Tabelle mit einer Spalte für jedes Attribut und einer Zeile für jedes Beispiel dargestellt werden. Tabelle 2.2 zeigt einen Datensatz mit den oben definierten Attributen *rund*, *Anzahl* und *Länge*. Die Zeilen der Tabelle entsprechen den Instanzen $(\text{nein}, 0, 0)$, $(\text{ja}, 10, \pi)$ und $(\text{ja}, 77, \sqrt{2})$.

rund	Anzahl	Länge
nein	0	0
ja	10	π
ja	77	$\sqrt{2}$

Tabelle 2.2: Ein Datensatz als Tabelle dargestellt.

2.1.2 Regeln

Eine Regel stellt Abhängigkeiten zwischen Attributen dar. Für den knappen Datensatz aus Tabelle 2.2 gilt zum Beispiel die Aussage „Wenn *rund* den Wert *ja* hat, dann hat *Länge* einen irrationalen Wert“. Die Sprache, in der solche Aussagen (Hypothesen) formalisiert werden, ist die *Hypothesensprache* des Lerners. Die Hypothesen eines Regel-Lerners sind Implikationen und werden daher Regeln genannt. Dies soll nun formalisiert werden.

Definition 2.7 (Basisaussage (über ein Attribut))**.** Für ein Attribut A und eine Teilmenge P von D_A heißt ein Term der Form $A \subseteq P$ *Basisaussage* über A. B_A bezeichnet die Menge aller Basisaussagen über A ($B_A := \{A \subseteq P | \forall P \in 2^{D_A}\}$).

Im Falle einer einelementigen Teilmenge schreibt man statt $A \subseteq \{x\}$ einfach $A = x$. Für geordnete Attribute schreibt man statt $A \subseteq \{x | x \in D_A \wedge x \leq a\}$ auch einfach $A \leqq a$. Ebenso schreibt man $A \geqq a$ statt $A \subseteq \{x | x \in D_A \wedge a \leq x\}$

Die Basisaussagen über *rund* sind:

$$\text{rund} \subseteq \emptyset,\ \text{rund} = \text{ja},\ \text{rund} = \text{nein},\ \text{rund} \subseteq \{\text{ja}, \text{nein}\}$$

Beispiele für Aussagen über *Anzahl*:

$$\text{Anzahl} \subseteq \{2n | n \in \mathbb{N}\},\ \text{Anzahl} = 0,\ \text{Anzahl} \leqq 5, \text{Anzahl} \geqq 6$$

Definition 2.8 (Gültigkeit einer Basisaussage)**.** Sei $i : \mathcal{A} \mapsto \bigcup_{A \in \mathcal{A}} D_A$ eine Instanz von $\mathcal{A}$ mit $A \in \mathcal{A}$ und $A \subseteq P_A$ eine Basisaussage über A, dann *gilt* $A \subseteq P_A$ für i genau dann, wenn $i(A) \in P_A$.

Für die beiden letzten Beispiele des Datensatzes aus Tabelle 2.2 gilt $\text{rund} = \text{nein}$, für das erste Beispiel gilt $\text{rund} = \text{ja}$. Während die Aussage $\text{rund} \subseteq \{\text{ja}, \text{nein}\}$ für alle drei Beispiele gilt, gilt die Aussage $\text{rund} \subseteq \emptyset$ für kein Beispiel und darüber hinaus für keine Instanz der Attributmenge. Dies lässt sich verallgemeinern:

Satz 2.1. *Sei $\mathcal{A}$ eine Attributmenge mit $A \in \mathcal{A}$, dann gilt die Aussage $A \subseteq \emptyset$ für keine Instanz von $\mathcal{A}$ und die Aussage $A \subseteq D_A$ für jede Instanz von $\mathcal{A}$.*

Beweis. Nach Definition gilt $A \subseteq \emptyset$ für eine Instanz i genau dann, wenn $i(A) \in \emptyset$. Dies kann für kein i gelten, da $\emptyset$ keine Elemente hat.

Ebenso gilt $A \subseteq D_A$ genau dann für i, wenn $i(A) \in D_A$. Genau dies wird per Definition von jeder Instanz gefordert. □

Aus den (atomaren) Aussagen über die einzelnen Attribute lassen sich mit den Operatoren $\vee$ und $\wedge$ logische Aussagen über Attributmengen konstruieren.

Definition 2.9 (Aussagen über eine Attributmenge)**.** Die Menge der *Basisaussagen* über einer Attributmenge $\mathcal{A}$ ist $S_0^{\mathcal{A}} := \bigcup_{A \in \mathcal{A}} B_A$.

$S_{n+1}^{\mathcal{A}} := \left\{A \vee B | A, B \in S_n^{\mathcal{A}}\right\} \cup \left\{A \wedge B | A, B \in S_n^{\mathcal{A}}\right\} \cup S_n^{\mathcal{A}}$ definiert *zusammengesetzte Aussagen* mit höchstens n Operatoren.

Die Menge $S_{\mathcal{A}}$ aller *Aussagen über* $\mathcal{A}$ ist definiert durch $S_{\mathcal{A}} := \bigcup_{n \in \mathbb{N}} S_n^{\mathcal{A}}$

Beispiele für Aussagen über $\{\text{rund}, \text{Anzahl}, \text{Länge}\}$ sind:

$$\text{rund} = \text{ja} \vee (\text{Anzahl} = 0 \wedge \text{Länge} = 0) \tag{2.1}$$

$$\text{rund} = \text{ja} \wedge \text{Länge} \subseteq \mathbb{R} \setminus \mathbb{Q} \tag{2.2}$$

$$\text{Länge} \subseteq \{x | x \in \mathbb{R} \wedge x < 0\} \tag{2.3}$$

$$\text{Länge} \subseteq \{x | x \in \mathbb{R} \wedge x < 0\} \vee \text{rund} = \text{nein} \tag{2.4}$$

Satz 2.2 (Erweiterung von Aussagen)**.** *Jede Aussage über eine Teilmenge $\mathcal{B}$ von $\mathcal{A}$ stellt eine Aussage über $\mathcal{A}$ dar: $\mathcal{B} \subseteq \mathcal{A} \Rightarrow S_{\mathcal{B}} \subseteq S_{\mathcal{A}}$.*

Beweis. Da sich $S_{\mathcal{A}}$ und $S_{\mathcal{B}}$ nach Definition 2.7 auf die gleiche Weise aus den jeweiligen Basisaussagen $S_0^{\mathcal{A}}$ und $S_0^{\mathcal{B}}$ bilden, bleibt zu zeigen, dass $S_0^{\mathcal{B}} \subseteq S_0^{\mathcal{A}}$, also $\bigcup_{A \in \mathcal{B}} B_A \subseteq \bigcup_{A \in \mathcal{A}} B_A$. Für $\mathcal{B} \subseteq \mathcal{A}$ ist dies offensichtlich der Fall. □

Definition 2.10 (Gültigkeit von Aussagen)**.** Sei $i : \mathcal{A} \mapsto \bigcup_{A \in \mathcal{A}} D_A$ eine Instanz von $\mathcal{A}$ und $s \in S_{\mathcal{A}}$ eine Aussage über $\mathcal{A}$, dann *gilt* s für i genau dann, wenn

- s ist eine Basisaussage, die für i gilt, oder
- s ist eine Aussage der Form $u \wedge v$ und sowohl u als auch v gilt für i, oder
- s ist eine Aussage der Form $u \vee v$ und u oder v gilt für i.

Statt „s gilt für i“ wird gelegentlich auch „s deckt i ab“ verwendet.

Für den Datensatz aus Tabelle 2.2 gelten die Aussagen 2.1 und 2.2 für alle Instanzen, Aussage 2.3 für keine Instanz und Aussage 2.4 nur für die erste Instanz.

Satz 2.3. *Zu jeder Instanz i über $\mathcal{A}$ lässt sich eine Aussage konstruieren, die für genau diese Instanz und keine andere Instanz über $\mathcal{A}$ gilt: $s_i := \bigwedge_{A \in \mathcal{A}} A = i(A)$.*

Beweis. s_i gilt für eine Instanz j genau dann, wenn die Basisaussagen $A \doteq i(A)$ für jedes $A \in \mathcal{A}$ gelten. Nach Definition 2.7 ist dies genau dann der Fall, wenn $j(A) \in \{i(A)\}$ für alle $A \in \mathcal{A}$. Dies wiederum ist äquivalent zu $\forall A \in \mathcal{A}.j(A) = i(A)$ und entspricht $j = i$. □

Aussagen lassen sich nach Definition 2.9 aus vorhandenen Aussagen durch Verknüpfung mit $\vee$ oder $\wedge$ bilden. Dadurch sind beliebig lange und beliebig redundante Aussagen möglich. Beispielsweise lassen sich aus einer einzigen Basisaussage $A \subseteq P_A$ unendlich viele Aussagen bilden. Von ihrer Bedeutung unterscheiden sich diese Aussagen jedoch untereinander nicht, denn sie alle gelten genau dann, wenn die Basisaussage $A \subseteq P_A$ gilt. Ähnlich lassen sich sehr komplex aussehende Aussagen bilden, die einfache Tautologien oder Widersprüche darstellen. Begriffe wie Äquivalenz von Aussagen, Implikation und die kanonische Normalform dienen dazu, die riesige Menge an Aussagen handhabbar zu halten.

Definition 2.11 (Äquivalenz)**.** Zwei Aussagen p und q über eine Attributmenge heißen *äquivalent* ($p \leftrightarrow q$), wenn für jede Instanz i genau dann p gilt wenn auch q für i gilt.

Äquivalente Aussagen gelten also für die gleichen Instanzen. Eine etwas schwächere Einschränkung ist die der Implikation.

Definition 2.12 (Implikation)**.** Seien p und q Aussagen über $\mathcal{A}$, dann sagt man p *impliziert* q wenn für alle von p abgedeckten Instanzen von $\mathcal{A}$ q gilt. Man schreibt auch $p \rightarrow q$.

Offensichtlich implizieren sich zwei Aussagen genau dann gegenseitig, wenn sie äquivalent sind.

Statt von Implikation zwischen Aussagen wird häufig auch von Generalisierung und Spezialisierung gesprochen. Eine Aussage p, die nur eine Teilmenge der von einer Aussage q abgedeckten Instanzen abdeckt, wird dabei als Spezialisierung von q angesehen, denn q deckt mehr Instanzen ab und wird daher auch als Generalisierung von p aufgefasst. Nach Definition von $p \rightarrow q$ entspricht diese Teilmengenbeziehung der Implikation.

Definition 2.13 (Spezialisierung und Generalisierung)**.** Seien p und q zwei Aussagen über einer Attributmenge $\mathcal{A}$ mit $p \rightarrow q$. Dann heißt p *Spezialisierung* von q und q heißt *Generalisierung* von p.

Implikation ist reflexiv, da für alle von einer Regel abgedeckten Instanzen diese Regel gelten muss. Weiterhin ist Implikation transitiv, denn wenn $p \rightarrow q$ und $q \rightarrow r$, dann gilt für jede von p abgedeckte Instanz q, wenn aber q die Instanz abdeckt, muss auch r für diese Instanz gelten. Für jede von p abgedeckte Instanz gilt r, also $p \rightarrow r$. Damit stellt Implikation eine Quasiordnung auf Aussagen dar. Zu einer partiellen Ordnung fehlt die Asymmetrie, da es zusammengesetzte Aussagen gibt, die sich gegenseitig implizieren. Abbildung 2.3 zeigt die Struktur der Implikationen zwischen den Basisaussagen

über rund. Die Pfeile entsprechen den Implikationen. Reflexive und durch Transitivität erklärbare Pfeile sind zur besseren Lesbarkeit weggelassen.

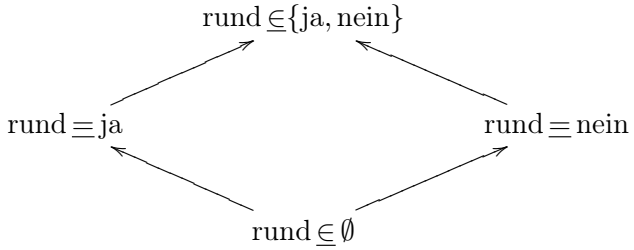


Abbildung 2.3: Die Basisaussagen über rund und ihre Implikationen

Ergänzt man die Basisaussagen über rund nur um die beiden Aussagen

$$\text{rund} = \text{nein} \vee \text{rund} = \text{ja}$$

und

$$\text{rund} = \text{nein} \wedge \text{rund} = \text{ja},$$

so erhält man die in Abbildung 2.4 dargestellte Struktur. Die beiden Aussagen sind also äquivalent zu bereits in Abbildung 2.3 vorhandenen Basisaussagen.

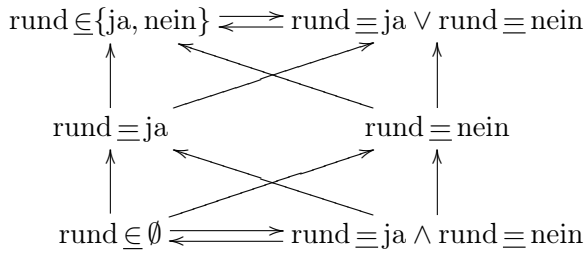


Abbildung 2.4: Aussagen über rund und ihre Implikationen

Auf den Äquivalenzklassen der Aussagen stellt die Implikation eine Ordnungsrelation dar. Die fehlende Asymmetrie ergibt sich aus der Tatsache, dass gegenseitige Implikation genau der Äquivalenz entspricht. Identifiziert man unter den syntaktischen Aussagen also die semantisch äquivalenten, erhält man einen vollständigen Verband von Aussagen wie in Abbildung 2.5.

Für die Handhabung von Aussagen sind Normalformen hilfreich. Im Rahmen dieser Arbeit wird nur die disjunktive Normalform benötigt:

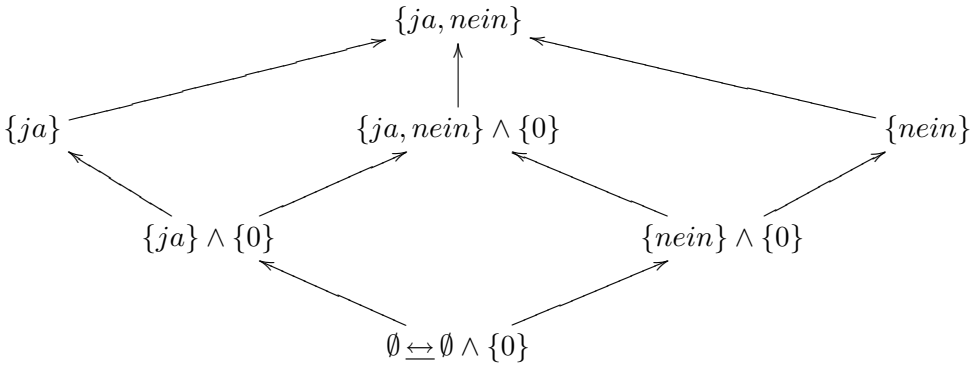


Abbildung 2.5: Aussagen über rund und Anzahl und ihre Implikationen. Statt $A \in P_A$ ist aus Platzgründen nur P_A notiert.

Definition 2.14 (disjunktive Normalform)**.** Eine Aussage über eine Attributmenge $\mathcal{A}$ ist in *disjunktiver Normalform*, wenn sie eine Disjunktion von Konjunktionen von Basisaussagen über $\mathcal{A}$ darstellt.

Definition 2.15 (kanonische disjunktive Normalform)**.** Eine Aussage in disjunktiver Normalform liegt in *kanonischer disjunktiver Normalform* vor, wenn sie außerdem in jeder Konjunktion genau eine Aussage über jedes Attribut enthält.

Beispiele für Aussagen in disjunktiver Normalform sind unter anderem alle Aussagen in Abbildung 2.5. Gegenbeispiele sind

$$(\text{rund} \in \{\text{ja}, \text{nein}\} \ \vee \ \text{Länge} = 0) \ \wedge \ \text{rund} \in \{\text{ja}, \text{nein}\}$$

oder

$$((\text{rund} \in \{\text{ja}, \text{nein}\} \ \vee \ \text{Länge} = 0) \ \wedge \ \text{Länge} = 0) \ \vee \ \text{rund} \in \{\text{ja}, \text{nein}\}.$$

Kanonische disjunktive Normalformen über rund und Anzahl sind zum Beispiel:

$$\text{rund} \in \{\text{ja}, \text{nein}\} \ \wedge \ \text{Anzahl} = 0$$

oder

$$(\text{rund} = \text{nein} \ \wedge \ \text{Anzahl} = 0) \ \vee \ (\text{rund} = \text{ja} \ \wedge \ \text{Anzahl} = 0) \ \vee \ (\text{rund} \in \emptyset \ \wedge \ \text{Anzahl} = 0)$$

Satz 2.4. *Jede Aussage in disjunktiver Normalform lässt sich in eine äquivalente Aussage in kanonischer disjunktiver Normalform transformieren, indem man in jeder Konjunktion für jedes fehlende Attribut A die Basisaussage $A \in D_A$ ergänzt.*

Beweis. Die so konstruierte Aussage ist offensichtlich in kanonischer disjunktiver Normalform. Also bleibt zu zeigen, dass sie äquivalent zur ursprünglichen Aussage ist. Sei also p die ursprüngliche Aussage und p' die daraus konstruierte. Weiter sei i eine beliebige Instanz, die jedoch von p abgedeckt wird und j eine Instanz, für die p nicht gilt. Zu zeigen ist, dass p' für i aber nicht für j gilt.

Da p für i gilt, muss mindestens eine der Konjunktionen für i gelten. Eine davon sei k. Nun enthält p' eine Konjunktion k', die aus k nur durch Ergänzung von Basisaussagen der Form $A \subseteq D_A$ hervorgeht. Die Basisaussagen, die k' mit k gemeinsam hat, gelten für i, da k für i gilt. Die ergänzten Aussagen gelten nach Satz 2.1 für jede Instanz, also auch für i. Damit gilt k' und somit auch p' für i.

Angenommen p' gelte für j. Dann gilt eine der Konjunktion k' von p' für j. Nach der Konstruktion von p' gibt es dann eine Konjunktion k von p, die sich von k' nur durch das Fehlen einiger Basisaussagen der Form $A \subseteq D_A$ unterscheidet. Da k' für j gilt, gelten alle Basisaussagen aus k' für j. Alle Basisaussagen von k sind jedoch auch Basisaussagen von k' und gelten somit für j. Also muss auch k für j gelten. Dies steht jedoch im Widerspruch zu der Voraussetzung, dass k nicht für j gilt. Die Annahme, p' gelte für j muss daher falsch sein.

□

Satz 2.5. *Jede Aussage über eine Attributmenge $\mathcal{A}$ ist äquivalent zu einer Aussage über $\mathcal{A}$ in kanonischer disjunktiver Normalform.*

Beweis. Der Beweis erfolgt durch Konstruktion einer Aussage in kanonischer disjunktiver Normalform aus den von der ursprünglichen Aussage abgedeckten Instanzen. Sei p also eine Aussage über $\mathcal{A}$ und I_p die Menge aller von p abgedeckten Instanzen. Nach Satz 2.3 lässt sich jede Instanz durch eine Konjunktion beschreiben, die offenbar über jedes Attribut genau eine Basisaussage enthält. Konstruiert man solche Aussagen für jedes $i \in I_p$ und verknüpft diese durch $\vee$, so erhält man eine Aussage p' in kanonischer disjunktiver Normalform: $\bigvee_{i \in I_p} \bigwedge_{A \in \mathcal{A}} A \subseteq \{i(A)\}$.

Zu zeigen bleibt noch, dass p' genau die Instanzen aus I_p abdeckt. Sei $i \in I_p$, dann wird i nach Satz 2.3 von der Konjunktion $\bigwedge_{A \in \mathcal{A}} A \subseteq \{i(A)\}$ abgedeckt. Da diese Konjunktion disjunktiv mit anderen Konjunktionen verknüpft ist, deckt auch die Disjunktion p' die Instanz i ab.

Sei nun also $j \notin I_p$. Angenommen, p' gilt für j, so muss eine der Konjunktionen $\bigwedge_{A \in \mathcal{A}} A \subseteq \{i(A)\}$ mit $i \in I_p$ für j gelten. Dazu muss aber $\forall A \in \mathcal{A}.A \subseteq \{i(A)\}$ für j gelten. Dies entspricht $\forall A \in \mathcal{A}.j(A) \in \{i(A)\}$ und somit $j = i$, also $j \in I_p$. Aus diesem Widerspruch zur Voraussetzung $j \notin I_p$ lässt sich folgern, dass die Annahme, p' gilt für j, falsch sein muss. Für $j \notin I_p$ gilt p' also nicht. □

Definition 2.16 (Regel)**.** Eine *Regel* über einer Attributmenge $\mathcal{A}$ ist ein Term $p \rightarrow c$ mit $p \in S_{\mathcal{P}}$ und $c \in S_{\mathcal{C}}$ für $\mathcal{P} \subseteq \mathcal{A}$ und $\mathcal{C} \subseteq \mathcal{A}$ mit $\mathcal{C} \cap \mathcal{P} = \emptyset$. p heißt die *Vorbedingung (Prämisse, Körper)* und c heißt die *Folgerung (Konsequenz, Kopf)* der Regel.

Beispiele für Regeln über *rund*, *Anzahl* und *Länge*:

$$\text{rund} \subseteq \{\text{ja}\} \rightarrow \text{Länge} \subseteq \mathbb{R} \setminus \mathbb{Q} \tag{2.5}$$

$$(\text{Anzahl} \subseteq \{0\} \wedge \text{Länge} \subseteq \{0\}) \vee$$
$$(\text{Anzahl} \subseteq \mathbb{N} \setminus \{0\} \wedge \text{Länge} \subseteq \mathbb{R} \setminus \{0\}) \rightarrow \text{rund} \subseteq \{\text{nein}\} \tag{2.6}$$

$$\text{Anzahl} \subseteq \{0\} \wedge \text{Länge} \subseteq \{0\} \rightarrow \text{rund} \subseteq \{\text{ja}\} \tag{2.7}$$

Definition 2.17 (Abdeckung)**.** Eine Regel *deckt* eine Instanz *ab*, wenn die Vorbedingung für die Instanz gilt.

Definition 2.18 (Korrektheit)**.** Eine Regel heißt *korrekt* bezüglich eines Datensatzes, wenn für jedes von ihr abgedeckte Beispiel des Datensatzes ihre Folgerung gilt.

Tabelle 2.3 zeigt, welche Regeln die jeweiligen Instanzen im Datensatz-Beispiel abdecken. Regel 2.5 ist dabei korrekt, die beiden anderen Regeln sind nicht korrekt.

rund	Anzahl	Länge	abdeckende Regeln
nein	0	0	2.6, 2.7
ja	10	π	2.5, 2.6
ja	77	$\sqrt{2}$	2.5, 2.6

Tabelle 2.3: Abdeckung am Datensatz-Beispiel

Der Kopf einer Regel wird häufig in disjunktiver Normalform angegeben. Für die Abdeckung braucht man so nur testen, ob eine der Konjunktionen für die Instanz gilt.

2.2 Klassifikation und Regeln

Unter Klassifikation versteht man die Einteilung der Beispiele eines Datensatzes in verschiedene Klassen anhand der Werte eines ausgezeichneten Attributs. Dieses Attribut wird Klassenattribut genannt und seine Werte werden auch als Klassen oder Label bezeichnet. Alternativ kann man eine Klassifikation auch als Funktion der Instanzen über der um das Klassenattribut reduzierten Attributmenge in die Domäne des Klassenattributs auffassen. Im maschinellen Lernen versucht man, aus einem gegebenen Datensatz eine solche Funktion zu lernen. Daher wird im Folgenden diese Sichtweise zur Formalisierung verwendet.

Definition 2.19 (Klassifikation)**.** Eine *Klassifikation* f über einer Attributmenge $\mathcal{A}$ nach einem Attribut $C \in \mathcal{A}$ ist eine Funktion von $\prod_{A \in \mathcal{A} \setminus \{C\}} D_A$ nach D_C. Das Attribut C wird dabei als *Klassenattribut* zu f bezeichnet.

Klassifikation	Klassenattribut
$f(i) = \begin{cases} \text{ja} & i(\text{Anzahl}) > 0 \\ \text{nein} & i(\text{Anzahl}) \leq 0 \end{cases}$	rund
$f(i) = \begin{cases} \text{ja} & i(\text{Länge}) > i(\text{Anzahl}) \\ \text{nein} & i(\text{Länge}) \leq i(\text{Anzahl}) \end{cases}$	rund
$f(i) = \begin{cases} 0 & i(\text{rund}) = \text{ja} \\ 1 & i(\text{rund}) = \text{nein} \end{cases}$	Anzahl
$f(i) = \log_2(i(\text{Anzahl}) + 1)$	Länge

Tabelle 2.4: Beispiele für Klassifikationen

Tabelle 2.4 zeigt Beispiele für Klassifikationen über $\{\text{rund}, \text{Anzahl}, \text{Länge}\}$ nach verschiedenen Attributen.

Klassifikationen über endliche Mengen endlicher Attribute lassen sich durch Regeln ausdrücken. Für jede Klasse $c \in D_C$ wird eine Regel mit Kopf $C \equiv c$ formuliert, deren Körper genau die Instanzen von $\mathcal{A} \setminus \{C\}$ abdeckt, für welche die Klassifikation den Wert c annimmt. Solche Regeln bezeichnet man daher als Klassifikationsregeln.

Definition 2.20 (Klassifikationsregel)**.** Eine Regel über einer endlichen Menge $\mathcal{A}$ endlicher Attribute heißt *Klassifikationsregel* für $C \in \mathcal{A}$, wenn ihr Kopf der Form $C \equiv c$ entspricht. Der Wert c wird als *Klasse* der Regel bezeichnet.

Satz 2.6. *Zu jeder Klassifikation f über einer endlichen Menge endlicher Attribute $\mathcal{A}$ nach einem Attribut $C \in \mathcal{A}$ gibt es eine Menge $\{r_c | c \in D_C\}$ von Klassifikationsregeln für C über $\mathcal{A}$, so dass*

- $r_c = p \rightarrow C \equiv c$ *und*
- r_c *deckt* i *ab* $\Leftrightarrow f(i|_{\mathcal{A}\setminus C}) = c$

Beweis. Zum Beweis wird eine Regel der Form $r_c = p \rightarrow C \equiv c$ für jede Klasse $c \in C$ konstruiert und anschließend gezeigt, dass für alle diese Regeln die zweite Bedingung erfüllt ist.

Für jede Instanz i von $\mathcal{A} \setminus C$ mit $f(i) = c$ konstruiert man nach Satz 2.3 die Aussage s_i. All diese Aussagen verknüpft man durch $\vee$ zu einer Aussage p in disjunktiver Normalform und bildet daraus die Regel $r_c := p \rightarrow C \equiv c$.

Angenommen, eine Instanz j wird von r_c abgedeckt, dann muss nach Definition 2.17 die Disjunktion p für j gelten. Also muss eine der Aussagen s_i für j gelten und damit für alle $A \in \mathcal{A} \setminus C$ auch $j(A) = i(A)$, also $j|_{\mathcal{A}\setminus C} = i$. Damit folgt auch $f(j|_{\mathcal{A}\setminus C}) = f(i) = c$.

Sei nun j eine Instanz mit $f(j|_{\mathcal{A}\setminus C}) = c$, dann ist nach obiger Konstruktion die Aussage $s_{j|_{\mathcal{A}\setminus C}}$ eine der Konjunktionen der Vorbedingung von r_c. Diese Konjunktion gilt offensichtlich für j und somit deckt r_c die Instanz j ab. □

Die Klassifikationen aus Abbildung 2.4 sind in Tabelle 2.5 äquivalenten Regelmengen gegenübergestellt. Da Anzahl und Länge nicht endlich sind, kann nicht jede Klassifikation als Regel dargestellt werden.

