Schema-Matching

Seminar Informationsintegration and Informationsqualität

Ausarbeitung von Andreas Rumpf
Betreuer: Boris Stumm

Technische Universität Kaiserslautern
Lehrgebiet Datenverwaltungssysteme
Sommersemester 2006

Zusammenfassung Schema Matching ist ein Problem, das vielen Anwendungen wie Data Warehouses, Schema-Integration und Ontologien-Matching zugrunde liegt. In diesem Aufsatz wird eine Einführung in dieses umfangreiche Themengebiet gegeben und erläutert, wie man die unterschiedlichen Ansätze einordnen kann. Fünf Ansätze werden im Detail dargestellt, um anschließend ihre Effektivität zu bewerten.

Inhaltsverzeichnis

1 Allgemeines

Ein Schema ist eine (hierarchische) Struktur, die beschreibt, wie die Daten (*Instanzen*) in einer Datenbank abgelegt werden, über welche Attribute sie verfügen, welchen Typ diese Attribute haben und welche Querverweise es zwischen den verschiedenen Datensätzen gibt.

Schema-Matching bezeichnet den Prozess, ein Schema auf ein anderes abzubilden. Die Abbildung (*Match*) enthält Informationen, die nötig sind, um ein Schema in ein anderes zu überführen, so dass die Semantik erhalten bleibt.

Oft wird Schema Matching manuell von Experten (der entsprechenden Domäne) durchgeführt. Dies ist ein sehr zeitaufwändiger und mühsamer Prozess und ohne maschinelle Unterstützung für größere Schemata kaum zu bewältigen, da man leicht eine Korrespondenz übersehen kann.

In diesem Aufsatz geht es um Verfahren, die eine automatische Generierung einer solchen Abbildung ermöglichen und damit dem Experten einen Großteil der Arbeit abnehmen; er muss das Ergebnis verifizieren und gegenenfalls korrigieren.

Es kann allerdings passieren, dass ein Schema Informationen enthält, die in dem anderen nicht vorkommen. Diese Informationen tauchen dann auch im generierten Match nicht auf; die hier behandelten Systeme können damit nicht umgehen. Für viele Anwendungsgebiete, in denen ein festes Ziel-Schema definiert wurde (z. B. Data Warehouses), spielt dies aber keine große Rolle.

1.1 Anwendungsgebiete

Schema Matching ist ein grundlegendes Verfahren, das in vielen Bereichen wie z. B. Data Warehouses, Schema-Integration und Ontologien-Matching Anwendung findet.

Bei der Schema-Integration geht es darum, eine globale Sicht auf verschiedene Datenbank-Schemata zu erstellen. Selbst wenn Schemata genau die gleiche Domäne modellieren, unterscheiden sie sich in ihrer Struktur, Terminologie und in den konrekten verwendeten Datentypen, da sie unabhängig voneinander entwickelt wurden. Als erster Schritt einer Schema-Integration muss herausgefunden werden, wie die Schemata zusammenhängen; es ist ein Schema-Matching-Prozess nötig. Anschließend können die zusammenhängenden Elemente in ein integriertes Schema übernommen werden.

Ein verwandtes Problem ist es, unterschiedliche Datenquellen in einem Data Warehouse zu vereinen. Ein Data Warehouse ist eine Datenbank, die dem Management als Entscheidungshilfe dienen soll. Dafür müssen aus verschiedenen Datenquellen Informationen extrahiert werden. Daten werden aus dem Quellformat in das Data-Warehouse-Format transformiert. Um zu diesen Transformationen zu gelangen, muss man zunächst die Beziehungen zwischen Quell- und Data-Warehouse-Format herausfinden. Dies ist ein Schema-Matching-Problem.

Es gibt viele verschiedene Ansätze, das Schema-Matching-Problem zu lösen. Man kann sie auf verschiedene Arten klassifizieren, was im folgenden Abschnitt erläutert wird.

1.2 Begriffsbestimmung

Man unterscheidet drei Arten von Kardinalitäts-Problemen beim Schema Matching:

1:1 Hier wird genau ein Objekt (Attribut, etc.) des einen Schemas auf ein Objekt des anderen abgebildet.

1:n

n:1 Hier muss ein Objekt auf eine Menge von Objekten abgebildet werden. Ein Beispiel wäre eine Adresse, die in einem Schema als eine Zeichenkette abgelegt wird, im anderen Schema aber als Tupel bestehend aus (Straße, Hausnummer, Stadt).

n:m Hier muss eine Menge von Objekten auf andere Menge abgebildet werden, ohne dass einzelne Objekte direkt aufeinander abbildbar wären.

1:n- und n:m-Abbildungen werden auch *indirekte Mappings* genannt. Die Zahl der Fälle, in denen indirekte Mappings auftreten, liegt zwischen 22% und 35% [Empley04]. Trotzdem beschränken sich die meisten Schema-Matching-Algorithmen auf 1:1 Kardinalitäten, was daran liegt, dass indirekte Mappings schwieriger zu finden sind.

Rahm und Bernstein [Bernstein01] unterscheiden die verschiedenen Lösungsansätze nach folgenden Kriterien:

Instanzen- versus schemabasierte Matcher Diese Unterscheidung liegt diesem Aufsatz zugrunde: Ein Matcher kann entweder über die Instanzen, also die konkreten Datensätze, Übereinstimmungen zwischen Schemata finden oder über das Schema selbst, das dazu in einer konkreten Form (SQL-Datenbankschema, XML-Schema, etc.) vorliegen muss. Natürlich sind auch hybride Verfahren denkbar, die beide Informationsquellen nutzen.

Element- versus Struktur-basierte Matcher Diese Unterscheidung ist nur bei schemabasierten Matchern sinnvoll: Ein Matcher kann einzelne Elemente (z. B. Attribute) aufeinander abbilden oder komplexere Gebilde wie Objekte, Strukuren und Relationen.

Ein instanzenbasierter Matcher wird lediglich auf der Element-Ebene arbeiten können, da die Instanzen-Ebene im Wesentlichen unstrukturiert ist.

Linguistische versus Constraint-basierte Matcher Hier geht es darum, welcher Teil der Information genutzt wird: Werden z. B. die Namen der Attribute, ihre sprachliche Beschreibung, usw., spricht man von einem linguistischen Matcher. Oft verwendet ein linguistischer Matcher zusätzliche Informationsquellen wie Wörterbücher, Thesauri und Synonymlexika. Ein Constraint-basierter Matcher nutzt die Typen und Beschränkungen der Attribute. Ein Attribut vom Typ „Zahl“ kann z. B. keinen Namen enthalten. Beschränkungen können Wertebereiche, Kardinalitäten, sowie Beziehungen innerhalb eines Schemas umfassen.

2 Schema-basierte Verfahren

Die meisten Systeme zum Schema-Matching beruhen auf schemabasierten Verfahren. Sie haben den Vorteil, dass sie schneller als instanzenbasierte Verfahren sind, da jene aus der (umfangreichen) Menge von Instanzen die benötigten Informationen extrahieren müssen.

Dieser Abschnitt behandelt die drei Verfahren *Cupid*, *Simiarity Flooding* und *S-Match* im Detail.

2.1 Das System Cupid

Das System *Cupid* [Rahm01] wurde von Jayant Madhavan, Philip Bernstein und Erhard Rahm im Auftrag der Microsoft Corporation entwickelt. Es hat das Ziel, möglichst *generisch* zu sein, d. h. auf verschiedenen Problemfeldern, in verschiedenen Domänen einsetzbar und möglichst unabhängig vom verwendeten Datenmodell zu sein.

Cupid betrachtet ein Schema als einen gerichteten Graphen, dessen Knoten den Elementen (Attributen) entsprechen und dessen Kanten die Struktur des Schemas modellieren. Dazu ein Beispiel: Das Datenbank-Schema links entspricht dem Graphen auf der rechten Seite in Abbildung 1.

```
CREATE TABLE Kunde(
  Kundennr INT,
  Name     VARCHAR(100),
  "Straße" VARCHAR(100),
  Stadt    VARCHAR(100),
  PLZ      VARCHAR(10),
  PRIMARY KEY (Kundennr)
);
```

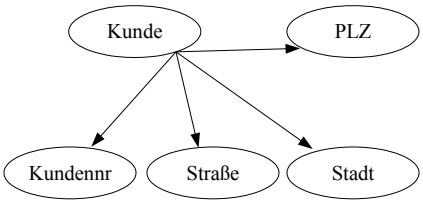

Abbildung 1. Ein Schema und sein Graph

Cupid setzt sich aus zwei Teilen zusammen: dem linguistischem Matcher und dem strukturellen Matcher.

