Automatische Datengenerierung aus HTML-Dokumenten

Matthias Hanitzsch

Technische Universität München
hanitzsc@in.tum.de

Zusammenfassung. Die Menge der im Internet verfügbaren Informationen wird immer größer. Damit steigt auch der Bedarf an Werkzeugen, um Daten aus diesen Informationen zu generieren. Wrapper sind Funktionen, die diese Aufgabe übernehmen. Sie durchsuchen Internetseiten und filtern relevante Daten aus diesen heraus. Interessant ist vor allem die maschinelle Erstellung solcher Wrapper. Dies soll an Hand von Beispielseiten, in denen die relevanten Daten markiert sind, erfolgen. Hier werden insgesamt 5 Arten von Wrappern vorgestellt. Dabei wird untersucht, wieviele Beispiele zum Lernen notwendig sind, wieviele Internetseiten sich damit in der Praxis verarbeiten lassen und wie das Laufzeitverhalten der einzelnen Algorithmen ist.

1 Einführung

Im Internet steht eine Vielzahl von Informationen zur Verfügung: Telefonverzeichnisse, Kataloge, Wettervorhersagen, Veranstaltungskalender und vieles mehr. Viele dieser Informationen liegen in HTML (Hyper Text Markup Language) vor. HTML ist eine Auszeichnungssprache, die es ermöglicht, die Struktur von Texten (Überschriften, Absätze, Tabellen usw.) zu beschreiben. Weiterhin lassen sich Bilder, Klänge und ähnliche Dinge in Seiten einbinden. Eines der wichtigsten Merkmale von HTML ist die Möglichkeit von Querverweisen zwischen Seiten. HTML-Seiten sind semistrukturiert und die maschinelle Datengenerierung aus diesen ist deshalb nicht immer einfach. Eine Möglichkeit der Gewinnung von Daten sind sogenannte Wrapper. Das sind Funktionen, die ein HTML-Dokument als Eingabe erhalten und relevante Daten aus diesem extrahieren, ohne jedoch, wie etwa beim Data Mining, nach Beziehungen zwischen Daten zu suchen oder Schlüsse daraus abzuleiten.

Solche Wrapper manuell zu erstellen, birgt aber verschiedene Probleme. Zum einen ist es notwendig, die innere Struktur des HTML-Dokuments genau zu kennen. Zum anderen kann sich die betreffende Internet-Seite recht schnell ändern. Schon kleinste Unterschiede im Layout können bewirken, dass der Wrapper seine Aufgabe nicht mehr erfüllen kann. Er muss also wieder angepasst werden. Das kann mitunter einen recht hohen Zeitaufwand für Entwicklung, Test und Pflege eines solchen Wrappers bedeuten und damit sind manuell erstellte Wrapper in vielen Fällen nicht sehr praktikabel.

Abb. 1. Eine fiktive Internetseite mit Ländern und ihrer Landesvorwahl.

Eine Alternative dazu ist das maschinelle Lernen von Wrappern. Dazu werden Techniken entwickelt, die es ermöglichen an Hand von Beispieldokumenten, in denen die relevanten Daten markiert sind, Wrapper automatisch zu erstellen. Kushmerick [6] beschreibt sechs Klassen von Wrappern, insbesondere die Algorithmen, um diese maschinell zu lernen. Wir wollen vier dieser Klassen untersuchen. Dabei interessiert uns vor allem, wieviele Internetseiten damit in der Praxis verarbeitet werden können, wie aufwendig die Markierung der Beispielseiten und der Vorgang des Lernens an sich ist.

In Abschnitt 2 wird zunächst das Problem der Wrappergenerierung untersucht. In Abschnitt 3 wird dann die Wrapper-Klasse LR zusammen mit den erforderlichen Techniken beschrieben. In den Abschnitten 4 bis 5 werden drei andere Klassen eingeführt, die im Wesentlichen eine Erweiterung der ersten sind. In Abschnitt 6 wird die Qualität der Klassen und der zum Lernen notwendige Aufwand untersucht. Ein etwas anderer Ansatz, der Verbesserungen bringt, wird in Abschnitt 7 beschrieben. Den Abschluss bildet Abschnitt 8, in dem die Erkenntnisse zusammengefasst werden.