Klassifikation	Regelmenge
$f(i) = \begin{cases} \text{ja} & i(\text{Anzahl}) > 0 \\ \text{nein} & i(\text{Anzahl}) \leq 0 \end{cases}$	$\text{Anzahl} \sqsubseteq \mathbb{N} \setminus 0 \rightarrow \text{rund} = \text{ja}$ $\text{Anzahl} \sqsubseteq \{0\} \rightarrow \text{rund} = \text{nein}$
$f(i) = \begin{cases} \text{ja} & i(\text{Länge}) > i(\text{Anzahl}) \\ \text{nein} & i(\text{Länge}) \leq i(\text{Anzahl}) \end{cases}$	nicht darstellbar
$f(i) = \begin{cases} 0 & i(\text{rund}) = \text{ja} \\ 1 & i(\text{rund}) = \text{nein} \end{cases}$	$\text{rund} = \text{ja} \rightarrow \text{Anzahl} = 0$ $\text{rund} = \text{nein} \rightarrow \text{Anzahl} = 1$
$f(i) = \log_2(i(\text{Anzahl}) + 1)$	nicht darstellbar

Tabelle 2.5: Klassifikationen und Regelmengen

Zwar lässt sich jede Klassifikation über einer endlichen Menge von endlichen Attributen durch eine Menge von Klassifikationsregeln darstellen, doch nicht jede Menge von Klassifikationsregeln beschreibt auch eine Klassifikation. Zum einen werden eventuell nicht alle Instanzen von diesen Regeln abgedeckt. Ein einfaches Beispiel dafür erhält man, wenn man aus den Regelmengen in Tabelle 2.5 jeweils eine Regel entfernt. Andererseits muss eine Regelmenge nicht funktional sein, sondern eine Instanz kann von Regeln für verschiedene Klassen abgedeckt werden. Vereinigt man die Regelmengen der beiden Klassifikationen für rund aus Tabelle 2.5, erhält man eine solche nicht funktionale Regelmenge.

Das erste Problem kann durch Hinzufügen einer sogenannten Default-Regel gelöst werden: Man ergänzt einen Term der Form $\rightarrow C = c$, die sogenannte Default-Regel, um auszudrücken, dass alle Beispiele, die nicht durch andere Regeln abgedeckt sind, zur Klasse c gehören.

Um das zweite Problem zu lösen, kann man eine Priorisierung der Regeln vornehmen. Statt von einer Menge von Regeln geht man von einer Liste von Regeln aus, die nacheinander zu beachten sind. Wird ein Beispiel von einer Regel abgedeckt, so werden die nachfolgenden Regeln nicht mehr angewandt.

Durch diese Maßnahmen gelangt man von Klassifikationsregeln zu Entscheidungslisten. Diese von Rivest (1987) untersuchten Listen sind meist deutlich einfacher als äquivalente Regelmengen. Außerdem sind Entscheidungslisten funktional und mit Default-Regel auch total, so dass jeder Liste auch eine eindeutige Klassifikation entspricht.

Wie in Satz 2.6 gezeigt, lassen sich alle Klassifikationen über endlichen Mengen endlicher Attribute durch Regelmengen ausdrücken. Doch die konstruierten Regelmengen sind umfangreich und komplex, da jede Regel letztlich alle zu einer Klasse gehörenden Instanzen kodiert.

Das Ziel des Regel-Lernens ist es, möglichst einfache Regeln zu finden, welche die Klassifikation möglichst gut beschreiben. Abschnitt 2.2.1 befasst sich mit der Einfachheit von Regeln, während Abschnitt 2.2.2 die Qualität von Regeln zum Thema hat.

Die dabei zu lernenden Regeln müssen für jede Instanz genau eine Klasse liefern, also selbst eine Klassifikation darstellen.

Definition 2.21 (Hypothese (Klassifikationsregeln))**.** Eine *Hypothese* h zu einer Klassifikation f über einer Attributmenge $\mathcal{A}$ ist eine Menge von Klassifikationsregeln für das Klassenattribut C zu f, so dass jede Instanz i von $\mathcal{A}$ von genau einer Regel aus h abgedeckt wird. Die Klasse c dieser Regel wird als *Voraussage* von h für i bezeichnet. Man schreibt dafür $h(i) = c$.

In der Praxis gibt man meistens Einschränkungen an die zulässigen Hypothesen vor. So kann aus Effizienzgründen eine Beschränkung auf bestimmte Teilmengen oder logische Verknüpfungen sinnvoll sein. Oder man beschränkt sich auf eine bestimmte Form von Regeln, die für die Nutzung der gelernten Hypothesen von Vorteil ist. Die Menge der zulässigen Regeln bezeichnet man dann als die verwendete Hypothesensprache.

Definition 2.22 (Hypothesensprache)**.** Eine *Hypothesensprache* ist eine Menge von Hypothesen.

2.2.1 Einfache Regeln

Geht man davon aus, dass der gegebene Datensatz die zu lernende Klassifikation f fehlerfrei und vollständig beschreibt, so kann man durch Kodierung der Instanzen nach Satz 2.3 für jede Klasse c die Menge der Instanzen, für die f den Wert c liefert, als Klassifikationsregel kodieren. Die so erhaltene exakte Beschreibung von f lässt sich dann mit Hilfe logischer Äquivalenzumformungen auf eine einfachere Form reduzieren.

In der Praxis des maschinellen Lernens ist jedoch nicht immer von fehlerfreien Daten auszugehen. Bei der Datenerfassung kann es zu Messfehlern, Irrtümern oder Ähnlichem kommen. Vollständige Daten sind sogar noch seltener, da die meisten interessanten Probleme zu umfangreich sind, um überhaupt alle Beispiele erfassen zu können.

Definition 2.23 (unvollständiger Datensatz)**.** Ein Datensatz d über $\mathcal{A}$ heißt *unvollständig* bezüglich einer Klassifikation über $\mathcal{A}$ für $C \in \mathcal{A}$, wenn es Instanzen von $\mathcal{A} \backslash \{C\}$ gibt, deren zugehörige Aussagen s_i (nach Satz 2.3) kein Beispiel aus d abdecken.

Jeder endliche Datensatz über einer Attributmenge mit mindestens einem unendlichen Attribut außer dem Klassenattribut ist unvollständig. Doch auch viele Datensätze über endlichen Attributen sind unvollständig. So ist auch der Datensatz aus Tabelle 2.7 unvollständig bezüglich jeder Klassifikation über seiner Attributmenge, die in Tabelle 2.6 dargestellt ist.

Attribut	Domäne
outlook	{sunny, overcast, rainy}
temperature	{hot, mild, cool}
humidity	{high, normal}
windy	{TRUE, FALSE}
play	{yes, no}

Tabelle 2.6: Die Attribute des Wetter-Datensatzes in Tabelle 2.7

outlook	temperature	humidity	windy	play
sunny	hot	high	FALSE	no
sunny	hot	high	TRUE	no
overcast	hot	high	FALSE	yes
rainy	mild	high	FALSE	yes
rainy	cool	normal	FALSE	yes
rainy	cool	normal	TRUE	no
overcast	cool	normal	TRUE	yes
sunny	mild	high	FALSE	no
sunny	cool	normal	FALSE	yes
rainy	mild	normal	FALSE	yes
sunny	mild	normal	TRUE	yes
overcast	mild	high	TRUE	yes
overcast	hot	normal	FALSE	yes
rainy	mild	high	TRUE	no

Tabelle 2.7: Der Wetter-Datensatz aus der weka-Bibliothek mit den Attributen aus Tabelle 2.6

Konstruiert man nach Satz 2.6 eine Regelmenge aus einem unvollständigen Datensatz, so repräsentiert diese Regelmenge keine Klassifikation, da sie nicht alle Instanzen der Attributmenge abdeckt. Um eine Klassifikation zu erhalten, benötigt man also generellere Regeln. Generellere Regeln sind auch meist verständlicher, da sie aus kürzeren Konjunktionen bestehen.

Definition 2.24 (fehlerhafter Datensatz)**.** Ein Datensatz d über $\mathcal{A}$ heißt *fehlerhaft* bezüglich einer Klassifikation f über $\mathcal{A}$ für $C \in \mathcal{A}$, wenn es ein Beispiel i in d mit $i(C) \neq f(i|_{\mathcal{A}\setminus\{C\}})$ gibt.

Der Datensatz aus Tabelle 2.7 ist zum Beispiel bezüglich der Klassifikation

$$f(i) = \begin{cases} \text{yes} & i(\text{outlook}) = \text{overcast} \\ \text{no} & \end{cases}$$

für play fehlerhaft, denn $f((\text{sunny}, \text{mild}, \text{normal}, \text{TRUE})) = \text{no}$, aber es gibt ein Beispiel $(\text{sunny}, \text{mild}, \text{normal}, \text{TRUE}, \text{yes})$ im Datensatz.

Aus fehlerhaften Daten lässt sich die zugrunde liegende Klassifikation nicht mehr zuverlässig bestimmen. Um überhaupt etwas aus fehlerhaften Daten lernen zu können, müssen Regeln zugelassen werden, die nicht korrekt bezüglich des Datensatzes sind. Auch auf fehlerhaften Daten haben sich einfachere, generellere Regeln als erfolgreicher erwiesen (Holte, 1993), während komplexere, speziellere Regeln meist sehr stark von Fehlern beeinflusst werden.

Ein weiteres Problem des Lernens ist, dass nicht alle Klassifikationen durch Regeln ausgedrückt werden können. Zum einen tritt dies bei unendlichen oder unendlich vielen Attributen auf. Da in Regeln nur einstellige Prädikate, nämlich die Basisaussagen über den Attributen, zugelassen sind, können im unendlichen Fall einige zweistellige Prädikate wie Länge > Anzahl nicht ausgedrückt werden. Im endlichen Fall lassen sich solche Relationen wieder auf einstellige Prädikate und damit Basisaussagen zurückführen, indem man einfach für jedes Paar der Relation eine Konjunktion aus den entsprechenden Basisaussagen formuliert. Eine solche Konstruktion liefert im Falle unendlicher Attribute jedoch unendlich viele Konjunktionen.

Andererseits kann dieses Problem auch schon auf endlichen Attributen auftreten, wenn man die Hypothesensprache einschränkt. So macht es in der Praxis durchaus Sinn, zum Beispiel aus Gründen der Effizienz statt $\in$ nur $\leqq$ zu verwenden. Unter solchen Einschränkungen lassen sich nicht mehr alle Instanzen durch Aussagen darstellen. Zum Beispiel trifft eine Aussage $A \leqq a$ auch für alle $b \leq a \in A$ zu.

Damit kann das Ziel des Lernens nur eine möglichst gute Approximation der Klassifikation durch eine Hypothese sein. Das Problem des Lernens von Klassifikationsregeln besteht also darin, aus der Hypothesensprache diejenige Hypothese mit den wenigsten Abweichungen von der gesuchten Klassifikation zu finden.

Definition 2.25 (Klassifikationsproblem für Regel-Lerner)**.** Gegeben sei ein Datensatz d über einer Attributmenge $\mathcal{A}$ und ein Klassenattribut $C \in \mathcal{A}$ sowie eine Hypothesensprache $\mathcal{H}$. Die Beispiele in d entsprechen bis auf eine Fehlerquote von e_d einer unbekannten Klassifikation f für C.

Finde eine Hypothese $h \in \mathcal{H}$, so dass für alle Hypothesen $h' \in \mathcal{A}$ gilt:

$$\left| \left\{ i \in \prod_{A \in \mathcal{A} \setminus C} D_A \,\middle|\, f(i) \neq h(i) \right\} \right| \leq \left| \left\{ i \in \prod_{A \in \mathcal{A} \setminus C} D_A \,\middle|\, f(i) \neq h'(i) \right\} \right|$$

Da die den Beispielen zugrunde liegende Klassifikation dem Lerner nicht bekannt ist, kann die Forderung nach einer minimalen Anzahl an Abweichungen nicht direkt überprüft werden. Der folgende Abschnitt stellt einige Methoden vor, mit denen sich die Qualität einer Hypothese dennoch abschätzen lässt.

2.2.2 Evaluation

Eine ausführliche Betrachtung verschiedenster Methoden zur Bewertung von Hypothesen findet sich in (Witten und Frank, 2005). In diesem Abschnitt werden die wesentlichen Ideen für die Bewertung von Regeln vorgestellt.

Ziel des Lernens ist es, Hypothesen zu finden, die auf unbekannten Instanzen zuverlässige Vorraussagen treffen. Um dies zu testen, wird die Evaluation grundsätzlich nicht auf dem Datensatz durchgeführt, aus dem der Lerner die Hypothese entwickelt. Der für das Lernen verwendete Datensatz wird als Trainingsmenge und der für die Bewertung verwendete als Testmenge bezeichnet. Im Idealfall ist der Schnitt beider Mengen leer.

Um von der gemessenen Fehlerrate auf der Testmenge auf die tatsächliche Fehlerrate bezüglich der Klassifikation zu schließen, sind statistische Verfahren notwendig. Die Evaluation auf jedem einzelnen Beispiel der Testmenge wird dabei als Bernoulli-Experiment aufgefasst, so dass die Evaluation auf dem Gesamten einer Bernoulli-Reihe entspricht. Mit Hilfe eines Signifikanztests lassen sich dann Konfidenzintervalle bestimmen. Somit läßt sich schließen, mit welcher Wahrscheinlichkeit die tatsächliche Fehlerrate in einem bestimmten Intervall liegt. Die Größe der Intervalle ist dabei abhängig von der Größe der Testmenge: je größer die Testmenge, desto kleiner die Intervalle.

Für eine zuverlässige Evaluation ist also eine große Testmenge nötig. Doch in der Praxis sind die verfügbaren Datensätze oft nicht ausreichend groß, um genug Material sowohl für die Test- als auch für die Trainingsmenge zu liefern. Um dennoch möglichst viel der vorhandenen Daten für die Evaluation nutzen zu können, wendet man die sogenannte Cross-Validation an.

Bei diesem Verfahren wird der Datensatz in mehrere (meist 10) Teile gleicher Länge zerlegt. Anschließend wird eine auf einem Teil gelernte Hypothese auf den anderen Teilen getestet. Dies wird für die anderen Teile wiederholt, so dass jeder Teil als

Testmenge für eine auf den anderen Teilen gelernte Hypothese verwendet wurde. Der Durchschnitt der so erhaltenen Fehlerraten wird dann zur Bewertung des Lerners auf dem ursprünglichen Datensatz verwendet. Da verschiedene Partitionierungen zu sehr unterschiedlichen Ergebnissen führen können, wird auch die Cross-Validation mehrfach durchgeführt und aus den Ergebnissen wieder ein Mittelwert gebildet. Diese Ergebnisse lassen sich nicht direkt zur Bewertung einer Hypothese verwenden, da verschiedene Hypothesen evaluiert werden. Jedoch lassen so Aussagen über die Qualität des Lerners auf den gegebenen Daten machen.

2.3 Lerner für Klassifikationsregeln

Die Aufgabe eines Lerners ist, für jede Klasse eine Regel zu lernen, die den in Kapitel 2.2.2 vorgestellten Kriterien entspricht. Für die Suche nach einer solchen Regel wird die durch Implikation gegebene Verbandstruktur auf den Regeln ausgenutzt. Abbildung 2.6 zeigt einen solchen Verband für zwei binäre Attribute. Die Basisaussagen sind durch Konkatenation der Elemente der jeweiligen Mengen abgekürzt.

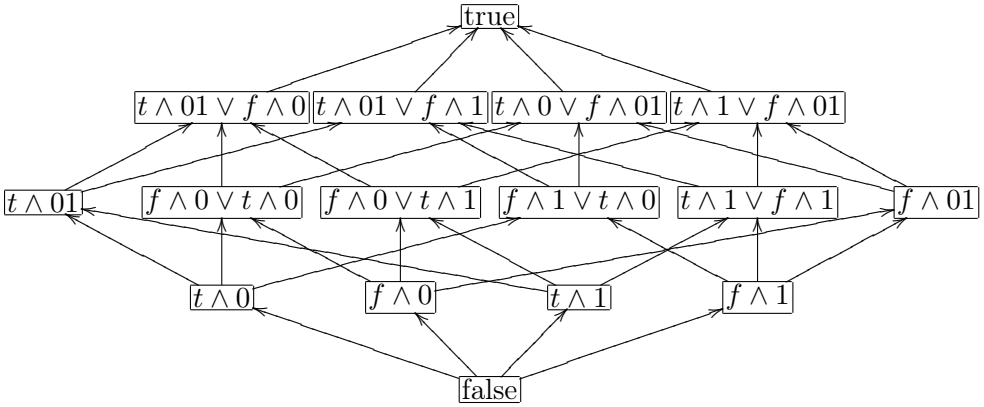


Abbildung 2.6: Verband der Regeln über zwei binären Attributen

Da nach Satz 2.5 jede Regel als Disjunktion von Konjunktionen dargestellt werden kann, wird in der Praxis nicht im Verband aller Regeln gesucht. Stattdessen durchsucht man den Verband der konjunktiven Regeln und bildet aus mehreren konjunktiven Regeln eine Theorie. Die Theorie wird dabei ähnlich zu den in Abschnitt 2.2 vorgestellten Entscheidungslisten verwendet und entspricht einer Regel in disjunktiver Normalform.

Definition 2.26 (konjunktive Regel)**.** Eine *konjunktive Regel* ist eine Regel, deren Körper aus einer Konjunktion von Basisaussagen mit höchstens einer Basisaussage zu jedem Attribut besteht, so dass jede Basisaussage des Körpers die Form $A \in \{x\}$ hat.

Konjunktive Regeln sind sehr viel einfacher und damit eingeschränkter als allgemeine Regeln. Jedoch sind sie für Menschen einfacher zu verstehen und bilden eine Verbandstruktur, die sehr viel einfacher zu durchsuchen ist als der Verband aller Regeln. Außerdem lassen sich Basisaussagen als Disjunktionen von konjunktiven Regeln ausdrücken, so dass sich jede Regel als Verknüpfung von konjunktiven Regeln ausdrücken lässt. Abbildung 2.7 zeigt den Verband der konjunktiven Regeln über zwei binären Attributen. Die Basisaussagen sind wieder durch Konkatenation der Elemente abgekürzt.

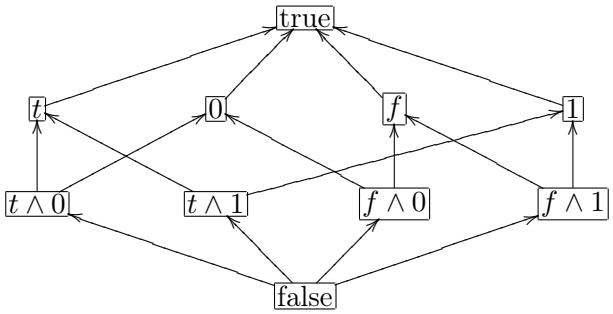


Abbildung 2.7: Verband der konjunktiven Regeln

Während im Verband aller Regeln (siehe Abbildung 2.6) jede Spezialisierung einer Regel maximal ein Beispiel weniger abdeckt, fallen bei der Spezialisierung auf dem Verband der konjunktiven Regeln (siehe Abbildung 2.7) alle Beispiele weg, die für das gewählte Attribut nicht den gewählten Wert haben. Eine Suche auf dem Verband aller Regeln muss also in vielen kleinen Schritten auf eine einzelne Regel zusteuern, während einzelne konjunktive Regeln in wenigen großen Schritten gefunden werden können, jedoch mehrere Regeln gefunden werden müssen.

Für das Lernen einer Theorie ist die in (Fürnkranz, 1999) untersuchte Separate&Conquer Methode weit verbreitet. Diese Methode wird auch als Grundlage dieser Arbeit dienen. Das Ziel dieser Methode ist, eine möglichst kleine Menge konjunktiver Regeln zu lernen, so dass jedes Beispiel des Datensatzes von mindestens einer Regel abgedeckt wird. Dazu wird eine anfangs leere Theorie schrittweise um Regeln ergänzt. Zum Lernen der Regeln werden nur die bisher nicht abgedeckten Beispiele verwendet. Algorithmus 2.1 ist die generische Variante für diese Methode aus (Fürnkranz, 1999). Dort findet sich eine detaillierte Auseinandersetzung mit der Methode.

Der generische Algorithmus wird durch konkrete Definition der verwendeten Funktionen instanziiert. Wie in (Fürnkranz, 1999) beschrieben, kann man so sämtliche Separate&Conquer Algorithmen erhalten. Eine Implementierung dieser Methode in Java findet sich in (Thiel, 2005).

Algorithmus 2.1 : Separate&Conquer-Algorithmus aus (Fürnkranz, 1999)

Eingabe: Beispielmenge *Examples*
Theory $\longleftarrow \emptyset$
while `Positive`(*Examples*) $\neq \emptyset$ **do**
  Rule $\longleftarrow$ `FindBestRule`(*Examples*)
  if `RuleStoppingCriterion`(*Theory*, *Rule*, *Examples*) **then**
    exit **while**
  Examples $\longleftarrow$ *Examples*$\backslash$`Cover`(*Rule*, *Examples*)
  Theory $\longleftarrow$ *Theory* $\cup$ *Rule*
Theory $\longleftarrow$ `PostProcess`(*Theory*)
return *Theory*

3 Entwurf

Das Ziel dieser Arbeit ist es, eine erweiterbare Umgebung zu schaffen, in der Regel-Lerner einfach formuliert werden können. Für den Fall der Separate&Conquer-Lerner bietet das in (Thiel, 2005) entworfene SeCo-Framework eine solche Umgebung, in der man einen Lerner durch Auswahl vorgegebener sowie eigener Bausteine zusammensetzen kann.

Die Idee des folgenden Entwurfes ist nun, eine neue Datenabstraktion zu schaffen, auf der das SeCo-Framework aufbauen kann. Diese Abstraktion soll dabei aber allgemein genug sein, um darin beliebige Regel-Lerner zu formulieren. Auf dieser Grundlage kann dann das SeCo-Framework auf allgemeine Regel-Lerner erweitert werden.

3.1 Kodierung von Daten und Regeln

Wie in Abschnitt 2.1.1 angedeutet beschränkt man sich in der Praxis auf endliche Datensätze über endlichen Mengen endlicher Attribute. Der Hauptgrund ist, dass man so alle Objekte vom Attribut über die Instanz hin zum Datensatz unter Verwendung endlich vieler Symbole kodieren kann. Im Folgenden sind daher alle Attribute, Attributmengen und Datensätze endlich.

Tabelle 3.1 zeigt einen Beispieldatensatz aus der in (Witten und Frank, 2005) vorgestellten weka-Bibliothek, der im Folgenden als Grundlage für Beispiele dienen wird. Die Attribute sind outlook mit den Werten $\{\text{sunny}, \text{overcast}, \text{rainy}\}$, temperature mit Domäne $\{n | n \in \mathbb{Z} \wedge n < 150 \wedge n \geq -150\}$ und natürlicher Ordnung, humidity mit Wertebereich $\{n | n \in \mathbb{N} \wedge n < 100\}$ und ebenfalls üblicher Ordnung sowie die beiden binären Attribute windy $\{\text{TRUE}, \text{FALSE}\}$ und play $\{\text{yes}, \text{no}\}$. Für eine Klassifikation mit Klassenattribut play beschreibt die Regel in Abbildung 3.1 die Klasse yes.

$$\begin{aligned} \text{outlook} &= \text{overcast} \vee \\ \text{outlook} = \text{rainy} \wedge \text{windy} &= \text{FALSE} \vee \\ \text{outlook} = \text{sunny} \wedge \text{humidity} &\leq 70 \\ &\rightarrow \text{play} = \text{yes} \end{aligned}$$

Abbildung 3.1: Eine Regel für die Klasse *yes* im Wetter-Beispiel

outlook	temperature	humidity	windy	play
sunny	85	85	FALSE	no
sunny	80	90	TRUE	no
overcast	83	86	FALSE	yes
rainy	70	96	FALSE	yes
rainy	68	80	FALSE	yes
rainy	65	70	TRUE	no
overcast	64	65	TRUE	yes
sunny	72	95	FALSE	no
sunny	69	70	FALSE	yes
rainy	75	80	FALSE	yes
sunny	75	70	TRUE	yes
overcast	72	90	TRUE	yes
overcast	81	75	FALSE	yes
rainy	71	91	TRUE	no

Tabelle 3.1: Der Wetter-Datensatz, ein Beispieldatensatz aus der weka-Bibliothek

3.1.1 Regeln als Bitvektoren

Wie in Abschnitt 2.1.2 beschrieben, besteht eine Klassifikationsregel aus Vorbedingung und Klasse. Die Klasse ist dabei nur ein Attributwert, dessen Kodierung der verwendeten Datenkodierung entspricht. Die Vorbedingung wird in disjunktiver Normalform angenommen, so dass sie als Menge von Konjunktionen aufgefasst werden kann. Diese Konjunktionen lassen sich als Listen von Teilmengen der Domänen darstellen, indem man statt der Basisaussagen $A \subseteq P_A$ einfach die Teilmengen P_A notiert und die konjunktiven Verknüpfungen ($\wedge$) impliziert.

Um die Teilmengen den Attributen zuordnen zu können, wird dabei von der kanonischen disjunktiven Normalform (siehe Definition 2.15) ausgegangen, die nach Satz 2.4 aus der disjunktiven Normalform gebildet wird. So wird die Aussage

$$\text{outlook} = \text{sunny} \wedge \text{humidity} \leq 70$$

durch die Liste

$$\{\text{sunny}\}, \{n | n \in \mathbb{Z} \wedge -150 < n < 150\}, \{n | n \in \mathbb{N} \wedge n < 100\}, \{\text{TRUE}, \text{FALSE}\}, \{\text{yes}, \text{no}\}$$

kodiert.

Solche Listen lassen sich mittels der jeweiligen Datenkodierung bilden, indem jede Menge als Liste ihrer Elemente kodiert wird. Allerdings lassen sie sich auch unabhängig von der Datenkodierung kodieren: Eine Teilmenge der Domäne D_A wird durch Angabe ihrer charakteristischen Funktion kodiert, indem man die Funktionswerte (0 oder 1)

gemäß einer bei Definition des Attributs gegebenen Reihenfolge notiert. Man erhält so einen Bitvektor mit einem Bit für jeden Attributwert, das anzeigt, ob das Attribut den entsprechenden Wert annehmen darf. Zur Veranschaulichung zeigt Tabelle 3.2 die Teilmengen der Domänen des Attributs outlook und Tabelle 3.3 die Teilmengen der Domäne des Attributs windy.

Menge	sunny	overcast	rainy
$\emptyset$	0	0	0
{sunny}	1	0	0
{overcast}	0	1	0
{rainy}	0	0	1
{overcast, rainy}	0	1	1
{sunny, rainy}	1	0	1
{sunny, overcast}	1	1	0
{sunny, overcast, rainy}	1	1	1

Tabelle 3.2: Kodierung von Teilmengen endlicher Domänen am Beispiel outlook

Menge	TRUE	FALSE
$\emptyset$	0	0
{TRUE}	1	0
{FALSE}	0	1
{TRUE, FALSE}	1	1

Tabelle 3.3: Kodierung von Teilmengen endlicher Domänen am Beispiel windy

Nachdem nun Teilmengen der Domänen als Bitvektoren kodiert sind, kann man Listen dieser Teilmengen wiederum als Bitvektoren kodieren, indem man die kodierten Teilmengen konkateniert. Jedes Bit steht für einen Wert eines Attributs. Eine so kodierte Aussage deckt eine Instanz ab, wenn für jedes Attribut das Bit gesetzt ist, das für den Wert der Instanz in diesem Attribut steht. In Tabelle 3.4 sind die Vorbedingungen der Regeln aus Abbildung 3.1 in Bitvektorkodierung dargestellt. Die Punkte stehen für die 400 ausgelassenen Bits der Attribute temperature und humidity, welche in den beiden ersten Zeilen alle gesetzt sind. In Zeile 3 sind die letzten 29 Bits für humidity nicht gesetzt.

So lässt sich nun jede Regel als ein Paar (c, b) aus der Klasse c und einer Liste von Bitvektoren b kodieren. Die Regel aus Abbildung 3.1 ist in Abbildung 3.2 auf diese Weise dargestellt.

Vorbedingung	Bitvektor
outlook $=$ overcast	010...11
outlook $=$ rainy $\wedge$ windy $=$ FALSE	001...01
outlook $=$ sunny $\wedge$ humidity $\leqq$ 70	100...11

Tabelle 3.4: Regeln und ihre Bitvektorkodierung

$$(\text{yes}, (010\ldots11, 001\ldots01, 100\ldots11))$$

Abbildung 3.2: Eine Regel als Paar aus Klasse und Liste von Bitvektoren

outlook	temperature	humidity	windy	play
0	235	85	1	1
0	230	90	0	1
1	233	86	1	0
2	220	96	1	0
2	218	80	1	0
2	215	70	0	1
1	214	65	0	0
0	222	95	1	1
0	219	70	1	0
2	225	80	1	0
0	225	70	0	0
1	222	90	0	0
1	231	75	1	0
2	221	91	0	1

Tabelle 3.5: Das Wetter-Beispiel in Ganzzahlkodierung

Attribut	outlook			temperature	humidity	windy	
Instanz	0			235	85	1	
	↙	↓	↘	⋮	⋮	↙	↘
Index	0	1	2			0	1
	↓			⋮	⋮		↓
Regel	0	1	0	...	...	1	1
Match		−		+	+	+	

Abbildung 3.3: Matching für Ganzzahlkodierung am Beispiel outlook $=$ overcast mit negativem Ergebnis

3.1.2 Daten als Zahlen

Für die im vorherigen Abschnitt beschriebene Regelkodierung sind die Werte der jeweiligen Attribute in eine Reihenfolge zu bringen, also durchzunummerieren. Die Werte der Attribute können dann durch diese Indizes kodiert und in einer Liste gespeichert werden. Fehlende Werte werden als -1 kodiert.

