Information extraction from text

Spring 2003, Part 3

Helena Ahonen-Myka

Information extraction from semi-structured text

- IE from Web pages
  - HTML tags, fixed phrases etc. can be used to guide extraction
- IE from other semi-structured data
  - e.g. email messages, rental ads, seminar announcements

2

WHISK

- Soderland: Learning information extraction rules for semi-structured and free text, Machine Learning, 1999

3

Semi-structured text (online rental ad)

Capitol Hill - 1 br twnhme. Fplc D/W W/D. Undrgrnd Pkg incl $675. 3 BR, upper flr of turn of ctry HOME. incl gar, grt N. Hill loc $995. (206) 999-9999
<i> (This ad last ran on 08/03/97.)
 </i> <hr>

4

2 case frames extracted:

- Rental:
  - Neighborhood: Capitol Hill
  - Bedrooms: 1
  - Price: 675
- Rental:
  - Neighborhood: Capitol Hill
  - Bedrooms: 3
  - Price: 995

5

Semi-structured text

- the sample text (rental ad) is not grammatical nor has a rigid structure
  - we cannot use a natural language parser as we did before
  - simple rules that might work for structured text do not work here

6

Rule representation

- WHISK rules are based on a form of regular expression patterns that identify
  - the context of relevant phrases
  - the exact delimiters of the phrases

7

Rule for number of bedrooms and associated price

- ID:: 1
- Pattern:: * (*Digit*) ' BR' * '$' (*Number*)
- Output:: Rental {Bedrooms $1}{Price $2}
  - * : skip any number of characters until the next occurrence of the following term in the pattern (here: the next digit)
  - single quotes: literal -> exact (case insensitive) match
  - *Digit*: a single digit; *Number*: possibly multi-digit

8

Rule for number of bedrooms and associated price

- parentheses (unless within single quotes) indicate a phrase to be extracted
  - the phrase within the first set of parentheses (here: *Digit*) is bound to the variable $1 in the output portion of the rule
- if the entire pattern matches, a case frame is created with slots filled as labeled in the output portion
- if part of the input remains, the rule is re-applied starting from the last character matched before

9

2 case frames extracted:

- Rental:
  - Bedrooms: 1
  - Price: 675
- Rental:
  - Bedrooms: 3
  - Price: 995

10

Disjunction

- The user may define a semantic class
  - a set of terms that are considered to be equivalent
  - *Digit* and *Number* are special semantic classes (built-in in WHISK)
  - user-defined class: *Bdrm* = (brs|br|bds|bdrm|bd|bedrooms|bedroom|bed)
  - a set does not have to be complete or perfectly correct: still it may help WHISK to generalize rules

11

Rule for neighborhood, number of bedrooms and associated price

- ID:: 2
- Pattern:: *(*Nghbr*) *(*Digit*) ' ' *Bdrm* * '$' (*Number*)
- Output:: Rental {Neighborhood $1} {Bedrooms $2}{Price $3}
  - assuming the semantic classes *Nghbr* (neighborhood names for the city) and *Bdrm*

12

IE from Web

- information agents
- extraction rules: wrappers
- learning of extraction rules: wrapper induction
- wrapper maintenance

- active learning
- unsupervised learning

13

Information agents

- data is extracted from a web site and transformed into structured format (database records, XML documents)
- the resulting structured data can then be used to build new applications without having to deal with unstructured data
  - e.g., price comparisons
- challenges:
  - thousands of changing heterogeneous sources
  - scalability: speed is important -> no complex processing possible

14

What is a wrapper?