Der linguistische Matcher Das linguistische Matching ist für die Element-Ebene (also Blatt-Ebene) zuständig. Es besteht aus drei Phasen:

1. Normalisierung
   Auch wenn Elemente aus verschiedenen Schemata dasselbe Objekt in der Realität bezeichnen, unterscheiden sie sich oft in ihrer Namensgebung: Hierfür sind Abkürzungen, unterschiedliche Schreibweisen (z. B. Zusammenschreibung oder mit Bindestrichen) oder Synonyme verant wortlich.

Daher werden die Namen zunächst in einzelne Bestandteile zerlegt („tokenization“). Anschließend werden Abkürzungen und Akronyme [1] vervollständigt („Expansion“) sowie unwichtige Füllwörter (Artikel, Prä positionen, etc.) entfernt („Elimination“). Hierbei hilft ein allgemeines und ein domänenspezifisches Wörterbuch. Es wird versucht, den semantischen Inhalt eines Namens zu erfassen. Ein Problem dabei sind Homonyme; diese führen zu falschen Treffern wenn sie später nicht durch den strukturellen Matcher beseitigt werden.

2. Kategorisierung
   Anschließend werden die Elemente entsprechend ihrer Semantik und ihrem Datentyp kategorisiert. Die Kategorie muss nicht eindeutig sein, ein Element kann mehreren Kategorien angehören. Auch wird die Position eines Elements in der Schema-Hierarchie berücksichtigt.
   Die Kategorisierung dient lediglich dazu, die potenziellen Vergleichskandidaten einzuschränken. Es werden in der dritten Stufe, dem Vergleich, nur solche Elemente verglichen, die in derselben Kategorie gelandet sind.
3. Vergleich
   In dieser Phase werden Elemente aus Schema A mit Elementen aus Schema B verglichen. Ziel ist es, einen Ähnlichkeits-Koeffizienten zu bestimmen. Dieser gibt die Ähnlichkeit als numerischen Wert an, ein Wert von 1 bedeutet totale Übereinstimmung, 0 überhaupt keine.
   Hierfür wird wieder ein Wörterbuch verwendet, das Synonyme oder eng verwandte Unterbegriffe (Hyponyme) und Oberbegriffe (Hypernyme) kennt.

Ergebnis des „linguistic matchers“ ist eine Tabelle *lsim*, die für jedes Element-Paar einen Ähnlichkeits-Koeffizienten im Bereich 0-1 enthält.

Der strukturelle Matcher nutzt die Struktur der Schemata, um einen Ähnlichkeits-Koeffizienten *ssim* zu errechnen. *lsim* und *ssim* werden anschließend zu *wsim*, dem gewichteten Ähnlichkeits-Koeffizienten verrechnet. Dies geschieht mittels der Formel $wsim = w_{struct} \cdot ssim + (1 - w_{struct}) \cdot lsim$, wobei w_{struct} hier das Gewicht bei *ssim* bezeichnet.

Der folgende Algorithmus wird zum strukturellen Matching verwendet. Er geht davon aus, dass die Schemata keine Zyklen enthalten. Momentan kann Cupid nur solche Schemata verarbeiten. Es gibt aber effiziente Verfahren, wie man Schema-Graphen in Schema-Bäume transformieren kann, z. B. den *graph-to-tree-Operator* von Protoplasm [Bernstein04].

[1] Das ist ein Wort, das sich aus den Anfangsbuchstaben einer Beschreibung zusammensetzt. Ein Beispiel: AIDS - Acquired Immune Deficiency Syndrome

TreeMatch(SourceTree S, TargetTree T)
for each $s \in S, t \in T$ where s, t are leaves
    set $ssim(s,t) = datatype - compatibility(s,t)$
$S' = post - order(S), T' = post - order(T)$
for each $s \in S'$
    for each $t \in T'$
        compute $ssim(s,t) = structural - similarity(s,t)$
        $wsim(s,t) = w_{struct} \cdot ssim(s,t) + (1 - w_{struct}) \cdot lsim(s,t)$
        if $wsim(s,t) > th_{high}$
            $increase - struct - similarity(leaves(s), leaves(t), c_{inc})$
        if $wsim(s,t) < th_{low}$
            $decrease - struct - similarity(leaves(s), leaves(t), c_{dec})$

Der Algorithmus geht davon aus, dass die Matrix $lsim(s,t)$ durch den linguistischen Matcher bereits gefüllt wurde.

Die erste Schleife initialisiert für alle Blätter in S, T $ssim$ mit der Datentypkompatibilität, die ein Wert zwischen 0 bis 0,5 ist (nicht von 0 bis 1, da sich der Wert später noch verbessern kann). Dieser Wert wird durch ein Nachschlagen in einer vordefinierten Tabelle ermittelt.

Anschließend werden die beiden Schemata in *post-order* durchlaufen, was einer *bottom-up*-Strategie entspricht. Das heißt, dass die Blätter zuerst betrachtet werden. Dies ist ein wesentlicher Vorteil von Cupid: Die Struktur der Schemata wird berücksichtigt, aber der Algorithmus lässt sich dennoch nicht beirren, wenn die Struktur der beiden signifikant voneinander abweicht. Letztendlich stecken in den Blättern alle Informationen, die im Schema enthalten sind. Man vergleiche dies mit einem *top-down*-Verfahren: Wenn die Schemata an den Wurzeln sich schon unterscheiden, wird ein solches Verfahren kaum gute Ergebnisse liefern.

$structural - similarity(s,t)$ berechnet sich wie folgt: Wenn s und t Blätter sind, ist der Wert einfach der Initial-Wert. Haben zwei Blätter eine Ähnlichkeit, die größer als ein Schwellwert th_{accept} ist, spricht man von einer starken Verbindung („strong link“) zwischen ihnen. Dies sind also zwei Elemente, die wahrscheinlich aufeinander abbildbar sind.

Sind s und t keine Blätter (bzw. einer von beiden kein Blatt), dann ist berechnet sich der Wert wie folgt ($leaves(s)$ bezeichnet die Blätter des Teilbaumes, der bei s seine Wurzel hat):
$structural - similarity(s,t) = \frac{1}{\|leaves(s) \cup leaves(t)\|} \cdot$
$\|\{x | x \in leaves(s) \wedge \exists y \in leaves(t), stronglink(x,y)\}$
$\cup \{x | x \in leaves(t) \wedge \exists y \in leaves(s), stronglink(y,x)\}\|$

Es wird also die relative Anzahl von starken Verbindungen zwischen den Blättern von s und t berechnet. Die Idee hierbei ist, dass zwei Schema-Elemente zueinander ähnlich sind, wenn ihre Blätter es sind, auch wenn die Zwischenknoten nicht übereinstimmen, bzw. ein Schema über Zwischenknoten verfügt, die im anderen gar nicht vorkommen.

Anschließend wird $wsim$ nach der schon erläuterten Formel berechnet. Ist der Wert von $wsim$ größer als ein Schwellwert th_{high}, so wird $ssim$ von allen

Blättern von s und t um einen Betrag c_{inc} inkrementiert (natürlich so, dass dabei der Wert 1 nicht überschritten wird); ist er kleiner als th_{low}, so wird $ssim$ um c_{dec} dekrementiert.

Da die Bäume in *Post-Order*-Reihenfolge durchlaufen werden, ist folgendes sichergestellt: Wenn zwei Elemente s und t miteinander verglichen werden, wurden ihre Teilbäume bereits verglichen, so dass die Struktur ihrer Teilbäume automatisch beim Vergleich berücksichtigt wird.

Die konkreten Mappings ergeben sich wie folgt: Ist man nur an einer Element-auf-Element-Abbildung interessiert, wobei die Elemente Blätter sind, kann man einfach das entsprechende Element auswählen, welches die höchste Ähnlichkeit aufweist: Wähle für s das t mit $wsim(s,t)$ maximal. Man erhält so eine 1:1 Abbildung.

Um eine 1:n Abbildung zu erhalten, wählt man für s alle t mit $wsim(s,t >= th_{accept}$.

Ist man auch an Elementen interessiert, die keine Blätter sind, so ist das Verfahren ähnlich. Allerdings ist zu beachten, dass zunächst einmal ein weitere Postorder-Iteration über die Schemata nötig ist, die die Ähnlich keiten zwischen inneren Knoten erneut berechnet, da erst jetzt die endgültigen Ähnlichkeitens-Werte für die Blätter bekannt sind.