2 Das Problem der Wrappergenerierung

Wir wollen zuerst klären, was es bedeutet einen Wrapper maschinell zu lernen. Um das Verständnis zu erleichtern, soll die fiktive Internetseite in Abb. 1 dienen. Die Seite stellt eine Liste von Ländern zusammen mit ihrer Landesvorwahl dar, daneben befindet sich der entsprechende HTML-Code.

Ausgehend von einem relationalem Datenmodell wird jeder Informationsquelle eine Menge von K Attributen zugeordnet. Jedes *Attribut* entspricht einer Spalte in diesem relationalen Modell. Im obigen Beispiel gibt es also $K = 2$ Attribute.

Ein *Tupel* ist ein Vektor $\langle a_1, \ldots, a_K \rangle$ von K Strings. Der String a_j ist dabei der Wert des $j$ten Attributs im Tupel. Die Menge aller Tupel einer Seite wird als *Inhalt* bezeichnet. Zur Repräsentation des Inhalts dienen sogenannte *Label*. Diese stellen den Inhalt in Form von Indizes dar. Für unser Beispiel erhalten wir also das folgendes Label

$$L_{CC} = \left\{ \begin{array}{c} \langle < 50, 55 >, < 63, 66 > \rangle, \\ \langle < 78, 83 >, < 91, 93 > \rangle, \\ \langle < 105, 111 >, < 119, 122 > \rangle, \\ \langle < 134, 139 >, < 147, 149 > \rangle \end{array} \right\}.$$

Die erste Zeile enthält die Indizes für *Congo* und 242.

Die allgemeine Form eines Labels ist

$$L = \left\{ \begin{array}{c} \langle < b_{1,1}, e_{1,1} >, \ldots, < b_{1,k}, e_{1,k} >, \ldots, < b_{1,K}, e_{1,K} > \rangle, \\ \vdots \\ \langle < b_{m,1}, e_{m,1} >, \ldots, < b_{m,k}, e_{m,k} >, \ldots, < b_{m,K}, e_{m,K} > \rangle, \\ \vdots \\ \langle < b_{|L|,1}, e_{|L|,1} >, \ldots, < b_{|L|,k}, e_{|L|,k} >, \ldots, < b_{|L|,K}, e_{|L|,K} > \rangle \end{array} \right\}.$$

Die Seite enthält $|L| > 0$ Tupel, jedes von diesen hat $K > 0$ Attribute. Die Zahlen $1 \leq k \leq K$ sind die Indizes der Attribute, und die Zahlen $1 \leq m \leq |L|$ sind die Indizes der Tupel. Jedes Paar $\langle b_{m.k}, e_{m.k} \rangle$ stellt den Start- und Endindex des $k$ten Attributs im Tupel m dar.

Ein Wrapper sei nun eine Funktion W, die zu einer Seite P das Label L berechnet: $W(P) = L$. Damit besteht das Problem der Wrappergenerierung darin, zu einer gegebenen Menge von Bespielseiten einen Wrapper W zu erzeugen, der alle diese Beispielseiten verarbeiten kann:

EINGABE: Menge $\mathcal{E} = \{\ldots, < P_n, L_n >, \ldots\}$ von Beispielen, wobei P_n eine Seite und L_n das entsprechende Label ist.

AUSGABE: Wrapper W $\in \mathcal{W}$, so dass $\mathrm{W}(P_n) = L_n$ für alle Beispiele $< P_n, L_n >$ aus der Beispielmenge $\mathcal{E}$. $\mathcal{W}$ bezeichnet dabei eine Wrapper-Klasse.

3 Die Wrapper-Klasse LR

Grundlegende Idee bei der Wrapper-Klasse LR (Left Right) ist, dass zu jedem Attribut k ein linker Begrenzer l_k und ein rechter Begrenzer r_k existiert. Die Prozedur Exec_{LR} zeigt die Arbeitsweise eines LR-Wrappers:

```
procedure Exec_LR (wrapper ⟨l_1, r_1, ..., l_K, r_K⟩, page P)
  m ← 0
  while there are more occurences of l_1 in P
    m ← m + 1
    for each ⟨l_k, r_k⟩ ∈ {⟨l_1, r_1⟩, ..., ⟨l_K, r_K⟩}
      scan in P to next occurence of l_k; save position as b_{m,k}
      scan in P to next occurence of r_k; save position as e_{m,k}
  return label {..., ⟨⟨b_{m,1}, e_{m,1}⟩, ..., ⟨b_{m,K}, e_{m,K}⟩⟩, ...}
```

Ein LR-Wrapper ist also ein Vektor $\langle l_1, r_1, \ldots, l_K, r_K \rangle$ von $2K$ Strings, den Begrenzern. Die Prozedur Exec_{LR} durchsucht die Seite P nach diesen Begrenzern, ermittelt so die Indizes der Attribute in den Tupeln und gibt schließlich das entsprechende Label zurück. Zur Generierung eines LR-Wrappers muss also ein passender Vektor von Begrenzern l_k,r_k gefunden werden.