- a **wrapper** is a piece of software that can translate an HTML document into a structured form (~database tuple)
- critical problem:
  - How to define a set of extraction rules that precisely define how to locate the information on the page?
- for any item to be extracted, one needs an extraction rule to locate both the beginning and end of the item
  - extraction rules should work for all of the pages in the source

15

Learning extraction rules: wrapper induction

- adaptive IE
- learning from examples
  - manually tagged: it is easier to annotate examples than write extraction rules
  - how to minimize the amount of tagging or entirely eliminate it?
    - active learning
    - unsupervised learning

16

Wrapper induction system

- input: a set of web pages labeled with examples of the data to be extracted
  - the user provides the initial set of labeled examples
  - the system can suggest additional pages for the user to label
- output: a set of extraction rules that describe how to locate the desired information on a web page

17

Wrapper induction system

- after the system creates a wrapper, the wrapper verification system uses the wrapper to learn patterns that describe the data being extracted
  - if a change is detected, the system can automatically repair a wrapper by
    - using the same patterns to locate examples on the changed pages and
    - re-running the wrapper induction system

18

Wrapper induction methods

- Kushmerick et al: LR and HLRT wrapper classes
- Knoblock et al: STALKER

19

Wrapper classes LR and HLRT

- Kushmerick, Weld, Doorenbos: Wrapper induction for information extraction, IJCAI –97
- Kushmerick: Wrapper induction: Efficiency and expressiveness, Workshop on AI & Information integration, AAAI-98

20

LR (left-right) class

- a wrapper consists of a sequence of delimiter strings for finding the desired content
- in the simplest case, the content is arranged in a tabular format with *K* columns
- the wrapper scans for a pair of delimiters for each column
  - total of *2K* delimiters

21

LR wrapper induction

- the wrapper construction problem:
  - input: example pages
- associated with each information resource is a set of *K* attributes, each representing a column in the relational model
- a tuple is a vector $\langle A_1, \ldots, A_K \rangle$ of *K* strings
  - string A_k is the value of tuple's k^{th} attribute
  - tuples represent rows in the relational model
- the label of a page is the set of tuples it contains

22

Example: country codes

<HTML><TITLE>Some Country Codes</TITLE>

<BODY>

<B>Congo</B> <I>242</I>

<B>Egypt</B> <I>20</I>

<B>Belize</B> <I>501</I>

<B>Spain</B> <I>34</I>

<HR></BODY></HTML>

23

Label of the example page

{<Congo, 242>,

<Egypt, 20>,

<Belize, 501>,

<Spain, 34>}

24

Execution of the wrapper: procedure ccwrap_LR

- 1. scan for the string l_1=<B> from the beginning of the document
- 2. scan ahead until the next occurrence of r_1=</B>
- 3. extract the text between these positions as the value of the 1st column of the 1st row
- 4. similarly: scan for l_2=<I> and r_2=</I> and extract the text between these positions as the value of the 2nd column of the 1st row
- 5. the process starts over again and terminates when l_1 is missing (= end of document)

25

ccwrap_LR (page *P*)

while there are more occurrences in *P* of <B>

for each ⟨ l_k, r_k ⟩ in {⟨<B>,</B>⟩, ⟨<I>,</I>⟩}

scan in *P* to next occurrence of l_k

save position as start of k^{th} attribute

scan in *P* to next occurrence of r_k

save position as end of k^{th} attribute

return extracted pairs {..., ⟨ country, code ⟩, ...}

General template

- generalization of ccwrap_LR:
  - delimiters can be arbitrary strings
  - any number K of attributes
- the values $l_1, \ldots, l_K$ indicate the left-hand attribute delimiters
- the values $r_1, \ldots, r_K$ indicate the right-hand delimiters

27

executeLR (⟨ l_1,r_1⟩,..., ⟨ l_K,r_K⟩, page *P*)

while there are more occurrences in *P* of l_1

for each ⟨ l_k, r_k ⟩ in {⟨ l_1,r_1 ⟩,..., ⟨ l_K,r_K⟩}

scan in *P* to next occurrence of l_k

save position as start of next value A_k

scan in *P* to next occurrence of r_k

save position as end of next value A_k

return extracted tuples {..., ⟨ A_1,...,A_K ⟩ , ...}

LR wrapper induction

- the behavior of ccwrap_LR can be entirely described in terms of four strings <B>,</B>,<I>,</I>
- the LR wrapper induction problem thus becomes one of identifying $2K$ delimiter strings $l_1, r_1, \ldots, l_K, r_K$ on the basis of a set $E = \{\ldots, P_n, L_n, \ldots\}$ of examples