2.2 Similiarity Flooding

Wie auch Cupid, versucht das Verfahren des *Similarity Flooding* [Melnik01] eine möglichst generische Lösungsstrategie zum Schema-Matching-Problem zu liefern. Der Fokus richtet sich dabei aber auf 1:1 Mappings.

Zugrunde liegt dem Similiarity Flooding das Prinzip der *Fixpunktiteration*. Hierbei wird eine Operation so lange ausgeführt, bis sie keine Veränderung (des Ergebnisses) mehr bewirkt. Es gilt also für den letzten Schritt des Verfahrens: $f(x*) = x*$, wobei f die Operation symbolisiert und $x*$ den gefunden Fixpunkt.

Ähnlichkeiten-Propagierungs-Graph Wie bei Cupid werden Schemata als Graphen modelliert, diesmal aber als gerichtete. Um Ähnlichkeiten zwischen Elementen (als Knoten modelliert) zu errechnen, liegt die Heuristik zugrunde, dass zwei Elemente von verschiedenen Schemata ähnlich sind, wenn ihre Nachbarn einander ähnlich sind. Die Ähnlichkeit *fließt* von einem benachbarten Knoten zum nächsten während der Fixpunktiteration, daher der Name *Similarity Flooding*.

Die interne Datenstruktur, die für das Similarity Flooding benutzt wird, ist ein markierter gerichteter Graph: Eine Kante wird als Tripel (s,p,o) dargestellt, wobei s für den Quellknoten, o für den Zielknoten und p für die Markierung der Kante stehen.

Der Algorithmus arbeitet aber nicht auf zwei solchen Graphen A und B (es sollen ja zwei Schemata verglichen werden), sondern diese werden zunächst einmal in eine andere Datenstruktur, dem *Ähnlichkeiten-Propagierungs-Graphen*

(Similarity Propagation Graph), überführt. Auch dieser ist ein gerichteter markierter Graph. Um zu dieser Datenstruktur zu kommen, wird zuerst ein *paarweiser Verbindungsgraph* (Pairwise Connectivity Graph) PCG von A und B definiert: $((x,y),p,(x',y')) \in PCG(A,B) \Leftrightarrow (x,p,x') \in A \wedge (y,p,y') \in B$. Im PCG geht jeder Knoten aus einer Knotenkombination von $A \times B$ hervor, er repräsentiert ein Mapping von x nach y. Die Idee dahinter ist die folgende: Wenn a zu b ähnlich ist, dann ist a_1 zu b_1 ähnlich, da man jeweils über die Kante $L1$ von a zu a_1 bzwv̇on b zu b_1 gelangt. Daraus resultiert die Kante $((a,b),L1,(a_1,b_1))$ im PCG.

Ein Ähnlichkeiten-Propagierungs-Graph errechnet sich aus dem PCG, indem zu jeder Kante $((x,y),p,(x',p'))$ ihre Rückrichtung $((x',y'),p,(x,y))$ hinzugefügt wird.

Die Kanten im Ein Ähnlichkeiten-Propagierungs-Graph erhalten eine Zahl zwischen 0 und 1 als Markierung, den sogenannten Propagierungs-Koeffizienten. Es gibt verschiedene Verfahren, diese Koeffizienten zu bestimmen. Im Beispiel liegt der Gedanke zugrunde, dass die Koeffizienten der ausgehenden Kanten vom selben Typ (derselben Markierung) in der Summe 1 ergeben sollen. Daher haben die beiden Kanten vom Typ $L1$ nur einen Koeffizienten von 0.5, da sowohl die Knoten $a1, b1$ und $a2, b1$ über $L1$ zur Ähnlichkeit vom Knoten a, b beitragen (und damit zur Ähnlichkeit zwischen a und b). Nun wird auch ersichtlich, warum sämtliche anderen Koeffizienten in Abbildung 2 1.0 sind. Es gibt noch weitere Verfahren, um die Koeffizienten zu bestimmen [Melnik01].

Abbildung 2 veranschaulicht den gesamten Prozess:

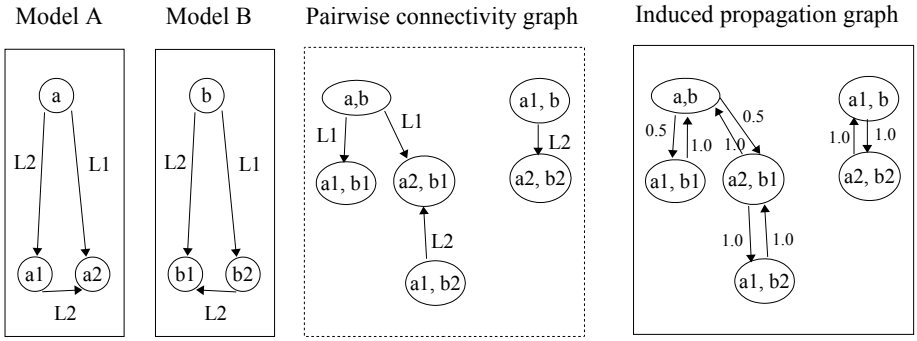

Abbildung 2. Zwei Modelle werden in ihren Ähnlichkeiten-Propagierungs-Graph überführt

Fixpunktiteration Sei $\sigma(x, y)$ eine totale Funktion $A \times B \rightarrow [0, 1]$. σ ist das Ähnlichkeitsmaß oder auch das Mapping zwischen A und B. Dieses Mapping wird nun iterativ bestimmt, dabei sei σ^i das Mapping nach der i-ten Iteration und σ^0 das initiale Mapping. Als initiales Mapping kann man z. B. einen String-Matcher verwenden. Will man nicht so viel Aufwand betreiben, ist es auch möglich einfach $\sigma(x, y)^0 = 1.0 \forall x \in A, y \in B$ zu definieren.

σ^{i+1} berechnet sich dann wie folgt:

$\sigma^{i+1}(x, y) = \sigma^i(x, y) +$
$\sum_{(a_u,p,x)\in A,(b_u,p,y)\in B} \sigma^i(a_u, b_u) * \omega((a_u, b_u), (x, y)) +$
$\sum_{(x,p,a_v)\in A,(y,p,b_v)\in B} \sigma^i(a_v, b_v) * \omega((a_v, b_v), (x, y))$

Anschließend müssen die Werte noch normalisiert werden, indem sie durch den maximal auftretenden Wert dividiert werden.

Diese Operation wird solange durchgeführt, bis entweder die Euklidische Länge des Differenzen-Vektors $\Delta(\sigma^n, \sigma^{n-1}) \leq \epsilon$ für ein $n \geq 0$ oder eine Maximal-Anzahl von Iterationen erreicht wird (leider konvergiert das Verfahren nicht immer).

Tabelle 1 enthält die Werte nach fünf Iterationen für das Beispiel:

Tabelle 1. Fixpunkt-Werte

Knoten	(a,b)	(a2,b1)	(a1,b2)	(a1,b1)	(a1,b)	(a2,b2)
Wert	1.0	0.91	0.69	0.39	0.33	0.33

Man sieht, dass a am ähnlichsten zu b ist und b der einzig sinnvolle Kandidat für a ist. In der Tabelle sind alle sinnvollen Abbildungen aufgelistet (*Multimapping*). Bei n Tupeln ergeben sich 2^n mögliche Teilmengen, die als Mapping infrage kommen. Es gibt zwei Kriterien, nach denen man entscheidet kann, welche man auswählen sollte:

Cumulative similarity Die Paare sind so zu wählen, dass die Summe ihrer Ähnlichkeiten maximal wird. Dies ist das intuitive Vorgehen.

Stabile Ehe Man betrachtet das Auswahl-Problem als Problem der „stabilen Ehe“. Hierbei hat man n Männer und n Frauen, die jeweils eine Rangfolge von bevorzugten (andersgeschlechtlichen) Partnern haben. Ziel ist es nun, die Paare so auszuwählen, dass es keine Paare (x, y) und (a, b) gibt, bei denen x b lieber mag als y und b x lieber mag als a. Dies wäre nämlich eine instabile Situation. Bezogen auf das Schema-Matching-Problem betrachtet man also die Elemente von Schema A als Frauen und die von Schema B als Männer.

Diese beiden Kriterien können zu unterschiedlichen Ergebnissen kommen, wie das Beispiel in Tabelle 2 zeigt:

Das Cumulative-Similarity-Kriterium würde hier als Mapping $M_1 = (a, y), (x, b)$ vorschlagen, da es mit einer Ähnlichkeit von $0.81 + 0.54 = 1.35$ die höchste aufweist. $M_2 = (a, b), (x, y)$ weist nur eine Ähnlichkeit von $1.0 + 0.27 = 1.27$ auf. Aber M_2 erfüllt das Stabile-Ehe-Kriterium, M_1 nicht, da in M_1 a b gegenüber y bevorzugt (1.0 versus 0.81) und b a gegenüber x (1.0 versus 0.54).