Dazu wird aus der Menge der möglichen Kandidaten ein Begrenzer ausgewählt und auf seine Gültigkeit überprüft. Die Menge $\text{Cands}_l(k, \mathcal{E})$ ist dabei die Menge der Kandidaten für den Begrenzer l_k unter der Beispielmenge $\mathcal{E}$. Diese Menge besteht aus allen Suffixen des kürzesten Strings links des Attributs k in allen Beispielen aus $\mathcal{E}$. Entsprechendes gilt für die Menge $\text{Cands}_r(k, \mathcal{E})$. Sie besteht aus allen Präfixen des kürzesten Strings rechts des Attributs k in allen Beispielen aus $\mathcal{E}$.

Damit ein Kandidat u für einen Begrenzer r_k gültig ist, muss er die folgenden Bedingungen erfüllen:

1. *Bedingung* $\mathcal{C}_r^A$*:* Der String u darf kein Teilstring irgendeines Attributs k in irgendeinem Beispiel sein.
2. *Bedingung* $\mathcal{C}_r^B$*:* Der String u muss ein Präfix des Texts sein, der unmittelbar nach dem Attribut k in jedem Beispiel folgt.

Ist eine dieser Bedingungen verletzt, wird jeder Wrapper, der den Begrenzer $r_k = u$ enthält, zumindest bei einem der Beispiele scheitern. Wird die Bedingung $\mathcal{C}_r^A$ verletzt, dann ist das Attribut k zu kurz. Ist $\mathcal{C}_r^B$ verletzt, dann ist es zu lang.

Ein Kandidat u für einen Begrenzer l_k muss die folgenden Bedingungen erfüllen:

1. *Bedingung* $\mathcal{C}_l^A$*:* Der String u muss ein geeignetes Suffix des Texts sein, der unmittelbar vor jedem Attribut k in jedem Beispiel enthalten ist.
2. *Bedingung* $\mathcal{C}_l^B$*:* Für l_1 darf u kein Teilstring des Seitenrests (nach dem letzten Tupel) in irgendeinem Beispiel sein.

Enthält ein Wrapper den Begrenzer $l_k = u$, so wird er bei zumindest einem Beispiel scheitern. Bei Verletzung der Bedingung $\mathcal{C}_l^A$ wird zumindest einer der Startindizes $b_{m,k}$, die Exec_{LR} berechnet, falsch sein. Ist $\mathcal{C}_l^B$ nicht erfüllt, so wird Exec_{LR} versuchen, zu viele Attribute zu extrahieren.

Wir können jetzt die Prozedur Learn_{LR} zur Generierung eines LR-Wrappers angeben:

```
procedure Learn_LR (examples E)
   for each 1 ≤ k ≤ K
      for each u ∈ Cands_l(k, E):
         if Valid_l(u, k, E) then l_k ← u and terminate this loop
   for each 1 ≤ k ≤ K
      for each u ∈ Cands_r(k, E):
         if Valid_r(u, k, E) then r_k ← u and terminate this loop
 return LR wrapper ⟨l_1, r_1, ..., l_K, r_K⟩
```

Die Prozedur $\text{Valid}_l(u, k, \mathcal{E})$ überprüft hierbei die Gültigkeit der oben angebebenen Bedingungen für einen Begrenzer l_k, und die Prozedur $\text{Valid}_r(u, k, \mathcal{E})$ überprüft diese für einen Begrenzer r_k.

4 Die Wrapper-Klasse HLRT

Wie wir an der Prozedur Exec_{LR} sehen, durchsucht ein LR-Wrapper immer das gesamte Dokument nach Begrenzern. Das erfordert, dass für jedes Attribut k Begrenzer existieren, die im gesamten Dokument eindeutig sind. In unserem Beispiel aus Abb. 1 sind die Landesnamen fettgedruckt dargestellt. Würde jetzt noch mehr fettgedruckter Text, etwa eine Überschrift, hinzukommen, wäre es nicht mehr möglich, die Seite mit einem LR-Wrapper zu verarbeiten. Es gibt keinen l_1 Begrenzer, der zwischen fettgedruckten Landesnamen und anderem fettgedrucktem Text unterscheiden kann.