29

LR wrapper induction

- LR learning is efficient:
  - the algorithm enumerates over potential values for each delimiter
    - selects the first that satisfies a constraint that guarantees that the wrapper will work correctly on the training data
  - the $2K$ delimiters can all be learned independently

30

Limitations of LR classes

- an LR wrapper requires a value for l_1 that reliably indicates the beginning of the 1st attribute
  - this kind of delimiter may not be available
  - what if a page contains some bold text in the top that is not a country?
  - it is possible that no LR wrapper exists which extracts the correct information
  - -> more expressive wrapper classes

HLRT (head-left-right-tail) class of wrappers

```
<HTML><TITLE>Some Country Codes</TITLE>
<BODY> <B>Country Code List</B> <P>
<B>Congo</B> <I>242</I><BR>
<B>Egypt</B> <I>20</I><BR>
<B>Belize</B> <I>501</I><BR>
<B>Spain</B> <I>34</I><BR>
<HR> <B>End</B> </BODY></HTML>
```


HLRT class of wrappers

- HLRT (head-left-right-tail) class uses two additional delimiters to skip over potentially confusing text in either the head (top) or tail (bottom) of the page
  - head delimiter *h*
  - tail delimiter *t*
- in the example, a head delimiter *h*=<P> could be used to skip over the initial <B> at the top of the document
-> l_1 = <B> would work correctly

HLRT wrapper

```
<HTML><TITLE>Some Country Codes</TITLE>
<BODY><B>Country Code List</B><P>
<B>Congo</B> <I>242</I><BR>
<B>Egypt</B> <I>20</I><BR>
<B>Belize</B> <I>501</I><BR>
<B>Spain</B> <I>34</I><BR>
<HR><B>End</B> </BODY></HTML>
```


HLRT wrapper

- labeled examples:
  - <Congo,242>, <Egypt,20>, <Belize,501>, <Spain,34>


```
ccwrap_HLRT (page P)
  skip past first occurrence of <P> in P
  while next <B> is before next <HR> in P
     for each ⟨ l_k, r_k ⟩ in {⟨ <B>, </B> ⟩, ⟨ <I>, </I> ⟩ }
        skip past next occurrence of l_k in P
        extract attribute from P to next occurrence of r_k
  return extracted tuples
```

executeHLRT ($\langle h, t, l_1, r_1, \ldots, l_K, r_K \rangle$, page P)

skip past first occurrence of h in P

while next l_1 is before next t in P

  for each $\langle l_k, r_k \rangle$ in $\{\langle l_1, r_1 \rangle, \ldots, \langle l_K, r_K \rangle\}$

    skip past next occurrence of l_k in P

    extract attribute from P to next occurrence of r_k

return extracted tuples

HLRT wrapper induction

- task: how to find the parameters $h, t, l_1, r_1, \ldots, l_K, r_K$?
- input: a set $E = \{\ldots, \langle P_n, L_n \rangle, \ldots\}$ of examples, where each P_n is a page and each L_n is a label of P_n
- output: a wrapper W such that $W(P_n) = L_n$ for every $\langle P_n, L_n \rangle$ in E

38

BuildHLRT(labeled pages $E = \{\ldots, \langle P_n, L_n \rangle, \ldots\}$)

for $k = 1$ to K

  r_k = any common prefix of the strings following each (but not contained in any) attribute k

for $k = 2$ to K

  l_k = any common suffix of the strings preceding each attribute k

for each common suffix l_1 of the pages' heads

 for each common substring h of the pages' heads

  for each common substring t of the pages' tails

   if (a) h precedes l_1 in each of the pages' heads; and

    (b) t precedes l_1 in each of the pages' tails; and

    (c) t occurs between h and l_1 in no page's head;

    (d) l_1 doesn't follow t in any inter-tuple separator

   then return <$h, t, l_1, r_1, \ldots, l_K, r_K$>

Problems

- missing attributes
- multi-valued attributes
- multiple attribute orderings
- disjunctive delimiters
- nonexistent delimiters
- typographical errors and exceptions
- sequential delimiters
- hierarchically organized data