Tabelle 2. Fixpunkt-Werte

Knoten	(a,b)	(a,y)	(x,b)	(x,y)
Wert	1.0	0.81	0.54	0.27

Welches Kriterium das bessere ist, hängt vom Anwendungsfall ab; in der Regel scheidet Stabile-Ehe-Kriterium besser ab [Melnik01].

2.3 Semantic Schema Matching

Eine gänzlich andere Herangehensweise zum Schema-Matching-Problem wählten 2005 Fausto Giunchiglia, Pavel Shvaiko und Mikalai Yatskevich in ihrem *S-Match System* [Giunchiglia05]. Wie Cupid und das Similarity-Flooding arbeitet das S-Match System auf XML-Schemata.

Die grundlegende Idee ist es, aus den XML-Element-Namen und den XML-Strukturen der Schemata A und B die *Semantik* zu extrahieren. Anschließend wird sie in aussagenlogische Formeln abgebildet, um dann das Matching auszuwählen, dass keinen Widerspruch zu den Formeln enthält.

S-Match betrachtet XML-Daten als Baumstrukturen, die Knoten sind markiert mit den entsprechenden XML-Tags. Das *Konzept* einer Markierung ist die aussagenlogische Formel, die die Menge von Instanzen beschreibt, die unter den Knoten fallen. Zwei Konzepte K_1, K_2 können auf vier verschiedene Arten zusammenhängen:

1. Sie können semantisch äquivalent sein ($K_1 = K_2$).
2. K_1 kann in K_2 enthalten sein, K_2 ist *allgemeiner* als K_1 ($K_2 \supseteq K_1$).
3. Der umgekehrte Fall: K_2 ist spezieller als K_1 ($K_2 \subseteq K_1$).
4. Die Konzepte können disjunkt sein: $K_1 \perp K_2$.

Ist die Relation unbekannt, wird dies durch die *idk* (I don't know) Relation ausgedrückt. Eine Überlappung ($K_1 \cap K_2 \neq \emptyset$) kann nicht festgestellt werden; auch sie ist *idk*. Man kann eine Ordnung für die verschiedenen Relationen definieren: $=$ ist die stärkste, dann kommen (auf gleicher Stufe) $\supseteq$ und $\subseteq$, $\perp$ und mit *idk* die schwächste Relation.

Formale Definition des Problems Sei $N1$ die Knoten-Anzahl von Schema A, $N2$ die von Schema B und $n1_i$ der i-te Knoten von A und $n2_j$ der j-te von B. Ein Mapping-Element ist ein Tupel $(ID_{ij}, n1_i, n2_j, R)$, wobei R die Relation darstellt und ID_{ij} ein eindeutiger Bezeichner für das Mapping von $n1_i$ nach $n2_j$ ist.

Das semantische Matching-Problem ist definiert als: Seien die Bäume $T1$, $T2$ gegeben. Gesucht sind die Tupel $(ID_{ij}, n1_i, n2_j, R')$ für alle $i = 1, \ldots, N1$, $j = 1, \ldots, N2$, so dass R' die stärkste Relation ist, die für die Konzepte der Knoten $n1_i, n2_j$ gilt.

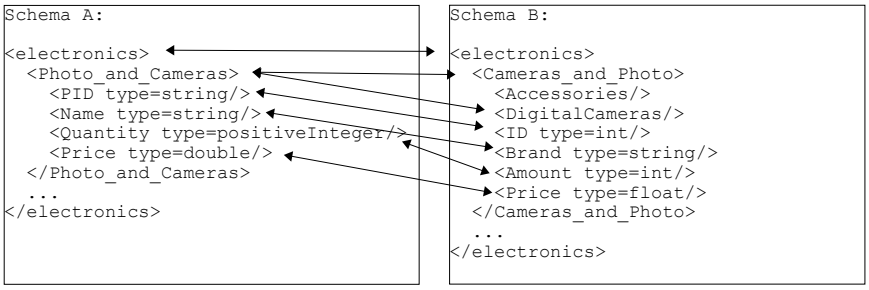

Abbildung 3. Zwei XML-Schema und ihre Abbildung

Der Baum-Matching-Algorithmus Am Beispiel, das Abbildung 3 zeigt, wird nun der Algorithmus vorgeführt.

1. Ermittle für alle Markierungen L in den beiden Bäumen ihr Konzept. Markierungen werden als Kurzbeschreibungen der Instanzen angesehen, die unter den Knoten fallen. Um die Semantik aus diesen Kurzbeschreibungen zu extrahieren, wird zuerst eine Satzzerlegung (Tokenization) ausgeführt. S-Match benutzt anschließend *WordNet* [Fellbaum98], um die Bedeutung der einzelnen Wörter zu analysieren. So ist die Bedeutung der Markierung „Photo and Cameras“ „sowohl Fotos als auch Kameras“. Als nächstes werden *Lemmas* aus den Semantiken extrahiert, z. B. *cameras* $\rightarrow$ *camera*. Komplexe aussagenlogische Formeln werden gebildet, indem Atome kombiniert werden mittels den der Mengenlehre entsprechenden Operatoren $\vee\wedge\ \rightarrow\leftrightarrow$. Das Konzept von „Photo and Cameras“ ist $\langle Cameras, senses_{WN\#2}\rangle \vee \langle Photo, senses_{WN\#1}\rangle$, wobei $senses_{WN\#2}$ die Disjunktion aller Bedeutungen, die WordNet „Photo“ zuordnet.
   Im folgenden sei nun die Unterscheidung zwischen einer Markierung und ihrem Konzept aufgehoben.
2. Berechne für alle Knoten N in den beiden Bäumen ihr Konzept. Der Begriff des Konzepts wird hier auf Knoten erweitert: Das Konzept eines Knotens N ist die Konjunktion aller Formeln auf dem Weg von der Wurzel bis zu N. Im Beispiel ist das Konzept des Knotens „DigitalCameras“ $= Electronics \wedge CamerasandPhoto \wedge DigitalCameras$.
3. Berechne die Relationen für jedes Paar von Markierungen. Dies wird mit Hilfe einer Bibliothek von semantischen Element-Matchern erledigt, die z. T. ähnlich wie in Cupid funktionieren, oder auf WordNet aufbauen.
4. Berechne die Relationen für alle Paare von Knoten zwischen den beiden Bäumen.

Der letzte Schritt wird durch folgenden Algorithmus vollzogen:

```
1  treeMatch(source, target: Tree): array of array of relation =
2  var
3    cLabsMatrix, cNodesMatrix, relMatrix:
       array of array of relation
4    axioms, context1, context2: formula
5    i, j: integer
6  begin
7    cLabsMatrix := fillCLabMatrix(source, target)
8    for each sourceNode in source do
9      i := getNodeId(sourceNode)
10     context1 := getCnodeFormula(sourceNode)
11     for each targetNode in target do
12       j := getNodeId(targetNode)
13       context2 := getCnodeFormula(targetNode)
14       relMatrix := extractRelMatrix(cLabMatrix, sourceNode,
                                       targetNode)
15       axioms := mkAxioms(relMatrix)
16       result[i][j] := nodeMatch(axioms, context1, context2)
17     end for
18   end for
19   return result
20 end
```

$cLabsMatrix$ ist die Matrix, die die Relationen der Markierungen enthält. Sie wird von der Bibliothek der semantischen Element-Matcher gefüllt (Aufruf von $fillCLabMatrix$). Die zwei Schleifen iterieren über alle Paare von Knoten. $getCnodeFormula(x)$ berechnet die Formel für Knoten x nach Schritt 2. In Zeile 14 wird die Relation, die zwischen den Markierungen von $sourceNode$ und $targetNode$ besteht, in die Matrix $relMatrix$ eingetragen.

