Scalable Attribute-Value Extraction from Semi-Structured Text

Submitted for Blind Review

Abstract

This paper describes a general methodology for extracting attribute-value pairs from web pages. Attribute-value extraction occurs in two phases: candidate generation, in which syntactically likely attribute-value pairs are annotated by scanning for common mark-up structures; and candidate filtering, in which semantically improbable annotations are removed. We describe three types of candidate generators and two types of candidate filtering techniques, all of which are designed to be massively parallelizable and to scale to the entire Web if desired. The best generation and filtering combination in our experiments achieves 70% F-measure on a hand-annotated corpus of 258 web pages.

1 Introduction

Lists of attribute-value pairs are a widespread and traditional way of displaying information in many fields, including artificial intelligence [17] and linguistics [15]. A simple example is given in Figure 1, which contains information about Pittsburgh, Pennsylvania. The attributes are the labels in the left-hand column, and the values are the corresponding entries in the right-hand column. Other typical data records that use attribute-value pairs include the specifications of many manufactured products (e.g., *"Hard Drive: 160 GB, Weight: 3.4 lb"*), and lists of author, publisher, year values in bibliographic records.

Many formal and structured data representation methods use attribute-value pairs to specify information about classes or objects. Much data in relational tables can usefully be thought of as attribute-value data, the attributes being specified as the columns in a database schema, and the values being the entries in individual table cells. In a "subject-verb-object" model of entities (seen recently in triple-stores such as the RDF graphs of the Semantic Web [5], traceable also to Aristotle's *Categories*), the verbs correspond to attributes and the objects to values.

If this information is well-organized, it can be used to serve many information needs: for example, given the attribute values in Figure 1, and relatively simple processing of numerical attributes, a query for *"cities in Pennsylvania founded before 1800"* or *"cities with population greater than 300,000"* could be matched exactly against Pittsburgh, whereas with keywords alone this match would be at best approximate.

Figure 1. Attributes of the City of Pittsburgh

Country	United States
Commonwealth	Pennsylvania
Founded	November 25, 1758
Population (2000)	334,563

Source: Wikipedia

In this paper, we present a general framework for extracting attribute-value pairs from web pages. Specifically, we restrict our attention to attribute-value pairs that are expressed in structural contexts such as tables and colon-delimited pairs. The main motivation is that a large number of attribute-value pairs that exist on the Web are encoded in such formats, and identifying these formats is relatively straightforward. On the other hand, since structural clues only provide weak indications of the presence of attribute-value pairs, a separate *candidate filtering* step is needed to identify attribute-value pairs that are semantically probable. We describe two approaches to candidate filtering. First, we use attribute whitelists to identify structural contexts that are rich in *known* attributes. Second, we treat candidate filtering as a binary classification task, and use the passive-aggressive algorithm [7] to generalize across previously unseen attributes. All methods described in this paper are designed to be massively parallelizable and to scale to the entire Web if desired. On a hand-annotated corpus of 258 web pages, our best candidate generation and filtering techniques achieve up to 70% F-score and up to 92% precision. Given a corpus of approximately 100 million web pages, our techniques are able to extract 1 billion attribute-value pairs from half of the documents at 70–75% precision.

2 Related Work

Information extraction (particularly relation extraction) using fixed lexico-syntactic patterns to generate candidates is a well-established technique in NLP, often traced to the work of Hearst [9], and it was not long before such methods

for candidate generation became combined with statistical analysis to help classify the candidates generated [10, 4, 2].

Most such work has been devoted to the acquisition of WordNet-style relations between pairs of concepts. Work specifically directed towards extracting attributes of concepts was performed by Poesio and Almuhareb [14]. Their system generates candidates using the pattern *"the X of the Y (is Z)"*, the hypothesis being that X is an attribute of the concept described by the noun phrase Y, and Z, if it appears, is the corresponding value. This pattern also generates many examples that are not attributes, and like ours leads to the problem of candidate filtering. However, because we focus on colon-delimited and table structures in web pages, the false positives are less to do with linguistic variation, which is almost boundless, and more to do with formatting opportunism.

The past few years have seen a surge of interest in open-domain information extraction, in which *unrestricted* relational tuples are extracted from heterogeneous corpora with minimal human intervention. Shinyama and Sekine [16] described an approach that involves clustering of documents into topics. Within each cluster, named-entity recognition, co-reference resolution and syntactic parsing are performed in order to identify relations between entities. Since their work requires document clustering and deep linguistic analysis, it is difficulit to apply in the Web scale.