40

Problems

- Missing attributes
  - complicated pages may involve missing or null attribute values
  - if the corresponding delimiters are missing, a simple wrapper will not process the remainder of the page correctly
  - a French e-commerce site might only specify the country in addresses outside France
- Multi-valued attributes
  - a hotel guide might list the cities served by a particular chain, instead of giving <chain, city> pairs for each city

41

Problems

- Multiple attribute orderings
  - a movie site might list the release date before the title for movies prior to 2003, but after the title for recent movies
- Disjunctive delimiters
  - the same attribute might have several possible delimiters
  - an e-commerce site might list prices with a bold face, except that discount prices are rendered in red

42

Problems

- Nonexistent delimiters
  - the simple wrappers assume that some irrelevant background tokens separate the content to be extracted
  - this assumption may be violated
  - e.g. how can the department code be separated from the course number in strings such as COMP4016 and GEOL2001?
- Typographical errors and exceptions
  - errors may occur in the delimiters
  - even a small badly formatted part may make a simple wrapper to fail on entire page

43

Problems

- Sequential delimiters
  - the simple wrappers assumed a single delimiter per attribute
  - it might be better to scan for several delimiters in sequence
  - e.g. to extract the name of a restaurant from a review, it might be simpler to scan for <B>, then to scan for <BIG> from that position, and finally to scan for <FONT>, rather than to force the wrapper to find a single delimiter
- Hierarchically organized data
  - an attribute could be an embedded table

44

STALKER

- hierarchical wrapper induction
- Muslea, Minton, Knoblock:
  A Hierarchical approach to wrapper induction

45

STALKER

- a page is a tree-like structure
  - leaves are the items that are to be extracted
  - internal nodes represent lists of *k*-tuples
  - each item in a tuple can be either a leaf or another list (= embedded list)
- a wrapper can extract any leaf by determining the path from the root to the corresponding leaf

46

Tokenization of text

- a document is a sequence of tokens
  - words (strings)
  - numbers
  - HTML tags
  - punctuation symbols
- token classes generalize tokens:
  - Numeric, AlphaNumeric, Alphabetic, Word
  - AllCaps, Capitalized
  - HtmlTag
  - Symbol
  - also: user-defined classes

47

1: <p> Name: Yala <p> Cuisine: Thai<p><i>

2: 4000 Colfax, Phoenix, AZ 85258 (602) 508-1570

3: </i>
 <i>

4: 523 Vernon, Las Vegas, NV 89104 (702) 578-2293

5: </i>
 <i>

6: 403 Pico, LA, CA 90007 (213) 798-0008

7: </i>

Extraction rules

- the extraction rules are based on **landmarks** (= groups of consecutive tokens)
  - landmarks enable a wrapper to locate the content of an item within the content of its parent
- e.g. identify the beginning of the restaurant name:
  - R1 = SkipTo()
  - start from the beginning of the parent (= whole document) and skip everything until you find the landmark

49

Extraction rules

- the effect of applying R1 consists of consuming the prefix of the parent, which ends at the beginning of the restaurant's name
- similarly: the end of a node's content
  - R2 = SkipTo()
  - R2 is applied from the end of the documents towards its beginning
  - R2 consumes the suffix of the parent

50

Extraction rules

- R1: a start rule; R2: an end rule
- the rules are not unique, e.g., R1 can be replaced by the rules
  - R3 = SkipTo(Name) SkipTo()
  - R4 = SkipTo(Name *Symbol HtmlTag*)
- these rules **match correctly**
  - start rules SkipTo(:) and SkipTo(<i>) would **match incorrectly**
  - start rule SkipTo(<table>) would **fail**

51

Disjunctive rules

- extraction rules allow the use of disjunctions
- e.g. if the names of the recommended restaurants appear in bold, but the other in italics, all the names can be extracted using the rules
  - start rule: either SkipTo() or SkipTo(<i>)
  - end rule: either SkipTo() or SkipTo(Cuisine) SkipTo(</i>)
- a disjunctive rule matches if at least one of its disjuncts matches

52

Extracting list items

- e.g. the wrapper has to extract all the area codes from the sample document
- the agent starts by extracting the entire list of addresses LIST(Addresses):
  - start rule: SkipTo(<p><i>) and
  - end rule: SkipTo(</i>)