Auch Weka kodiert nominelle Attribute als Indizes zu einem geordneten Wertebereich und fehlende Werte durch -1.

Das Wetter-Beispiel in dieser Kodierung zeigt Tabelle 3.5. Die Werte sind dabei wie folgt indiziert; für outlook: sunny $= 0$, overcast $= 1$, rainy $= 2$, für temperatur ab -150 durchzählend (Wert+150), für humidity durchzählend ab 0, für windy TRUE $= 0$, FALSE $= 1$ und für play yes $= 0$, no $= 1$.

Beim Matching muss für jedes Attribut nur nachgeschlagen werden, ob das in der Instanz angegebene Bit des Bitvektors gesetzt ist. Abbildung 3.3 zeigt ein Beispiel für Matching bei einer nicht abgedeckten Instanz (negatives Ergebnis), Abbildung 3.4 eines mit abgedeckter Instanz (positives Ergebnis).

Sei $\{A_1, \ldots, A_k\}$ die Menge der Attribute, $(c, ((b_{1,1}, \ldots, b_{1,n}), \ldots (b_{m,1}, \ldots, b_{m,n})))$ eine kodierte Regel, $(z_1, \ldots, z_k)$ eine kodierte Instanz und $o_1 = 0$, $o_{i+1} = o_i + |D_{A_i}|$ die Offsets zu den Attributen, dann entspricht

$$\exists j \leq m \in \mathbb{N}. \forall i \leq k \in \mathbb{N}. b_{j,z_i+o_i} = 1$$

der Gültigkeit der Regel für die Instanz.

3.1.3 Daten als Bitvektoren

Wie in Satz 2.3 gezeigt, lässt sich aus jedem Beispiel eine Aussage erzeugen, die für die vorgegebene Attributmenge genau dieses Beispiel beschreibt. Da Aussagen in Regeln bereits als Bitvektoren kodiert werden, liegt es nahe, Aussagen aus Beispielen ebenfalls als Bitvektoren zu kodieren.

Den Bitvektor zu einer Instanz kann man auch ohne den Umweg über Regeln direkt aus der Instanz konstruieren. Wieder steht jedes Bit für einen Wert eines Attributs. Die Reihenfolge wird bei der Definition der Attributmenge vorgegeben. Ein Bit wird dann gesetzt, wenn das entsprechende Attribut in der Instanz den entsprechenden Wert hat. Andernfalls wird das Bit nicht gesetzt. Der so konstruierte Bitvektor ist identisch zu dem, der sich für die das Beispiel beschreibende Regel ergibt.

Fehlende Werte werden repräsentiert, indem für das Attribut gar kein Bit gesetzt wird. Abbildung 3.6 stellt das Wetter-Beispiel aus Abbildung 3.1 in dieser Bitvektorkodierung da. Die 300 Bits für temperature sowie die 100 Bits für humidity sind dabei aus Platzgründen hexadezimal ohne führende Nullen notiert.

Das Matching bei dieser Kodierung ist ein Test, ob alle im Bitvektor der Instanz gesetzten Bits auch in einem Bitvektor der Regel gesetzt sind. Ein bitweises Und von

Attribut	outlook			temperature	humidity	windy	
Instanz	1			233	86	1	
	↙	↓	↘	⋮	⋮	↙	↘
Index	0	1	2			0	1
		↓		⋮	⋮		↓
Regel	0	1	0	...	...	1	1
Match		+		+	+	+	

Abbildung 3.4: Matching für Ganzzahlkodierung am Beispiel outlook $=$ overcast mit positivem Ergebnis

outlook	temperature	humidity	windy	play
100	...10000000000000000	...4000	01	01
100	...200000000000000000	...200	10	01
010	...40000000000000000	...2000	01	10
001	...800000000000000000	...8	01	10
001	...2000000000000000000	...80000	01	10
001	...10000000000000000000	...20000000	10	01
010	...20000000000000000000	...400000000	10	10
100	...2000000000000000000	...10	01	01
100	...10000000000000000000	...20000000	01	10
001	...4000000000000000000	...80000	01	10
100	...4000000000000000000	...20000000	10	10
010	...2000000000000000000	...200	10	10
010	...100000000000000000	...1000000	01	10
001	...4000000000000000000	...100	10	01

Tabelle 3.6: Das Wetter-Beispiel kodiert in Bitvektoren

Instanz-Bitvektor und Regel-Bitvektor muss also wieder den Instanz-Bitvektor ergeben. Sei $(c, ((b_{1,1}, \ldots, b_{1,n}), \ldots (b_{m,1}, \ldots, b_{m,n})))$ eine kodierte Regel und $(i_1, \ldots, i_n)$ eine kodierte Instanz, dann entspricht

$$\exists j \leq m \in \mathbb{N}. \forall k \leq n \in \mathbb{N}. i_k = 1 \rightarrow b_{j,k} = 1$$

der Gültigkeit der Regel für die Instanz.

Abbildung 3.5 visualisiert das Matching anhand einer Regel und einer nicht abgedeckten Instanz aus dem Wetter-Beispiel. Ein Beispiel für eine abgedeckte Instanz zeigt Abbildung 3.6.

Attribut	outlook			temperature	humidity	windy	
Regel	0	1	0	...	...	1	1
Instanz	1	0	0	...	...	0	1
bitweise Und	0	0	0	...	...	0	1
xor Instanz (Mismatch)	1	0	0	...	...	0	0

Abbildung 3.5: Matching von Bitvektoren am Beispiel outlook = overcast bei nicht abgedeckter Instanz

Attribut	outlook			temperature	humidity	windy	
Regel	0	1	0	...	...	1	1
Instanz	0	1	0	...	...	0	1
bitweise Und	0	1	0	...	...	0	1
xor Instanz (Mismatch)	0	0	0	...	...	0	0

Abbildung 3.6: Matching von Bitvektoren am Beispiel outlook = overcast bei abgedeckter Instanz

3.1.4 Kompakte Bitvektoren

Die bisher vorgestellte Bitvektorkodierung benötigt für Attribute mit vielen Werten extrem lange Bitvektoren. Für jeden Wert jedes Attributs fällt ein Bit an. Bei 64 Werten pro Attribut sind also für jedes Attribut 64 Bit notwendig. Weka speichert pro Attribut einen `long`, also ebenfalls 64 Bit. Allerdings kann Weka damit bis zu $2^{64} - 1$ Werte pro Attribut handhaben.

Dieses Problem umgeht der im folgenden vorgestellte Kompromiss aus Bitvektor- und Ganzzahldarstellung, allerdings auf Kosten des sehr eleganten Matchings.

Anstatt die Indizes der Ganzzahldarstellung mit 32 oder mehr Bits zu repräsentieren, wird der Index für das Attribut A auf $\lceil \log_2(|D_A| + 1) \rceil$ Bits gespeichert. So lassen sich

genau die benötigten Indizes sowie ein Wert zur Anzeige fehlender Attribute speichern. Den so kodierten Wetter-Datensatz zeigt Tabelle 3.7.

Speichert man fehlende Werte als 0 ist bei der Dekodierung noch eine Subtraktion nötig. Andernfalls ist zur Erkennung fehlender Werte ein Vergleich mit einer attributspezifischen Konstante notwendig. Zur Vereinheitlichung der Handhabung werden daher ein fehlender Wert als 0 und Indizes um 1 erhöht kodiert.

outlook	temperature	humidity	windy	play
01	011101100	1010110	10	10
01	011100111	1011011	01	10
10	011101010	1010111	10	01
11	011011101	1100001	10	01
11	011011011	1010001	10	01
11	011011000	1000111	01	10
10	011010111	1000010	01	01
01	011011111	1100000	10	10
01	011011100	1000111	10	01
11	011100010	1010001	10	01
01	011100010	1000111	01	01
10	011011111	1011011	01	01
10	011101000	1001100	10	01
11	011011110	1011100	01	10

Tabelle 3.7: Das Wetter-Beispiel kodiert in kompakten Bitvektoren

Das Matching läuft analog zur Ganzzahldarstallung. Der entsprechende Teil des Bitvektors muss zu einer Ganzzahl dekodiert werden, die das zu prüfende Bit des Regel-Bitvektors bezeichnet. Dieser Prozess wird in Abbildung 3.7 für den Fall einer nicht abgedeckten Instanz und in Abbildung 3.8 mit einer abgedeckten Instanz dargestellt.

Sei $\{A_1, \ldots, A_k\}$ die Menge der Attribute, $(c, ((b_{1,1}, \ldots, b_{1,n}), \ldots (b_{m,1}, \ldots, b_{m,n})))$ eine kodierte Regel, $((z_{1,1+\lfloor \log_2 |D_{A_1}| \rfloor}, \ldots z_{1,1}), \ldots, (z_{k,1+\lfloor \log_2 |D_{A_k}| \rfloor} \ldots z_{k,1}))$ eine kodierte Instanz und $o_1 = 0$, $o_{i+1} = o_i + |D_{A_i}|$ die Offsets zu den Attributen, dann entspricht

$$\exists j \leq m \in \mathbb{N}. \forall i \leq k \in \mathbb{N}. b_{j, \sum_{l=1}^{1+\lfloor \log_2 |D_{A_i}| \rfloor} z_{i,l} 2^{l-1} - 1} = 1$$

der Gültigkeit der Regel für die Instanz.

Attribut	outlook			temperature	humidity	windy	
Instanz	0	1		011101100	1010110	1	0
dekodiert	0			235	85	1	
	↙	↓	↘	⋮	⋮	↙	↘
Index	0	1	2			0	1
	↓			⋮	⋮		↓
Regel	0	1	0	...	...	1	1
Match		−		+	+	+	

Abbildung 3.7: Matching von kompakten Bitvektoren am Beispiel outlook = overcast bei nicht abgedeckter Instanz

Attribut	outlook			temperature	humidity	windy	
Instanz	1	0		011101010	1010111	1	0
dekodiert	1			233	86	1	
	↙	↓	↘	⋮	⋮	↙	↘
Index	0	1	2			0	1
		↓		⋮	⋮		↓
Regel	0	1	0	...	...	1	1
Match		+		+	+	+	

Abbildung 3.8: Matching von kompakten Bitvektoren am Beispiel outlook = overcast mit abgedeckter Instanz

3.1.5 Kodierung geordneter Attribute

Ein mengenwertiges Attribut über einer Basismenge X lässt sich platzsparend kodieren, indem man statt für jede Teilmenge von X nur für jedes Element von X ein Bit kodiert. Teilmengen werden dann durch das Setzen mehrerer Bits kodiert. Tabelle 3.8 stellt die beiden Varianten am Beispiel einer dreielementigen Menge gegenüber.

Teilmenge	einfacher Bitvektor	effizienter Bitvektor
$\emptyset$	10000000	000
$\{1\}$	01000000	100
$\{2\}$	00100000	010
$\{3\}$	00010000	001
$\{2,3\}$	00001000	011
$\{1,3\}$	00000100	101
$\{1,2\}$	00000010	110
$\{1,2,3\}$	00000001	111

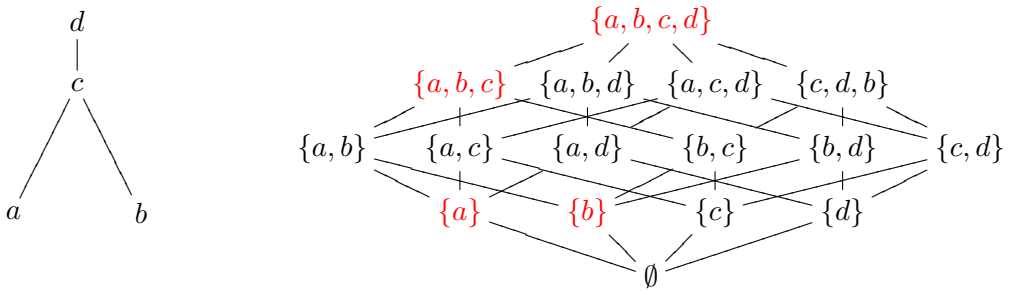
Tabelle 3.8: effiziente Kodierung der Teilmengen von $\{1,2,3\}$

Das Matching lässt sich durch Erzeugen des einfachen Bitvektors auf das Matching für einfache Bitvektoren (siehe Abschnitt 3.1.3) reduzieren. Für die Konstruktion des einfachen Bitvektors bietet sich eine Tabelle wie Tabelle 3.8 an.

Generell lassen sich geordnete Attribute so durch Bitvektoren kodieren, dass die Ordnungsrelation durch ein logisches Und auf Bitvektoren überprüft werden kann. Ordnet man jedem Element x einer Domäne D_A die Menge $\{y \in D_A | y \leq x\}$ zu, erhält man eine Einbettung der geordneten Menge D_A in 2^{D_A} geordnet bezüglich Mengeninklusion. Abbildung 3.9 zeigt die Einbettung an einem Beispiel. Im Potenzmengenverband sind die eingebetteten Elemente der ursprünglichen Ordnungs rot hervorgehoben. Diese Potenzmenge kann man dann effizient kodieren, indem man wieder für jede einlementige Menge ein Bit verwendet.

Mit Gleichheit als Ordnungsrelation entspricht diese Kodierung genau der einfachen Bitvektorkodierung, da jedes Element x der Domäne auf die Menge x abgebildet und mit einem Bit kodiert wird.

Wie am Beispiel der mengenwertigen Attribute zu sehen ist, lassen sich einige Ordnungen effizienter kodieren. Statt $|D_A|$ Bits braucht man zur Kodierung eines mengenwertigen Attributs nur $log_2 D_A$ Bits. Das Problem, die kürzeste Bitvektorkodierung für eine Ordnung zu finden ist in (Habib u. a., 2004) bezüglich seiner Komplexität untersucht worden. Im Allgemeinen ist das Problem NP-vollständig, doch für baumartige Ordnungen wird ein Algorithmus vorgestellt, der Bitvektorkodierungen erzeugt, deren Länge sich von der minimal möglichen höchstens um den Faktor 4 unterscheidet. Die Laufzeit des Algorithmus ist dabei quadratisch in der Anzahl der Elemente.



$$a \rightarrow 1000, b \rightarrow 0100, c \rightarrow 1110, d \rightarrow 1111$$

Abbildung 3.9: Einbettung einer Ordnung (links) in ihre Potenzmenge (rechts) und die daraus resultierende Bitvektorkodierung (unten)

3.1.6 Alternative Kodierungen für Regeln

Die effiziente Kodierung mengenwertiger Attribute lässt sich auch für Regeln verwenden. Dadurch wird jedoch die Hypothesensprache eingeschränkt, da nur noch Basisaussagen der Form $A \subseteq a$ zugelassen werden. Für nicht mengenwertige Attribute verwendet man die bereits vorgestellte Kodierung, die der effizienten Kodierung eines mengenwertigen Attributs mit der Domäne als Basismenge entspricht.

Mit der gleichen Kodierung für Beispiele entspricht das Matching einer einfachen Bitvektor-Verknüpfung wie im Falle der einfachen Bitvektoren (siehe Abschnitt 3.1.3).

Im Falle der anderen Datenkodierungen könnte man das Matching für nicht mengenwertige Attribute aus den vorhergehenden Abschnitten übernehmen. Jedoch wären dann die Attribute unterschiedlich zu behandeln. Für eine einheitliche Behandlung aller Attribute ist es daher erforderlich, nicht mengenwertige Attribute als mengenwertige Attribute aufzufassen, die nur einelementige Teilmengen als Werte annehmen.

Anhand einer Tabelle wie Tabelle 3.8 für jedes Attribut lässt sich aus der effizient kodierten Regel wieder ein einfacher Bitvektor konstruieren, für den das Matching wie in den vorhergehenden Abschnitten beschrieben abläuft.

Alternativ kann man die effiziente Kodierung der Teilmengen als Binärkodierung der für die Datenkodierung benötigten Indizes auffassen. Die Reihenfolge der Teilmengen wird also durch ihre Kodierung bestimmt. Der in der Ganzahldarstellung kodierte Index wird binär kodiert und anschließend als effiziente Kodierung einer Teilmenge aufgefasst. Damit entspricht die kompakte Bitvektorkodierung genau der effizienten Teilmengenkodierung. Für die einfache Bitvektorkodierung muss zunächst der Index des gesetzten Bits bestimmt werden. Abbildung 3.9 stellt die verschiedenen Datenkodierungen am Beispiel der Menge $\{1, 2, 3\}$ gegenüber.

Teilmenge	Index	einfacher Bitvektor	effizienter Bitvektor
$\emptyset$	0	10000000	000
$\{3\}$	1	01000000	001
$\{2\}$	2	00100000	010
$\{2,3\}$	3	00010000	011
$\{1\}$	4	00001000	100
$\{1,3\}$	5	00000100	101
$\{1,2\}$	6	00000010	110
$\{1,2,3\}$	7	00000001	111

Tabelle 3.9: Datenkodierungen für effiziente Teilmengenkodierung der Regeln

3.2 Vergleich der Kodierungen

Abschnitt 3.1 stellt zwei Regelkodierungen mit jeweils drei dazu passenden Datenkodierungen vor. Nach einem Vergleich der beiden Regelkodierungen werden die Datenkodierungen gegenübergestellt. Anschließend folgt eine Diskussion über die Kombinationen der Datenkodierungen mit den Regelkodierungen.

3.2.1 Vergleich der Regelkodierungen

Die beiden Regelkodierungen unterscheiden sich deutlich, da die alternative Kodierung für mengenwertige Attribute nur Basisaussagen der Form $A \leq a$ zulässt, während die einfache Kodierung beliebige Basisaussagen erlaubt.

Durch diese Einschränkung wird die Länge der Bitvektoren für mengenwertige Attribute von $|D_A| = |2^X| = 2^{|X|}$ auf $\log_2 |D_A| = log_2 2^{|X|} = X$ reduziert.

Deckt eine effizient kodierte Regel eine Instanz i ab, dann deckt sie auch diejenigen Instanzen ab, die sich von i lediglich dadurch unterscheiden, dass sie in mengenwertigen Attributen kleinere Werte als i annehmen. Dieses Problem lässt sich mit Entscheidungslisten (siehe Abschnitt 2.2) jedoch umgehen, indem man zunächst die Regeln für die Ausnahmen notiert.

Trotz dieser Unterschiede kann man die einfache Kodierung als eine Variante der effizienten ansehen. Dabei werden alle Attribute als mengenwertig mit ihrer ursprünglichen Domäne als Basismenge angesehen. Man kann dann für jedes mengenwertige Attribut entscheiden, ob man die Basismenge oder deren Potenzmenge als Domäne angibt. Im ersten Fall erhält man die Variante mit kürzeren Bitvektoren und eingeschränkter Hypothesensprache, im zweiten die einfache Kodierung.

3.2.2 Vergleich der Datenkodierungen

Allen drei Kodierungen ist gemein, dass die Werte der Attribute zunächst nummeriert werden. In der Ganzzahldarstellung werden diese Indizes einfach als Zahlen kodiert. Die Anzahl der dafür verwendeten Bits ist abhängig von der Implementierung, aber konstant. Sei c diese Anzahl Bits, die zur Speicherung einer Ganzzahl verwendet wird. Das erste Bit repräsentiert das Vorzeichen, also lassen sich 2^{c-1} Indizes darstellen. Attribute mit mehr als 2^{c-1} Werten lassen sich in dieser Darstellung nicht kodieren. Für Attribute mit deutlich weniger Werten benötigt diese Kodierung wiederum unnötig viel Platz. Den Extremfall bilden binäre Attribute, die in der Praxis sehr häufig auftreten. Für ein binäres Attribut sind nur zwei Werte und damit 1 Bit notwendig, so dass die Ganzzahlkodierung $c - 1$ überflüssige Bits benötigt.

Dieses Problem löst die kompakte Bitvektorkodierung. Anstatt die Zahl in einer festen Zahl Bits zu speichern, wird für jedes Attribut die Anzahl an benötigten Bits bestimmt. Somit können beliebig große Attribute kodiert werden und der Speicherbedarf ist minimal. Allerdings müssen die Start- und Endindizes der Attribute bei jedem Zugriff auf einen Attributwert berechnet oder mit dem Attribut gespeichert werden. Da für das Matching auf einzelne Attributwerte zugegriffen werden muss, wird die Speichereffizienz mit einer recht hohen Laufzeitineffizienz erkauft.

Die einfache Bitvektorkodierung optimiert die Laufzeit des Matching auf Kosten der Speichereffizienz. Für jeden Attributwert wird ein Bit verwendet und damit ab c Werten pro Attribut sogar der Speicherbedarf der Ganzzahlkodierung erreicht und sehr schnell überschritten. Für den Zugriff auf einzelne Attributwerte ist ein Speichern oder Berechnen der Indizes wie im Falle der kompakten Bitvektorkodierung nötig. Allerdings werden diese Indizes nicht beim Matching verwendet, welches prinzipiell nur aus zwei logischen Verknüpfungen, einem Und und einem anschließenden exklusiven Oder, besteht.

Da die Bitvektoren in der einfachen Kodierung recht lang werden, stellt sich die Frage, ob eine Sparse-Kodierung der Bitvektoren dieses Problem beseitigen kann.

3.2.3 Sparse-Kodierung

Eine Sparse-Kodierung speichert nur die Indizes der gesetzten (oder nicht gesetzten) Bits. Bei großen Bitvektoren mit kaum gesetzten (oder nicht gesetzten) Bits kann man so den Speicherbedarf reduzieren, wenn die gespeicherten Indizes weniger Platz beanspruchen als die nicht mehr gespeicherten nicht gesetzten (oder gesetzten) Bits.

Um einen Index abzuspeichern, benötigt man c Bits. Für einen Bitvektor der Länge l ergibt sich in einer Sparse-Kodierung also ein Speicherbedarf von nc Bits. Dabei ist n die Anzahl der gesetzten Bits im Bitvektor. Um überhaupt den Speicherbedarf reduzieren zu können, muss $l > nc$ sein. Die Anzahl der gesetzten Bits muss also kleiner als $\frac{l}{c}$ sein.

Für die Datenkodierung wird in der einfachen Bitvektordarstellung pro Attribut ein Bit gesetzt. Die Anzahl der gesetzten Bits entspricht also der Anzahl der Attribute $|\mathcal{A}|$. Die Länge des Bitvektors ergibt sich als $l = \sum_{A \in \mathcal{A}} |D_A|$. Dies lässt sich auch über die Anzahl der Attribute und die durchschnittliche Attributwertigkeit $w := \frac{\sum_{A \in \mathcal{A}} |D_A|}{|\mathcal{A}|}$ ausdrücken: $l = w|\mathcal{A}|$. Für eine Reduzierung des Speicherbedarfs muss $l > nc$ gelten, also $w|\mathcal{A}| > |\mathcal{A}|c$ und damit $w > c$.

Sobald die Attribute also durchschnittlich mehr Werte haben, als man Bits braucht, um eine Ganzzahl abzuspeichern, liefert eine Sparse-Kodierung der Instanzen eine Reduzierung des Speicherbedarfs.

Unter diesen Voraussetzungen unterscheidet sich eine Sparse-Kodierung für Instanzen nur unwesentlich von einer Ganzzahlkodierung. Statt eines Indexes für das entsprechende Attribut, auf den noch die Position des Attributs im Bitvektor addiert werden muss, wird in der Sparse-Kodierung ein Index direkt in den Bitvektor gespeichert.

Die Anzahl der gesetzten Bits für eine Regel lässt sich nicht so exakt abschätzen. Ist ein Attribut für eine Regel irrelevant, so sind alle Bits für dieses Attribut gesetzt. Ist es relevant, so ist mindestens ein Bit für das Attribut nicht gesetzt. Es kann jedoch auch für ein relevantes Attribut nur ein einziges Bit gesetzt sein. Für ein Attribut mit m Werten kann jede der Zahlen von 1 bis m als Anzahl gesetzter Bits auftreten. Durchschnittlich ergeben sich also $\frac{\sum_{i=1}^{m} m}{m} = \frac{m^2+m}{2m} = \frac{m+1}{2}$ gesetzte Bits. Für $|\mathcal{A}|$ Attribute mit durchschnittlich w Werten ergeben sich im Durchschnitt also insgesamt $|\mathcal{A}|\frac{w+1}{2}$ gesetzte Bits. Um eine Reduzierung des Speicherbedarfs zu erreichen, muss $l > nc$ gelten, also $w|\mathcal{A}| > c|\mathcal{A}|\frac{w+1}{2}$ und damit auch $w > c\frac{w+1}{2}$. Dies lässt sich zu $(2-c)w > c$ umformen.

Für $c > 2$ lässt sich dies zu $w < \frac{c}{2-c}$ auflösen. Da $2 - c$ dann negativ wird, ist dies für kein positives w der Fall, also ist keine Reduzierung möglich.

Falls $c = 2$ ergibt sich $0 > 2$. Auch in diesem Fall ist keine Reduzierung möglich.

Für $c < 2$ erhält man $w > \frac{c}{2-c}$. Da in diesem Falle nur noch $c = 1$ in Frage kommt, erhält man $w > 1$. Eine Reduzierung ist also möglich, wenn Ganzzahlen nur noch 1 Bit lang sind. Damit gibt es aber nur noch zwei mögliche Indizes, der Bitvektor darf also nicht mehr als 2 Bits lang sein. Bei durchschnittlich zwei Werten pro Attribut ist auch nur ein Attribut möglich. Für die Praxis ist dieser Fall also irrelevant.

Eine Sparse-Kodierung bringt für Regeln also im Durchschnitt keinen Vorteil. Zur Datenkodierung lohnt sie sich, wenn die durchschnittliche Attributwertigkeit die Anzahl der Bits zur Speicherung einer Ganzzahl übersteigt. Jedoch muss dann für das Matching ähnlich der Ganzzahlkodierung das durch den Index gegebene Bit der Regel überprüft werden.

Bei der effizienten Teilmengenkodierung ergeben sich kürzere Bitvektoren, bei denen auch mehr als ein Bit pro Attribut gesetzt sein kann. Mit jedem so kodierten Attribut steigt also die durchschnittliche Wertigkeit, ab der eine Sparse-Kodierung eine Reduzierung des Speicherbedarfs zur Folge hat.

3.2.4 Kombinationen der Kodierungen

Die einfache Bitvektorkodierung für Regeln bietet mit der einfachen Bitvektorkodierung für Daten ein sehr effizientes Matching. Sie ist auch weniger speicherintensiv als die Ganzzahlkodierung, solange die durchschnittliche Attributwertigkeit kleiner als die Anzahl der Bits für eine Ganzzahl ist. In diesem Bereich der Attributwertigkeiten stellt sich damit nur die Frage, ob die schlechtere Laufzeit der kompakten Bitvektorkodierung für die bessere Speichereffizienz in Kauf genommen werden soll.

Für höhere durchschnittliche Attributwertigkeiten bleibt die kompakte Kodierung eindeutig diejenige mit dem geringsten Speicherbedarf. Die Ganzzahlkodierung bietet nun eine bessere Speichereffizienz als die einfache Kodierung. Eine Sparse-Darstellung der einfachen Kodierung bietet jedoch die gleiche Speichereffizienz wie die Ganzzahlkodierung. Darüber hinaus entfallen dabei die Additionen des Attributindexes beim Matching. Die Sparse-Kodierung bietet also einen Kompromiss zwischen Speicher- und Laufzeiteffizienz. Optimale Laufzeit bei zunehmender Speicherineffizienz bietet die einfache Kodierung.