TextRunner [1] is a system for extracting open-domain relational tuples from free text. It employs a two-phase approach that combines candidate generation and filtering. For candidate generation, a noun-phrase chunker is used to locate plausible entities and relations. Then a Naive-Bayes classifier is used for candidate filtering, followed by additional filtering based on redundancy of tuples across the entire test corpus. The Naive-Bayes classifier is trained using documents that have been automatically labeled with tuples using pattern-based heuristics. Compared to previous methods, TextRunner runs considerably faster, since the required linguistic analysis is relatively lightweight (e.g. part-of-speech tagging). Like our method, it is scalable and can be massively parallelized. The main difference is that our method considers attribute-value pairs occurring in structural contexts, not free text. Because these contexts are both rich but are largely disjoint, our techniques should serve as a useful complement to systems like TextRunner.

More recently, Pasca and Van Durme [12] presented a method for extracting open-domain entity classes, along with their associated class attributes, from both web pages and query logs. It exploits the natural tendency of search engine queries to be simple noun phrases such as *"cast selection for kill bill"*, where class attributes (*"cast selection"*) are coupled with class instances (*"kill bill"*). Compared to our method, their system is mainly focused on ontology building and does not annotate instances of entity-attribute pairs in web pages. Our method also considers attribute-value pairs (e.g. *"Cast: Uma Thurman"*), not entity-attribute pairs.

There has been much work on the extraction of data from highly structured text such as automatically generated web pages. Most learning algorithms take advantage of the underlying mark-up structures of such text and learn string- or tree-based patterns (called *wrappers*) given a small set of training documents. Examples of wrapper induction methods include Stalker [11], BWI [8], and WL^2 [6]. While the learned wrappers are often highly site-specific, similar techniques have been used to extract large-scale knowledge bases from the Web. An example is Grazer [18], where a large number of site-specific wrappers are learned through bootstrapping. Starting with a set of initial facts (entity-attribute-value triples) scraped from Wikipedia[1], Grazer finds mentions of these facts in other web sites, for which site-specific wrappers are learned. Additional facts are then extracted using these site-specific wrappers and stored in a knowledge base. These facts are used to learn wrappers for yet other sites, from which more facts are extracted. In corroborating facts, entity resolution and attribute-value normalization are performed. These tasks can be very challenging. To maintain reasonable precision, heuristics are applied at every stage of processing, which severely limits the coverage of the knowledge base. Our method avoids these difficulties by focusing on the extraction of *instances* of attribute-value pairs, and ignoring entity-name recognition, which can be done as a separate task [3]. Our method relies on a small set of known mark-up structures which give good coverage. Hence no bootstrapping is required and error propagation is minimized.

3 Attribute-Value Extraction

Our attribute-value extraction algorithm focuses on three types of structural contexts that cover a large portion of the attribute-value data observed on the Web, namely:

1. *Two-column tables:* Tables with exactly two columns, with attributes in the left-hand column and values in the right-hand column. A sample two-column table is shown in Figure 1.

2. *Relational tables:* Two-dimensional tables that typically have many rows, with attribute across the first row and a set of values associated with a particular entity in each subsequent row. Figure 2 shows a sample relational table.

3. *Colon-delimited pairs:* Attribute-value pairs that appear in the stylized form *"Attribute: Value"*. The

[1] URL: `http://en.wikipedia.org/`

Figure 2. A sample relational table

Course	*Location*	*Type*
Big Island Country Club	Kailua Kona	Semi
Hilo Municipal Golf Course	Hilo	Public
Hamakua Country Club	Honokaa	Semi

Figure 3. Sample colon-delimited pairs

Price: $3,060,000
Type: Single Family
SqFt: 4,849
Price/SqFt: $631.06
Bedrooms: 4
Full Baths: 3

colon is used to symbolize the fact that the attribute name is a prompt for the value. Many web pages contain attribute-value data of this kind. Figure 3 shows a set of colon-delimited pairs associated with a house.

Obviously, these structural contexts are only weak indications of the presence of attribute-value pairs. Not all tables contain attribute-value data. Many of them are used for formatting web pages. Not all instances of the colon are used to separate attributes and values. They often appear before quotations, in expressions of time (e.g. *"2:00"*), and even in movie titles (e.g. *"Star Wars: Episode III — Revenge of the Sith"*). These structural contexts provide a huge number of candidate attribute-value pairs, but only a small fraction of them are good attribute-value pairs.