Um die Relationen zwischen zwei Knoten zu bestimmen, wird eine Hilfsvariable $axioms$ mit Prämissen (Axiomen) erzeugt, die eine Konjunktion aus relevanten Markierungs-Formeln ist, welche durch $relMatrix$ in Beziehung zueinander stehen. Ein Beispiel: Die Knoten *electronics* von Schema A und B sollen aufeinander abgebildet werden. Hierzu sind die folgenden Axiome nötig:
$(Electronics_1 = Electronics_2) \wedge (Cameras_1 = Cameras_2) \wedge$
$(Photo_1 = Photo_2) \wedge (Cameras_1 \supseteq DigitalCameras_2)$

Schließlich wird in Zeile 16 das Match zwischen den Knoten berechnet und in die Ergebnisvariable *result* geschrieben. Dies geschieht wie folgt:

Das Knoten-Match-Problem wird ein Problem aussagenlogischer Erfüllbarkeit transformiert. Um herauszufinden, ob eine Relation gültig ist, müssen die Prämissen, die ihr zugrundeliegen, allgemeingültig sein. Die Formel $axioms \rightarrow rel(context_1, context_2)$ muss für alle Variablenbelegungen wahr sein. Das ist genau dann der Fall, wenn $axioms \wedge \neg rel(context_1, context_2)$ unerfüllbar ist, was mit einem SAT-Solver entschieden werden kann.

Die verschiedenen Relationstypen werden durch entsprechende logische Operatoren ersetzt, was Tabelle 3 zeigt.

Tabelle 3. Abbildung von Relationen auf logische Operatoren

rel(a,b)	Transformation in Aussagenlogik
$a = b$	$a \leftrightarrow b$
$a \subseteq$	$a \rightarrow b$
$a \supseteq$	$b \rightarrow a$
$a \perp b$	$\neg(a \wedge b)$

Der Algorithmus *nodeMatch* macht sich genau diese Ersetzung zunutze und gibt die stärkste Relation zurück, die gültig ist:

nodeMatch(axioms, context1, context2: Formula) =
    if $axioms \wedge \neg(context_1 \leftrightarrow context_2)$ is unsatisfiable then
        return =
    elif $axioms \wedge \neg(context_1 \rightarrow context_2)$ is unsatisfiable then
        return $\subseteq$
    elif $axioms \wedge \neg(context_2 \rightarrow context_1)$ is unsatisfiable then
        return $\supseteq$
    elif $axioms \wedge (context_1 \wedge context_2)$ is unsatisfiable then
        return $\perp$
    else
        return *idk*
    end

Dies ist auch der größte Vorteil von S-Match: Es werden keine Ähnlichkeitskoeffizienten berechnet, sondern der Benutzer bekommt eine klare Relation, eine Beschreibung, *wie* die einzelnen Elemente zusammenhängen.

Das semantische Matching ist ein relativ neues Verfahren, das auch große Schemata schnell vergleichen kann [Giunchiglia05]. Nach den Autoren ist es auch im Punkt Match-Qualität den anderen Ansätzen überlegen.

3 Instanzenbasierte Verfahren

Im Gegensatz zu den schemabasierten Verfahren nutzen instanzenbasierte Verfahren die eigentlichen Datensätze um ein Match zu generieren. Bei den beiden vorgestellten Ansätzen handelt es sich jedoch eigentlich um hybride Verfahren, da sie zumindest die Möglich keiten bieten, Schema-Informationen mit zu verwenden. Sie sind aber im Gegensatz zu den obigen Verfahren nicht von solchen Metainformationen abhängig; sie können prinzipiell auch angewandt werden, wenn gar keine expliziten Schemata vorhanden sind, beispielsweise wenn die Daten im XML-Format vorliegen, aber die DTDs dazu fehlen. Die beiden Verfahren, die hier vorgestellt werden, sind allerdings nicht rein instanzenbasiert und daher auch auf Schemata angewiesen.

3.1 Machine-learning-Ansätze (LSD)

Das System *Learning Source Descriptions* (LSD) benutzt Machine-Learning-Techniken, um das Schema-Matching-Problem für DTD-Schemata zu lösen [Doan01]. DTD sind eine Beschreibungssprache, die dazu dient, XML-Grammatiken aufzustellen. Es wird vorausgesetzt, dass zunächst einmal XML als Bindeglied der verschiedenen Datenquellen benutzt wird.

Ähnlich wie bei Cupid setzt LSD auf einzelne Lern-Algorithmen, die auf Element-Ebene arbeiten und einem Algorithmus, der die einzelnen Lerner kombiniert, dem *Meta-Lerner*.

Das grundsätzlich Vorgehen ist wie folgt: Zunächst wird manuell ein Mapping zwischen einigen Quell-Schemata und dem Ziel-Schema (im DTD-Format) angegeben. Diese Beispiel-Mappings dienen dazu, die verschiedenen Lerner zu trainieren und sie an die Anwendungsdomäne anzupassen. Hierzu werden sowohl Daten aus den Instanzen als auch Daten aus den Schemata (z. B. die Bezeichner der Elemente) extrahiert. Diese Daten werden dazu benutzt, Fallbeispiele zum Trainieren der unterschiedlichen Lerner zu konstruieren und sind daher auf den jeweiligen Lerner angepasst. Anschließend werden die Lerner mithilfe der Fallbeispiele und der Zusatzdaten trainiert. Ist dies geschehen, so wird der Meta-Lerner trainiert. Er ist dafür verantwortlich festzulegen, welchen Anteil die einzelnen Lerner am Mapping haben sollten, d. h. er lernt ihre Gewichtung.

Nach der Trainingsphase wird das System dazu benutzt, neue Schemata auf das Ziel-Schema abzubilden. Dafür sorgt ein *Constraint Handler*, der auf Beschränkungen zurückgreift, um das Mapping-Ergebnis zu verbessern. Diese Constraints sind domänenspezifisch und werden vom Benutzer angegeben.

Sollte das gelieferte Mapping nicht den Anforderungen genügen, kann der Nutzer ein anderes anfordern; er kann auf diese Weise das System beeinflussen. Die gelernten Mappings werden zu den Beispielen hinzugefügt, so dass das System mit der Zeit immer weniger manuelle Mapping-Eingaben benötigt.

Es sei nun das ganze etwas formaler dargestellt: Schema-Matching wird als *Klassifikations-Problem* betrachtet. Für jedes Quell-DTD-Tag soll ein Ziel-DTD-Tag gefunden werden. Es seien die $c_1, \ldots, c_n$ die Tag-Namen des Ziel-DTD. Eine Menge von Trainingsbeispielen ist eine Menge von Tupeln $(x_1, c_{i1}), \ldots, (x_m, c_{im})$, wobei x_j für die Daten des Beispiels steht und c_j für die Klassifizierung, als den Namen des Ziel-Tags.

Es bezeichne weiterhin $s(c_j|x, L)$ den Sicherheits-Koeffizienten („confidence score“), der numerisch ausdrückt, wie sicher sich der Lerner L ist, dass x auf c_j abgebildet werden sollte. Je höher der Sicherheits-Koeffizient ist, desto sicherer ist sich der Lerner seiner Sache. Der Lerner erzeugt also eine Voraussage in Form einer Liste von Sicherheits-Koeffizienten:

$$\langle s(c_1|x, L), \ldots, s(c_n|x, L)\rangle$$

Hierbei gilt, dass sich alle Koeffizienten zu 1 aufsummieren müssen: $\sum_{j=1}^{n} s(c_j|x, L) = 1$.

LSD ist als generisches System ausgelegt, was in diesem Fall heißt, dass es durch zusätzliche Basis-Lerner erweitert werden kann, die dann durch den Meta-

Lerner kombiniert werden. Diese zusätzlichen Basis-Lerner können der Problem-Domäne angepasst sein. Standardmäßig hat LSD vier Lerner im Programm, auf die nun genauer eingegangen wird.

Der Name-Matcher Der Name-Matcher benutzt den Bezeichner eines XML-Elements, um ein Element auf ein anderes abzubilden. Ähnlich wie bei Cupid werden Synonyme berücksichtigt. Der Bezeichner eines Elements e enthält auch die Namen der Knoten auf dem Pfad von der Wurzel bis zu e. Im folgenden Beispiel ist der Bezeichner von „Straße“ also „Kunde Adresse Straße“.

```
<Kunde>
  <Adresse>
    <Straße>...</Straße>
    ...
  </Adresse>
  ...
</Kunde>
```

Der Name-Matcher speichert alle (Quell-Tag, Ziel-Tag) Paare, die aus den Trainingsbeispielen entnommen werden. Für ein XML-Element e kommen die Elemente des Ziel-Schemas infrage, die nur um einen Ähnlichkeits-Wert δ von e abweichen. Die Ähnlichkeit wird mittels *WHIRL* (Word-based Information Representation Language) [Cohen98] berechnet, einem Verfahren, das die Semantik von Wörter erfassen soll. Es wird auch die Häufigkeit von Wörtern berücksichtigt, so dass häufige Wörter eine geringere Gewichtung bekommen. Der Wert δ wird aus den Beispielen berechnet.