Im Falle mengenwertiger Attribute bietet sich die effiziente Teilmengenkodierung für Regeln an, falls die Einschränkung der Hypothesensprache auf Regeln mit Basisaussagen der Form $A \leqq a$ akzeptabel ist. Für die Ganzzahlkodierung wird das Matching noch aufwändiger, da zusätzlich der Teilvektor für das Attribut aus der Regel extrahiert, als Zahl aufgefasst und mit dem Index verglichen werden muss. Außerdem müssen für nicht mengenwertige Attribute mit m Werten 2^m Indizes verwendet werden, da die Binärkodierung einer einelementigen Teilmenge genau ein Bit enthält, also in der Ganzzahldarstellung eine Zweierpotenz liefert. Statt c sind also nur noch $\log_2 c$ Werte pro Attribut möglich.

Für die einfache Bitvektorkodierung müssen nun für mengenwertige Attribute im Bitvektor Dekodierungen und Vergleiche vorgenommen werden. Der große Vorteil, das effiziente Matching, fällt also weg.

Auch für kompakte Bitvektoren ist eine Vorgehensweise wie im Falle der Ganzzahldarstellung notwendig.

Das optimale Matching bietet in diesem Falle die effiziente Teilmengenkodierung, also kompakte Kodierung für mengenwertige und einfache Kodierung für die übrigen Attribute. Hier sind wieder nur zwei logische Operationen auf Bitvektoren notwendig. Der Speicherbedarf ist geringer als bei der einfachen Bitvektorkodierung. Auf mengenwertigen Attributen entspricht er der kompakten Kodierung. Je höher der Anteil an mengenwertigen Attributen ist, desto höher ist die Speichereffizienz.

4 Implementierung

In diesem Kapitel wird die Implementierung der Datenabstraktion und eines darauf aufbauenden, einfachen Separate&Conquer-Lerners beschrieben. Zunächst stellt Abschnitt 4.1 die Funktionalität der einzelnen Komponenten dar. Anschließend wird in Abschnitt 4.2 die Implementierung dieser Funktionalität erläutert.

4.1 Schnittstellen

Um die konkrete Implementierung angebotener Funktionalität austauschbar zu halten, werden Schnittstellen definiert, die die Funktionalität beschreiben. Neben der syntaktischen Definition in Java, die die Signaturen der angebotenen Methoden der implementierenden Klassen beschreibt, ist es meist notwendig, semantische Eigenschaften der Methoden zu fordern. Diese Eigenschaften sichern das korrekte Zusammenspiel verschiedener Implementierungen.

In den folgenden zwei Unterabschnitten werden diese Eigenschaften zusammen mit den Definitionen der Schnittstellen erläutert. Abschnitt 4.1.1 stellt die Schnittstellen zur Datenabstraktion vor, die sich im Wesentlichen aus den in Abschnitt 2.1 definierten Begriffen ableiten, während in Abschnitt 4.1.2 Schnittstellen für die Komponenten des Lerners, die sich aus dem Algorithmus in Abschnitt 2.3 ergeben, definiert werden.

4.1.1 Datenabstraktion

Die Datenabstraktionsschicht bietet die Funktionalität der in Abschnitt 2.1 definierten Begriffe Attribut, Beispiel, Datensatz und Regel. Jedem dieser Begriffe wird dazu eine Schnittstelle zugeordnet. Tabelle 4.1 zeigt die Zuordnung von Begriffen zu Schnittstellen.

Außerdem definiert die Schnittstelle `IDataFactory` die Funktionalität zur Generierung eines Datensatzes aus eingelesenen Daten.

Begriff	Attribut	Beispiel	Datensatz	Regel
Schnittstelle	IAttribute	IExample	IDataSet	IRule

Tabelle 4.1: Zuordnung Schnittstellen zu Grundbegriffen

Schnittstelle IAttribute

Abbildung 4.1 zeigt die Schnittstelle `IAttribute`, die die Funktionalität eines Attributes beschreibt. Nach Definition 2.1 besteht ein Attribut aus einem Namen und einer Domäne. Der Name kann durch die Methode `getName()` gelesen und mit der Methode `setName()` verändert werden. Die Methode `getValues()` erlaubt das Auslesen der Domäne als Feld von Zeichenketten. Über dieses Feld können die Namen der einzelnen Werte, nicht aber ihre Anzahl oder sonstige Struktur geändert werden. Mit `valueCount()` kann direkt die Wertigkeit bestimmt werden.

Wie in Abschnitt 3.1 beschrieben, werden die Werte der Attribute als Zahlen (Ganzzahl oder Bitvektor) kodiert. Die Methoden `getIndex()` und `getValue()` konvertieren zwischen den Darstellungen als Zeichenkette und als Zahl.

Schnittstelle IExample

Ein Beispiel ist eine Instanz, ordnet also jedem Attribut einen Wert zu (vergleiche Definition 2.4). Wie in Abbildung 4.2 zu sehen, ist neben dieser durch `getValue()` realisierten Funktionalität noch weitere Funktionalität für die Verwendung im Lerner definiert.

Mit `checkValue()` lässt sich überprüfen, ob das Beispiel für das angegebene Attribut den gegebenen Wert hat. Diese Methode erlaubt es der konkreten Implementierung, eine effizientere Methode zum Testen auf einen gegebenen Wert zur Verfügung zu stellen als den Wert des Beispiels für das Attribut tatsächlich zu bestimmen. Dies wird durch die Bitvektorkodierung ausgenutzt (siehe Abschnitt 4.2.1).

Den Definitionsbereich des Beispiels liefert `getAttributes()`. Wie in Abschnitt 2.1.1 beschrieben wird eine Reihenfolge der Attribute vorgegeben, daher wird der Definitionsbereich als Liste und nicht als Menge zurückgegeben.

Schnittstelle IDataSet

Datensätze sind nach Definition 2.6 Folgen von Beispielen. Für den Zugriff auf Beispiele definiert die Schnittstelle `IDataSet` (siehe Abbildung 4.3) die Methode `getExamples()`. Mit `getAttributes()` lässt sich der gemeinsame Definitionsbereich der Beispiele als Liste von Attributen abrufen. Auf den Beispielen des Datensatzes muss die Methode `getAttributes()` eine Liste mit identischen Attributen liefern.

Die Identität der Attribute garantiert effiziente Manipulation von Attributen über einen ganzen Datensatz hinweg. Eine Änderung an einem Attribut muss so nicht an alle Beispiele weitergereicht werden.

Mit `getName()` und `setName()` lässt sich der Name des Datensatzes auslesen und verändern. Dieser Name wird zur Identifikation des Datensatzes – unter anderem bei der Ausgabe – verwendet.

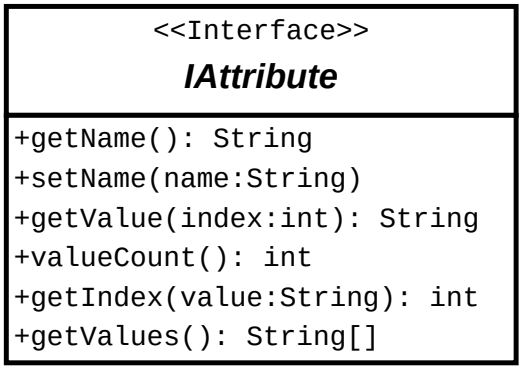


Abbildung 4.1: Die Schnittstelle `IAttribute`

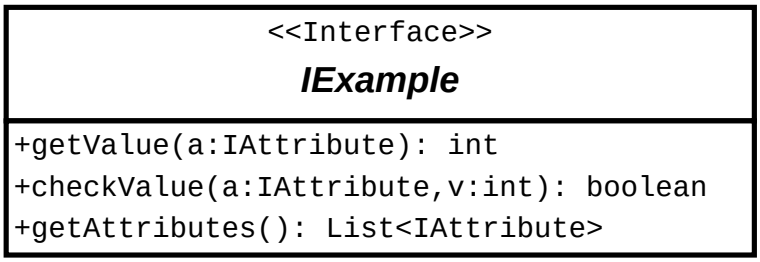


Abbildung 4.2: Die Schnittstelle `IExample`

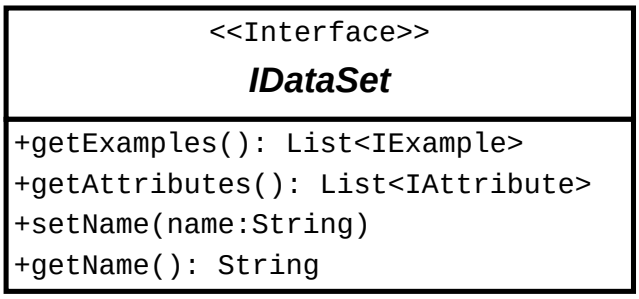


Abbildung 4.3: Die Schnittstelle `IDataSet`

Schnittstelle IDataFactory

Die Schnittstelle `IDataFactory` (siehe Abbildung 4.4) modelliert das Einlesen eines Datensatzes. Bevor mit dem Hinzufügen von Beispielen begonnen werden kann, muss die Angabe der Attribute abgeschlossen sein. Methoden für diese Zustandsverwaltung während der Eingabe stellt die abstrakte Klasse `AbstractDataFactory` zur Verfügung. Diese kann bei Implementierung von `IDataFactory` als Basisklasse verwendet werden.

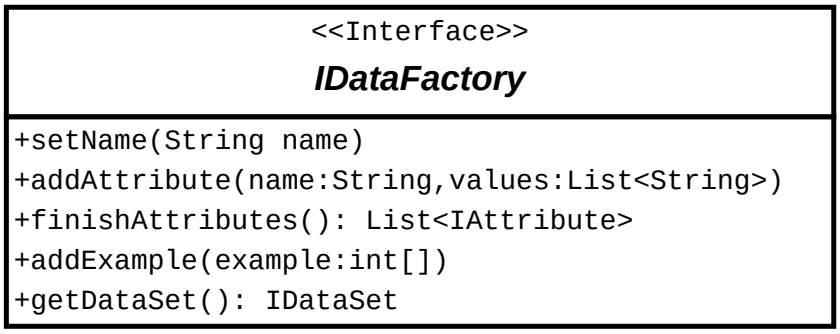


Abbildung 4.4: Die Schnittstelle `IDataFactory`

Zunächst müssen mit `addAttribute()` Attribute hinzugefügt werden. Die Attribute werden dabei als Paar aus Name und Domäne angegeben. Konkrete IAttribute-Objekte müssen von der Implementierung daraus erzeugt werden.

Ist die Angabe von Attributen abgeschlossen, so ist dies durch Aufruf der Methode `finishAttributes()` zu signalisieren. Diese Methode gibt außerdem eine Liste der Attribute zurück. Erst danach kann mit der Angabe von Beispielen durch `addExample()` begonnen werden. Beispiele werden dabei als Feld von Werten angegeben. Die Länge des Feldes muss der Anzahl der Attribute entsprechen. Für die Zuordnung von Attributen zu Werten wird die Reihenfolge der Attribute verwendet. Konkrete `IExample`-Objekte muss die Implementierung erzeugen.

Der Name des Datensatzes kann durch `setName()` gesetzt werden. Sind alle Angaben gemacht, lässt sich der Datensatz mittels `getDataSet()` generieren und zurückgeben.

Schnittstelle IRule

Nach Definition 2.16 besteht eine Regel aus einem Körper und einem Kopf. Die Schnittstelle `IRule` (siehe Abbildung 4.5) bietet jedoch nur die Funktionalität des Körpers an. Der Kopf wird implizit in der Schnittstelle ITheory (siehe Abschnitt 4.1.2) verwaltet.

Wie in Abschnitt 3.1.1 beschrieben, lassen sich Aussagen als Mengen von Attributwerten kodieren. Die Methode `getValues` liefert die zur Basisaussage über das angegebene Attribut gehörende Menge. Mit `setValues()` lässt sich die Basisaussage und damit die Regel verändern.

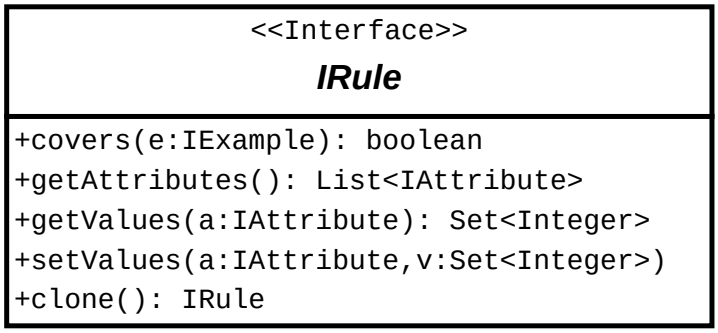


Abbildung 4.5: Die Schnittstelle `IRule`

Die Methode `getAttributes()` liefert den Definitionsbereich der Regel. Mit `covers()` kann geprüft werden, ob das gegebene Beispiel von der Regel abgedeckt wird. Dazu muss das Beispiel den selben Definitionsbereich haben.

Um eine Regel vor Veränderungen zu schützen, kann mit `clone()` eine Kopie angelegt werden, auf der man dann Veränderungen durchführen kann, ohne die ursprüngliche Regel zu beschädigen.

4.1.2 Komponenten des Lerners

Der hier vorgestellte Lerner basiert auf einer Erweiterung des Algorithmus aus Abschnitt 2.3 zur Unterstützung von mehr als zwei Klassen. Diese Erweiterung ist in Algorithmus 4.1 zu sehen. Dabei wird der ursprüngliche Algorithmus für jede Klasse ausgeführt. Jedoch werden die Klassen vorher nach ihrer Häufigkeit im Datensatz aufsteigend sortiert und für jede Klasse nur auf den bisher noch nicht abgedeckten Beispielen gelernt. So werden zunächst die weniger häufigen Klassen durch spezielle Regeln abgedeckt und schließlich für die häufigeren Klassen generellere Regeln gelernt.

Algorithmus 4.1 : Angepasster Separate&Conquer Algorithmus

Eingabe: Beispielmenge *Examples*, Klassenattribut *ClassAttribute*
Theory $\longleftarrow \emptyset$
for *Class* $\in$ `Sort`($D_{ClassAttribute}$) **do**
  while `Positive`(*Examples*, *Class*) $\neq \emptyset$ **do**
    Rule $\longleftarrow$ `FindBestRule`(*Examples*, *Class*)
    if `StoppingCriterion`(*Theory*, *Rule*, *Examples*) **then**
      exit **while**
    Examples $\longleftarrow$ `RemoveCovered`(*Examples*, *Rule*)
    Theory $\longleftarrow$ *Theory* $\cup$ *Rule*
return `PostProcess`(*Theory*)

Außerdem nutzt der Lerner für die Funktion FindBestRule() die Instanziierung gemäß Algorithmus 4.2. Dabei handelt es sich um eine Greedy-Tiefensuche, die nur Verfeinerungen der jeweils besten Verfeinerung betrachtet.

Algorithmus 4.2 : Tiefensuchalgorithmus zum Finden einer konjunktiven Regel

Eingabe: Beispielmenge *Examples*, Klassenattribut *ClassAttribute*, Klasse *Class*
BestRule ⟵ `InitRule`(*Examples*, *ClassAttribute*, *Class*)
BestRuleEvaluation ⟵ `Evaluate`(*BestRule*, *Examples*, *ClassAttribute*, *Class*)
Refinements ⟵ `RefineRule`(*BestRule*)
while *Refinements* $\neq \emptyset$ **do**
  BestRefinement ⟵ $argmax_{r \in Refinements}$`Evaluate`($r$, *Examples*)
  Evaluation ⟵ `Evaluate`(*BestRefinement*, *Examples*, *ClassAttribute*, *Class*)
  if *Evaluation* > *BestRuleEvaluation* **then**
    BestRule ⟵ *BestRefinement*
  Refinements ⟵ `RefineRule`(*BestRefinement*)
return *BestRule*

Daraus ergeben sich für den Lerner zwei Arten benötigter Funktionalität. Zum einen sind die generischen Komponenten StoppingCriterion(), PostProcess(), InitRule(), Evaluate() und RefineRule() zu instanziieren, zum anderen muss die Menge der noch nicht abgedeckten Beispiele verwaltet werden.

Für die Verwaltung der Beispiele bietet sich eine Datenstruktur an, die Methoden für die Funktionen Positive() und RemoveCovered() anbietet. Diese Datenstruktur kann außerdem die für die generischen Komponenten notwendigen Informationen zur Verfügung stellen. Weiterhin bietet sich eine Schnittstelle für das verwalten der Theorie an, vor allem in Hinblick auf den späteren Einsatz zur Klassifikation von Instanzen.

Funktion	**Schnittstelle**
StoppingCriterion()	`IStoppingCriterion`
PostProcess()	`IPostProcessor`
InitRule()	`IRuleFactory`
Evaluate()	`IRuleEvaluator`
RefineRule()	`IRefinementOperator`

Tabelle 4.2: Schnittstellen für generische Funktionen des Algorithmus

Die Zuordnung der generischen Funktionen zu Schnittstellen ist in Tabelle 4.2 dargestellt. Die entsprechenden Schnittstellen verfügen nur über jeweils eine Methode zur Realisierung der entsprechenden Funktionalität. Die einzige Ausnahme stellt hier `IRuleEvaluator` dar. Dieser Schnittstelle wurde eine Variante von `evaluate()` hinzugefügt, die statt einer Regel eine Liste von Regeln akzeptiert und ein Feld von Bewertungen liefert. Diese Variante erlaubt es, mehrfache Iterationen über alle Beispiele zur

Bewertung mehrerer Regeln zu vermeiden. Im Folgenden werden die Methoden dieser Schnittstellen kurz beschrieben.

Zunächst werden die Schnittstellen `IClassificationData` und `ITheory` beschrieben, die die Datenstrukturen bilden, mit denen die generischen Komponenten des Lerners arbeiten. Anschließend werden die den generischen Funktionen entsprechenden Schnittstellen vorgestellt.

Schnittstelle IClassificationData

Die Schnittstelle `IClassificationData` (vergleiche Abbildung 4.6) verwaltet die noch abzudeckenden Beispiele. Daher erweitert sie die Schnittstelle `Iterator<IExample>` und bietet die Methode `iterator()` für die Iteration in Schleifen an. Weiterhin lässt sich mit `totalExampleCount()` die Anzahl aller enthaltenen Beispiele bestimmen.

Den gemeinsamen Definitionsbereich der Beispiele liefert `getAttributes()`. Mit `removeCovered()` lassen sich alle von der gegebenen Regel abgedeckten Beispiele entfernen. Der Definitionsbereich der Regel muss mit dem der Beispiele übereinstimmen, also muss `rule.getAttributes()` eine Liste liefern, die die selben Instanzen der Schnittstelle `IAttribute` wie die von `getAttributes()` gelieferte Liste in der gleichen Reihenfolge enthält.

Im Unterschied zum Datensatz (vergleiche Abschnitt 4.1.1) enthält diese Datenstruktur Klasseninformationen. So lässt sich mit `getClassAttribute()` das Klassenattribut abfragen. Die Methode `getCounts()` liefert die Anzahlen der Beispiele der einzelnen Klassen in einem durch die Klassen indizierten Feld.

Außerdem verwaltet `IClassificationData` die aktuelle Klasse des Lernprozesses.

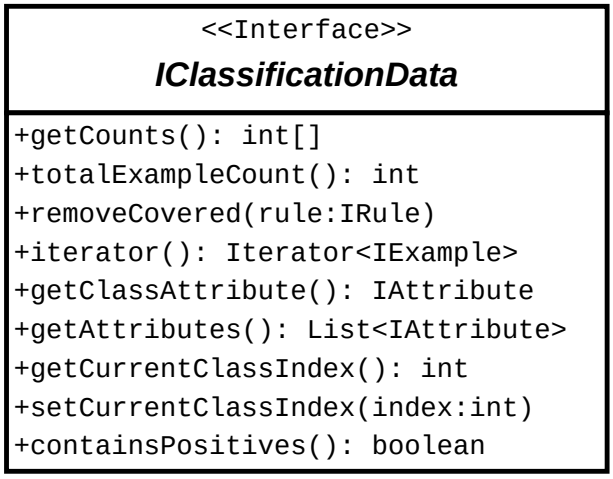


Abbildung 4.6: Die Schnittstelle `IClassificationData`

Durch die Methoden `getCurrentClassIndex()` und `setCurrentClassIndex()` lässt sich die Klasse abfragen und setzen. Mit `containsPositives()` kann dann die Anzahl der Beispiele der aktuellen Klasse ermittelt werden.

Schnittstelle ITheory

Das Ergebnis des Lernprozesses ist eine Theorie, die zur Klassifikation von Instanzen genutzt werden kann. Eine solche Theorie wird durch die in Abbildung 4.7 gezeigte Schnittstelle `ITheory` modelliert. Die Methode `getClassification()` dient der Klassifikation von Instanzen, die ebenfalls als Example kodiert werden können. Das Attribut, nach dem klassifiziert wird, lässt sich mit `getClassAttribute()` bestimmen. Mit `getRulesForClass()` kann die Liste der Regeln für eine gegebene Klasse abgefragt werden. Diese Methode kann auch zur Manipulation der Theorie insbesondere während des Lernprozesses eingesetzt werden.

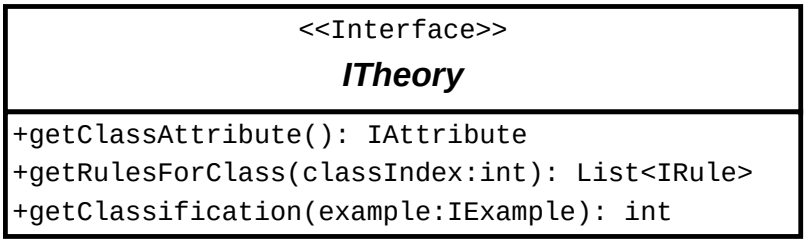


Abbildung 4.7: Die Schnittstelle `ITheory`

Schnittstelle IStoppingCriterion

Die Schnittstelle `IStoppingCriterion` enthält nur die Methode `holds()` (siehe Abbildung 4.8), die angibt, ob die Regel `r` der Theorie `t` hinzugefügt werden oder das Lernen von Regeln für die aktuelle Klasse von `d` beendet werden soll.

Abbildung 4.8: Die Schnittstelle `IStoppingCriterion`

Aufrufe von `t.getClassAttribute()` und `d.getClassAttribute()` müssen die selbe Instanz von IAttribute liefern, der duch `i` bezeichnete Wert muss ein zulässiger Wert für dieses Attribut sein und die Listen `r.getAttributes()` und `d.getAttributes()` müssen die selben Instanzen von `IAttribute` in gleicher Reihenfolge enthalten.

Schnittstelle IPostProcessor

Abbildung 4.9 zeigt die Schnittstelle `IPostProcessor` mit ihrer einzigen Methode `process()`. Diese erwartet die zu bearbeitende Theorie als Parameter vom Typ `ITheory` und führt eine Nachbearbeitung der Theorie durch. In der Regel wird ein sogenanntes Post-Pruning durchgeführt, bei dem Regeln nach bestimmten Kriterien auf ihre Relevanz geprüft und gegebenenfalls entfernt werden.

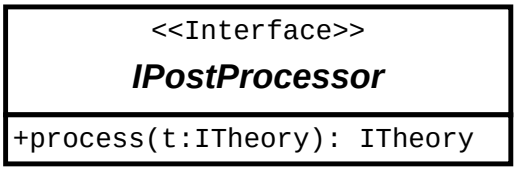


Abbildung 4.9: Die Schnittstelle `IPostProcessor`

Schnittstelle IRuleFactory

Die Methode `getInitRule()` der Schnittstelle `IRuleFactory` Bestimmt aus den gegebenen Beispielen eine Startregel für die Suche (siehe Abbildung 4.10).

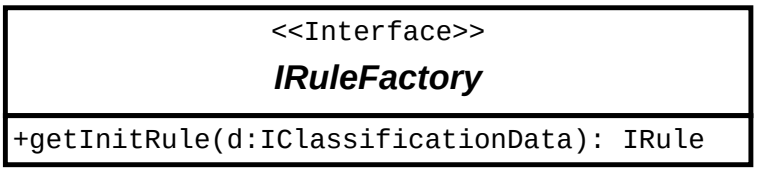


Abbildung 4.10: Die Schnittstelle `IRuleFactory`

Schnittstelle IRuleEvaluator

Zur Bewertung von Regeln bietet die Schnittstelle `IRuleEvaluator` (siehe Abbildung 4.11) zwei Methoden an. Mit `evaluate(IRule r, IClassificationData d)` wird ein Maß

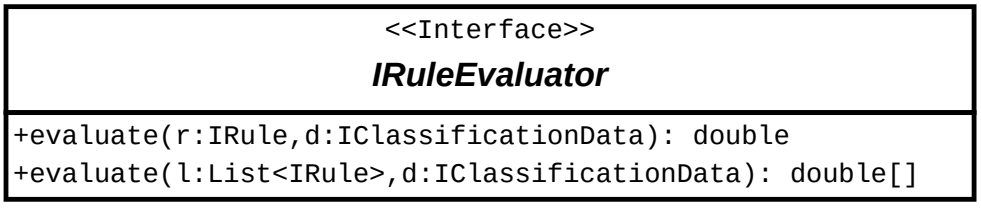


Abbildung 4.11: Die Schnittstelle `IRuleEvaluator`

für die Qualität der Regel `r` auf den Beispielen `d` berechnet. Ein höherer Wert steht für bessere Qualität. Die Listen `r.getAttributes()` und `d.getAttributes()` müssen die selben Instanzen von `IAttribute` in der gleichen Reihenfolge enthalten.

Außerdem ermittelt `evaluate(List<IRule> l, IClassificationData d)` Qualitätsmaße für die Regeln in `l`. Die Maße werden im Feld in der Reihenfolge der Regeln in `l` abgelegt. Diese Methode kann Optimierungen gegenüber einer wiederholten Verwendung der Methode für nur eine Regel implementieren.

Für jede Regel aus `l` muss `getAttributes()` eine Liste liefern, die die selben Attribute wie `d.getAttributes()` in der gleichen Reihenfolge enthält.

Schnittstelle IRefinementOperator

Die Methode `refine()` der Schnittstelle `IRefinementOperator` (siehe Abbildung 4.12) bestimmt eine Menge von Verfeinerungen der Regel `r`. Die Verfeinerungen unterscheiden sich dabei von `r` nicht auf den Attributen in der Menge `c`.

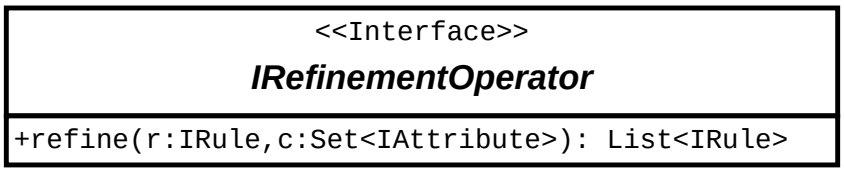


Abbildung 4.12: Die Schnittstelle `IRefinementOperator`

Für alle Regeln in der Liste muss `getAttributes()` eine Liste liefern, die die selben Attribute wie `r.getAttributes()` in gleicher Reihenfolge enthält.

Schnittstelle IRuleFinder

Für die Suche nach einer konjunktiven Regel bietet die Schnittstelle `IRuleFinder` (zu sehen in Abbildung 4.13) die Methode `findBestRule()`. Diese liefert eine aus den Beispielen in `d` gelernte Regel, die sich auf den Attributen in `c` nicht von der durch eine Instanz von `IRuleFactory` gegebenen Startregel unterscheidet.

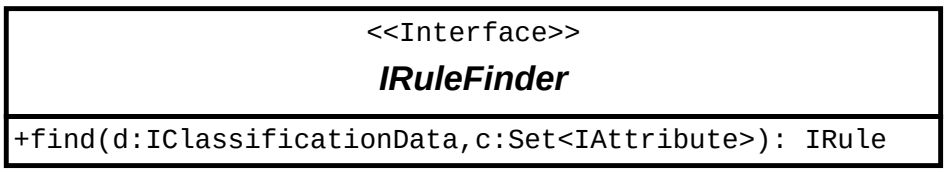


Abbildung 4.13: Die Schnittstelle `IRuleFinder`

4.2 Realisierung

Zu jeder der im vorherigen Abschnitt beschriebenen Schnittstellen für die Datenabstraktion und die Komponenten des Lerners wurden möglichst einfache, erweiterbare und effiziente Implementierungen in Form von Java Klassen geschrieben. Für die Datenabstraktion wurden dabei sowohl die Ganzzahlkodierung als auch die Bitvektorkodierung durch eigene Klassen realisiert. Auf die wichtigsten Details dieser Realiserung geht Abschnitt 4.2.1 ein. Anschließend befasst sich Abschnitt 4.2.2 mit der Realisierung des Lerners.