For attribute-value pairs to be meaningful, there needs to be a certain amount of consistency in the attribute-value data. In particular:

1. There must be some consistency in attribute naming.
2. There must be some consistency in value representations for any given attribute.

Furthermore, attributes of the same entity often appear in groups, such as tables and lists (see Figures 1–3). Therefore, the fact that *Price* and *Type* are attributes of a house in Figure 3 should be a strong indication that *Price/SqFt* is a related attribute. Similarly, the fact that *United States* is not an attribute in Figure 1 should strongly indicate that Figure 1 is not a relational table. In this work, we exploit these types of consistency in order to identify attribute-value pairs from structural contexts. Specifically, attribute-value extraction is divided into the following two sub-tasks:

1. *Candidate generation:* Candidate attribute-value pairs are found by using simple HTML-based extractors called *candidate generators*. There is a candidate generator for each of the following types of structural contexts: two-column tables, relational tables, and colon-delimited pairs.
2. *Candidate filtering:* Unwanted candidate attribute-value pairs are filtered out by one of the following methods: *attribute whitelists* and *feature-based candidate filtering*.

In the next few sections, we will describe each of these components in detail.

3.1 Candidate Generators

The construction of candidate generators is relatively straightforward. For relational tables, we wrote a candidate generator that analyzes every HTML table in a given web page, and determines whether the table is used for formatting. For example, the code skips over table rows that contain form elements such as text boxes — an indication that the table is actually an HTML form. It also skips over table rows that contain a nested table — an indication that the outer table is used for page layout. Table rows containing cells that span across multiple columns are ignored. The entire table is ignored when there are too few valid rows remaining. Then attributes are extracted from the first row of the table, and values are extracted from all subsequent rows.

The candidate generator for two-column tables is similar, except that attributes are extracted from the first column and values from the second column. Only tables with exactly two columns are considered. While it is possible to construct a candidate generator for tables with many columns where each non-header *column* corresponds to a distinct entity, such tables are quite rare in practice, and therefore we do not consider these cases. Note that all two-column tables are also potential relational tables, in which case candidates are generated using both candidate generators, and the bad candidates are removed in the candidate filtering step.

For colon-delimited pairs, we wrote a simple list item scraper that uses a set of regular expressions to identify colon-delimited pairs. We use HTML line breaking tags to separate a given web page into lines and extract colon-delimited pairs that take a single line.

3.2 Attribute Whitelists

For candidate filtering, one simple approach is to match the candidate attribute-value pairs against a list of known attribute names. This list is known as an *attribute whitelist* and can be obtained from a suitable ontology or knowledge base, such as Freebase[2]. In this work, we use an open-domain fact repository extracted from the Web by Grazer

[2]URL: `http://www.freebase.com/`

[18]. In this repository, there are more than 123K unique attributes, 8,747 of which appear in at least 10 facts. We use this list of 8,747 attributes as an attribute whitelist.

This attribute whitelist is very short. In order to extract attributes that are not on the list, we use the following strategy: Assume that all candidate attribute-value pairs come in groups. For two-column tables and relational tables, attribute-value pairs from the same table are treated as a group. Also colon-delimited pairs with no intervening spaces are treated as a group. Attribute-value pairs from different candidate generators are in different groups. Then for each group of candidate attribute-value pairs, if more than a certain fraction of the attributes are in the attribute whitelist, then we assume that all of the candidate attribute-value pairs are good. For example, in Figure 1, if *Country*, *Commonwealth* and *Founded* are in the attribute whitelist, and the minimum fraction is 66%, then *Population (2000)* would be recognized as a good attribute as well, because $75\% > 66\%$ of the attributes are in the attribute whitelist. On the other hand, if the minimum fraction is 80%, then all attribute-value pairs from this table would be discarded.

This simple strategy turns out to be quite effective, even though only attributes are involved in the filtering process, not values. A possible explanation is that for tables (or other table-like representations such as lists of colon-delimited pairs), if the first row or column looks like a header, then it is likely that the rest of the table contains actual data. This assumption might not hold for other types of structural contexts.