Der Name-Matcher liefert für domänen-spezifische oder konkrete (z. B. Preis, Kunde) Namen gute Ergebnisse; für vage oder partielle Namen („Büro“ für „Büro-Telefonnummer“ stehend) funktioniert er schlecht.

Der Content-Matcher Der Content-Matcher benutzt auch Whirl und ist dem Name-Matcher sehr ähnlich. Der Unterschied besteht darin, dass anstelle von Tag-Namen ihr Inhalt auf Instanzenebene verglichen wird. Jedes Trainingsbeispiel wird daher als (Quell-Inhalt, Ziel-Tag) abgespeichert.

Er funkioniert umso besser, je länger die Texte sind, die verglichen werden sollen, oder z. B. bei Farbwerten, die sich eindeutig von anderen Inhalten abgrenzen lassen.

Der Naive Bayes-Klassifikator Ein Naiver-Bayes-Klassifikator basiert auf dem Satz von Bayes, der angibt, wie man aus der Kenntnis der Wahrscheinlichkeiten der Ereignisse $P(X = a)$, $P(Y = b)$ und $P(Y = b|X = a)$ die Wahrscheinlichkeit von $P(X = a|Y = b)$ berechnen kann, also die Wahrscheinlichkeit, dass die Zufallsvariable X den Wert a annimmt, unter der Voraussetzung, dass Y den Wert b angenommen hat:

$$P(X = a|Y = b) = P(X = a) \cdot \frac{P(Y = b|X = a)}{P(Y = b)}$$

Das Klassifikationsproblem entspricht dem Problem herauszufinden, mit welcher Wahrscheinlichkeit ein Dokument repräsentiert durch einen Vektor $\boldsymbol{x} = (x_1, \ldots, x_d)$ im d-dimensionalen Raum in die Klasse c_j einzuordnen ist. Der Vektor $\boldsymbol{x}$ ist hier eine Menge von Wörtern, aus denen der Text aus einer Instanz besteht.

Der Text wird dem Ziel-Tag c zugeordnet, für das $P(c, \boldsymbol{x})$ den maximalen Wert annimmt. Nach obiger Formel ist dies gleichbedeutend mit dem Problem folgende Wahrscheinlichkeiten zu bestimmen:

$P(\boldsymbol{x})$ Dies ist die generelle Auftrittswahrscheinlichkeit der Wörter in $\boldsymbol{x}$. Da sie für alle Klassen c gleich ist, kann sie bei der Berechnung vernachlässigt werden; als Maß für die Klassifikation reicht $P(\boldsymbol{x}|c) \cdot P(c)$ aus.

$P(c)$ Dies ist die Wahrscheinlichkeit, dass ein Text der Klasse c angehört. Sie wird durch die relative Anzahl der Texte, die in den Trainingsdaten als c klassifiziert sind, approximiert.

$P(\boldsymbol{x}|c)$ Dies ist die Wahrscheinlichkeit, dass die Wortmenge x in einem Text auftaucht, von dem bekannt ist, dass er der Klasse c angehört. Wird die Unabhängigkeit der Wörter x_i angenommen, kann $P(\boldsymbol{x}, c)$ als

$$P(\boldsymbol{x}|c) = \prod_{i=1}^{d} P(x_i|c)$$

ausgedrückt werden. Dieser Annahme verdankt der Lerner sein Adjektiv „naiv“, da sie in der Praxis nicht zutrifft.
Die Einzelwahrscheinlichkeiten $P(x_i|c)$ werden als $\frac{n(x_i,c)}{n(c)}$ abgeschätzt. $n(c)$ bezeichnet die Anzahl der relevanten Wörter $\boldsymbol{x}$ in den zu c gehörenden Dokumenten und $n(x_i, c)$ die Anzahl der Vorkommen von x_i in den zu c gehörenden Dokumenten.

Der Naive Bayes-Klassifikator leistet gute Dienste bei Daten, die sich aufgrund von herausstechenden oder oft auftretenden Wörtern unterscheiden lassen.

Der XML-Klassifikator Der XML-Klassifikator baut auf dem Naiven Bayes-Klassifikator auf und erweitert diesen um die Fähigkeit, die Baumstruktur, die XML bietet, zu nutzen, um die Match-Ergebnisse zu verbessern. Wie der Naive Bayes-Lerner betrachtet er die Texte als Menge von Wörtern (*tokens*), die voneinander unabhängig sind und multipliziert ihre Wahrscheinlichkeiten.

Zunächst wird ein Baum erzeugt, der die Struktur der XML-Datei widerspiegelt, als Wurzel des Baumes ist ein Pseudoknoten eingeführt. Die inneren Knoten sind die Tags in der XML-Datei, die Blätter sind die eigentlichen Daten. Anschließend werden die inneren Knoten (Tags) durch ihre Ziel-Tags erzetzt. Ihre Ziel-Tags werden durch die anderen Lernverfahren von LSD ermittelt. Dann werden Tokens generiert: Es gibt *Knoten-Tokens* und *Kanten-Tokens*. Als Knoten-Tokens werden einfach die Markierungen der Knoten (inklusive der inneren Knoten) genommen. Die Kanten-Tokens setzen sich aus den Markierungen der Knoten zusammen, die die Kante verbindet.

Die unterschiedlichen Tokentypen sind nützlich, Unterscheidungen für die Klassifikation zu treffen, die dem Naiven Bayes-Lerner nicht zugänglich sind.

3.2 SEMINT

Das System *Semantic Integrator* (SEMINT) [Li00] benutzt Neuronale Netze, um Ähnlichkeiten zwischen Attributen in verschiedenen Datenbanksystemen zu berechnen. Wie LSD ist es kein rein instanzenbasiertes System, da es auch Informationen aus den Schemata nutzt.

SemInt verfügt über verschiedene Parser, die auf Schemata eines Datenbanksystems spezialisiert sind (z. B. Oracle 7, SQL), um die benötigten Informationen automatisch extrahieren zu können. Zu diesen Informationen gehören:

Die Typen der Attribute SemInt konvertiert die Datenbanktypen in sein eigenes Typen-System, das die folgenden unterscheidet: Character, Zahl, Datum, Rowid (also eine eindeutige Identifikationsnummer) und Raw (sonstige Binärdaten).
Zusätzlich werden Präzision, also die Anzahl gültiger Stellen in einer Zahl, Skalierungsfaktor (Scale), d. h. die Anzahl an Stellen hinter dem Dezimalkomma und Länge vermerkt. Länge bezieht sich für Character auf die Länge der Zeichenkette, für alle anderen Typen hat es keine Bedeutung.

Constraints Hierunter fallen Eigenschaften der Attribute: Ob sie der Primär- oder Fremdschlüssel sind, Wert- und Bereichsbeschränkungen haben, ob NULL ein zulässiger Wert ist und sogar Zugriffsbeschränkungen.

Die Instanzen werden dazu benutzt, um statistische Daten über die Attribute zu ermitteln. Zu diesen gehören Durchschnittswerte, Verteilungen, Varianzen, Muster sowie Minimal- und Maximalwerte. SemInt berechnet diese Werte automatisch, indem es SQL-Anfragen generiert. Für Zeichenketten beziehen sich diese Werte auf die Anzahl von Buchstaben, Ziffern, etc.

Die statistischen Daten erlauben es, Attribute zu unterscheiden, die in ihren Typen und Constraints übereinstimmen, aber sich in ihren konkreten Werten unterscheiden. Es wird nicht direkt mit den Daten der Instanzen gearbeitet, sondern mit ihren Patterns und Verteilungen. Daher ist das System fehlertoleranter (es können auch fehlerbehafte Daten in der Datenbank liegen) und schneller, da für die Statistiken nicht alle Daten, sondern nur ein kleiner Teil benötigt wird.

In SemInt werden die unterschiedlichen Charakteristika (Typ, Constraints, Verteilung), die berücksichtigt werden, um ein Attribut zu beschreiben, *Diskriminatoren* genannt.

Das Klassifikator-Netz Nachdem alle relevanten Daten aus der Datenbank extrahiert wurden, wird mit ihnen ein neuronales Netz trainiert, der *Klassifikator*, der anhand von N Diskriminatoren M *Cluster* zu unterscheiden lernt.

Idealerweise fällt jedes Attribut in eigenes Cluster. In der Praxis ist das aber nicht zu erreichen, da z. B. „Vorname“ und „Nachname“ durch statitistische Werte sich kaum unterscheiden lassen; „Vorname“ und „Nachname“ würden

in denselben Cluster fallen. Clusterbildung hat zudem den Vorteil, das Verfahren schneller zu machen, da beim Mapping mit einem anderen Schema statt P verschiedenen Attributen nur M verschiedene Cluster berücksichtigt werden müssen. Abbildung 4 zeigt das Netz des Klassifikators.