53

Extracting list items

- the wrapper has to iterate through the content of LIST(Addresses) and to break it into individual addresses
  - in order to find the start of each address, the wrapper repeatedly applies a start rule SkipTo(<i>)
  - each successive rule-matching starts where the previous one ended
  - similarly the end of each address: end rule SkipTo(</i>)
  - three addresses found: lines 2, 4, and 6
- the wrapper applies to each address the area-code start rule SkipTo('(') and end rule SkipTo(')')

54

More difficult extractions

- instead of area codes, assume the wrapper has to extract ZIP codes
  - e.g. '85258' from 'AZ 85258'
- list extraction and list iteration remain unchanged
- ZIP code extraction is more difficult, because there is no landmark that separates the state from the ZIP code
- SkipTo rules are not expressive enough, but they can be extended to a more powerful extraction language

55

More difficult extractions

- e.g., we can use either the rule
  - R5 = SkipTo(,) SkipUntil(*Numeric*), or
  - R6 = SkipTo(*AllCaps*) NextLandmark(*Numeric*)
- R5: "ignore all tokens until you find the landmark ',', and then ignore everything until you find, but do not consume, a number"
- R6: "ignore all tokens until you encounter an AllCaps word, and make sure that the next landmark is a number"

56

Advantages of STALKER rules

- nesting is possible
  - hierarchical extraction allows to wrap information sources that have arbitrary many levels of embedded data
- free ordering of items
  - as each node is extracted independently of its siblings, also documents that have missing items or items appearing in various orders can be processed

57

Landmarks and landmark automata

- each argument of a SkipTo() function is a landmark
- a group of SkipTo()s represents a landmark automaton
  - a group must be applied in a pre-established order
  - = extraction rules are landmark automata
- a linear landmark = a sequence of tokens and wildcards
  - a wildcard = a class of tokens (*Numeric, HtmlTag*...)

58

Landmark automaton

- a landmark automaton *LA* is a nondeterministic finite automaton with the following properties:
  - the initial state s_0 has a branching-factor of k
  - exactly k accepting states (one/branch)
  - all k branches that leave s_0 are sequential *LA*s
    - from each non-accepting state S_i there are exactly two possible transitions: a loop to itself, and a transition to the next state
  - linear landmarks label each non-looping transition
  - all looping transitions have the meaning "consume all tokens until you encounter the linear landmark that leads to the next state"

59

Learning extraction rules

- input: a set of sequences of tokens that represent the prefixes that must be consumed by the new rule
- the user has to
  - select a few sample pages
  - use a graphical user interface (GUI) to mark up the relevant data
- GUI generates the input format

60

The user has marked up the area codes:

E1: 513 Pico, Venice, Phone: 1-800-555-1515

E2: 90 Colfax, Palms, Phone: (818) 508-1570

E3: 523 1st St., LA , Phone: 1-888-578-2293

E4: 403 Vernon, Watts , Phone: (310) 798-0008

Training examples: the prefixes of the addresses that end immediately before the area code (underlined)

Learning algorithm

- STALKER uses sequential covering
  - begins by generating a linear *LA* that covers as many as possible of the 4 positive examples
  - tries to create another linear *LA* for the remaining examples, and so on
  - once all examples are covered, the disjunction of all the learned *LA*s is returned

Learning algorithm

- the algorithm tries to learn a minimal number of **perfect disjuncts** that cover all examples
- a perfect disjunct is a rule that
  - covers at least one training example and
  - on any example the rule matches, it produces the correct result

Learning algorithm; example

- the algorithm generates first
  - the rule R1 = SkipTo('('), which
    - accepts the positive examples E2 and E4
    - rejects both E1 and E3, because R1 cannot be matched on them
- 2nd iteration:
  - only the uncovered examples E1 and E3 are considered
  - rule R2 = SkipTo(Phone) SkipTo(<b>)
- rule "either R1 or R2" is returned