3.3 Expanded Attribute Whitelists

The grouping information generated by the candidate extractors can also be used to expand or refine existing attribute whitelists. The basic idea is to find all candidate attributes in a corpus (e.g. the Web), and then count the number of times each candidate attribute appears in the same group as a known attribute. Specifically, for each candidate attribute a, compute the following score $f(a)$:

$$f(a) = \sum_{g \in G} n(a, g) \cdot n'(A, a, g) \quad (1)$$

where G is the set of groups of candidate attributes in the entire corpus, A is an attribute whitelist, $n(a, g)$ is the number of times a appears in group g, and $n'(A, a, g)$ is the number of times any attribute in A except a appears in group g. If $f(a)$ is large, then a is a frequent attribute that co-occurs with known attributes frequently. A new attribute whitelist is formed by collecting all attributes a with sufficiently high $f(a)$. As we will see in Section 4, this new attribute whitelist tends to be of higher quality than the attribute whitelist derived from the fact repository alone.

3.4 Feature-Based Candidate Filtering

There are two main weaknesses in the attribute whitelist approach. First, despite the use of grouping information to identify unknown attributes, the matching of known attributes is done using exact match, which can be quite fragile. Slight variations of known attributes can be treated as completely unknown (e.g. *Population* vs *Population (2000)*), and this can hurt the coverage of the attribute-value extractor. Second, the attribute whitelist approach completely ignores value representations, which can be extremely useful in identifying non-attribute-value pairs (e.g. *Population* is usually associated with numbers but not the word *Country*). This information is especially important when there is no strong signal from the structural context, as is the case with an isolated colon-delimited pair.

We tackle these problems using feature-based candidate filtering. Each candidate attribute-value pair is turned into a real-valued feature vector, $\mathbf{x} = (x_1, x_2, \ldots, x_n)$. Each feature x_i encodes a certain aspect of the attribute-value pair, such as the frequency of a certain token in the attribute. This feature vector $\mathbf{x}$ is then mapped to either $+1$ or -1: $+1$ means the candidate is a good attribute-value pair, and -1 means it is bad. This mapping can be done using any suitable binary classifier, which is learned using a set of training examples, i.e. candidate attribute-value pairs coupled with gold-standard labels ($+1$ or -1). We will describe the training process in more detail later in this section.

We use the following feature set in candidate filtering. Note that none of these features are language- or domain-specific. Knowledge about languages and domains will be injected through the training data. For brevity, we assign a unique identifier to each feature type. These identifiers are shown in brackets (e.g. *[id]*). A feature *x* of type *[id]* is written as *id:x*.

- *Attribute tokens [at]:* The value of the feature *at:s* is the frequency of the token *s* in the attribute. For example, for the attribute-value pair *Length x Width: 50 cm x 30 cm*, the value of *at:length* is 1.
- *Attribute prefix [ap]:* The value of the feature *ap:s* is 1 if *s* is the first token of the attribute; 0 otherwise. For example, for the attribute-value pair *Length x Width: 50 cm x 30 cm*, the value of *ap:length* is 1.
- *Attribute suffix [au]:* The value of the feature *au:s* is 1 if *s* is the last token of the attribute; 0 otherwise.
- *Attribute signatures [as]:* The value of the feature *as:s'* is the frequency of *s'* being the case signature of the attribute tokens. The case signature of a token is the concatenation of its character types (e.g. letters, digits), with repeated types conflated into a single instance.

- *Attribute prefix signature [aps]:* The value of the feature *aps:s'* is 1 if *s'* is the case signature of the first token of the attribute; 0 otherwise.
- *Attribute suffix signature [aus]:* Similar to *[aps]*, but for the last token of the attribute.
- *Attribute length [al]:* The value of the feature *al:n* is 1 if the attribute consists of *n* tokens; 0 otherwise.
- *Value tokens [vt]:* Similar to *[at]*, but for the value tokens. For example, for the attribute-value pair *Length x Width: 50 cm x 30 cm*, the value of *vt:cm* is 2.
- *Value prefix [vp]:* Similar to *[ap]*, but for the first token of the value.
- *Value suffix [vu]:* Similar to *[au]*, but for the last token of the value.
- *Value signatures [vs]:* Similar to *[as]*, but for the value tokens.
- *Value prefix signature [vps]:* Similar to *[aps]*, but for the first token of the value.
- *Value suffix signature [vus]:* Similar to *[aus]*, but for the last token of the value.
- *Value length [vl]:* Similar to *[al]*, but for the length of the value.
- *Group attribute prefix [apg]:* The value of the feature *apg:s* is the frequency of *s* being the first token of any other attribute in the same group.
- *Group attribute suffix [aug]:* The value of the feature *aug:s* is the frequency of *s* being the last token of any other attribute in the same group.

The last two feature types, *[apg]* and *[aug]*, capture the tendency that good attributes often appear in groups, so that less common attributes would be recognized when good attributes are nearby.