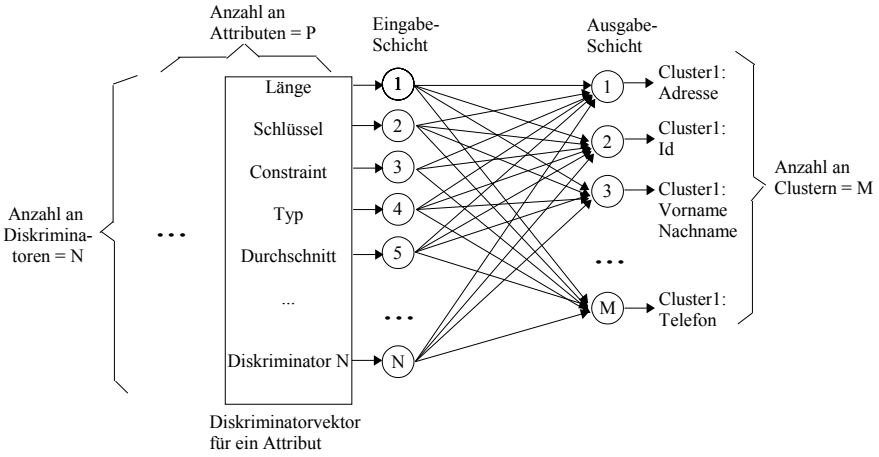

Abbildung 4. Neuronales Netz des Klassifikators

Neuronale Netze sind im Allgemeinen vereinfachte Modelle eines biologischen Nervensystems. Die Knoten sind Nervenzellen, die Neuronen, die mittels Axonen miteinander verbunden sind. Ein Neuron hat mehrere Eingänge und einen Ausgang. In der Natur verarbeitet ein Neuron elektrische Impulse; ab einer bestimmten Schwell-Spannung sendet es selbst einen Impuls. In der mathematischen Modellierung ist jede Eingabe und die Ausgabe eine Zahl von 0 bis 1. Die Eingaben werden gewichtet und aufsummiert.

Damit der Klassifikator die Eingaben verwerten kann, müssen sie normiert werden. Für numerische Werte in einem Intervall $[a, b]$ geschieht dies nach der Formel: $f(x) = 1/(1+k^{-x})$. k wird unabhängig von den Intervall-Grenzen immer auf 1.01 gesetzt, damit man die Werte von verschiedenen Attributen unterscheiden kann. Ein Beispiel: Angenommen man würde für ein Attribut A einfach eine lineare Normalisierung der Form $f(x) = x/max_A$ wählen. Sei $max_A = 100$ und $max_B = 1000$. Dann wären für $a \in A = 10$ und $b \in B = 100$ ihre normalisierten Werte $f(a) = 10/100 = 100/1000 = f(b)$ gleich, obwohl sie ursprünglich verschieden sind. Es käme zu einer falschen Übereinstimmung (false match). Umgekehrt könnte es auch zu einer falsche Ablehnung (false drop) kommen, beispielsweise für $a \in A = 10$ und $b \in B = 10$ wären ihre normalisierten Werte $f(a) = 10/100 \neq 10/1000 = f(b)$.

Für Längenangaben wird die Funktion $f(length) = 2 \cdot (1/(1+k^{-length}) - 0.5)$ benutzt, da Längen keine negativen Werte annehmen können.

Bei Boolsche Werten wird falsch auf 0, wahr auf 1 und Null auf 0.5 abgebildet (falls es vorkommt).

Die fünf Typen, die SemInt kennt, müssen auch als Zahl im Bereich 0-1 kodiert werden. Allerdings kann man nicht einfach Character auf 0, Zahl auf 0.25, Datum auf 0.5, etc. abbilden, da das implizieren würde, dass ein Datum ähnlicher zu einer Zahl ist als zu einem Character, was je nach Anwendungsfall sinnvoll sein kann oder nicht. Solche Ähnlichkeiten sollen gelernt und nicht vorprogrammiert werden. Daher wird ein Typ wird als Fünf-Tupel kodiert: Character ist $(1, 0, 0, 0, 0)$, Zahl ist $(0, 1, 0, 0, 0)$, etc.

Die Datenbank-Parser geben dem Klassifikator für jedes Attribut einen Diskriminatorvektor, der alle normalisierten Daten über das Attribut enthält. Der Diskriminatorvektor in SemInts Implementierung enthält 20 Werte. Ein Attribut wird als Punkt in einem 20-dimensionalen Raum repräsentiert. Der euklidische Abstand zwischen zwei Punkten ist ein Maß für die Ähnlichkeit zwischen den entsprechenden Attributen, der maximale Abstand beträgt $sqrt(20)$, da jede Dimension den Bereich 0-1 umfasst.

Der Klassifikator benutzt den *Self-Organizing Map Algorithm* [Kohonen87] um die Punkte in Cluster einzuteilen. Punkte, die nahe beieinander liegen, werden zu einem Cluster zusammengefasst. Dabei darf der Radius eines Clusters einen bestimmten Maximalwert nicht übersteigen. Dieser Maximalwert wurde empirisch ermittelt und beträgt standardmäßig $\sqrt{20}/10$ (er kann vom Benutzer überschrieben werden).

Die Ausgabe des Klassifikators ist ein Vektor der Größe M, der den Cluster beschreibt, der zu der Eingabe gehört. Cluster 1 entspricht $(1, 0, 0, \ldots, 0)$, Cluster 2 $(0, 1, 0, \ldots, 0)$, etc.

Das Back-Propagation-Netz Mit der Ein- und Ausgabe des Klassifikators wird nun ein Back-Propagation-Netz trainiert. Es handelt sich hierbei um einen *supervised* Lerner, das heißt er benötigt zum Lernen die korrekte Ausgabe.

Der Lernalgorithmus funktioniert wie folgt: Die Gewichte der Kanten sind anfangs zufällige Werte zwischen -0.5 und 0.5, da über die Zwischenschicht jeder Eingabeknoten mit jedem Ausgabeknoten verbunden sind $(N \cdot (N + M)/2 \cdot = (N + M)^2/2$ Kanten), kann jeder Diskriminator einen Beitrag zu jedem Cluster liefern. In der *Forward-Propagation-Phase* wird zu jeder Eingabe die Ausgabe, die sich aus den momentanen Gewichten ergibt, berechnet.

Anschließend werden die Ist-Ausgaben mit den Soll-Ausgaben verglichen, die Differenz bestimmt den Fehler. Dieser Fehlervektor wird nun rückwärts durch das Netz propagiert (Backward Propagation) und dabei werden die Gewichte angepasst. Das ganze wird wiederholt, bis der Fehler klein genug ist, also er unter einem Schwellwert bleibt. Der Schwellwert kann vom Benutzer eingestellt werden. Der Prozess endet also, wenn die Ausgabeknoten für jede Eingabe das richtige Cluster anzeigen.

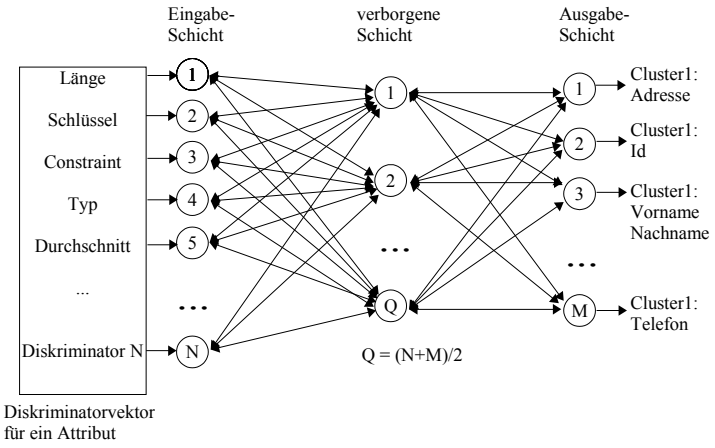

Abbildung 5. Das Back-Propagation Netz

Nach dem Training des Back-Propagation-Netzes kann es benutzt werden, um die Attribute anderer Datenbanken in Cluster einzuteilen. Für jedes Attribut werden die Cluster mit dem höhsten Ähnlichkeitswerten als Mapping vorgeschlagen. Der Benutzer kann hier angeben, wie hoch der Ähnlichkeitswert mindestens sein soll und wie viele Cluster maximal vorgeschlagen werden.