Die Klasse HLRT (Head Left Right Tail) benutzt zwei weitere Begrenzer, um den Bereich, innerhalb dessen nach Attributen gesucht werden soll, eingrenzen zu können. Der *head*-Begrenzer zeigt den Anfang des Suchbereichs und der *tail*-Begrenzer dessen Ende an. Ein HLRT-Wrapper ist also ein Vektor $\langle h, t, l_1, r_1, \ldots, l_K, r_K \rangle$ von $2K + 2$ Strings.

Die Arbeitsweise eines HLRT-Wrappers, nachfolgend dargestellt durch die Prozedur Exec_{HLRT}, ist ähnlich zu der eines Wrappers der Klasse LR. Hier wird aber zuerst nach dem Begrenzer h gesucht. Erst an dieser Stelle beginnt die Extraktion von Attributen. Abgebrochen wird, wenn der Begrenzer t vor dem nächsten l_1 gefunden ist.

```
procedure Exec_HLRT (wrapper ⟨h, t, l_1, r_1, ..., l_K, r_K⟩, page P)
    m ← 0
    scan in P to next occurence of h
    while the next l_1 in P is before the next t
      m ← m + 1
      for each ⟨l_k, r_k⟩ ∈ {⟨l_1, r_1⟩, ..., ⟨l_K, r_K⟩}
        scan in P to next occurence of l_k; save position as b_{m,k}
        scan in P to next occurence of r_k; save position as e_{m,k}
    return label {..., ⟨⟨b_{m,1}, e_{m,1}⟩, ..., ⟨b_{m,K}, e_{m,K}⟩⟩, ...}
```

Bei der Generierung eines LR-Wrappers konnten die Begrenzer unabhängig voneinander betrachtet werden. Nun beeinflussen sich aber die Begrenzer h, t und l_1. Deshalb ergeben sich in der Prozedur Learn_{HLRT} die drei ineinandergeschachtelten for-Loops. Beim Lernen der anderen Begrenzer kann allerdings auf Learn_{LR} zurückgegriffen werden.

```
procedure Learn_HLRT (examples E)
   ⟨·, r_1, ..., l_K, r_K⟩ ← Learn_LR(E)
   for each u_{l_1} ∈ Cands_l(1, E)
     for each u_h ∈ Cands_h(E)
       for each u_t ∈ Cands_t(E)
         if Valid_{l_1,h,t}(u_{l_1}, u_h, u_t, E) then
           l_1 ← u_{l_1}, h ← u_h, t ← u_t, and terminate all loops
   return HLRT wrapper ⟨h, t, l_1, r_1, ..., l_K, r_K⟩
```

5 Weitere Wrapper-Klassen

Der Vorteil der Klasse HLRT gegenüber LR ist, dass nur noch in einem bestimmten Bereich nach Attributen gesucht wurde. Es gibt noch zwei weitere Klassen von Wrappern, die einen ganz ähnlichen Ansatz verwenden.

Ein OCLR-Wrapper (Open Close Left Right) ist ein Vektor $\langle o, c, l_1, r_1, \ldots, l_K, r_K \rangle$ von $2K+2$ Strings, der wie HLRT zwei weitere Begrenzer benutzt. Mit dem *open*-Begrenzer wird hier der Beginn eines Tupels, und mit dem *close*-Begrenzer dessen Ende angezeigt. Die Prozeduren Exec_{OCLR} und Learn_{OCLR} funktionieren in gleicher Weise wie bei den anderen beiden Klassen.

Die Wrapper-Klasse HOCLRT (Head Open Close Left Right Tail) ist sozusagen die logische Fortsetzung der drei bisherigen Wrapper. Ein HOCLRT-Wrapper ist ein Vektor $\langle h, t, o, c, l_1, r_1, \ldots, l_K, r_K \rangle$ von $2K+4$ Strings und kombiniert die Funktionalität von HLRT und OCLR. Er benutzt *head*- und *tail*-Begrenzer, um den Suchbereich, sowie *open*- und *close*-Begrenzer, um die Tupel einzugrenzen.