We also introduce feature conjunctions. The value of the feature conjunction *&:x,y* (read: *x and y*) is the product of the values of the features *x* and *y*. For example, for the attribute-value pair *Length x Width: 50 cm x 30 cm*, the value of *&:au:width,vu:cm* is 1. If two features are conjoined, then one of them must be attribute-related and the other one value-related. The purpose of these feature conjunctions is to associate attributes with values in a linear classifier.

One advantage of this feature-vector representation of attribute-value pairs is that, for two attribute-value pairs that look similar, their feature-vector representations should be similar as well. So any belief that a certain attribute-value pair is good or bad should generalize across other attribute-value pairs with similar feature-vector representations. This gives us better robustness compared to exact match in the attribute whitelist approach.

To learn a binary classifier, we use the online passive-aggressive algorithm with the linear kernel [7]. The resulting binary classifier is a linear classifier, $\text{sign}(\mathbf{w} \cdot \mathbf{x} + b)$, where $\mathbf{w} \in \mathbb{R}^n$ and $b \in \mathbb{R}$. The learning algorithm performs iterative updates to the linear classifier so that the new classifier remains as close as possible to the current one while achieving at least a unit margin on the most recent training example. This learning algorithm is chosen because it allows overlapping features (e.g. *[at]*, *[ap]*), has good generalization properties, and is very fast in practice.

3.5 Training Feature-Based Filters

This section describes the training process of the feature-based candidate filter in more detail. Learning a binary classifier requires a set of training examples with gold-standard labels ($+1$ or -1), and there need to be both good attribute-value pairs and bad attribute-value pairs in the training set. While good attribute-value pairs can be obtained from a suitable knowledge base, bad attribute-value pairs can be hard to come by, since we need to know that they are bad.

We construct the training set using the same fact repository used for extracting attribute whitelists (Section 3.2). Each fact in the fact repository has at least one source URLs. The basic idea is to find all documents that are sources of known facts, collect all candidate attribute-value pairs from these documents, then label these attribute-value pairs as $+1$ if they match any known facts, or -1 otherwise.

Since the fact repository is quite sparse, this basic approach can lead to many false negatives. Also for a given known fact, there may not be any exact match in any of its source documents because of attribute-value normalization or content drift. We alleviate these problems by doing the following:

1. Allow partial matching.
2. If an attribute-value pair is labeled positive, then label the entire group as positive as well.
3. Use the attribute whitelist described in Section 3.2 to identify additional positive examples.

The resulting training set consists of 17M positive examples and 28M negative examples. The positive examples are estimated to be 95% clean (i.e. 5% of them are mislabeled), and the negative examples are estimated to be 70% clean. We further reduce the false negative rate by removing negative examples that have similar feature-vector representations as the positive examples, based on average cosine

similarity. Specifically, we remove all negative examples $\mathbf{x}$ with their similarity score, $s(\mathbf{x})$, below a certain threshold:

$$s(\mathbf{x}) = \frac{1}{|P(\mathbf{x})|} \sum_{\mathbf{x}' \in P(\mathbf{x})} \text{sim}(\mathbf{x}', \mathbf{x}) \quad (2)$$

where $P(\mathbf{x})$ is the set of positive examples with non-zero cosine similarity with $\mathbf{x}$, and $\text{sim}(\mathbf{x}', \mathbf{x})$ is the cosine similarity between $\mathbf{x}'$ and $\mathbf{x}$. This removes 5M negative examples from the training set, making the negative training examples 85% clean.

To better handle labeling noise in the training data, we use a soft-margin version of the passive-aggressive update with a low aggressiveness parameter [7]. To avoid overfitting, we also remove from the feature set any features that occur less than 200 times in the training set. If two features are conjoined, then each one of them must occur at least 1000 times in the training set. There are about 289K features in the resulting feature set.

4 Experiments

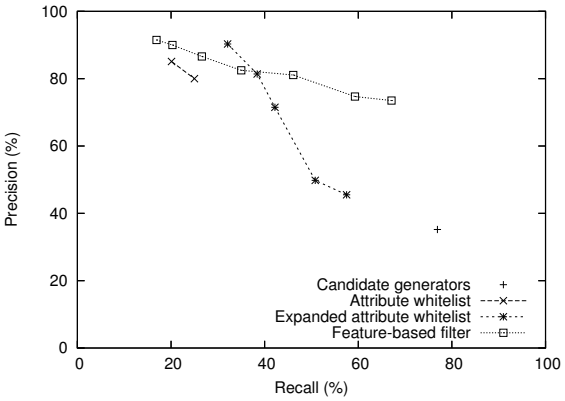
Our experiments were run over a web repository consisting of approximately 100 million HTML documents in English. Since running our extractors over 100 million documents can be time-consuming, we created the following data sets for quick experimentation:

1. *Development set:* From the web repository, we randomly sampled 343,217 documents. We call this unannotated data set the development set.