STALKER (Examples)
  Let RetVal = ∅ (a set of rules)
  While Examples ≠ ∅
    aDisjunct = **LearnDisjunct**(Examples)
    remove all examples covered by aDisjunct
    add aDisjunct to RetVal
  return RetVal

LearnDisjunct (Examples)
  Terminals = Wildcards ∪ **GetTokens** (Examples)
  Candidates = **GetInitialCandidates** (Examples)
  While Candidates ≠ ∅ Do
    Let D = **BestDisjunct** (Candidates)
    If D is a perfect disjunct Then return D
    For each t in Terminals Do
      Candidates = Candidates ∪ **Refine**(D, t)
    remove D from Candidates
  return best disjunct

LearnDisjunct

- GetTokens
  - returns all tokens that appear at least once in each training example
- GetInitialCandidates
  - returns one candidate for each token that ends a prefix in the examples, and
  - one candidate for each wildcard that matches such a token

67

LearnDisjunct

- BestDisjunct
  - returns a disjunct that accepts the largest number of positive examples
  - if there are many, returns the one that accepts fewer false positives
- Refine
  - landmark refinements: make landmarks more specific
  - topology refinements: add new states in the automaton

68

Refinements

- a refining terminal t: a token or a wildcard
- landmark refinement
  - makes a landmark l more specific by concatenating t either at the beginning or at the end of l
- topology refinement
  - adds a new state S and leaves the existing landmarks unchanged
  - if a disjunct has a transition from A to B labeled by a landmark l ($A \rightarrow^{l} B$), then the topology refinement creates two new disjuncts in which the transition is replaced either by $A \rightarrow^{l} S \rightarrow^{t} B$ or by $A \rightarrow^{t} S \rightarrow^{l} B$

69

Example

- 1st iteration: LearnDisjunct() generates 4 initial candidates
  - one for each token that ends a prefix (in R1 and R2)
  - one for each wildcard that matches such a token (in R3 and R4)
  - R1 is a perfect disjunct -> LearnDisjunct() returns R1 and 1st iteration ends

70

Example

- 2nd iteration: LearnDisjunct() is invoked with the uncovered training examples E1 and E3
  - computes the set of refining terminals
    - {Phone   : , . *HtmlTag Word Symbol*}
  - generates the initial candidate rules R5 and R6
  - both candidates accept the same false positives -> refinement is needed

71

Example

- 2nd iteration continues: LearnDisjunct()
  - selects randomly the rule to be refined: R5
  - refines R5: topological refinements R7, ..., R16 and landmark refinements R17 and R18
  - R7 is a perfect disjunct
  - returns rule "either R1 or R7"

72

Wrapper maintenance

- information agents have no control over the sources from which they extract data
- the wrappers rely on the details of the formatting of a page
  - if the source modifies the formatting, the wrapper will fail
- two challenges
  - wrapper verification
  - wrapper re-induction

73

Wrapper verification

- determine whether the wrapper is still operating correctly
- problem:
  - either the formatting (delimiters) or the content to be extracted may have changed
  - the verification algorithm should be able to distinguish between these two
  - e.g. agent checks the Microsoft stock price three times at a stock-quote server:
    - values: +3.10, -0.61, <b><IMG src="advert.gif" />
    - How to know that the first two are OK, but the third probably indicates a defective wrapper?

74

Wrapper verification

- possible solution:
  - the algorithm learns a probabilistic model of the data extracted by wrapper during a period when it is knowing to be operating correctly
  - model captures various properties of the training data: length or fraction of numeric characters of the extracted data
  - to verify afterwards, the extracted data is evaluated against the learned model to estimate the probability that the wrapper is operating correctly

75

Wrapper re-induction

- learning a revised wrapper
- possible solution:
  - after the wrapper verification algorithm notices that the wrapper is broken, the learned model is used to identify probable target fragments in the new and unannotated documents
  - this training data is then post-processed to remove noise, and the data is given to a wrapper induction algorithm

76

What about XML?

- XML does not eliminate the need for Web IE
  - there will still be numerous old sites that will never export their data in XML
  - different sites may still use different document structures
    - person's name can be one element or two elements (first name, family name)
  - different information agents may have different needs (e.g. the price with or without the currency symbol)

77