6 Bewertung der Wrapper-Klassen

In diesem Abschnitt soll untersucht werden, wie gut die vorgestellten Wrapper-Klassen auf Internetseiten in der Praxis anwendbar sind. Da die Markierung von Beispielseiten einen nicht unerheblichen Arbeitsaufwand bedeuten kann, ist eine andere wichtige Frage, wieviele Beispielseiten im Durchschnitt nötig sind, um einen dieser Wrapper zu lernen. Schliesslich werden wir auch noch die Laufzeit der Algorithmen betrachten.

Um die eben genannten Fragen zu klären, wurden recht umfangreiche Tests durchgeführt. Dazu wurden 448 Internetseiten, die bei www.search.com gelistet waren, herangezogen. Von diesen wurden wiederum 30 Seiten per Zufall ausgewählt. Als nächstes wurden für jede dieser Seiten die Ergebnisse von 10 Anfragen gesammelt und die darin relevanten Daten von Hand markiert. Die Anzahl der Attribute K je Tupel reichte von 2 bis 18.

Nun wurde untersucht, wieviele der 30 Internetseiten von den einzelnen Wrapper-Klassen verarbeitet werden konnten. Dabei wurde folgendes Ergebnis erreicht: LR 16 (53%), HLRT 17 (57%), OCLR 16 (53%), HOCLRT 17 (57%). Insgesamt konnten 18 (60%) der 30 untersuchten Seiten mit den Wrapper-Klassen verarbeitet werden.

Die restlichen 40% der Seiten haben verschiedene Eigenschaften, die eine Verarbeitung unmöglich machten. So stellen zum Beispiel fehlende Attribute ein Problem dar, oder Attribute in unterschiedlicher Reihenfolge. Weiterhin erfordern die Wrapper, dass Attribute eindeutige Begrenzer haben. Auch das war nicht immer der Fall.

Als nächstes wurde die Frage geklärt, wieviele Beispielseiten zum Lernen eines Wrappers nötig sind. In diesem Zusammenhang soll das PAC-Modell (von *probably approximately correct*) vorgestellt werden, das im Bereich des maschinellen Lernens eine wichtige Rolle spielt. Mit diesem Modell wird es möglich,

eine obere Schranke für die notwendige Anzahl von Beispielseiten anzugeben. Wir gehen dabei von einem Zielwrapper $W_T \in \mathcal{W}$ aus. Dieser berechnet für alle Seiten aus der Beispielmenge das entsprechende Label. Ziel ist es nun, einen Wrapper $W \in \mathcal{W}$ zu finden, der W_T so nahe wie möglich kommt. Im besten Fall sollen W und W_T identisch sein.

Als Kriterium dafür dient die Funktion Error. Sie gibt die Wahrscheinlichkeit dafür an, dass der Wrapper W zu einer Seite P ein anderes Label L berechnet als der Zielwrapper W_T:

$$\text{Error}(W) = \text{Prob}[W(P) \neq W_T(P)].$$

Je näher Error(W) an 0 ist, desto näher ist der Wrapper W am Zielwrapper W_T. Wir geben nun einen Parameter $0 < \epsilon < 1$ an und wollen sicherstellen, dass Error(W) $< \epsilon$ gilt, egal wie klein ϵ gewählt wird. Natürlich können wir das nicht garantieren. Die Beispielseiten zum Lernen des Wrappers sind alle zufällig gewählt und können mitunter auch irreführend sein. Auch hier können wir nur mit einer bestimmten Wahrscheinlichkeit sagen, dass die Bedingung Error(W) $< \epsilon$ hält. Dazu wird ein weiterer Parameter $0 < \delta < 1$ eingeführt. Nun wird also verlangt, dass bei gegebenem ϵ und δ der Lernalgorithmus mit der Wahrscheinlichkeit $1 - \delta$ einen Wrapper erzeugt, so dass Error(W) $< \epsilon$ gilt.

Mit diesem Modell lässt sich herleiten, dass die Anzahl der Beispiele der folgenden Ungleichung genügen muss:

$$|\mathcal{E}| > \frac{1}{\epsilon}(\ln|\mathcal{W}| - \ln\delta).$$

Damit ergeben sich bei den einzelnen Wrapper-Klassen für die Anzahl der zum Lernen notwendigen Beispielseiten obere Schranken von einigen Tausend Seiten (abhängig von ϵ und δ). Das ist für die Praxis noch nicht sehr hilfreich und in [5] finden sich Ansätze zur weiteren Verbesserung des Modells. Dennoch ist das PAC-Modell ein wichtiges Hilfsmittel, um die Güte eines Lernalgorithmus zu beurteilen.