2. *Evaluation set:* From the development set, we randomly sampled 258 documents. A human annotator then hand-labeled all attribute-value pairs in these documents in any structural contexts (i.e. not in free text). All attribute-value pairs must be about some entities (i.e. *"Can you say the X of the Y is Z?"* [14]). However, the entities can be implicit and were not annotated. There are 3,088 annotated attribute-value pairs in this evaluation set. The annotation task was relatively straightforward. An experiment that involved three human annotators with minimal instructions showed that the inter-annotator agreement was quite high, with 80% average pairwise F-score.

We evaluated our attribute-value extractors using precision and recall. *Precision* is the fraction of extracted attribute-value pairs that are correct, and *Recall* is the fraction of gold-standard attribute-value pairs that are extracted. Partial credit is given for partially correct extractions as follows. Given an attribute-value pair $p_1 = (a_1, v_1)$, and a gold-standard attribute-value pair $p_2 = (a_2, v_2)$, the partial

Figure 4. Precision-recall curves for attribute-value extractors on the evaluation set



overlap score between p_1 and p_2 is defined as:

$$\begin{cases} \frac{|a_1 \cap a_2| + |v_1 \cap v_2|}{|a_1 \cup a_2| + |v_1 \cup v_2|} & \text{if } |a_1 \cap a_2| > 0 \text{ and } |v_1 \cap v_2| > 0 \\ 0 & \text{otherwise} \end{cases}$$

which is the number of overlapping non-HTML characters divided by the total number of non-HTML characters. Both attributes and values need to be matched for a non-zero score. Note that the score is 1.0 for exact match.

Figure 4 shows the precision-recall curves for various attribute-value extractors on the evaluation set. *Candidate generators* shows the performance of all three candidate generators combined without candidate filtering (precision: 35%, recall: 77%). *Attribute whitelist* shows the performance of the candidate generators with candidate filtering using the attribute whitelist extracted from the Grazer fact repository. The curve was obtained by varying the size of the whitelist, i.e. removing attributes that are less common. *Expanded attribute whitelist* is the corresponding curve for the attribute whitelist expanded using co-occurrence statistics derived from the web repository. *Feature-based filter* shows the performance of the candidate generators with the learned feature-based candidate filter. The curve was obtained by shifting the learned linear separator, i.e. varying the value of b.

Figure 4 shows that out of all attribute-value pairs that occur in any structural contexts, our candidate generators capture 77% of them. All of our candidate filters provide better precision than the candidate generators. In particular, the expanded attribute whitelist provides the best recall at 82% precision or above, and the feature-based candidate filter provides the best recall at lower levels of precision. The feature-based candidate filter also achieves the best overall F-score (precision: 74%, recall: 67%, F-score: 70%), and can be tuned to achieve the best overall precision (92%).

Figure 5. Number of extractions from the development set at 80% overall precision

	Attribute whitelist	*Expanded attribute whitelist*	*Feature-based filter*
Two-column tables	23,554	102,596	77,217
Relational tables	206,955	687,640	640,935
Colon-delimited pairs	348,408	578,041	447,897
Total	**578,917**	**1,368,277**	**1,166,049**

Figure 6. Sample extractions

Rider: Catherine Cheatley *Team:* CRW *Time:* 1.00' 13"
Web host established on: 1999 *Phone support availability:* 631.495.xxxx *Web hosting plan name:* Starter *Hosting platform:* RedHat Linux
Address: 2xxx, rue de Lorimier, Longueuil, QC *Telephone:* 450-463-xxxx *Category:* Tile ceramic mfrs & distrs, granite

Both the expanded attribute whitelist and the feature-based candidate filter outperform the original attribute whitelist at 80% precision.

We obtained similar results from the development set. Figure 5 shows the number of attribute-value pairs extracted from the development set at 80% overall precision. Both the expanded attribute whitelist and the feature-based candidate filter provide better coverage than the original attribute whitelist. Relational tables account for about half of the extracted attribute-value pairs, and colon-delimited pairs account for about 40%. Figure 6 shows some sample extracted attribute-value pairs.