4.2.1 Datenabstraktion

Die Implementierungen der Schnittstellen `IAttribute`, `IDataSet` sind komplett unabhängig von der gewählten Kodierung der Beispiele. Für die Schnittstelle `IExample` wurde je eine Klasse für Ganzzahl- und Bitvektorkodierung geschrieben. Außerdem gibt es passend zu diesen beiden Klassen Implementierungen von `IDataFactory`, die entsprechende Instanzen der `IExample`-Implementierungen erzeugen. Der generelle Prozess der IDataFactory wurde dabei in der abstrakten Klasse `AbstractDataFactory` implementiert. Die Schnittstelle `IRule` wurde für die in Abschnitt 3.1.1 beschriebene Bitvektorkodierung von Regeln realisiert.

Zunächst folgt eine Beschreibung der gemeinsamen Klassen. In den folgenden zwei Unterabschnitten werden dann die unterschiedlichen Implementationen für die beiden Kodierungen vorgestellt.

Durch die Klasse `SimpleAttribute` wird die Schnittstelle `IAttribute` realisiert. Die Implementierung speichert den Namen des Attributs als Zeichenkette und die Domäne als Feld von Zeichenketten. Die Funktionalität der Schnittstelle ist trivial implementiert, indem einfach der Name, das Feld, die Länge des Feldes oder der Eintrag an einer bestimmten Stelle zurückgegeben werden. Für die Methode `getIndex()` wird das Feld in einer Schleife durchgegangen, bis eine Übereinstimmung gefunden ist. Die Implementierung bietet zwei Konstruktoren an, die jeweils den Namen und die Domäne als Argumente erwarten. Die Domäne kann als Feld oder als `Collection<String>` angegeben werden. Im letzteren Fall wird daraus mit der Methode `toArray()` der Schnittstelle `Collection<String>` ein entsprechendes Feld erzeugt.

Ebenfalls trivial ist die Implementierung von `IDataSet` durch die Klasse `DataSet`. Zur Verwaltung von Attributen und Beispielen werden zwei Listen (`List<IAttribute>` und `List<IExample>`) gespeichert, die bei Aufruf der entsprechenden Methoden zurückgegeben werden. Der Name wird wieder als einfache Zeichenkette gespeichert.

Wie in der Beschreibung der Schnittstelle erläutert, gibt es zwei Zustände für die Dateneingabe durch eine Instanz von `IDataFactory`. Zunächst werden die Attribute mit `addAttribute()` hinzugefügt und anschließend mit `finishAttributes()` zur Eingabe von Beispielen durch `addExample()` umgeschalten.

Die Klasse `AbstractDataFactory` implementiert das generelle Verhalten der Schnittstelle `IDataFactory`. Sie verwaltet den Namen des Datensatzes, Listen der Attribute und Beispiele und den Zustand der Dateneingabe.

Die Methode `setName()` ist trivial, `getDataSet()` gar nicht implementiert. Die Methoden `addAttribute()` und `addExample()` prüfen den Zustand und fügen dann das Attribut beziehungsweise Beispiel der Liste hinzu. Zur Erstellung von Attributen und Beispielen werden dabei abstrakte Methoden verwendet, die in den Unterklassen für die verwendeten Implementierungen angepasst werden müssen. Mit `finishAttributes()` wird der Zustand umgeschaltet und die Liste der Attribute zurückgegeben.

Aufwändiger ist die Implementierung der Schnittstelle `IRule`. Zur Speicherung der eigentlichen Regeldaten verwaltet die Klasse `BitvectorRule` neben einem `BitVector` eine Zuordnung von Attributen zu den jeweiligen Startindizes und eine Liste von Attributen, für die eine unerfüllbare Basisaussage gemacht wird. Außerdem hat jede Instanz von `BitvectorRule` einen Zeiger auf eine Liste der Attribute. Die Klasse verfügt weiterhin über zwei konstante Zeichenketten für die Textrepräsentation der unerfüllbaren und allgemeingültigen Regel.

Die Methoden `clone()` und `getAttributes()` sind sehr einfach implementiert. Erstere konstruiert eine neue Regel mit einer Kopie des Bitvektors und identischer Attributliste, Attributmenge und Indexzuordnung. Die zweite gibt einfach den Verweis auf die Liste zurück.

Sehr ähnlich zueinander sind die Methoden `setValues()` und `getValues()` aufgebaut. In einer Schleife werden alle Werte des entsprechenden Attributs durchlaufen. Vom Startindex des Attributs im Bitvektor ausgehend werden nacheinander die Bits durchgegangen. In `getValues()` wird bei einem gesetzten Bit der aktuelle Wert des Attributs an die verkettete Liste angehängt, die nach der Schleife zurückgegeben wird.

Für `setValues()` wird das entsprechende Bit gesetzt oder gelöscht, je nachdem, ob der aktuelle Wert des Attributs in der übergebenen Menge von Werten enthalten ist. Ist die übergebene Menge von Werten leer, so entsteht dadurch die unerfüllbare Regel, da für ein Attribut die Basisaussage $A \subseteq \emptyset$ gefordert wird, die nach Satz 2.1 für keine Instanz gelten kann. Daher wird in diesem Fall das Attribut der Menge von Attributen mit unerfüllbaren Basisaussagen hinzugefügt. Andernfalls wird das Attribut aus dieser Menge entfernt.

Um in der Methode `covers()` die konkrete Kodierung des Beispiels ausnutzen zu können, wurden zwei Implementierungen der Methode für die jeweiligen Klassen zur Beispielkodierung vorgenommen. Die Methode stellt daher fest, welche Beispielkodierung vorliegt und ruft die entsprechende Implementierung auf, sofern die Regel überhaupt Instanzen abdecken kann. Bei der Ergänzung von Beispielkodierungen ist also eine Änderung an der Regelklasse notwendig. Dies wurde in Kauf genommen, um nicht für jede Datenkodierung eine eigene Regelklasse schreiben zu müssen, die sich alle nur in dieser Methode unterscheiden. Die beiden Methoden werden mit den Implementierungen der jeweiligen Datenkodierung beschrieben.

Bitvektorkodierung

```
public class BitVectorExample implements IExample {

private final BitVector data;
private final Map<IAttribute, Integer> startIndexes;
private final List<IAttribute> attributes;

public int getValue(IAttribute attribute) {
  final int start = startIndexes.get(attribute);
  final int end = start+attribute.valueCount()-1;
  return data.indexOfFromTo(start, end, true)-start;
}

public void setValue(IAttribute attribute, int value) {
  final int start = startIndexes.get(attribute);
  final int end = startIndex+attribute.valueCount()-1;
  data.replaceFromToWith(startIndex, endIndex, false);
  if (value >= 0)
    data.set(startIndex + value);
}

public boolean checkValue(IAttribute attribute, int value) {
  if (value < 0)
    return getValue(attribute) < 0;
  final int index = startIndexes.get(attribute)+value;
  return data.get(index);
}}
```

Listing 4.1: Die Klasse `BitVectorExample` (gekürzt um triviale Methoden)

Listing 4.1 zeigt die Klasse `BitVectorExample`, gekürzt um den Konstruktor und die trivialen Methoden `getData()` und `getAttributes()`. Diese Klasse implementiert `IExample` unter Verwendung der Bitvektorkodierung. Die Methode `getData()` ist nicht Teil der Schnittstelle `IExample`, sondern dient dem direkten Zugriff auf den Bitvektor.

Neben dem Bitvektor und der obligatorischen Liste der Attribute wird eine Zuordnung von Attributen zu Indizes verwaltet. Diese wird verwendet, um auf die Bits für einzelne Attribute zuzugreifen. Gespeichert ist dort der erste Index für die Bits des jeweiligen Attributs. Zur Bestimmung des letzten Bits wird darauf die Anzahl der Attributwerte addiert und anschließend eins abgezogen. Die verwendete Bitvektorko-

dierung arbeitet mit inklusiven Indizes, so dass immer das letzte Bit und nicht das imaginäre Bit dahinter angegeben werden muss.

In der Methode `getValue()` wird durch einen Aufruf von `indexOfFromTo()` das erste gesetzte Bit für das Attribut bestimmt. Ist kein Bit gesetzt, also kein Wert vorhanden, gibt letztere Methode den Wert -1 zurück, was auch bei `getValue()` zu einem negativen Ergebnis führt.

Um den Wert für ein Attribut zu setzen, werden zunächst mit `replaceFromToWith()` alle Bits des Attributs gelöscht und anschließend das Bit für den gewünschten Wert mit `set()` gesetzt. Gerade bei Attributen mit vielen Werten kann `replaceFromToWidth` einige Optimierungen vornehmen, die in einer einfachen Iteration über alle Bits des Attributs nicht möglich sind.

Für die Überprüfung auf einen konkreten Wert mittels `checkValue()` wird nur gezielt das Bit für den entsprechenden Wert des gegebenen Attributs mit `get()` abgefragt. Soll auf einen fehlenden Wert überprüft werden, ist dies so nicht möglich und es wird durch `getValue()` zunächst der tatsächliche Wert für das Attribut festgestellt.

Die Klasse `BitvectorDataFactory` erbt von `AbstractDataFactory` und implementiert somit `IDataFactory` für die Bitvektorkodierung. Die Methoden `getDataSet()` und `createAttribute()` sind trivial durch Konstruktoraufrufe realisiert. Zur Konstruktion von Beispielen wird in `createExample()` zunächst auf Basis eines leeren Bitvektors eine Instanz von `BitVectorExample()` erzeugt und anschließend in einer Schleife über die gegebenen Attribute für jedes Attribut der entsprechende Wert gesetzt. Die Methode `getAttributes()` ist so überschrieben, dass zunächst eine Zuordnung von Attributen zu Startindizes vorgenommen und anschließend die entsprechende Methode der Basisklasse aufgerufen wird. Beim Aufbau der Zuordnung in Form einer `HashMap<IAttribute, Integer>` wird außerdem die Anzahl der Bits für den in `createExample()` erzeugten Bitvektor festgestellt.

In Listing 4.2 ist die Methode `covers()` für die Bitvektorkodierung dargestellt. Die beiden Bitvektoren werden dabei in Blöcken von 64 Bit verglichen. Der eigentliche Vergleich entspricht dem in Abschnitt 3.1.3 vorgestellten Verfahren. Wird in einem der Blöcke festgestellt, dass die Regel das Beispiel nicht abdeckt, wird der Vergleich abgebrochen und sofort das Ergebnis geliefert.

Ganzzahlkodierung

Die Klasse `IntegerExample` implementiert die Schnittstelle `IExample` für die Ganzzahlkodierung. Bis auf den trivialen Konstruktor und die triviale Methode `getAttributes()` ist sie in Listing 4.3 zu sehen.

Wie bei der Implementierung für die Bitvektorkodierung wird eine Liste von Attributen und eine Zuordnung von Attributen zu Indizes verwaltet. Die Daten selbst werden in einem Feld von Ganzzahlen gespeichert. Die `Map<IAttribute, Integer>` speichert, an welcher Position im Feld der Wert des entsprechenden Attributs zu finden ist.

```
public boolean covers(BitVectorExample example) {
  BitVector data = example.getData();
  final long[] other = data.elements();
  final long[] mine = rule.elements();
  int index = rule.size();
  if (index != data.size())
    return false;
  final long part = ~(0xffffffffffffffffL << (index & 0x3f));
  index = index >>> 6;
  if (part != 0 &&
  (((other[index] & mine[index]) ^ other[index]) & part) != 0)
    return false;
  while (index-- > 0)
    if (((other[index] & mine[index]) ^ other[index]) != 0)
      return false;
  return true;
}
```

Listing 4.2: Die Methode `BitvectorRule.covers(BitVectorExample)`

```
public class IntegerExample implements IExample {

  private final int[] data;
  private final Map<IAttribute, Integer> indexes;
  private final List<IAttribute> attributes;

  public int getValue(IAttribute attribute) {
    int index = indexes.get(attribute);
    return data[index];
  }

  public void setValue(IAttribute attribute, int value) {
    data[indexes.get(attribute)] = value;
  }

  public boolean checkValue(IAttribute attribute, int value) {
    return data[indexes.get(attribute)] == value;
  }
}
```

Listing 4.3: Die Klasse `IntegerExample` (gekürzt um triviale Methoden)

Die übrigen Methoden sind sich sehr ähnlich. Der Index für das gegebene Attribut wird nachgeschlagen und anschließend der Zugriff auf das Feld durchgeführt.

Ähnlich der Klasse `BitvectorDataFactory` ist auch die Klasse `IntegerDataFactory` aufgebaut, die ebenfalls von `AbstractDataFactory` erbt und somit die Implementierung der Schnittstelle `IDataFactory` für die Ganzzahlkodierung darstellt.

Auch hier sind die Methoden `createAttribute()` und `getDataSet()` trivial durch Konstruktoraufrufe realisiert. Zur Konstruktion von Beispielen wird in `createExample()` der Konstruktor von `IntegerExample` mit einer Kopie des gegebenen Feldes aufgerufen. Die Methode `getAttributes()` ist so überschrieben, dass zunächst eine Zuordnung von Attributen zu Indizes vorgenommen und anschließend die entsprechende Methode der Basisklasse aufgerufen wird.

```
public boolean covers(IntegerExample example) {
  for (IAttribute attribute : example.getAttributes()) {
    int value = example.getValue(attribute);
    if (value >= 0 &&
    !rule.get(startIndexes.get(attribute) + value))
      return false;
  return true;
}
```

Listing 4.4: Die Methode `BitvectorRule.covers(IntegerExample)`

Listing 4.4 zeigt die Methode `covers()` der Ganzzahlkodierung. Für jedes Attribut wird geprüft, ob der Wert des Beispiels in der Menge der Regel enthalten ist.

4.2.2 Komponenten des Lerners

Die generischen Komponeneten des Lerners sind größtenteils trivial implementiert. So sind die Schnittstellen `IStoppingCriterion` und `IPostProcessor` durch die Klassen `NoStoppingCriterion` und `NoPostProcessing` so implementiert, dass das Abbruchkriterium nie erfüllt ist und die Theorie unverändert zurückgegeben wird. Für die Schnittstelle `IRuleEvaluator` wurde mit der Klasse `Accuracy` eine Abschätzung der Genauigkeit durch die Differenz von korrekt abgedeckten positiven und fälschlich abgedeckten negativen Beispielen gewählt. Die Klasse `BitvectorRuleFactory` implementiert `IRuleFactory` und liefert eine Regel, die alle Instanzen der gegebenen Attributmenge abdeckt, indem sie eine Regel aus einem Bitvektor voller Einsen konstruiert.

Auch die Implementierung von `ITheory` durch `Theory` ist weitestgehend trivial. Es wird das Klassenattribut, die Klassen als Feld von Ganzzahlen und die Regeln für die einzelnen Klassen als Liste von Listen von Regeln verwaltet. Der Konstruktor erzeugt für jede Klasse eine leere Liste von Regeln. Die Liste von Listen ist da-

bei eine `ArrayList`, um einfachen, wahlfreien Zugriff auf die Listen für die einzelnen Klassen zu erlauben. Die Listen für die einelnen Klassen sind wiederum Instanzen von `LinkedList` und damit für Manipulationen (Hinzufügen und Entfernen) bestens geeignet. Die Implementierung von `getClassAttribute()` ist trivial, die von `getRulesForClass()` sehr einfach, da nur die entsprechende Liste anhand des Index aus der `ArrayList` ausgewählt wird. Für die Bestimmung der Klassifikation eines Beispiels mittels `getClassification()` wird für jede Klasse die Liste der Regeln durchgegangen, bis eine Übereinstimmung gefunden ist. Dann wird die aktuelle Klasse zurückgegeben.

Etwas aufwändiger ist die Implementierung von `IClassificationData` durch die Klasse `ClassificationData`. Jede Instanz verwaltet eine Liste der Attribute, eine Liste der Beispiele, das Klassenattribut, ein Feld von Anzahlen der Beispiele jeder Klasse und den Index der aktuellen Klasse. Mit `setCurrentClassIndex()` wird dieser Index gesetzt und über `getCurrentClassIndex()` zurückgegeben. Ebenfalls trivial sind die Methoden `getCounts()`, `getAttributes()`, `getClassAttribute()`, `containsPositives()` und `totalExampleCount()` implementiert. Der Konstruktor erzeugt eine neue verkettete Liste von Beispielen, um diese effizienzer manipulieren zu können. Außerdem werden dabei die Anzahlen der Beispiele für die einzelnen Klassen festgestellt. In `removeCovered()` wird mittels des von `iterator()` gelieferten Iterators über die Beispiele iteriert und im Falle, dass die Regel das Beispiel abdeckt, die Methode `remove()` des Iterators aufgerufen. Die Hauptarbeit steckt also im Iterator. Dieser verwaltet einen Iterator der unterliegenden Liste und einen Zeiger auf das zuletzt zurückgegebene Beispiel. Beim Entfernen von Beispielen stellt der Iterator die Klasse des entfernten Beispiels fest und reduziert die Anzahl für die entsprechende Klasse um eins.

5 Ergebnisse

Mit der in Kapitel 4 vorgestellten Implementierung einer konkreten SeCo-Instanz ist die prinzipielle Eignung der Umgebung für die Entwicklung eines Regel-Lerners erwiesen. Der implementierte Lerner ist stark an das SeCo-Framework angelehnt, so dass eine Portierung des SeCo-Framework von Weka auf die neue Umgebung ohne größere Probleme erfolgen kann. Dies ist jedoch nicht Teil dieser Arbeit.

Das Ziel dieser Arbeit ist, eine Umgebung als Grundlage für einen Lerner zu schaffen, die effizient und erweiterbar ist. Inwieweit diese Ziele von der Implementierung erreicht werden, wird in den Abschnitten dieses Kapitels erörtert.

5.1 Effizienz der Kodierungen

Zunächst stellt sich die Frage, welche der beiden implementierten Kodierungen für welche Datensätze effizienter bezüglich Speicherbedarf und Laufzeit sind. Um die verschiedenen Implementierungen im praktischen Einsatz bezüglich ihrer Effizienz zu vergleichen, wurden Messungen zum Speicherbedarf und zur Laufzeit des Lerners durchgeführt. Um die Ergebnisse dieser Messungen einschätzen zu können, folgen zunächst jedoch einige theoretische Überlegungen.

5.1.1 Theoretische Überlegungen zur Effizienz

Die Implementierungen für beide Datenkodierungen unterscheiden sich nur in der verwendeten Realisierungen von `IDataFactory` und `IExample` sowie der dadurch ausgewählten Implementierung der Methode `covers()`. Ersteres wirkt sich auf den Speicherbedarf und letzteres auf die Laufzeit aus. Um die Messungen sinnvoller beurteilen zu können ist es daher hilfreich, die Auswirkung dieser Unterschiede zunächst durch Berechnungen zu Speicherbedarf und Laufzeit abzuschätzen.

Als Grundlage für die Berechnungen wird eine Menge $\mathcal{A}$ von Attributen angenommen. Mit a wird die Anzahl der Attribute bezeichnet ($a := |\mathcal{A}|$), mit w die durchschnittliche Wertigkeit eines Attributs ($w := \frac{\sum_{A \in \mathcal{A}} |D_A|}{|\mathcal{A}|}$). Außerdem bezeichnet n die Anzahl der Beispiele im Datensatz.

Typ	Speicherbedarf
int	4 Bytes
long	8 Bytes
double	8 Bytes
`int[]`	12 + 4×Länge Bytes
`long[]`	12 + 8×Länge Bytes
`double[]`	12 + 8×Länge Bytes
Objekte	8 Bytes + Speicherbedarf der Felder

Tabelle 5.1: Speicherbedarf von Java-Typen

Speicherbedarf

Zur Berechnung des Speicherbedarfs sind einige Annahmen über den Speicherverbrauch der grundlegenden Typen von Java notwendig. Der tatsächliche Speicherverbrauch kann jedoch wegen unterschiedlicher Implementierungen der virtuellen Maschine variieren. Aufgrund von Experimenten in (Roubtsov, 2002), die auf der Testumgebung zu den gleichen Ergebnissen führten, liegen den folgenden Berechnungen die Annahmen aus Tabelle 5.1 zugrunde. Um die Rechnung zu vereinfachen und von der konkreten Java-Implementierung unabhängige Ergebnisse zu erzielen, wird im Folgenden die Ausrichtung an 8-Byte-Blöcken ignoriert.

Die Klasse `IntegerExample` (siehe Abschnitt 4.2.1) speichert die Daten in einem Feld `int[]` und braucht dafür $(12 + 4a)n$ Bytes.

In `BitVectorExample` (siehe Abschnitt 4.2.1) wird ein BitVector mit drei Feldern verwendet: ein `int`, ein `long` und ein `long[]`. Entscheidend ist also die Länge des `long[]`, welche sich aus der Länge l des repräsentierten Bitvektors als $\lceil l/64 \rceil$ ergibt. Die Länge des Bitvektors hängt wiederum von der gewählten Kodierung ab.

Für jedes Attribut verwendet die simple Kodierung ein Bit pro Wert (vgl. Abschnitt 3.1.3). Die Länge des Bitvektors ergibt sich also als Summe der Wertigkeiten der Attribute $\sum_{A \in \mathcal{A}} |D_A|$. Dies entspricht dem Produkt aw. Damit braucht diese Kodierung $(24 + 8\lceil \frac{aw}{64} \rceil)n$ Bytes.

Der Speicherbedarf hängt also von den Anzahlen der Attribute und Beispiele und den Wertigkeiten der Attribute ab. Während beide Implementierungen in den Anzahlen der Attribute und Beispiele linearen Speicherbedarf haben, unterscheiden sie sich stark in der Abhängigkeit von der Wertigkeit der Attribute: Für `IntegerExample` ist der Bedarf konstant, für die simple Kodierung verhält er sich linear in der Wertigkeit der Attribute. Abbildung 5.1 zeigt die Abhängigkeit von der Anzahl der Attribute für feste Wertigkeiten, Abbildung 5.2 stellt die Abhängigkeit von der durchschnittlichen Wertigkeit für feste Attributanzahlen dar.

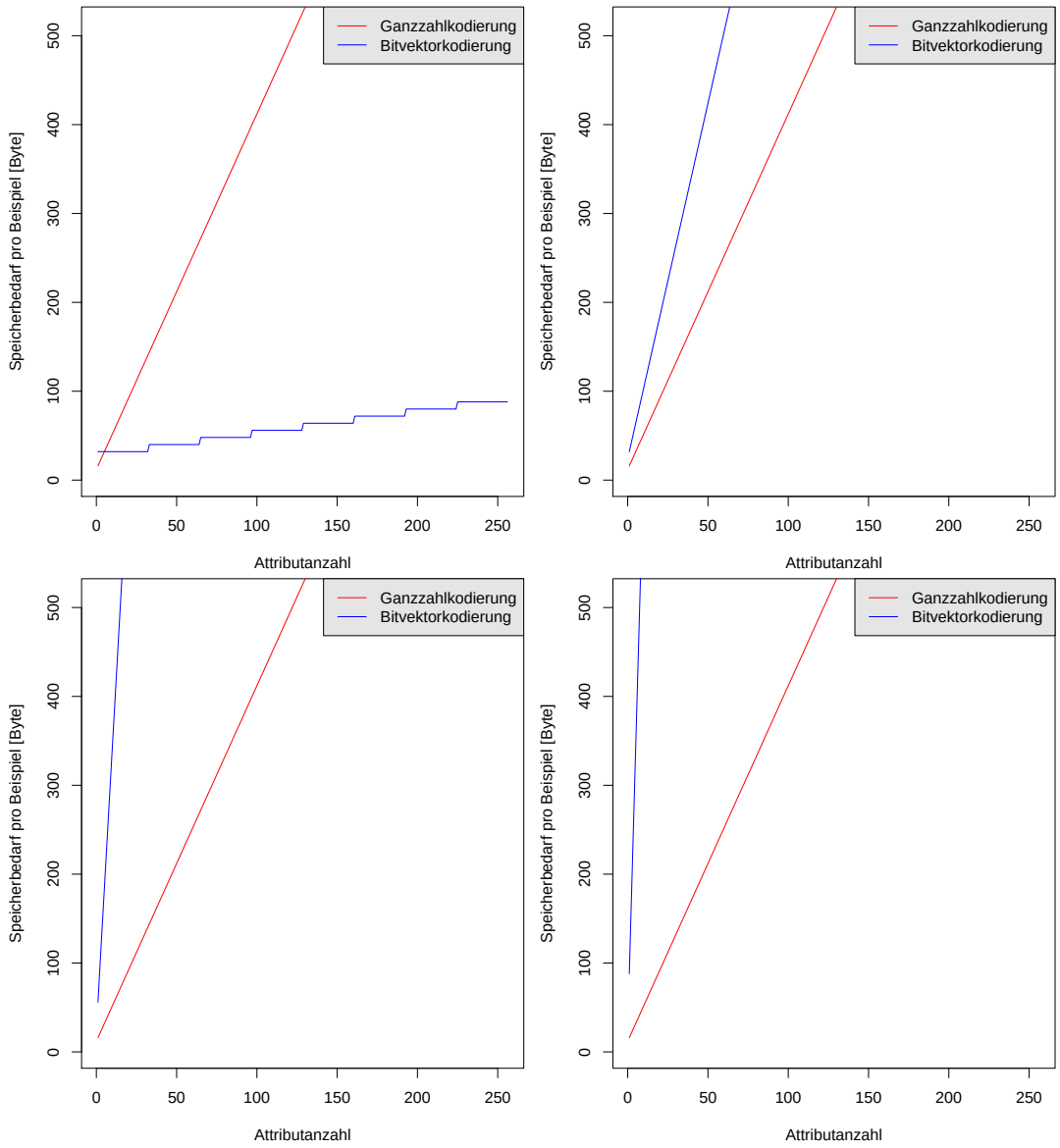


Abbildung 5.1: Speicher abhängig von der Attributanzahl für durchschnittlich 2 (oben links), 64 (oben rechts), 256 (unten links) und 512 (unten rechts) Werte

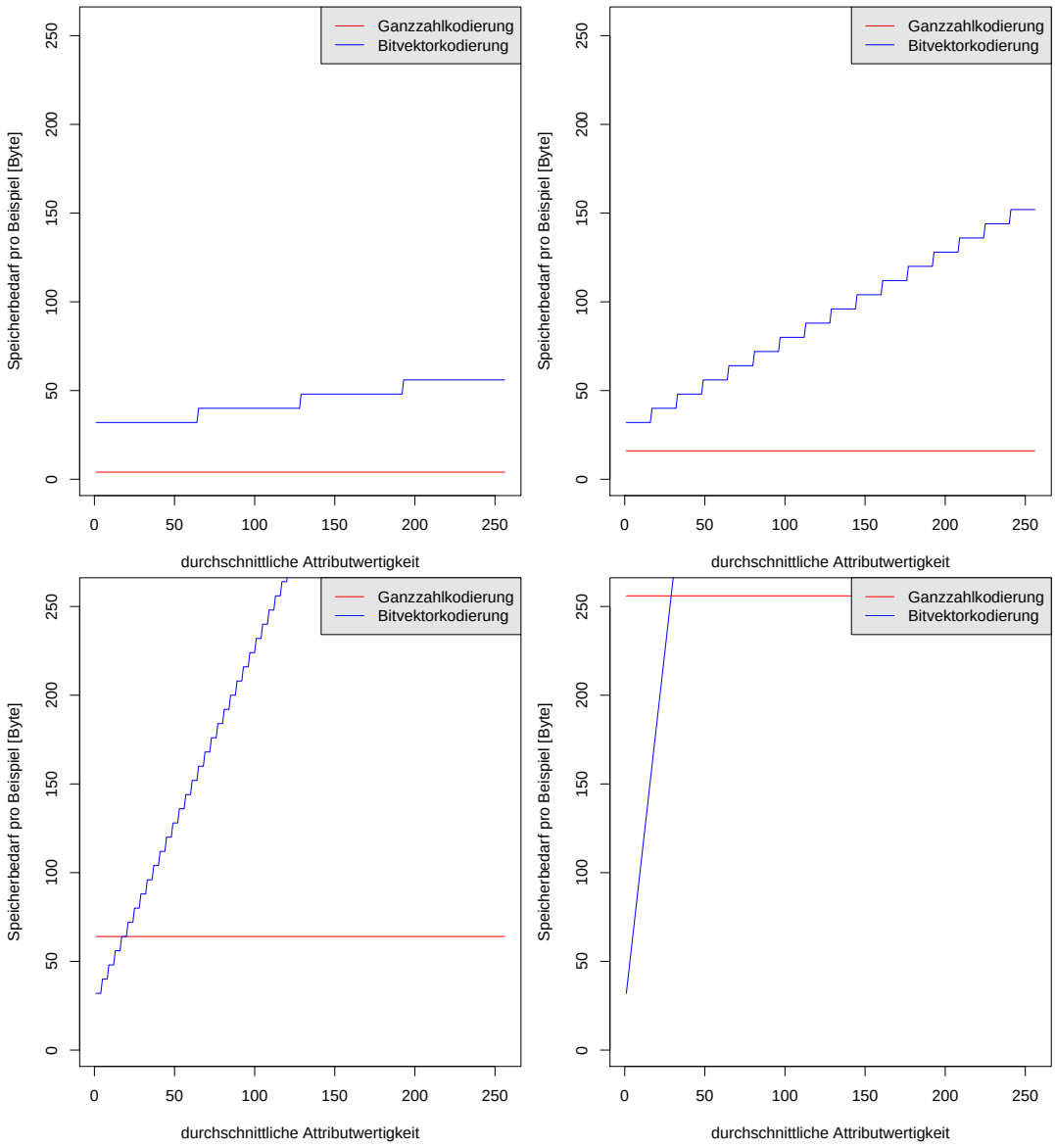


Abbildung 5.2: Speicher über durchschnittlicher Attributwertigkeit für 1 (oben links), 4 (oben rechts), 16 (unten links) und 64 (unten rechts) Attribute.