In Tests mit den 30 zufällig gewählten Internetseiten hat sich gezeigt, dass im Durchschnitt etwa 2-5 Beispielseiten notwendig sind, um einen Wrapper zu lernen. Das ist durchaus ein zufriedenstellendes Ergebnis. Wären 50, 100 oder mehr Beispiele zum Lernen nötig, dann hätten diese Wrapper kaum mehr praktische Bedeutung. So können aber schon mit wenigen Beispielseiten Wrapper erzeugt werden.

Natürlich ist auch die Laufzeit der Lernalgorithmen interessant. Dazu wurden folgenden Abschätzungen für deren Komplexität ermittelt:

LR :	$O(KM^2\|\mathcal{E}\|^2V^2)$
HLRT :	$O(KM^2\|\mathcal{E}\|^4V^6)$
OCLR :	$O(KM^4\|\mathcal{E}\|^2V^6)$
HOCLRT :	$O(KM^4\|\mathcal{E}\|^4V^{10})$

mit $V = \max_{1 \leq i \leq n} |P_i|$ maximale Länge der Beispielseite, $M = \sum_{i=1}^{n} |L_i|$ Gesamtzahl der Tupel in den Beispielen, jedes Tupel K Attribute.

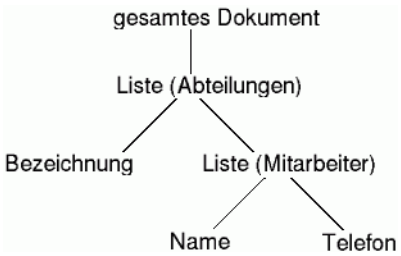

Abb. 2. Struktur einer Seite, die Abteilungen und ihre Mitarbeiter auflistet.

Die Wrapper können also in polynomieller Zeit gelernt werden. Die teilweise recht hohen Grade lassen relativ lange Ausführungszeiten erwarten. Auch die Parameter M und V sind normalerweise recht groß (V z.B. zwischen 899 und 57116). Deshalb verläuft das Lernen in einigen Fällen tatsächlich auch langsam (länger als 15 min). In anderen Fällen wurden aber auch Ausführungszeiten von unter einer Sekunde gemessen. Der Grund dafür ist, dass üblicherweise mehrere gültige Wrapper für eine Seite existieren. Wie schnell ein Wrapper gefunden wird, hängt auch davon ab, in welcher Reihenfolge mögliche Kandidaten für die Begrenzer getestet werden. Dabei ist es für sehr kurze oder sehr lange Kandidaten weniger wahrscheinlich, gültig zu sein. Hier ist es sinnvoll, die Kandidaten so zu ordnen, dass zuerst durchschnittlich lange Kandidaten getestet werden.

7 Ein anderer Ansatz: Stalker

Hier sollen knapp die grundlegenden Ideen von Stalker, einem weiteren Algorithmus zur Wrappergenerierung, beschrieben werden. Eine ausführliche Beschreibung findet sich in [8].

Internetseiten haben oft eine bestimmte hierarchische Struktur: Informationen sind in Form von Listen von Tupeln dargestellt. Diesen Umstand macht sich Stalker zu nutze und arbeitet intern mit einer baumartigen Darstellung der Struktur einer Seite. In dieser Baumstrukur befinden sich relevante Daten in den Blättern. Die inneren Knoten stellen Listen von k-Tupeln dar. Jedes Element eines solchen Tupels kann entweder ein Blatt oder wieder eine Liste sein. In Abb. 6 ist ein Beispiel eines solchen Baumes dargestellt.

Um nun Daten aus einem Dokument extrahieren zu können, benutzt Stalker die Baumstruktur der Seite und eine Menge von Regeln. Für jeden Knoten im Baum benötigt der Wrapper eine Regel, um ihn aus seinem Elternknoten extrahieren zu können. Stellt ein Knoten eine Liste dar, ist weiterhin eine Regel notwendig, mit der sich die Liste in ihre Tupel zerlegen lässt. Für einen gegebenen Strukturbaum und eine Menge von Regeln, besteht die Gewinnung von Daten darin, dem Pfad P von der Wurzel zum jeweiligen Knoten zu folgen und in jedem Schritt die entsprechenden Regeln anzuwenden. Diese Regeln haben die Form SkipTo(...) bzw. SkipUntil(...). Der Grundgedanke ist also, nach

Teilstrings (sogenannten *landmarks*) zu suchen, die eindeutig den Beginn eines Knotens innerhalb seines Elternknotens festlegen.