SEMINT hat den großen Vorteil, dass es sich selbst Trainingsdaten generiert; daher ist nur eine minimale Benutzer-Interaktion erforderlich. Ein Nachteil ist, dass das System nur schlecht mit String-Daten umgehen kann, da diese schwer in eine numerische Repräsentation für die neuronalen Netze gebracht werden können.

Wie man sieht, gibt es die unterschiedlichsten Ansätze, das Schema-Matching-Problem zu lösen. Bemerkenswert ist, dass keines der vorgestellten Verfahren auf Schema-Informationen völlig verzichtet.

4 Vergleich der unterschiedlichen Verfahren

Ein Vergleich zwischen den Systemen ist schwierig, da die Verfahren zum Teil auf unterschiedlichen Datenmodellen aufbauen, unterschiedliche Eingaben erfordern und Ausgaben liefern und für unterschiedliche Zwecke eingesetzt wurden.

Das Maß des Vergleichs hier soll die *Arbeitserleichterung* des Benutzers sein. Dabei kommt es zum einen auf Vollständigkeit an: Wurden alle Korrespondenzen gefunden? Der *Recall* ist ein Maß hierfür. Zum anderen kommt es auf die Korrektheit an: Die *Präzision* (Precision) gibt an, wieviele Korrespondenzen gültig waren im Verhältnis zu allen gefundenen.

Sei A die Menge der real-vorkommenden Korrespondenzen und B die der generierten. Die Präzision und der Recall sind dann formal definiert als: $Precision = \frac{|A \cap B|}{|B|}$ und $Recall = \frac{|A \cap B|}{|A|}$. Im Allgemeinen ist der Recall wichtiger, da der Benutzer evtl. nur unter den vorgeschlagenen Korrespondenzen die richtigen auswählt und nicht davon ausgeht, dass andere gültige vom System übersehen wurden. Auf der anderen Seite darf das System nicht zu viele ungültige Korrespondenzen vorschlagen, denn sonst reduziert das die Arbeitserleichterung.

Es ist leicht eine Größe auf Kosten der anderen zu verbessern: Schlägt ein System alle möglichen Korrespondenzen vor, ist der Recall maximal und die Präzision minimal (im Idealfall sind beide Größen 1). Man muss also beide Größen berücksichtigen. Hierfür bietet die Information-Retrieval-Forschung folgende Möglichkeiten [Do02]:

- $F - Measure(\alpha) = \frac{Precision \cdot Recall}{(1-\alpha) \cdot Precision + \alpha \cdot Recall}$, wobei $\alpha \in (0,1)$ angibt, was stärker zu gewichten ist, denn es gilt: $\lim_{\alpha \to 1} F - Measure(\alpha) = Precision$ und $\lim_{\alpha \to 0} F - Measure(\alpha) = Recall$ Wenn Präzision und Recall gleich gewichtet werden ($\alpha = 0.5$), ergibt sich folgende Formel:
- $F - Measure = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$, die das Harmonische Mittel von Präzision und Recall darstellt.
- $Overall = Recall \cdot (2 - \frac{1}{Precision})$; dieses Maß wurde extra für das Problem des Schema Matchings entwickelt [Melnik01]. Es basiert auf der Idee, die Arbeit, die nach dem Matching-Prozess anfällt, zu quantifizieren. Die zugrundeliegende Annahme ist, dass es keine Arbeit kostet ein korrektes Match-Paar zu identifizieren und gleich viel Arbeit kostet ein falsches Match-Paar (falsches Positiv) zu löschen wie ein fehlendes (falsches Negativ) hinzuzufügen, obwohl letzteres in der Regel mehr Zeit in Anspruch nimmt, da man in den unsprünglichen Schemata nachsehen muss.
  Sei $|A| = m$, $|B| = n$ und $|A \cap B| = c$. Es müssen $n - c$ Paare gelöscht und $m - c$ hinzugefügt werden. Bei der manuellen Durchführung eines Matchings müssen m Paare hinzugefügt werden, die Kosten betragen m. Daher ist $\frac{(n-c)+(m-c)}{m}$ ein Maß für die Arbeit, die nach dem automatischen Matching aufgewandt werden muss; $1 - \frac{(n-c)+(m-c)}{m} = \frac{c}{m}(2 - \frac{n}{c} = Recall \cdot (2 - \frac{1}{Precision})$ ist ein Maß für die Arbeitserleichterung. Overall kann negative Werte annehmen, was anschaulich bedeutet, dass man mit der maschinellen „Unterstützung“ mehr Arbeit hat als man ohne sie hätte.

Sowohl F-Measure als auch Overall nehmen für $Precision = Recall = 1$ ihren maximalen Wert (1) an. F-Measure ist ein optimistischeres Maß als Overall [Do02].

Tabelle 4 listet die Durschnittswerte für die Systeme LSD, SEMINT, Similiarity Flooding (SF) und für Cupid.

Die Werte für Cupid stammen nicht von dessen Autoren, sondern aus [Giunchiglia05] und beziehen sich auf andere Test-Schemata, was erklären könnte, warum Cupid bedeutend schlechter bei Overall abschneidet. Beim Overall-Wert von SEMINT wurde berücksichtigt, dass SEMINT nur Cluster von At-

Tabelle 4. Vergleich der Systeme

	Cupid	LSD	S-Match	SEMINT	SF
Precision	0.45	0.8	1.0	0.78	-
Recall	0.6	0.8	0.98	0.86	-
F-Measure	0.51	0.8	0.99	0.81	-
Overall	-0.13	0.6	0.98	0.48	0.6

tributen aufeinander abbildet, daher ist der Wert geringer als bei den anderen Verfahren.

Man sieht, dass S-Match am besten abschneidet. Allerdings wurde es auch nur für zwei Matching-Aufgaben ausführlich getestet. LSD und SF schneiden gleich gut ab. Dies ist bemerkenswert für das Similarity Flooding, da es ein allgemeiner Graphenalgorithmus ist, der nicht dafür entwickelt wurde, mit speziellen Schema-Matching-Verfahren zu konkurieren [Melnik01].

Ein grundlegendes Problem bei Messungen wie dieser ist, dass nur wenige Test-Schemata zur Verfügung stehen; so wurde Cupid mit 2 Match-Aufgaben getestet, LSD mit 20, S-Match mit 2, SEMINT mit 5 und SF mit 9. Leider sind umfangreichere und bessere Benchmarks kaum vorhanden [Do02], mit denen man bestimmter sagen könnte, welcher Ansatz der effektivste ist. Vermutlich schneiden hybride Verfahren, die sowohl Instanzen als auch Schemata betrachten, am besten ab.

Literatur

[Bernstein01] Erhard Rahm, Philip A. Bernstein: A survey of approaches to automatic schema matching The VLDB Journal 10, 2001

[Bernstein04] Philip A. Bernstein, Sergey Melnik, Michalis Petropoulos, Christoph Quix: Industrial-strength schema matching SIGMOD Record, 33(4):38-43, 2004

[Rahm01] Jayant Madhavan, Philip A. Berstein, Erhard Rahm: Generic Schema Matching with Cupid Proceedings of the 27th VLDB Conference, 2001

[Doan01] AnHai Doan, Pedro Domingos, Alon Halevy: Reconciling Schemas of Disparate Data Sources: A Machine-Learning Approach ACM SIGMOD 2001

[Melnik01] Sergey Melnik, Hector Garcia-Molina, Erhard Rahm: Similarity Flooding: A Versatile Graph Matching Algorithm 2001

[Empley04] David W. Embley, Li Xu, Yihong Ding: Automatic Direct and Indirect Schema Mapping: Experiences and Lessons Learned SIGMOD Record, Vol. 33, No. 4, 2004

[Do02] Hong-Hai Do, Sergey Melnik, Erhard Rahm: Comparison of Schema Matching Evaluations Proc GI-Workshop „Web and Databases“ Erfurt, Oktober 2002

[Li00] Wen-Syan Li, Chris Clifton: SEMINT: A tool for identifying attribute correspondences in hererogenous databases using neural networks Data Knowl. Eng. 33(1): 49-84, 2000

[Giunchiglia05] Fausto Giunchiglia, Pavel Shvaiko, Mikalai Yatskevich: Semantic Schema Matching Technical Report DIT-05-014, 2005

[Cohen98] W. Cohen, H. Hirsh: Joins that generalize: Text classification using Whirl Proc. of the Fourth Int. Conf. on Knowledge Discovery and Data Mining (KDD), 1998

[Fellbaum98] Christiane Fellbaum: WordNet: An Electronic Lexical Database Bradford Books, 1998

[Kohonen87] T. Kohonen: Adaptive associative and self-organizing functions in neural computing Appl. Optics 26 4910-4918, 1987