Laufzeit

Der Unterschied in den Laufzeiten des Lerners bei Verwendung der unterschiedlichen Kodierungen lässt sich auf die verschiedenen Implementierungen der Methode `covers()` zurückführen. Listing 5.1 zeigt dazu die wesentlichen Methoden aus der Implementierung des verwendeten Algorithmus.

In `learnTheory` wird durch wiederholtes Aufrufen von `findBestRule()` für jede Klasse eine Menge konjunktiver Regeln gelernt. Die Anzahl der Regeln für jede Klasse hängt von der den Daten zugrunde liegenden Klassifikation ab. Da jedoch jede gelernte Regel mindestens ein Beispiel abdeckt, ergibt sich n als obere Schranke. In dieser Methode wiederum werden in einer Tiefensuche verschiedene Regeln auf der Menge der noch nicht abgedeckten Beispiele evaluiert. Dazu wird festgestellt, welche dieser Beispiele von der neuen Regel abgedeckt werden. Pro Aufruf der Methode `evaluate()` ergeben sich also höchstens n Aufrufe der Methode `covers()`. Aufgrund des gewählten Verfeinerungsoperators ist a die maximale Tiefe der Suche. Pro Suchschritt werden maximal aw Aufrufe von `evaluate()` durchgeführt. Die Gesamtlaufzeit ist also höchstens $a^2wn^2 \cdot f_{\text{covers}}$. Der Term f_{covers} unterscheidet sich dabei für die verschiedenen Implementierungen.

Wie in Abschnitt 4.2.1 beschrieben, führt die Methode `covers` der Bitvektorkodierung prinzipiell ein bitweises Und gefolgt von einem bitweisen exklusiven Oder durch. Da lange Bitvektoren nur in Blöcken entsprechend der Wortbreite der verwendeten Maschine verarbeitet werden können, wird sowohl das Und als auch das exklusive Oder wortweise durchgeführt. Unabhängig von der konkret verwendeten Maschine realisiert die verwendete Bitvektorimplementierung einen Bitvektor als Feld von `long` und führt alle logischen Operationen auf die entsprechenden bitweisen Operationen für `long` in Java zurück. Bei einer Wortbreite von 64 Bit –wie auf dem Testsystem und heute üblichen Rechnern– entspricht das genau den wortweisen logischen Operationen.

Für jeden solchen Block werden also zwei logische Operationen ausgeführt. Die Anzahl der Blöcke ergibt sich aus der Länge l des Bitvektors als $\lceil l/64 \rceil$. Die Länge wiederum entspricht dem Produkt aus durchschnittlicher Attributwertigkeit und Anzahl der Attribute. Die Methode `covers()` für die Bitvektorkodierung hat also eine Laufzeit von $f_{\text{covers}_{BV}} = 2\lceil aw/64 \rceil$.

In der Implementierung für die Ganzzahlkodierung (siehe Abschnitt 4.2.1) wird für jedes Attribut ein Bit an einem aus dem Wert des Beispiels für das Attribut berechneten Index getestet. Für die Berechnung des Index ist eine Addition sowie ein Zugriff auf eine Tabelle notwendig. Der Test des Bits benötigt vier logische Operationen und einen Feldzugriff. Die Laufzeit der Methode liegt also bei $f_{\text{covers}_{Int}} = 7a$.

Damit ergibt sich eine Gesamtlaufzeit von höchstens $7a^3wn^2$ für die Ganzzahl- und $a^3w^2n^2/32$ für die Bitvektorkodierung. In der Anzahl der Attribute ergibt sich für beide Kodierungen ein kubisches Verhalten der Laufzeit, bei der Ganzzahlkodierung jedoch mit größerem konstanten Faktor (ca. 7) als in der Bitvektorkodierung (ca. $1/32$).

```
public ITheory learnTheory(IDataSet ds, IAttribute ca) {
  initState(ds, ca); List<IRule> rules;
  Theory theory = new Theory(classAttribute, classes);
  for (int i = 0; i < classes.length; i++) {
    uncovered.setCurrentClassIndex(classes[i]);
    rules = theory.getRulesForClass(classes[i]);
    while (uncovered.containsPositives()) {
      IRule rule = finder.findBestRule(uncovered, ignored);
      if (rules.contains(rule))
        break;
      uncovered.removeCovered(rule);
      rules.add(rule);
    }
  }
  return postProcessor.process(theory);
}

public IRule findBestRule(IClassificationData uncovered,
Set<IAttribute> ignored) {
  IRule rule = ruleFactory.getInitRule(uncovered);
  double best = evaluator.evaluate(rule, uncovered);
  List<IRule> refinements = operator.refine(rule, ignored);
  double[] values; int index; IRule bestRefinement;
  while (! refinements.isEmpty()) {
    double maxEvaluation = Double.NEGATIVE_INFINITY;
    bestRefinement = null; index = 0;
    values = evaluator.evaluate(refinements, uncovered);
    for (IRule refinement : refinements) {
      if (values[index] > maxEvaluation) {
        bestRefinement = refinement;
        maxEvaluation = values[index];
      }
      index++;
    }
    if (maxEvaluation > best) {
      best = maxEvaluation; rule = bestRefinement;
    }
    refinements = operator.refine(bestRefinement, ignored);
  }
  return rule;
}
```

Listing 5.1: Implementierung des konkreten SeCo-Lerners

Andererseits geht die durchschnittliche Attributwertigkeit in der Ganzzahlvariante nur linear in die Berechnung der Laufzeit ein, während die Zeit für die Bitvektorvariante quadratisch mit der Wertigkeit der Attribut steigt. Die Anzahl der Beispiele geht in beiden Fällen quadratisch in die Lauftzeit ein.

5.1.2 Messungen zur Effizienz

Alle Messungen wurden unter Suns Java in der 64-Bit Server VM Version `1.5.0_07-b03` auf einem Dual Opteron-System durchgeführt. Die JVM wurde dabei mit Standardparametern gestartet.

Zur Messung wurde auf verschiedenen Datensätzen mit dem in Kapitel 4 vorgestellten Lerner unter Verwendung der jeweiligen Datenkodierungen eine Theorie gelernt und dabei der Speicherbedarf beziehungsweise die Laufzeit gemessen. Für diese Messungen wurden von der Weka-Seite (`http://www.cs.waikato.ac.nz/ml/weka/`) Datensätze zur Regression und zur Klassifikation verwendet. Diese wurden durch den Filter `weka.filters.unsupervised.attribute.Discretize` mit Standardparametern (Intervalle gleicher Breite mit maximal 10 Intervallen pro Attribut) diskretisiert. Eine detaillierte Liste dieser Datensätze findet sich in Anhang A.

Die in den folgenden Kapiteln vorgestellten Programme zur Erfassung von Speicherbedarf beziehungsweise Laufzeit für einen Datensatz wurden von einem Skript für jeden einzelnen der verbleibenden Datensätze aufgerufen.

Weil diese Datensätze teilweise sehr unterschiedliche Anzahlen an Attributen und Beispielen aufweisen und sich auch in der Attributwertigkeit deutlich unterscheiden, ist damit nur ein tendenzieller Vergleich möglich. Wie in Tabelle 5.2 zu sehen ist, gibt es einzelne, große Datensätze, während der Großteil weniger als 100 Beispiele aufweist (vergleiche auch Anhang A. Die Auswirkung der Anzahl der Beispiele läßt sich aufgrund dieser Daten also nur schlecht bestimmen.

Minimum	**Median**	**Durchschnitt**	**Maximum**
4.0	96.0	832.8	138200.0

Tabelle 5.2: statistische Werte zu den Beispielanzahlen der Datensätze

Um den Einfluss von Attributanzahl, Attributwertigkeit und Anzahl der Beispiele genauer zu untersuchen, wurde der Datensatz CoverType mit 581012 Beispielen und 54 Attributen als Grundlage für die Erzeugung von zueinander ähnlichen Datensätzen, die sich nur in Anzahl der Beispiele, Anzahl der Attribute oder Attributwertigkeit unterscheiden, verwendet. Auch auf diese Datensätze wurden die im Folgenden beschriebenen Methoden zur Ermittlung von Speicherbedarf und Laufzeit angewandt.

Messungen zum Speicherbedarf

Die in Kapitel 4 vorgestellten Implementierungen unterscheiden sich aufgrund ihrer unterschiedlichen Kodierungen deutlich im Speicherbedarf. Da die Verwaltungsinformationen pro Datensatz nur einmal anfallen und sich zwischen den Implementierungen nicht unterscheiden, wird der Speicherbedarf eines einzelnen Beispiels zum Vergleich der Implementierungen betrachtet.

```
class MemoryMonitor extends Thread {

  private long sleepTime;
  private MemoryMXBean memBean;
  private LearnerState state;
  private double avg_int, avg_bv;
  private long count_int, count_bv;

  public void run() {
    while (true) {
      long usage = memBean.getHeapMemoryUsage().getUsed();
      switch (state) {
        case OFF:
          break;
        case BV:
          count_bv++;
          avg_bv += ((usage - avg_bv) / count_bv);
          break;
        case INT:
          count_int++;
          avg_int += ((usage - avg_int) / count_int);
          break;
      }
      try {
        sleep(sleepTime);
      } catch (InterruptedException e) {
        break;
      }}}}
```

Listing 5.2: Thread für Speichermessung in Java (ohne triviale Methoden)

Aufgrund der dynamischen Speicherverwaltung in Java sind Messungen des Speicherbedarfs nicht zuverlässig möglich. Der von der JVM reservierte Speicher lässt nur sehr

ungenau auf den tatsächlich vom Prozess benötigten Speicher schließen, da die JVM zu Beginn einen Speicherbereich fester Größe reserviert und anschließend bei Bedarf um weitere Blöcke ebenfalls fester Größe erweitert.

Objekte, auf die nicht mehr zugegriffen werden kann, werden regelmäßig durch einen Garbage-Collector entfernt, so dass der von ihnen belegte Speicher wieder freigegeben werden kann. Jedoch muss dies nicht zu einer Reduzierung des von der JVM reservierten Speichers führen.

Seit Version 1.5 bietet Java Schnittstellen, über die Details zu Informationen der JVM abgefragt werden können (`java.lang.management`). Unter anderem kann darüber auch der aktuell durch Objekte belegte Speicher ermittelt werden. Leider ist diese Information aufgrund des oben beschriebenen Verhaltens unzuverlässig, da es keine Möglichkeit zur Kontrolle des Garbage-Collector gibt. Außerdem muss zu dieser Messung in der JVM ein weiterer Thread gestartet werden, der die Messung parallel zu einem laufenden Lerner durchführt. Dieser Thread verfälscht durch seinen eigenen Speicherbedarf das Messergebnis.

Um diese Verfälschung zu minimieren wurde eine Klasse geschrieben, die regelmäßig den aktuellen Speicherbedarf durch alle Objekte in der JVM ermittelt und inkrementell den Durchschnitt berechnet. Anhand der so gewonnenen Schätzungen zum durchschnittlichen Speicherbedarf für beide Kodierungen sollte so zumindest ein tendenzieller Vergleich der Kodierungen möglich sein. Listing 5.2 zeigt den zur Messung verwendeten Thread. Mit der Methode `setState()` wurde während der Messung die verwendete Kodierung angegeben. Die beiden Kodierungen wurden immer im Wechsel für einen Lernvorgang verwendet. Die Eingabe des Datensatzes wurde dabei im inaktiven Zustand durchgeführt, so dass der Speicherbedarf während des Einlesens der Dateien nicht erfasst wurde.

Die Abbildungen 5.3 und 5.4 zeigen den so auf den ersten Datensätzen ermittelten durchschnittlichen Speicherbedarf –umgerechnet auf jeweils ein Beispiel– in Abhängigkeit von Attributanzahl, Summe der Attributwertigkeiten und durchschnittlicher Attributwertigkeit.

Aufgrund der beschriebenen Unterschiede zwischen den Datensätzen sind keine eindeutigen Abhängigkeiten zu erkennen. Die lineare Approximation macht zwar Tendenzen über der Attributanzahl erkennbar, doch über der durchschnittlichen Attributwertigkeit ergibt sich keine Abhängigkeit. Die lineare Approximation für die Ganzzahlkodierung wird durch einen einzelnen Wert etwas nach oben gezogen, von einer Tendenz kann man jedoch nicht sprechen.

Um die Abhängigkeiten genauer zu untersuchen, wurden die gleichen Messungen auf einem hundertstel des Cover-Type-Datensatzes, also einem Datensatz mit 5810 Instanzen, mit verschiedenen Diskretisierungen durchgeführt. Diese Messungen liefern leider keine aussagekräftigen Ergebnisse und lassen lediglich den Schluss zu, dass die Methode zur Messung unzuverlässig ist. Zur Verdeutlichung ist in Abbildung 5.5 der gemessene Speicherbedarf über der Anzahl der Beispiele aufgetragen.

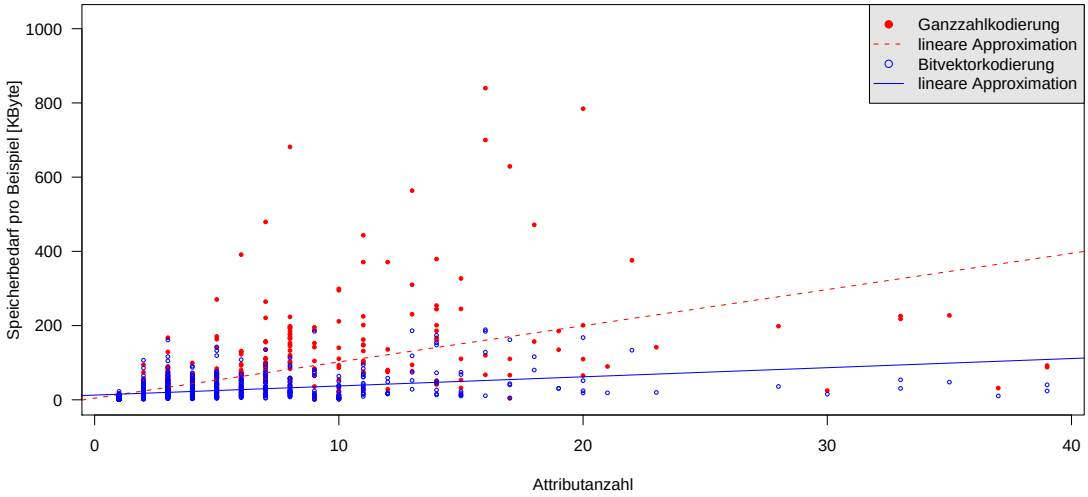


Abbildung 5.3: Speicherbedarf über Attributanzahl

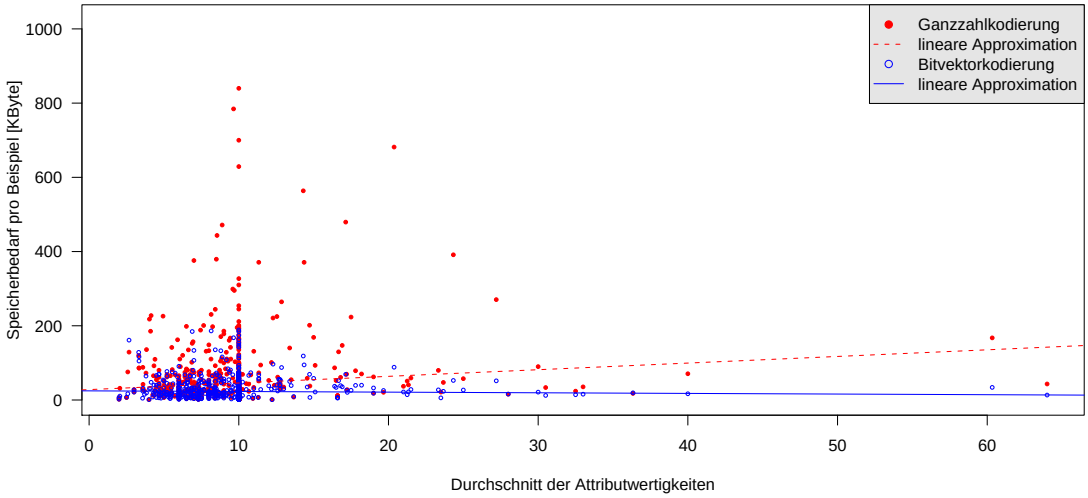


Abbildung 5.4: Speicherbedarf durchschnittlicher Attributwertigkeit

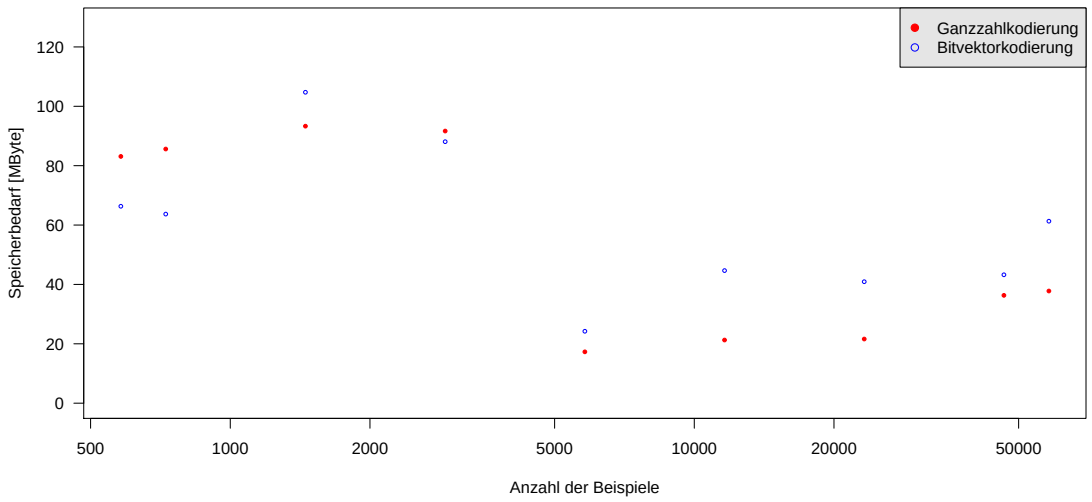


Abbildung 5.5: Speicherbedarf abhängig von der Anzahl der Beispiele

Aufgrund der beschriebenen Problematik zur Speichermessung unter Java sind die Ergebnisse der Speichermessung nicht eindeutig. Meist ist ein Vorteil der Ganzzahlkodierung tendenziell zu erkennen, aber gerade der hohe durchschnittliche Verbrauch bei wenig Attributen oder Beispielen macht deutlich, dass die Methode zur Messung nicht zuverlässig ist.

Messungen zur Laufzeit

Java bietet Methoden an, um die aktuelle Systemzeit oder die vom aktuellen Prozess verbrauchte CPU-Zeit abzufragen. Die Systemzeit eignet sich auf modernen Betriebsystemen aufgrund der großen Zahl parallel laufender Prozesse nicht zur Laufzeitmessung. Daher fällt die Wahl auf die in Nanosekunden angegebene CPU-Zeit, deren Genauigkeit ebenfalls vom verwendeten Betriebsystem und der Hardware abhängig ist. Auf dem Testsystem scheint sie im Bereich von 10-100 Mikrosekunden zu liegen.

Um also die Laufzeit des Lernprozesses abzuschätzen, wird vor und nach dem Lernen die CPU-Zeit abgefragt und anschließend die Differenz gebildet. Wegen der beschriebenen Ungenauigkeit ist dieser Wert für einen einzelnen Lernvorgang meist unbrauchbar. Im Extremfall ist die Differenz null, obwohl der Lernvorgang Zeit benötigt hat.

Diese Probleme versucht man bei Zeitmessungen durch Wiederholungen und Durchschnittsberechnung zu relativieren. Statt die Zeit für einen einzelnen Lernvorgang zu messen, wird dieser sehr oft wiederholt und die gesamte Zeit für diese Wiederholungen genommen. Der so berechnete Durchschnitt liegt für eine ausreichend große Zahl an Wiederholungen entsprechend nahe am gesuchten Wert. Listing 5.3 zeigt die Implementierung der Zeitmessung in Java.

```
public class TimingBenchmark {
private final static OperatingSystemMXBean osBean =
  (OperatingSystemMXBean) ManagementFactory.
  getOperatingSystemMXBean();

private final static long REPETITIONS = 100;

public static void main(String[] args) {
  try {
    IDataSet dsInt = new ArffReader(new FileReader(args[0]),
      new yarl.data.IntegerDataFactory()).readDataSet();
    IDataSet dsBV = new ArffReader(new FileReader(args[0]),
      new yarl.data.BitvectorDataFactory()).readDataSet();
    SimpleLearner.createLearner();
    long count = osBean.getProcessCpuTime();
    for (long i=0; i<REPETITIONS; i++)
      SimpleLearner.learn(dsBV);
    count = osBean.getProcessCpuTime() - count;
    System.out.print(count/(double)REPETITIONS+ ",");
    count = osBean.getProcessCpuTime();
    for (long i=0; i<REPETITIONS; i++)
      SimpleLearner.learn(dsInt);
    count = osBean.getProcessCpuTime() - count;
    System.out.print(count/(double)REPETITIONS);
  } catch (IOException e) {
  }}}
```

Listing 5.3: Zeitmessung in Java

Die Laufzeit des Lerners auf einem Datensatz ist abhängig von der Anzahl der Beispiele, der Anzahl der Attribute, der Wertigkeiten der Attribute und dem zu lernenden Konzept. Der Einfluss des zu lernenden Konzepts ist vernachlässigbar, da sich der Hypothesenraum aus den Wertigkeiten der Attribute ergibt. Um nicht mit einem hochdimensionalen Raum hantieren zu müssen, wird statt aller Attributwertigkeiten nur die durchschnittliche Attributwertigkeit betrachtet. Für die Charakterisierung eines Datensatzes bleiben also die drei Dimensionen Beispielanzahl, Attributanzahl und durchschnittliche Attributwertigkeit.

Um die Abhängigkeit von Attributwertigkeit und Attributanzahl zu untersuchen, eignet sich die Laufzeit pro Beispiel. Abbildung 5.6 zeigt sie in Abhängigkeit von der Anzahl der Attribute. Zunächst fällt auf, dass Datensätze mit gleicher Attributanzahl

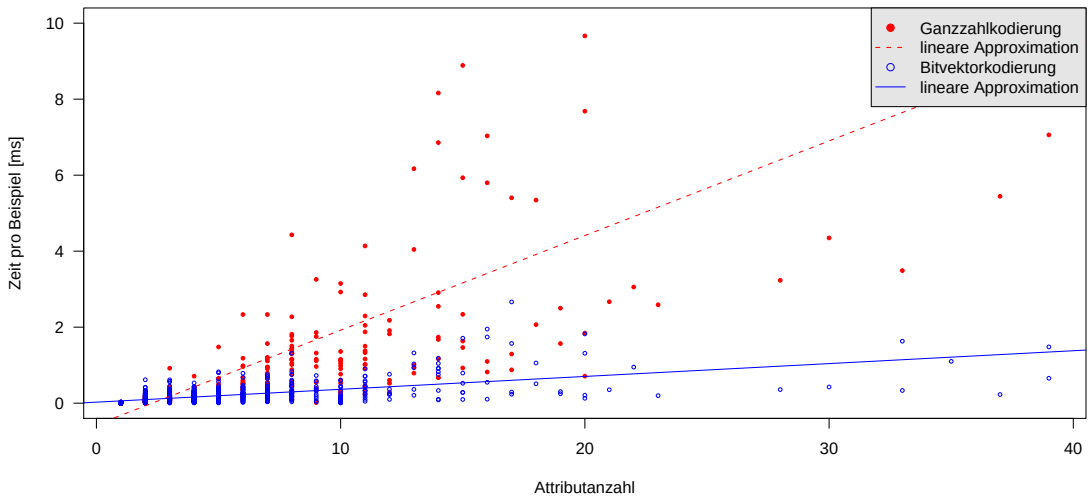


Abbildung 5.6: Laufzeit über Attributanzahl

in beiden Kodierungen stark variierende Laufzeiten aufweisen. Die Attributanzahl ist also selbst bei der Ganzzahlkodierung nicht das einzig ausschlaggebende Kriterium. Dennoch lässt sich bei beiden Kodierungen ein Anstieg bei steigender Attributanzahl erkennen, der bei der Ganzzahlkodierung etwas stärker ausfällt.

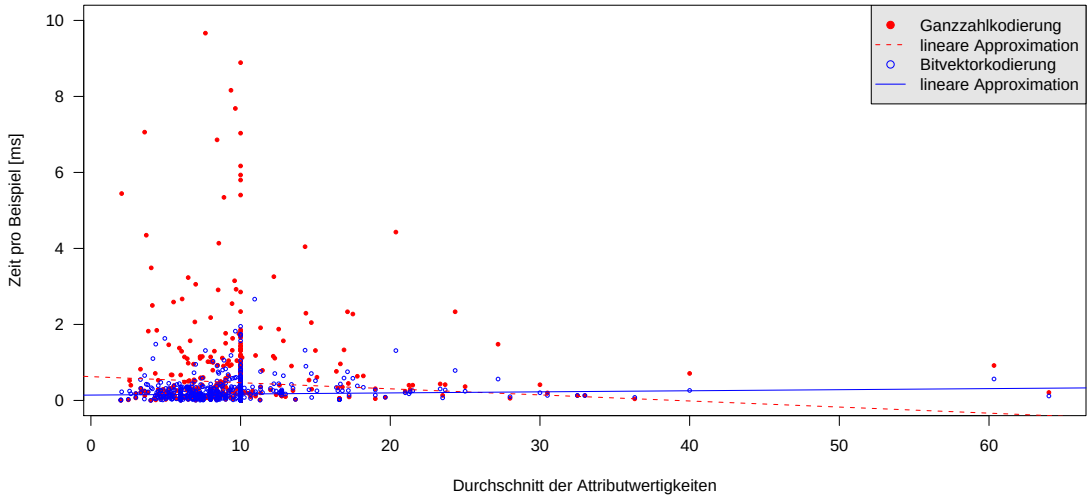


Abbildung 5.7: Laufzeit über durchschnittlicher Attributwertigkeit

Wie in Abbildung 5.7 zu sehen, hat die durchschnittliche Attributwertigkeit kaum Einfluss auf die Laufzeit. Dies liegt vor allem daran, dass die durchschnittlichen At-

tributwertigkeiten der Datensätze sich bei durchaus unterschiedlicher Komplexität des Lernproblems auf dem jeweiligen Datensatz nur wenig unterscheiden.

Auf dem Datensatz Cover-Type waren systematischere Messungen zum Einfluss der verschiedenen Parameter auf die Laufzeit möglich. Aus diesen Messungen lässt sich deutlich die Abhängigkeit von der jeweiligen Größe erkennen.