Zum Lernen der Regeln werden wie bei den anderen Wrappern Beispielseiten benutzt, in denen die relevanten Daten markiert sind. Angenommen wir haben die folgenden vier Beispiele und wollen Regeln lernen, um daraus die Vorwahlnummern zu extrahieren:

E1= Venice, Phone:1-800-555-1515,
E2= Palms, Phone:(818)-508-1570,
E3= LA, Phone:1-888-578-2293,
E4= Watts, Phone:(310) 798-0008.

Der Algorithmus Stalker geht dann so vor, dass er zuerst das kürzeste Beispiel (E2) betrachtet. Das letzte Zeichen vor der Vorwahl ist „(“und es gibt 2 Wildcards dafür: *Punctuation* und *Anything*. Somit erzeugt Stalker die 3 Kandidaten

$R1$ = SkipTo((),
$R2$ = SkipTo(*Punctuation*),
$R3$ = SkipTo(*Anything*).

Stalker benutzt nun eine Reihe von Kriterien, um aus verschiedenen Regeln eine auszuwählen. Hier wird die Regel $R1$ gewählt, da sie für die Beispiele E2 und E4 den Beginn der Vorwahl findet, jedoch für E1 und E3 nicht stoppt. Verbleiben also die Beispiele E1 und E3. Wieder wird nach gleichem Schema eine Regel gesucht. Am Ende liefert der Algorithmus die zwei Regeln

$R1$ = SkipTo((),
$R4$ = SkipTo(–).

Um die Vorwahl in den Beispielen zu finden wird nun entweder $R1$ oder $R4$ angewendet. Die Regeln für das Ende eines Attributs werden in gleicher Weise ermittelt.

Bei Tests hat sich gezeigt, dass Stalker mehr Internetseiten verarbeiten kann als die anderen vorgestellten Wrapper-Klassen. Stalker kann auch mit Seiten umgehen, in denen Attribute fehlen oder in unterschiedlicher Reihenfolge auftauchen. Wie in dem obigen Beispiel gezeigt, ist es auch möglich mehrere (durch *oder* verknüpfte) Regeln für einen Knoten zu haben. Dadurch entfällt die Forderung nach eindeutigen Begrenzern (hier *landmarks*) vor bzw. nach Attributen. Stalker wird damit um einiges flexibler und somit mächtiger.

8 Zusammenfassung

Mit der immer größer werdenden Menge von Informationen im Internet steigt sowohl das Interesse als auch die Notwendigkeit relevantes Material von irrelevantem zu trennen. Ein erster Ansatz waren manuell erstellte Wrapper, die dieses Filtern übernehmen. Da deren Erstellung aufwendig ist und sie bei fast jeder noch so kleinen Änderung der Informationsquelle angepasst werden müssen, wurde nach Wegen gesucht, den Prozess der Wrappergenerierung zu automatisieren.

Die hier vorgestellten Techniken ermöglichen es, mittels markierter Beispielseiten Wrapper automatisch zu generieren. Allerdings kann die Markierung von Beispielseiten durch den Benutzer einen nicht unerheblichen Aufwand bedeuten. Es müssen zuerst einmal genügend Beispiele gesammelt und die darin relevanten Daten markiert werden. Wieviele Beispiele notwendig sind, hängt von der Internetseite selbst und von dem zu verwendenden Wrapper ab. Ändert sich die Seite, muss der zugehörige Wrapper neu generiert werden. Das erfordert die erneute Markierung von Beispielen. Durch die Benutzung grafischer Systeme kann die Markierung zumindest erleichtert werden. [5] beschreibt Techniken, die diese Arbeit zum Teil automatisch erledigen. Hier werden Funktionen benutzt, die eine Zeichenkette als Zahl, Landesname oder ähnliches erkennen.

Das System RoadRunner [3] kommt ganz ohne die Markierung relevanter Daten in Beispielseiten aus. Zur Erzeugung eines Wrappers werden die Gemeinsamkeiten, Ähnlichkeiten und Unterschiede von je zwei Seiten des gleichen Typs untersucht. Unterschiedliche Teilstrings zwischen HTML-Tags bei sonst gleicher Struktur sind dann zum Beispiel ein Hinweis für zu extrahierende Daten. Besonders nützlich ist dieses System bei sehr datenintensiven Quellen. Die HTML-Seiten werden dann fast ausschließlich von Programmen erzeugt. Damit ist die Struktur der Dokumente gleich. Seiten unterscheiden sich dann (fast) nur in den relevanten Daten.