The recall of our attribute-value extractors is comparable to that of TextRunner [1]. From a corpus of 9 million web pages, TextRunner extracts 7.8 million facts of which 80% are correct. From a corpus of 0.3 million web pages (the development set), both the expanded attribute whitelist and the feature-based candidate filter extract more than 1 million facts of which 80% are correct. Note that there is probably little overlap between extractions from structured contexts and from free text. In Figure 6, we see that much of the extracted data comes from phone books and catalogs, which seldom appear in the form of free text. On the other hand, facts like *"President Bush – flew to – Texas"* are commonly found in free text (e.g. news articles), but not in tables. However, the recall figures indicate that there is plenty of attribute-value data encoded in structural contexts on the Web, and extraction from such contexts is relatively straightforward compared to extraction from free text.

Figure 7. Features with large weights

*au:_OOV_*	−0.001884
&:aus:0,vps:0	−0.001014
&:aus:Aa,vp:the	−0.000395
&:au:number,vus:0	0.000187
&:au:name,vps:Aa	0.000278
aug:date	0.001328

Using the learned feature-based candidate filter, we extracted 1.01 billion attribute-value pairs from the repository of 100 million web pages. The precision is estimated to be 70–75%. Of the 100 million web pages, 50.3 million contain extracted attribute-value pairs, or about half of the repository. The running time is less than 6 hours with 100 machines, or about 575 CPU hours. This again compares favorably with TextRunner, which takes 85 CPU hours to process 9 million web pages.

Figure 7 shows some of the most important features (i.e. large absolute feature weights) in the feature-based candidate filter. Among the most undesirable features are:

- The last token of the attribute being an out-of-vocabulary word.
- Both the last token of the attribute and the first token of the value being numbers (e.g. *"10:30 a.m."*).
- The last token of the attribute being a capitalized word and the first token of the value being the word *the* (e.g. *"Star Wars: The Force Unleashed"*).

Among the most desirable features are:

- The last token of the attribute being the word *number* and the last token of the value being a number (e.g. *"Serial number: 013209"*).
- The last token of the attribute being the word *name* and the first token of the value being a capitalized word (e.g. *"First name: John"*).
- The word *date* being the last token of some attribute within the same group.

Figure 8. Precision-recall curves for the feature-based filter on the evaluation set

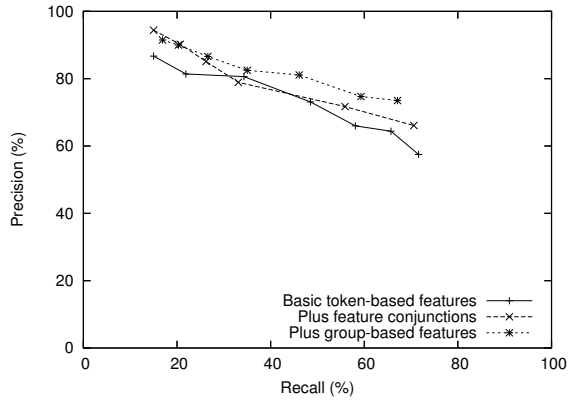


Figure 9. A two-column table with overlapping and incomplete extractions

Career Level:	Entry Level
Job Type:	Employee
Job Status:	Full Time
Job Shift:	First Shift
Salary:	From 35,000.00 to 45,000.00 USD

Extractions:

- Career level: entry level
- Job type: employee
- Job status: full time
- Entry level: employee *
- Entry level: full time *

Note that the feature conjunctions essentially form *attribute-specific value models* that associate certain types of values with certain types of attributes. These feature conjunctions turn out to be fairly useful, as shown in Figure 8, which compares the performance of the feature-based candidate filter using different feature sets. The group-based features (*[apg], [aug]*) are also useful at lower precision levels, but are not as useful at higher precision, where the expanded attribute whitelist outperforms (see Figure 4). There are two possible reasons for this:

1. The expanded attribute whitelist makes use of attribute co-occurrence information from the web repository. This extra information is important because it allows frequent, but previously unknown, attributes to be recognized. Interestingly, only the top 1,726 attributes of the expanded attribute whitelist were used in the 80% precision setting — almost one-fifth of the size of the original attribute whitelist.