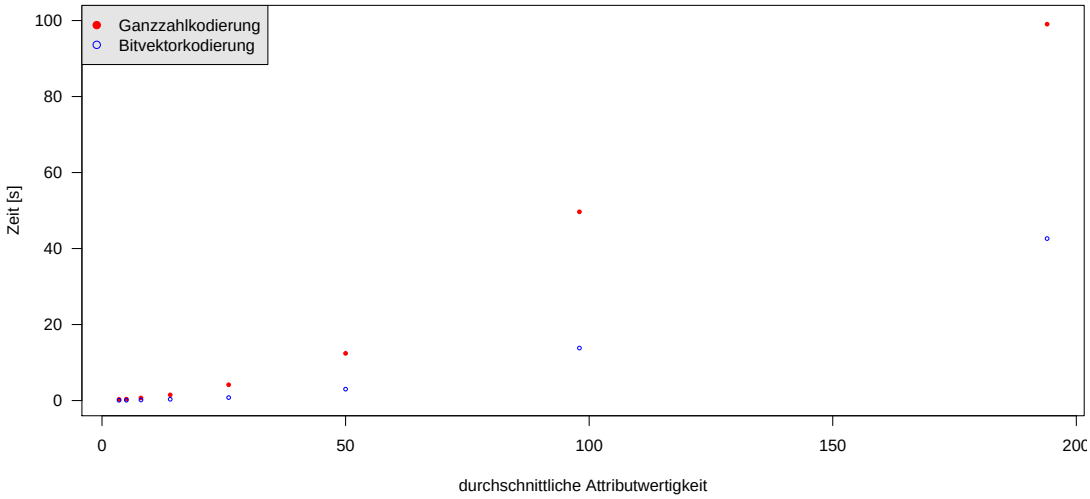


Abbildung 5.8: Laufzeit abhängig von der durchschnittlichen Attributwertigkeit

Abbildung 5.8 zeigt die Laufzeit auf dem Cover-Type-Datensatz in Abhängigkeit von der durchschnittlichen Attributwertigkeit. Der Verlauf der Kurven zeigt deutlich das grundlegende Verhalten der Laufzeit in Abhängigkeit von der Attributwertigkeit. Dieses wird durch die Tiefensuche bestimmt, deren Verzweigungsgrad direkt von dieser Größe abhängt. Die Unterschiede zwischen den Kodierungen weisen auf einen deutlichen Vorteil des Matchings in der Bitvektorkodierung hin, auch wenn diese theoretisch bezüglich des Matchings schlechter mit der durchschnittlichen Wertigkeit skaliert als die Ganzzahlkodierung.

Die Laufzeit in Abhängigkeit von der Anzahl der Attribute zeigt Abbildung 5.9. Um ansonsten konstante Bedingungen zu schaffen, wurden alle Attribute binär diskretisiert. Dadurch wird das Verhältnis zu Gunsten der Bitvektorkodierung verschoben. Mit höherer durchschnittlicher Attributwertigkeit steigt die Kurve für die Bitvektorkodierung steiler an, während die Kurve der Ganzzahlkodierung ihre Steigung beibehält.

In Abbildung 5.10 ist deutlich zu erkennen, dass der Zeitbedarf der Ganzzahlkodierung in der Anzahl der Beispiele einen deutlich höheren konstanten Faktor aufweist als der Zeitbedarf der Bitvektorkodierung. Die Messung wurde in logarithmischen Schritten in der Anzahl der Beispiele durchgeführt, um den Zeitaufwand in Grenzen zu halten. Daher ist die Abszisse logarithmisch skaliert.

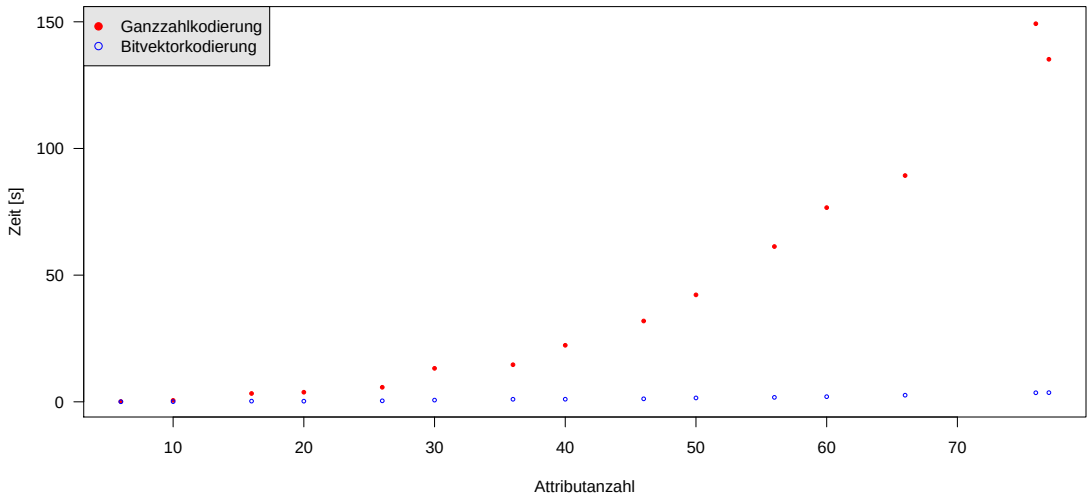


Abbildung 5.9: Laufzeit abhängig von der Attributanzahl

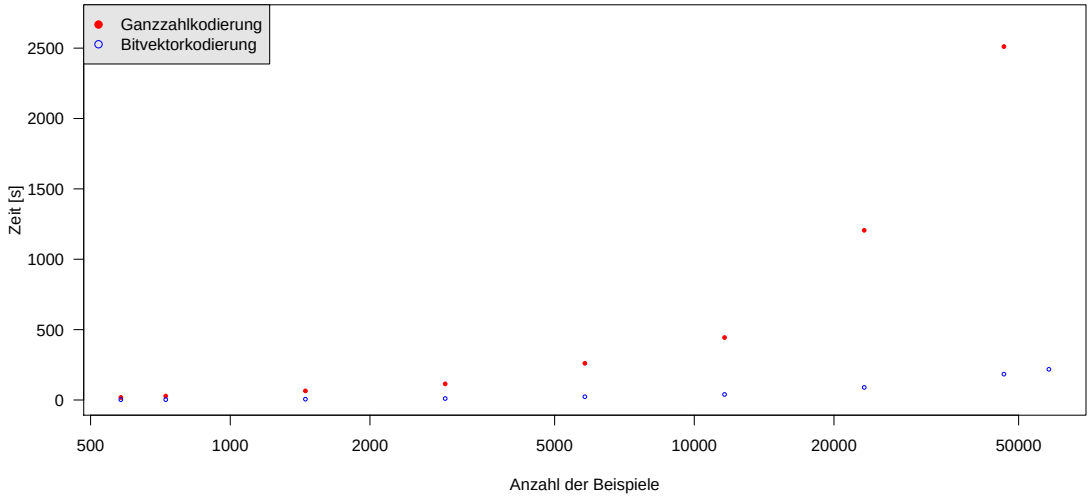


Abbildung 5.10: Laufzeit abhängig von der Anzahl der Beispiele

5.2 Erweiterbarkeit

Für den Einsatz im SeCo-Framework muss die Umgebung in mehrere Richtungen erweiterbar sein. Neben der Ergänzung verschiedenster Implementierungen der Komponenten des Lerners muss es möglich sein, numerische und strukturierte Attribute hinzuzufügen. Weitere Eingabeformate und Datenkodierungen sind ebenfalls als zukünftig notwendige Erweiterungen vorstellbar.

Generell sind durch den modularen Entwurf unter Verwendung von Schnittstellen die konkreten Implementierungen austauschbar. Jedoch besteht eine Abhängigkeit zwischen der Klasse `BitvectorRule` und der konkreten Datenkodierung. Diese Abhängigkeit lässt sich beseitigen, indem man das Matching in der Methode `covers()` nicht auf die konkrete Datenkodierung hin optimiert.

Aus Gründen der Effizienz ist die Erweiterbarkeit um neue Datenkodierungen also mit Änderungen an dieser konkreten Regelimplementierung verbunden. Solange nicht andere Regelimplementierungen hinzugefügt werden, ist dieser Aufwand durchaus vertretbar. Zieht man eine Erweiterbarkeit um andere Regelimplementationen in Betracht, so lässt sich diese Abhängigkeit leicht durch Beschränkung auf die Funktionalität der Schnittstelle `IExample` innerhalb der Methode `covers()` entfernen.

Eingabeformate sind durch die Schnittstelle `IDataFactory` vom Rest der Umgebung komplett entkoppelt. Daher steht einer Erweiterung in dieser Richtung nichts im Wege. Die Ausgabe lässt sich dann komplett über die Schnittstellen für die Datenrepräsentation lösen, also auch beliebig auf weitere Formate erweitern.

Das grundsätzliche Problem bei der Erweiterung um komplexere Attributtypen ist die Nutzung solcher Strukturen durch den Lerner. Ausgehend von der Schnittstelle `IAttribute` stehen keine weiteren Informationen zur Verfügung. Um solche Informationen zu nutzen, muss der Lerner also auf einer davon erbenden Schnittstelle arbeiten, die die erweiterten Informationen zur Verfügung stellt. Bei der Erweiterung um strukturierte Attribute ist also auf eine geeignete Wahl von Schnittstellen zu achten.

Für den Einsatz im SeCo-Framework werden die in dieser Arbeit vorgestellten Schnittstellen für die Komponenten des Lerners durch die entsprechenden, flexibleren Schnittstellen des SeCo-Framework ersetzt. Diese sind ebenfalls modular entworfen und lassen daher die problemlose Entwicklung verschiedenster Lerner zu.

6 Zusammenfassung und Ausblick

Die Grundlage der implementierten Umgebung für Regel-Lerner bildet das in Kapitel 2 vorgestellte mathematische Modell von Daten und Regeln. Ausgehend von einer Attributmenge werden Beispiele als Elemente des Kreuzproduktes der Domänen aufgefasst und Regeln als Beschreibungen für Mengen von Beispielen in Form von logischen Aussagen über den Domänen formalisiert.

Ähnlich wie in der formalen Begriffsanalyse (Ganter u. a., 2005) tauchen Verbände von Konzepten auf. Im Gegensatz zur formalen Konzeptanalyse wird hier jedoch nicht von vollständig und korrekt angegebenen Merkmalen ausgegangen, so dass sich der gesuchte Begriff (die gesuchte Hypothese) nicht direkt ergibt, sondern gar nicht im Verband vorkommen muss.

Für Attribute mit endlichen Domänen kann in diesem Modell jede Klassifikation als Regel formuliert werden. Außerdem ergibt sich für endliche Attribute eine sehr einfache Bitvektorkodierung, die in Kapitel 3 untersucht wurde. Für unendliche Attribute lässt sich mit den in Kapitel 2 definiterten Regeln als Hypothesen nicht jede Aussage über Beispiele formulieren. Schon ein einfacher Vergleich zweier Attribute wie Anzahl $>$ rund ist so nicht auszudrücken. Diese Einschränkung auf einstellige Prädikate weisen die Hypothesensprachen der meisten Lerner auf. Um sie zu beseitigen müsste die Sprache um mehrstellige Prädikate erweitert werden. Dann ist jedoch ein noch komplexerer Verband zu durchsuchen. Das Hinzufügen einer kleinen Anzahl von Relationen zur Hypothesensprache erscheint noch praktikabel. Das Finden von beliebigen Relationen durch den Lerner hingegen benötigt eine geeignete Bewertung solcher Relationen. Nicht wünschenswert ist zum Beispiel, dass der Lerner die durch den Kern der Klassifikationsfunktion gegebene Relation lernt. Alle nicht fehlerhaften Beispiele erfüllen die dadurch gegebene Bedingung, so dass sich daraus keine Implikation für die Klasse ergeben dürfte.

Die in Kapitel 3 vorgestellte Datenkodierung geht dann von endlichen Attributen aus, um eine einfache und einheitliche Kodierung zu erlauben. Eine Kodierung der Werte unendlicher Attribute durch endliche Bitvektoren ist prinzipiell unmöglich. Um weiterhin mit Bitvektoren zu arbeiten, könnte für jedes Attribut ein Bitvektor variabler Länge gewählt werden.

Jedoch geht damit der Vorteil gegenüber der Ganzzahlkodierung verloren und man erhält im Prinzip wieder eine Ganzzahlkodierung, da sich bei potentiell unendlichen Bitvektoren eine Sparse-Kodierung geradezu anbietet. Die Ganzzahlkodierung ist prinzipiell in der Lage mit unendlichen Attributen umzugehen.

Wie in Abschnitt 5.2 angedeutet, ist eine Erweiterung auf unendliche Attribute für den praktischen Einsatz der Umgebung unumgänglich. Zwar tauchen in der Praxis des maschinellen Lernens nur beschränkte numerische Attribute auf, doch ist beim Lernen meist nicht bekannt, was die tatsächlichen Schranken der Domäne eines Attributs sind, oder ob sie überhaupt existieren.

In der Praxis tauchen als unendliche Attribute vor allem numerische Attribute auf, auf denen Aussagen aus Intervallen aufgebaut werden. Solche Intervalle lassen sich noch durch die endliche Menge der Intervallgrenzen angeben, die Angabe einer Teilmenge einer Domäne als Liste der Elemente ist jedoch nicht mehr möglich. Prinzipiell sind für unendliche Attribute also andere Kodierungen der Aussagen notwendig. Eventuell bietet es sich an, beim Hinzufügen solcher Attributtypen die Regelkodierung flexibler zu gestalten.

Kapitel 4 beschreibt die vorgenommene Implementierung des Entwurfs. Dabei wurde auf der Grundlage endlicher Attribute gearbeitet, so dass eine Erweiterung auf unendliche Attribute Änderungen an den Schnittstellen nötig machen dürfte. Aus den in Kapitel 3 vorgestellten Kodierungen wurden für Regeln die einfache Bitvektorkodierung und für Daten die Ganzzahlkodierung und ebenfalls die einfache Bitvektorkodierung implementiert. Da die Ganzzahlkodierung im Wesentlichen einer Sparse-Kodierung der Bitvektorkodierung entspricht, wurde auf die Implementierung einer expliziten Sparse-Kodierung des Bitvektors verzichtet. Inwiefern sich eine Sparse-Kodierung für Regeln in bestimmten Fällen anbietet, wurde nicht näher untersucht.

Erwartungsgemäß verhält sich die Ganzzahlkodierung speichereffizienter, während die Bitvektorkodierung in der Laufzeit für das Matching besser abschneidet. Diese Erwartung sollte in Kapitel 5 überprüft werden. Auf den gesammelten Datensätzen von der Weka-Seite ließen sich leider keine klaren Aussagen dazu treffen. Vor allem der Vorteil der Ganzzahlkodierung war aufgrund der geringen Attributwertigkeiten nicht nachvollziehbar.

Daher wurden Messungen auf Teilen des Datensatzes CoverType durchgeführt, in denen gezielt jeweils eine der Größen Attributanzahl, Attributwertigkeit und Beispielanzahl variiert wurde. Für die Speichermessung lässt sich aufgrund der dynamischen Speicherverwaltung von Java keine eindeutige Aussage zum Speicherbedarf aus den Messungen ableiten, während sich die unterschiedlichen Entwicklungen der Laufzeiten für beide Kodierungen bei Veränderung der Größen klar erkennen lassen.

Ein konzeptionell und bezüglich der Erweiterbarkeit auf neue Attributtypen flexiblerer Ansatz für den Entwurf einer Umgebung für Regel-Lerner besteht darin, für jeden Attributtyp eine eigene Klasse zu entwerfen. Die anderen Komponenten müssten dann angeben, für welche Attributtypen sie verwendet werden können. Für die Beispiele wäre dann ein passender Relationstyp zur jeweiligen Attributmenge denkbar. Dieser Ansatz wäre näher am mathematischen Modell, ist jedoch mit dem Typsystem von Java nicht einfach umsetzbar. Außerdem stellt sich dabei die Frage, inwiefern dieses hohe Maß an Abstraktheit die Effizienz beeinflusst.

Den ersten praktischen Test der Erweiterbarkeit der in dieser Arbeit vorgestellten Umgebung wird der Einsatz im SeCo-Framework darstellen. Zwar sind dazu in erster Linie die Klassen des SeCo-Framework anzupassen, doch ohne eine Ergänzung um numerische Attribute wird die Umgebung die weka-Bibliothek nicht vollständig als Datenabstraktion für das SeCo-Framework ablösen können. Das Ziel, eine Datenabstraktion für das SeCo-Framework zu schaffen, ist damit nicht vollständig erreicht.

Das dem Entwurf zugrunde liegende, mathematische Modell für Daten und Regeln lässt sich problemlos auf unendliche und strukturierte Attribute erweitern. Dafür sind nur Änderungen an der Sprache der Basisaussagen notwendig, um Teilmengen über Relationen statt Angabe der Elemente zu formulieren.

Doch die Implementierung ist nur für endliche Attribute ohne Struktur ausgelegt. Hier sind grundlegende Änderungen nötig, um solche Attribute überhaupt kodieren zu können. Ferner bleibt zu untersuchen, inwiefern eine Bitvektorkodierung für derartige Attribute sinnvoll ist.

Literaturverzeichnis

[Fürnkranz 1999] Fürnkranz, Johannes: Separate-and-Conquer Rule Learning. In: *Artificial Intelligence Review* 13 (1999), February, Nr. 1, S. 3–54. – URL http://citeseer.ist.psu.edu/26490.html 1, 20, 21

[Ganter u. a. 2005] Ganter, Bernhard (Hrsg.) ; Stumme, Gerd (Hrsg.) ; Wille, Rudolf (Hrsg.): *Formal Concept Analysis, Foundations and Applications*. Bd. *3626*. Springer, 2005. (Lecture Notes in Computer Science). – ISBN 3-540-27891-5 73

[Habib u. a. 2004] Habib, M. ; Nourine, L. ; Raynaud, O. ; Thierry, E.: Computational aspects of the 2-dimension of partially ordered sets. In: *Theor. Comput. Sci.* 312 (2004), Nr. 2-3, S. 401–431. – ISSN 0304-3975 32

[Holte 1993] Holte, Robert C.: Very Simple Classification Rules Perform Well on Most Commonly Used Datasets. In: *Mach. Learn.* 11 (1993), Nr. 1, S. 63–90. – ISSN 0885-6125 17

[IEEE 1985] IEEE: *IEEE standard for binary floating-point arithmetic*. New York : Institute of Electrical and Electronics Engineers, 1985. – Note: Standard 754–1985

[Rivest 1987] Rivest, Ronald L.: Learning Decision Lists. In: *Machine Learning* 2 (1987), Nr. 3, S. 229–246. – URL citeseer.ist.psu.edu/rivest87learning.html 14

[Roubtsov 2002] Roubtsov, Vladimir: Java Tip 130: Do you know your data size? In: *JavaWorld.com* (2002). – URL http://www.javaworld.com/javaworld/javatips/jw-javatip130.html 58

[Thiel 2005] Thiel, Matthias: *Separate and Conquer Framework und disjunktive Regeln*, Fachbereich Informatik, Technische Universität Darmstadt, Diplomarbeit, 2005. – URL http://www.ke.informatik.tu-darmstadt.de/lehre/diplomarbeiten/2005/Thiel_Matthias.pdf 1, 20, 23

[Witten und Frank 2005] Witten, Ian H. ; Frank, Eibe: *Data Mining: Practical machine learning tools and techniques*. 2nd Edition. San Francisco : Morgan Kaufmann, 2005 18, 23

A Datensätze für Experimente

A.1 Datensätze von der Weka-Seite

Die folgenden Datensätze von der Weka-Seite wurden für Experimente zur Speicher- und Laufzeitmessung verwendet. Neben der Anzahl der Beispiele und Attribute ist die durchschnittliche Attributwertigkeit angegeben. Unter Verwendung der Standard-Parameter für `weka.filters.unsupervised.attribute.Discretize` sind diese Datensätze aus den Originalen von der Weka-Seite gewonnen worden.

Datei	**Beisp.**	**Attr.**	**Wert.**
arie_ben_david-ERA	1000	5	10
arie_ben_david-ESL	488	5	10
arie_ben_david-LEV	1000	5	10
arie_ben_david-SWD	1000	11	10
statlib-nominal-analcatdata_aids	50	5	6.4
statlib-nominal-analcatdata_asbestos	83	4	4.25
statlib-nominal-analcatdata_assessment	14	16	3.31
statlib-nominal-analcatdata_authorship	841	71	9.92
statlib-nominal-analcatdata_bankruptcy	50	7	14.57
statlib-nominal-analcatdata_benford	9	7	8.43
statlib-nominal-analcatdata_birthday	365	4	14.75
statlib-nominal-analcatdata_bondrate	57	12	11.33
statlib-nominal-analcatdata_boxing1	120	4	6.5
statlib-nominal-analcatdata_boxing2	132	4	6.75
statlib-nominal-analcatdata_braziltourism	412	9	7.11
statlib-nominal-analcatdata_broadway	95	10	15.1
statlib-nominal-analcatdata_broadwaymult	285	8	17.5
statlib-nominal-analcatdata_cancerrate	8	4	6.75
statlib-nominal-analcatdata_chall101	138	3	5
statlib-nominal-analcatdata_chall2	23	5	8
statlib-nominal-analcatdata_challenger	23	6	7
statlib-nominal-analcatdata_creditscore	100	7	6
statlib-nominal-analcatdata_currency	31	4	12.75
statlib-nominal-analcatdata_cyyoung8092	97	11	12.55

Datei	Beisp.	Attr.	Wert.
statlib-nominal-analcatdata_cyyoung9302	92	11	12.18
statlib-nominal-analcatdata_devils	82	2	8.5
statlib-nominal-analcatdata_dmft	797	5	5.4
statlib-nominal-analcatdata_donner	28	4	4.75
statlib-nominal-analcatdata_draft	366	6	11
statlib-nominal-analcatdata_esr	32	3	7.33
statlib-nominal-analcatdata_famufsu	14	4	7.25
statlib-nominal-analcatdata_fraud	42	12	2.58
statlib-nominal-analcatdata_germangss	400	6	4.67
statlib-nominal-analcatdata_halloffame	1340	18	83.28
statlib-nominal-analcatdata_happiness	60	4	5.5
statlib-nominal-analcatdata_homerun	163	28	6.5
statlib-nominal-analcatdata_hurricanes	57	2	9
statlib-nominal-analcatdata_japansolvent	52	10	13.4
statlib-nominal-analcatdata_lawsuit	264	5	6.8
statlib-nominal-analcatdata_mapleleafs	84	2	2.5
statlib-nominal-analcatdata_marketing	364	33	4.94
statlib-nominal-analcatdata_reviewer	379	9	44.78
statlib-nominal-analcatdata_votesurvey	48	5	7.2
statlib-nominal-backache	180	33	4.03
statlib-nominal-biomed	209	9	8.78
statlib-nominal-cars	406	8	8.5
statlib-nominal-cars_with_names	406	9	42.22
statlib-nominal-cjs	2796	35	10.2
statlib-nominal-cloud	108	8	8.25
statlib-nominal-collins	500	24	30.46
statlib-nominal-confidence	72	4	9
statlib-nominal-fl2000	67	17	12.59
statlib-nominal-irish	500	6	6.17
statlib-nominal-lupus	87	4	8
statlib-nominal-prnn_crabs	200	8	8
statlib-nominal-prnn_cushings	27	4	12.75
statlib-nominal-prnn_fglass	214	10	9.6
statlib-nominal-prnn_synth	250	3	7.33
statlib-nominal-prnn_virus3	38	18	6.94
statlib-nominal-prnn_viruses	61	18	8.89
statlib-nominal-profb	672	10	11.3
statlib-nominal-schizo	340	15	8.47
statlib-nominal-wseries	90	9	4.33
statlib-num-analcatdata_anscombe	11	8	10

Datei	Beisp.	Attr.	Wert.
statlib-num-analcatdata_apnea1	475	4	7.5
statlib-num-analcatdata_apnea2	475	4	7.5
statlib-num-analcatdata_apnea3	450	4	7.5
statlib-num-analcatdata_beetles	8	3	10
statlib-num-analcatdata_chlamydia	100	4	6.75
statlib-num-analcatdata_election2000	67	16	13.56
statlib-num-analcatdata_enrollment	56	3	8.33
statlib-num-analcatdata_floridashark	54	4	8.25
statlib-num-analcatdata_galapagos	30	8	12.38
statlib-num-analcatdata_gmat	38	5	8.4
statlib-num-analcatdata_gsssexsurvey	159	10	6
statlib-num-analcatdata_gviolence	74	10	16.4
statlib-num-analcatdata_hiroshima	649	3	7
statlib-num-analcatdata_hivcd4cd8	8	6	4.67
statlib-num-analcatdata_hivtrans1	4	5	10
statlib-num-analcatdata_hivtrans2	4	5	8.8
statlib-num-analcatdata_impeach	100	11	16.91
statlib-num-analcatdata_kotzehawk	13	14	10
statlib-num-analcatdata_lalottery	10	2	10
statlib-num-analcatdata_lda	15	4	6
statlib-num-analcatdata_menarche	25	3	10
statlib-num-analcatdata_michiganacc	108	5	9.2
statlib-num-analcatdata_mines	112	4	21.25
statlib-num-analcatdata_ncaa	120	20	4.3
statlib-num-analcatdata_neavote	100	4	30.5
statlib-num-analcatdata_negotiation	92	6	8.67
statlib-num-analcatdata_olympic2000	66	13	14.31
statlib-num-analcatdata_ossification	81	3	8
statlib-num-analcatdata_randomgen	40	3	8
statlib-num-analcatdata_runshoes	60	11	5.91
statlib-num-analcatdata_seropositive	132	4	8.25
statlib-num-analcatdata_sexpartners	1850	2	10
statlib-num-analcatdata_shampoo	15	2	10
statlib-num-analcatdata_soccer	10	5	9.6
statlib-num-analcatdata_supreme	4052	8	10
statlib-num-analcatdata_titanic	14	5	5.6
statlib-num-analcatdata_uklottery	49	2	10
statlib-num-analcatdata_uktrainacc	31	17	10
statlib-num-analcatdata_vehicle	48	5	4.4
statlib-num-analcatdata_vineyard	468	4	9.75

Datei	Beisp.	Attr.	Wert.
statlib-num-analcatdata_whale	228	8	8
statlib-num-analcatdata_wildcat	163	6	7.33
statlib-num-analcatdata_winterolympic	16	5	10.8
statlib-num-arsenic-female-bladder	559	5	16.6
statlib-num-arsenic-female-lung	559	5	16.6
statlib-num-arsenic-male-bladder	559	5	16.6
statlib-num-arsenic-male-lung	559	5	16.6
statlib-num-arsenic-us-female-bladder	21	4	12.75
statlib-num-arsenic-us-female-lung	21	4	12.75
statlib-num-arsenic-us-male-bladder	21	4	12.75
statlib-num-arsenic-us-male-lung	21	4	12.75
statlib-num-balloon	2001	3	10
statlib-num-baseball-hitter	322	24	23.79
statlib-num-baseball-pitcher	206	19	20.95
statlib-num-baseball-team	26	9	9.44
statlib-num-bodyfat	252	15	10
statlib-num-bolts	40	8	6.5
statlib-num-boston	506	14	9.36
statlib-num-boston_corrected	506	21	13.48
statlib-num-chatfield_4	235	13	10
statlib-num-christensen_ex12-3-1	24	7	6.57
statlib-num-christensen_ex5-1-2	42	2	10
statlib-num-christensen_ex5-7-12	15	2	10
statlib-num-christensen-llm_abort	72	5	6.8
statlib-num-christensen-llm_berkley	24	4	6
statlib-num-christensen-llm_boring	25	6	10
statlib-num-christensen-llm_chapman	200	8	9
statlib-num-christensen-llm_cushing	21	3	10
statlib-num-christensen-llm_oring	23	6	7.33
statlib-num-christensen-llm_tab10-1	30	4	8
statlib-num-christensen-llm_tab2-3	12	3	10
statlib-num-christensen-llm_tab2-4	24	3	10
statlib-num-christensen-llm_tab2-5	48	3	10
statlib-num-christensen-llm_tab2-6	32	3	10
statlib-num-christensen-llm_tab2-7	72	3	10
statlib-num-christensen-llm_tab2-8a	48	3	10
statlib-num-christensen-llm_tab2-8b	32	3	10
statlib-num-christensen-llm_tab4-15	16	5	3.6
statlib-num-christensen-llm_tab4-16	12	4	8
statlib-num-christensen-llm_tab4-17	27	6	10