Lixto [2] benutzt den Parsebaum eines HTML-Dokuments und lernt interaktiv mittels einer grafischen Benutzeroberfläche Regeln, um relevante Teile des Dokuments in XML umzuformen. Das erleichtert zudem die Weiterverarbeitung der Daten.

Wichtig in der Praxis ist auch, dass die Menge der Informationen oft zu groß ist, um sie auf einer einzigen Seite übersichtlich darstellen zu können (z.B. ein Telefonverzeichnis). Die Daten sind dann über mehrere Seiten verteilt und der Benutzer kann innerhalb dieser navigieren. Sogenannte Web-Crawler verfolgen selbständig Links und sammeln so die Daten von mehreren Seiten. Auch Lixto bietet diese Möglichkeit [1].

Interessant ist außerdem die Frage nach der Gültigkeit von Wrappern. In [7] wird Rapture beschrieben. Dieser Algorithmus benutzt Länge der Daten, Anteil besonderer Zeichen wie . , : (, um festzustellen, inwiefern die erwarteten und die tatsächlich erhaltenen Daten übereinstimmen. Dieses System findet unter anderem dann Anwendung, wenn sich die Daten oft ändern, ihre Form aber gleich bleibt (zum Beispiel also bei Aktienkursen).

Bei den Lernalgorithmen der vorgestellten Wrapper-Klassen ist der Stellenwert aller Beispielseiten gleich. *Boosting* [4] ist eine Technik, die einen Lernalgorithmus wiederholt auf eine Beispielmenge anwendet. Dabei werden Beispielseiten jedoch unterschiedlich gewichtet, so daß schwierigere Seiten einen höheren Stellenwert erhalten. Dadurch können mit recht einfachen Lernalgorithmen verlässliche Wrapper gelernt werden.

Ein anderes Gebiet für Verbesserungen ist die Ausführungsgeschwindigkeit besonders der Wrapper aus Abschnitt 3 bis 5. Da im allgemeinem mehrere Wrap-

per für eine Seite existieren, kann versucht werden durch Entwicklung entsprechender Heuristiken möglichst schnell einen gültigen Wrapper zu finden.

Es ist auch wichtig darauf hinzuweisen, dass Wrapper in der Praxis auch mit Ausnahmesituationen konfrontiert werden. Nicht gefundene Seiten, lange Wartezeiten und vieles mehr sollten deshalb wenn möglich auch berücksichtigt werden.

Literatur

1. R. Baumgartner, S. Flesca, G. Gottlob. Declarative information extraction, web crawling and recursive wrapping with Lixto. In: *Proceedings 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'2001)*, Band 2173 der *Lecture Notes in Computer Science*, S. 21-41. Springer-Verlag, 2001.
2. R. Baumgartner, S. Flesca, G. Gottlob. Visual web information extraction with Lixto. In: *Proceedings 27th International Conference on Very Large Data Bases (VLDB'2001)*, S. 119-128. Morgan Kaufmann, 2001.
3. V. Crescenzi, G. Mecca, P. Merialdo. RoadRunner: Towards automatic data extraction from large web sites. In: *Proceedings 27th International Conference on Very Large Data Bases (VLDB'2001)*, S. 109-118. Morgan Kaufmann, 2001.
4. D. Freitag, N. Kushmerick. Boosted wrapper induction. In: *Proceedings 17th National Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence (AAAI/IAAI'2000)*, S. 577-583. AAAI Press/The MIT Press, 2000.
5. N. Kushmerick. *Wrapper induction for information extraction.* Dissertation, Department of Computer Science & Engineering, University of Washington, Seattle, WA, 1997.
6. N. Kushmerick. Wrapper induction: Efficiency and expressiveness. *Artificial Intelligence*, 118(1-2):15-68, 1999.
7. N. Kushmerick. Wrapper verification. *World Wide Web*, 3(2):79-94, 2000.
8. I. Muslea, S. Minton, G. A. Knoblock. Hierarchical wrapper induction for semistructured information sources. *Autonomous Agents and Multi-Agent Systems*, 4(1/2):93-114, 2001.