Datei	Beisp.	Attr.	Wert.
statlib-num-christensen-llm_tab4-18	33	3	4.67
statlib-num-christensen-llm_tab4-19	39	3	7.33
statlib-num-christensen-llm_tab4-20	20	8	9
statlib-num-christensen-llm_tab4-21	12	2	10
statlib-num-christensen-llm_tab7-2	36	4	8
statlib-num-christensen-llm_tab8-1	14	3	4.67
statlib-num-christensen-llm_tab8-2	16	4	6
statlib-num-christensen-llm_tab8-3	72	5	8.4
statlib-num-christensen-llm_tab8-4	36	5	6.8
statlib-num-christensen-llm_tab8-6	42	3	10
statlib-num-christensen-llm_tension	16	5	3.6
statlib-num-christensen-llm_tenslr	8	5	5.2
statlib-num-christensen-llm_trauma	300	6	7.33
statlib-num-christensen_tab10-1	48	3	7.33
statlib-num-christensen_tab10-6	12	3	10
statlib-num-christensen_tab10-8	50	4	10
statlib-num-christensen_tab10-9	45	4	10
statlib-num-christensen_tab11-12	24	7	6.57
statlib-num-christensen_tab11-15	54	6	10
statlib-num-christensen_tab11-19	96	4	10
statlib-num-christensen_tab11-1	12	4	6
statlib-num-christensen_tab11-20	96	4	8
statlib-num-christensen_tab11-21	24	6	7.33
statlib-num-christensen_tab11-22	32	5	8.4
statlib-num-christensen_tab11-23	64	6	10
statlib-num-christensen_tab11-4	6	13	8.15
statlib-num-christensen_tab12-13	40	5	5.2
statlib-num-christensen_tab12-14	60	4	10
statlib-num-christensen_tab12-15	72	7	6.57
statlib-num-christensen_tab12-16	60	4	10
statlib-num-christensen_tab12-1	32	4	8
statlib-num-christensen_tab12-3a	24	7	6.57
statlib-num-christensen_tab12-3	72	6	6
statlib-num-christensen_tab13-1	20	7	10
statlib-num-christensen_tab13-2	12	5	10
statlib-num-christensen_tab13-3	32	5	10
statlib-num-christensen_tab13-4	60	7	10
statlib-num-christensen_tab13-5	45	16	10
statlib-num-christensen_tab16-13	32	4	8
statlib-num-christensen_tab16-18	20	4	10

Datei	Beisp.	Attr.	Wert.
statlib-num-christensen_tab16-1	12	3	4.67
statlib-num-christensen_tab16-23	15	5	8
statlib-num-christensen_tab16-24	20	7	7.43
statlib-num-christensen_tab16-25	40	3	10
statlib-num-christensen_tab16-26	5	6	9.17
statlib-num-christensen_tab16-7	36	3	10
statlib-num-christensen_tab17-10	16	9	5.56
statlib-num-christensen_tab17-3	24	5	5.2
statlib-num-christensen_tab17-6	16	5	5.2
statlib-num-christensen_tab18-1	54	5	10
statlib-num-christensen_tab2-3	20	2	10
statlib-num-christensen_tab4-10	10	2	10
statlib-num-christensen_tab4-11	6	2	10
statlib-num-christensen_tab4-12	8	3	10
statlib-num-christensen_tab4-13	10	2	6
statlib-num-christensen_tab4-14	10	3	10
statlib-num-christensen_tab4-15	7	3	10
statlib-num-christensen_tab4-1	27	3	10
statlib-num-christensen_tab4-2a	22	3	10
statlib-num-christensen_tab4-2	37	2	6
statlib-num-christensen_tab4-3	24	2	10
statlib-num-christensen_tab4-4	10	2	10
statlib-num-christensen_tab4-5	17	2	6
statlib-num-christensen_tab4-6	48	2	6
statlib-num-christensen_tab4-7	24	2	10
statlib-num-christensen_tab4-8	21	2	6
statlib-num-christensen_tab4-9	27	2	10
statlib-num-christensen_tab5-10	27	2	10
statlib-num-christensen_tab5-11	21	2	10
statlib-num-christensen_tab5-12	17	2	10
statlib-num-christensen_tab5-1	93	2	10
statlib-num-christensen_tab5-9	6	4	10
statlib-num-christensen_tab6-1	52	2	10
statlib-num-christensen_tab7-10	35	2	10
statlib-num-christensen_tab7-14	17	2	10
statlib-num-christensen_tab7-15	16	2	10
statlib-num-christensen_tab7-16	8	2	10
statlib-num-christensen_tab7-17	45	2	10
statlib-num-christensen_tab7-18	25	2	10
statlib-num-christensen_tab7-19	12	2	10

Datei	Beisp.	Attr.	Wert.
statlib-num-christensen_tab7-1	20	3	10
statlib-num-christensen_tab7-20	20	3	10
statlib-num-christensen_tab7-5	31	2	10
statlib-num-christensen_tab8-19	12	4	7.75
statlib-num-christensen_tab8-21	7	9	6.89
statlib-num-christensen_tab8-3a	24	3	7.33
statlib-num-christensen_tab8-3	12	2	10
statlib-num-christensen_tab9-10	28	3	10
statlib-num-christensen_tab9-11	40	3	10
statlib-num-christensen_tab9-12	25	3	8.33
statlib-num-christensen_tab9-13a	12	3	10
statlib-num-christensen_tab9-13	12	4	6
statlib-num-christensen_tab9-14	16	4	5.5
statlib-num-christensen_tab9-15	16	5	6.4
statlib-num-christensen_tab9-16	16	5	6.4
statlib-num-christensen_tab9-17	16	5	6.4
statlib-num-christensen_tab9-1	12	3	10
statlib-num-christensen_tab9-6	25	4	10
statlib-num-christensen_tab9-9	32	3	10
statlib-num-chscase_adopt	39	4	17.25
statlib-num-chscase_census2	400	8	10
statlib-num-chscase_census3	400	8	10
statlib-num-chscase_census4	400	8	10
statlib-num-chscase_census5	400	8	10
statlib-num-chscase_census6	400	7	10
statlib-num-chscase_chal	23	2	10
statlib-num-chscase_demand	27	11	10
statlib-num-chscase_djsp	161	3	60.33
statlib-num-chscase_empl	20	2	10
statlib-num-chscase_ers	28	12	7.5
statlib-num-chscase_funds	185	4	53.75
statlib-num-chscase_geyser1	222	3	10
statlib-num-chscase_geyser2	298	2	23.5
statlib-num-chscase_health	50	5	17.8
statlib-num-chscase_hockey1	24	4	13.5
statlib-num-chscase_hockey2	72	5	12.6
statlib-num-chscase_liinc	257	3	92.33
statlib-num-chscase_lischool	56	4	21.5
statlib-num-chscase_mort	20	4	10.5
statlib-num-chscase_nyseotc	30	2	10

Datei	Beisp.	Attr.	Wert.
statlib-num-chscase_pcb	37	3	19
statlib-num-chscase_ppp	44	5	16.8
statlib-num-chscase_return	18	3	10
statlib-num-chscase_sex	1850	2	10
statlib-num-chscase_vine1	52	10	10
statlib-num-chscase_vine2	468	3	10
statlib-num-chscase_vine3	16	4	10
statlib-num-chscase_vote	22	6	10
statlib-num-chscase_whale	228	10	10
statlib-num-colleges_aaup	1161	17	78.59
statlib-num-colleges_usnews	1302	35	47.29
statlib-num-cps_85_wages	534	11	5.45
statlib-num-csb_ch10	60	8	9
statlib-num-csb_ch11a	114	5	6.8
statlib-num-csb_ch11b	100	5	6.8
statlib-num-csb_ch12	1601	7	5.43
statlib-num-csb_ch13	2855	15	5.2
statlib-num-csb_ch14	130	19	6.63
statlib-num-csb_ch15	1032	12	8
statlib-num-csb_ch16a	885	3	10
statlib-num-csb_ch16b	612	4	10
statlib-num-csb_ch17	68	15	10
statlib-num-csb_ch18a	998	5	8.4
statlib-num-csb_ch18b	4340	5	6.8
statlib-num-csb_ch19a	1483	10	4.8
statlib-num-csb_ch19b	22432	10	6.5
statlib-num-csb_ch19c	20217	10	6.5
statlib-num-csb_ch19d	38586	10	7.3
statlib-num-csb_ch19e	49772	10	7.3
statlib-num-csb_ch19f	23455	10	7.3
statlib-num-csb_ch19g	31164	10	7.3
statlib-num-csb_ch1a	790	5	10
statlib-num-csb_ch1b	11	3	10
statlib-num-csb_ch20	3189	10	6.8
statlib-num-csb_ch21a	416	7	8.86
statlib-num-csb_ch21b	510	6	10
statlib-num-csb_ch2	50	6	10
statlib-num-csb_ch3a	51	5	18.2
statlib-num-csb_ch3b	89	13	11.46
statlib-num-csb_ch4a	51	3	10

Datei	Beisp.	Attr.	Wert.
statlib-num-csb_ch4b	6	5	10
statlib-num-csb_ch5	612	15	9.47
statlib-num-csb_ch6	148	7	10
statlib-num-csb_ch7	294	8	8
statlib-num-csb_ch8	54	5	8.4
statlib-num-csb_ch9	3240	4	6
statlib-num-detroit	13	14	10
statlib-num-diggle_table_a1	48	5	10
statlib-num-diggle_table_a2	310	9	9.89
statlib-num-diggle_table_a3	144	3	7.33
statlib-num-diggle_table_a4	18	5	10
statlib-num-diggle_table_a5	27	6	8.83
statlib-num-diggle_table_a6	79	20	9.65
statlib-num-disclosure_x_bias	662	4	10
statlib-num-disclosure_x_noise	662	4	10
statlib-num-disclosure_x_tampered	662	4	10
statlib-num-disclosure_z	662	4	10
statlib-num-dj30-1985-2003	138166	9	12.22
statlib-num-djdc0093	26612	2	10
statlib-num-hip	54	8	10
statlib-num-houses	20640	9	10
statlib-num-humandevel	130	4	40
statlib-num-hutsof99_logis	70	8	6.25
statlib-num-hutsof99_logis_d	70	12	4.83
statlib-num-hutsof99_tab1_01	38	4	6
statlib-num-hutsof99_tab2_01	150	2	10
statlib-num-hutsof99_tab2_02	150	2	10
statlib-num-hutsof99_tab2_03	150	2	10
statlib-num-hutsof99_tab3_01	15	3	10
statlib-num-hutsof99_tab3_05	45	2	10
statlib-num-hutsof99_tab3_07	11	3	10
statlib-num-hutsof99_tab3_11	70	7	6.86
statlib-num-hutsof99_tab3_11d	70	11	5.09
statlib-num-hutsof99_tab4_01	49	2	6
statlib-num-hutsof99_tab4_14	49	3	5
statlib-num-hutsof99_tab4_14d	49	3	5
statlib-num-hutsof99_tab5_01	4	3	2.67
statlib-num-hutsof99_tab5_04	18	4	4.5
statlib-num-hutsof99_tab5_07	10	4	4.5
statlib-num-hutsof99_tab5_08	10	4	5.75

Datei	Beisp.	Attr.	Wert.
statlib-num-hutsof99_tab5_10	12	4	5
statlib-num-hutsof99_tab5_11	12	4	6.75
statlib-num-hutsof99_tab5_13	9	3	4.67
statlib-num-hutsof99_tab5_16	6	3	3.33
statlib-num-hutsof99_tab5_17	6	3	3.33
statlib-num-hutsof99_tab6_11	42	17	10
statlib-num-iq_brain_size	20	9	8.22
statlib-num-kidney	76	7	6.86
statlib-num-longley	16	7	10
statlib-num-mu284	284	11	10
statlib-num-newton_hema	140	4	10.25
statlib-num-nflpass	26	7	12.29
statlib-num-no2	500	8	10
statlib-num-papir_1	30	22	7
statlib-num-papir_2	30	41	10
statlib-num-pbc	418	20	7.65
statlib-num-pbcseq	1945	19	61.26
statlib-num-places	329	10	41.9
statlib-num-plasma_retinol	315	14	8.43
statlib-num-pm10	500	8	10
statlib-num-pollen	3848	6	10
statlib-num-pollution	60	16	10
statlib-num-rabe_11	10	3	10
statlib-num-rabe_131	50	6	10
statlib-num-rabe_148	66	6	10
statlib-num-rabe_152	20	4	10
statlib-num-rabe_161	25	4	10
statlib-num-rabe_166	40	3	10
statlib-num-rabe_168	40	4	8
statlib-num-rabe_16	24	3	10
statlib-num-rabe_176	70	5	10
statlib-num-rabe_182	18	6	10
statlib-num-rabe_188	22	7	10
statlib-num-rabe_258	41	2	10
statlib-num-rabe_25	30	3	10
statlib-num-rabe_260	24	11	9.27
statlib-num-rabe_261	30	13	9.38
statlib-num-rabe_262	19	6	8.67
statlib-num-rabe_265	51	7	10
statlib-num-rabe_266	120	3	10

Datei	Beisp.	Attr.	Wert.
statlib-num-rabe_36	15	3	10
statlib-num-rabe_49	27	2	10
statlib-num-rabe_70	30	8	10
statlib-num-rabe_97	46	5	8.4
statlib-num-rmftsa_ctoarrivals	264	3	10.67
statlib-num-rmftsa_darwin	24	3	10
statlib-num-rmftsa_ladata	508	11	10
statlib-num-rmftsa_mtwashnh6597	12053	10	10.1
statlib-num-rmftsa_passengers1	144	3	7
statlib-num-rmftsa_propores	289	5	60.2
statlib-num-rmftsa_rainfall	107	2	10
statlib-num-rmftsa_sleepdata	1024	3	8
statlib-num-rmftsa_unemp	48	3	7
statlib-num-sensory	576	12	3.83
statlib-num-ships	40	5	6.2
statlib-num-sleep	62	11	14.73
statlib-num-sleuth_case0101	47	2	6
statlib-num-sleuth_case0102	93	2	6
statlib-num-sleuth_case0201	59	2	6
statlib-num-sleuth_case0202	15	2	10
statlib-num-sleuth_case0301	52	2	6
statlib-num-sleuth_case0302	743	2	6
statlib-num-sleuth_case0401	24	2	3
statlib-num-sleuth_case0402	28	3	4.67
statlib-num-sleuth_case0501	349	2	8
statlib-num-sleuth_case0502	46	2	8.5
statlib-num-sleuth_case0601	70	2	7.5
statlib-num-sleuth_case0602	84	2	8
statlib-num-sleuth_case0701	24	2	10
statlib-num-sleuth_case0702	10	2	7.5
statlib-num-sleuth_case0801	7	2	7
statlib-num-sleuth_case0802	76	3	8
statlib-num-sleuth_case0901	24	3	6
statlib-num-sleuth_case0902	96	5	27.2
statlib-num-sleuth_case1001	7	2	7
statlib-num-sleuth_case1002	20	3	7.67
statlib-num-sleuth_case1101	32	5	6.8
statlib-num-sleuth_case1102	34	9	7.44
statlib-num-sleuth_case1201	50	8	15
statlib-num-sleuth_case1202	93	7	8.14

Datei	Beisp.	Attr.	Wert.
statlib-num-sleuth_case1301	96	3	8
statlib-num-sleuth_case1302	29	3	7.33
statlib-num-sleuth_case1401	40	3	8
statlib-num-sleuth_case1402	30	5	8.4
statlib-num-sleuth_case1501	88	3	10
statlib-num-sleuth_case1502	108	2	10
statlib-num-sleuth_case1601	18	7	8.71
statlib-num-sleuth_case1602	20	4	8
statlib-num-sleuth_case1701	44	14	8.5
statlib-num-sleuth_case1702	30	8	3
statlib-num-sleuth_case2001	45	3	4.67
statlib-num-sleuth_case2002	147	7	5.43
statlib-num-sleuth_case2101	18	4	12
statlib-num-sleuth_case2102	14	4	7
statlib-num-sleuth_case2201	41	2	9.5
statlib-num-sleuth_case2202	59	4	8
statlib-num-sleuth_ex0112	14	2	6
statlib-num-sleuth_ex0116	10	3	10
statlib-num-sleuth_ex0211	122	2	6
statlib-num-sleuth_ex0221	59	2	6
statlib-num-sleuth_ex0222	94	2	6
statlib-num-sleuth_ex0321	35	3	7.33
statlib-num-sleuth_ex0325	29	4	12.75
statlib-num-sleuth_ex0326	47	3	7.33
statlib-num-sleuth_ex0329	36	2	6
statlib-num-sleuth_ex0330	20	3	10
statlib-num-sleuth_ex0331	96	2	6
statlib-num-sleuth_ex0428	15	2	10
statlib-num-sleuth_ex0429	12	2	6
statlib-num-sleuth_ex0430	13	2	8
statlib-num-sleuth_ex0431	58	3	4.67
statlib-num-sleuth_ex0519	30	4	7.75
statlib-num-sleuth_ex0523	52	2	10
statlib-num-sleuth_ex0524	23	2	6.5
statlib-num-sleuth_ex0619	50	2	7.5
statlib-num-sleuth_ex0620	41	2	7.5
statlib-num-sleuth_ex0724	107	3	9.33
statlib-num-sleuth_ex0725	38	3	7.67
statlib-num-sleuth_ex0727	10	2	10
statlib-num-sleuth_ex0728	60	3	10

Datei	Beisp.	Attr.	Wert.
statlib-num-sleuth_ex0816	12	2	8
statlib-num-sleuth_ex0817	15	2	10
statlib-num-sleuth_ex0818	17	2	10
statlib-num-sleuth_ex0820	21	2	10
statlib-num-sleuth_ex0821	16	2	7
statlib-num-sleuth_ex0822	18	3	12.67
statlib-num-sleuth_ex0914	36	4	10
statlib-num-sleuth_ex0915	38	3	10
statlib-num-sleuth_ex1014	25	3	8.33
statlib-num-sleuth_ex1023	17	3	7.33
statlib-num-sleuth_ex1024	62	5	17.2
statlib-num-sleuth_ex1115	22	2	10
statlib-num-sleuth_ex1120	18	2	10
statlib-num-sleuth_ex1122	11	4	10.25
statlib-num-sleuth_ex1123	60	7	17.14
statlib-num-sleuth_ex1217	60	17	12.94
statlib-num-sleuth_ex1220	30	7	10
statlib-num-sleuth_ex1221	42	11	14.36
statlib-num-sleuth_ex1317	28	3	6
statlib-num-sleuth_ex1318	38	3	4.67
statlib-num-sleuth_ex1414	36	6	5
statlib-num-sleuth_ex1415	36	6	5
statlib-num-sleuth_ex1509	200	2	10
statlib-num-sleuth_ex1511	37	3	10
statlib-num-sleuth_ex1512	114	3	10
statlib-num-sleuth_ex1605	62	6	10
statlib-num-sleuth_ex1611	21	4	12.75
statlib-num-sleuth_ex1612	11	4	10
statlib-num-sleuth_ex1613	36	3	7.33
statlib-num-sleuth_ex1614	12	3	10
statlib-num-sleuth_ex1708	12	14	10
statlib-num-sleuth_ex1713	18	6	11.33
statlib-num-sleuth_ex1714	47	8	10
statlib-num-sleuth_ex2011	24	2	6
statlib-num-sleuth_ex2012	120	3	7.33
statlib-num-sleuth_ex2015	60	8	9
statlib-num-sleuth_ex2016	87	11	8.55
statlib-num-sleuth_ex2017	12	5	8.8
statlib-num-sleuth_ex2114	8	5	3.8
statlib-num-sleuth_ex2115	20	3	9.33

Datei	Beisp.	Attr.	Wert.
statlib-num-sleuth_ex2220	17	2	8.5
statlib-num-sleuth_ex2224	42	4	8.25
statlib-num-sleuth_ex2225	24	3	6.67
statlib-num-sleuth_ex2226	24	2	7
statlib-num-sleuth_ex2227	47	4	10
statlib-num-sleuth_ex2228	90	7	5.71
statlib-num-sleuth_ex2413	71	4	6.25
statlib-num-smoothmeth_adptvisa	37	3	19
statlib-num-smoothmeth_airaccid	17	2	11
statlib-num-smoothmeth_basesal	118	2	64
statlib-num-smoothmeth_baskball	96	6	24.33
statlib-num-smoothmeth_birthrt	96	2	10
statlib-num-smoothmeth_caco2	50	2	7.5
statlib-num-smoothmeth_calibrat	14	2	8.5
statlib-num-smoothmeth_cars93	93	8	20.38
statlib-num-smoothmeth_cdrate	69	2	6
statlib-num-smoothmeth_diabetes	43	2	10
statlib-num-smoothmeth_elusage	55	3	25
statlib-num-smoothmeth_ethanol	88	2	10
statlib-num-smoothmeth_gascons	27	5	10
statlib-num-smoothmeth_geyser	222	2	10
statlib-num-smoothmeth_hckshoot	292	3	101.33
statlib-num-smoothmeth_jantemp	50	3	36.33
statlib-num-smoothmeth_marathon	55	2	32.5
statlib-num-smoothmeth_mbagrade	61	3	7.33
statlib-num-smoothmeth_mbasalry	91	2	6
statlib-num-smoothmeth_mbasurv	42	3	7
statlib-num-smoothmeth_mineexpl	55	2	10
statlib-num-smoothmeth_newscirc	19	3	13
statlib-num-smoothmeth_quake	2178	4	10
statlib-num-smoothmeth_racial	56	2	33
statlib-num-smoothmeth_safewatr	70	3	30
statlib-num-smoothmeth_salary	28	4	8.5
statlib-num-smoothmeth_salmon	28	3	10
statlib-num-smoothmeth_salyear	120	3	9
statlib-num-smoothmeth_schlvote	38	7	12.86
statlib-num-smoothmeth_sulfate	3321	2	10
statlib-num-smoothmeth_swissmon	100	4	10
statlib-num-smoothmeth_vineyard	52	4	10
statlib-num-smoothmeth_votfraud	22	2	10

Datei	Beisp.	Attr.	Wert.
statlib-num-smoothmeth_whale	228	3	10
statlib-num-socmob	1156	6	9.67
statlib-num-space_ga	3107	7	10
statlib-num-spdc2693	17610	2	10
statlib-num-stanford	103	11	7.82
statlib-num-strikes	625	7	10
statlib-num-tecator	240	125	10
statlib-num-transplant	131	4	10
statlib-num-tumor	86	8	9
statlib-num-veteran	137	8	6.25
statlib-num-vinnie	380	3	10
statlib-num-visualizing_animal	64	3	28
statlib-num-visualizing_barley	120	4	7
statlib-num-visualizing_bin_packing	275	2	10
statlib-num-visualizing_dating	19	2	10
statlib-num-visualizing_environmental	111	4	10
statlib-num-visualizing_ethanol	88	3	10
statlib-num-visualizing_fly	823	2	9.5
statlib-num-visualizing_food_web	113	2	6.5
statlib-num-visualizing_fusion_time	78	2	6
statlib-num-visualizing_galaxy	323	5	10
statlib-num-visualizing_ganglion	14	2	10
statlib-num-visualizing_hamster	73	6	10
statlib-num-visualizing_iris	150	5	8.6
statlib-num-visualizing_livestock	130	3	13.67
statlib-num-visualizing_melanoma	37	2	10
statlib-num-visualizing_ozone	132	2	10
statlib-num-visualizing_playfair	22	2	10
statlib-num-visualizing_polarization	355	2	10
statlib-num-visualizing_rubber	30	5	10
statlib-num-visualizing_run_time	36	4	5.25
statlib-num-visualizing_singer	235	2	9
statlib-num-visualizing_slope	44	4	10
statlib-num-visualizing_soil	8641	5	8.4
statlib-num-wind	6574	15	10
statlib-num-wind_correlations	45	47	10
statlib-num-witmer_brain_weight	39	3	19.67
statlib-num-witmer_burglaries_massachusetts	12	3	10.67
statlib-num-witmer_bush_perc_88_vote	50	3	21.33
statlib-num-witmer_census_1980	50	6	16.67

Datei	Beisp.	Attr.	Wert.
statlib-num-witmer_census_median_age	51	3	23.67
statlib-num-witmer_cereal	36	4	16.5
statlib-num-witmer_challenger	23	2	7
statlib-num-witmer_class_ratings_and_size	175	2	10
statlib-num-witmer_cleveland_temp	54	2	10
statlib-num-witmer_columbus_snow	17	3	10
statlib-num-witmer_dodgers_attendance	43	2	6
statlib-num-witmer_draft_lottery	366	2	10
statlib-num-witmer_florida_population	17	2	10
statlib-num-witmer_height_by_sex	76	2	6
statlib-num-witmer_hs_graduation_rates_1987	51	3	21
statlib-num-witmer_june_rain	41	2	10
statlib-num-witmer_larceny	60	2	10
statlib-num-witmer_laughter_and_funniness	43	3	7.33
statlib-num-witmer_nfl_1987	159	3	10
statlib-num-witmer_oberlin_temp_1988	366	3	10
statlib-num-witmer_percap_income	50	3	23.33
statlib-num-witmer_pres_vote_1976-1988	50	4	10
statlib-num-witmer_satm_satv_oberlin_math_majors	25	2	10
statlib-num-witmer_science_budget	96	2	10
statlib-num-witmer_shaw_and_wells	599	2	10
statlib-num-witmer_twins_batting_averages	14	3	11.33
statlib-num-witmer_water_oberlin	83	2	10
uci-nominal-anneal	898	39	4.33
uci-nominal-anneal.ORIG	898	39	3.59
uci-nominal-arrhythmia	452	280	7.68
uci-nominal-audiology	226	70	2.54
uci-nominal-autos	205	26	8.35
uci-nominal-balance-scale	625	5	8.6
uci-nominal-breast-cancer	286	10	5.3
uci-nominal-breast-w	699	10	9.2
uci-nominal-bridges_version1	107	13	16.85
uci-nominal-bridges_version2	107	13	15.38
uci-nominal-car	1728	7	3.57
uci-nominal-cmc	1473	10	4.5
uci-nominal-colic	368	23	5.52
uci-nominal-colic.ORIG	368	28	19.29
uci-nominal-credit-a	690	16	6.44
uci-nominal-credit-g	1000	21	6.1
uci-nominal-cylinder-bands	540	40	26.58

Datei	Beisp.	Attr.	Wert.
uci-nominal-dermatology	366	35	4.14
uci-nominal-diabetes	768	9	9.11
uci-nominal-ecoli	336	8	9.75
uci-nominal-flags	194	30	11.33
uci-nominal-glass	214	10	9.7
uci-nominal-haberman	306	4	8.5
uci-nominal-hayes-roth_test	28	5	7
uci-nominal-hayes-roth_train	132	5	8.8
uci-nominal-heart-c	303	14	6
uci-nominal-heart-h	294	14	5.36
uci-nominal-heart-statlog	270	14	9.43
uci-nominal-hepatitis	155	20	4.4
uci-nominal-hypothyroid	3772	30	3.7
uci-nominal-ionosphere	351	35	9.51
uci-nominal-iris	150	5	8.6
uci-nominal-kr-vs-kp	3196	37	2.05
uci-nominal-labor	57	17	6.06
uci-nominal-letter	20000	17	10.94
uci-nominal-liver-disorders	345	7	8.86
uci-nominal-lung-cancer	32	57	2.81
uci-nominal-lymph	148	19	4.11
uci-nominal-mfeat-factors	2000	217	10
uci-nominal-mfeat-fourier	2000	77	10

A.2 Teile des Cover-Type-Datensatzes

Für die Experimente zur Auswirkung der Anzahl der Beispiele wurde mit dem Filter `weka.filters.supervised.instance.Resample` ein Teil des Cover-Type-Datensatzes ausgewählt und mit `weka.filters.unsupervised.attribute.Discretize` diskretisiert. So wurden Datensätze mit $0,1\%$, $0,125\%$, $0,25\%$, $0,5\%$, 1%, 2%, 4%, 8% und 10% der ursprünglichen Beispielanzahl erzeugt.

Verschiedene durchschnittliche Attributwertigkeiten wurden erreicht, indem auf dem Datensatz mit 1% der ursprünglichen Beispiele nur die Attribute *hillshade_9am*, *hillshade_noon* und *hillshade_3pm* sowie das Klassenattribut betrachtet wurden. Die drei hillshade-Attribute stellen Indizes im Bereich von 0 bis 255 dar. Sie wurden manuell zu 2, 4, 8, 16, 64, 128 und 256 Werten diskretisiert.

Zum Erzeugen verschiedener Attributanzahlen wurde der Datensatz mit 1% der ursprünglichen Beispiele mit `weka.filters.unsupervised.attribute.Remove` auf ver-

schiedene Attribute reduziert, nachdem zunächst alle Attribute bis auf das Klassenattribut mit `weka.filters.supervised.attribute.Discretize` in mehrere binäre Attribute umgesetzt wurden.