2. For the attribute whitelist approach, filtering decisions are made based on groups, whereas for the feature-based approach, decisions are made for each candidate attribute-value pair. The latter approach may have undesirable consequences. For example, Figure 9 shows a table from which overlapping extractions were produced by the feature-based filter. Extractions overlap because the table was treated simultaneously as a two-column table and a relational table, and the feature-based filter was unable to tell whether *"Entry level: employee"* is a good attribute-value pair based on the fact that *"Career level: entry level"* was classified as good, and *"Career level: job type"* was classified as bad. Also note that the extractions are incomplete. The last two rows were not extracted even though it is clear that the table is a two-column table. This indicates that the current feature-based model may not be the most effective way to utilize grouping information. A way to incorporate grouping information into the model is to add an extra hidden variable to each group of candidates, and then use this group variable to influence individual filtering decisions in this group, and vice versa. Intuitively, this group variable indicates whether this group is valid or not. Variables for overlapping groups may also interact such that only one of them will be identified as valid. In other words, the candidate filtering problem would be solved using a tree-like graphical model, in which filtering decisions within groups are inter-dependent [13, 3].

5 Conclusions

This paper describes a general framework for extracting attribute-value pairs from web pages. Our work is focused on three types of structural contexts in which attribute-value pairs frequently occur: two-column tables, relational tables, and colon-delimited pairs. Atribute-value extraction is divided into two sub-tasks: candidate generation, in which potential attribute-value pairs occurring in the above structural contexts are identified, and candidate filtering, in which unlikely candidates are removed using attribute whitelists and feature-based candidate filtering. We have demonstrated the scalability of our methods by extracting from an extensive web repository, which results in 1 billion extracted attribute-value pairs within a few hundred CPU hours. On a smaller, fully-annotated evaluation set, we have shown that our methods can achieve up to 70% F-score and up to 92% precision.

This work can be seen as a basic component of the larger goal of extracting knowledge repositories from the Web in a bottom-up fashion. Under this goal, the immediate next task would be associating extracted attribute-value pairs with entities, forming entity-attribute-value triples. These triples would then be normalized using various entity resolution and attribute alignment and normalization techniques. All of these tasks are mutually dependent, so devising algorithms that are robust, accurate and scalable would be a very interesting challenge.

References

[1] M. Banko, M. J. Cafarella, S. Soderland, M. Broadhead, and O. Etzioni. Open information extraction from the web. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*, pages 2670–2676, Hyderabad, India, January 2007.

[2] M. Berland and E. Charniak. Finding parts in very large corpora. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics (ACL-99)*, pages 57–64, College Park, MD, June 1999.

[3] R. C. Bunescu and R. J. Mooney. Collective information extraction with relational Markov networks. In *Proceedings of the 42th Annual Meeting of the Association for Computational Linguistics (ACL-04)*, pages 439–446, Barcelona, Spain, July 2004.

[4] S. A. Caraballo. Automatic construction of a hypernym-labeled noun hierarchy from text. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics (ACL-99)*, pages 120–126, College Park, MD, June 1999.

[5] S. M. Cherry. Weaving a web of ideas. *IEEE Spectrum*, 39(9):65–69, September 2002.

[6] W. W. Cohen, M. Hurst, and L. S. Jensen. A flexible learning system for wrapping tables and lists in HTML documents. In *Proceedings of the 11th International World Wide Web Conference (WWW-02)*, pages 232–241, Honolulu, HI, May 2002.

[7] K. Crammer, O. Dekel, J. Keshet, S. Shalev-Shwartz, and Y. Singer. Online passive-aggressive algorithm. *Journal of Machine Learning Research*, 7:551–585, 2006.

[8] D. Freitag and N. Kushmerick. Boosted wrapper induction. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-00)*, pages 577–583, Austin, TX, July 2000.

[9] M. A. Hearst. Automatic acquisition of hyponyms from large text corpora. In *Proceedings of the 15th International Conference on Computational Linguistics (COLING-92)*, Nantes, France, August 1992.

[10] M. A. Hearst and H. Schütze. Customizing a lexicon to better suit a computational task. In *Proceedings of the ACL-SIGLEX Workshop on Acquisition of Lexical Knowledge from Text*, Columbus, Ohio, June 1993.

[11] I. Muslea, S. Minton, and C. A. Knoblock. Hierarchical wrapper induction for semistructured information sources. *Journal of Autonomous Agents and Multi-Agent Systems*, 4:93–114, 2001.

[12] M. Pasca and B. Van Durme. Weakly-supervised acquisition of open-domain classes and class attributes from web documents and query logs. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics (ACL-HLT-08)*, pages 19–27, Columbus, OH, June 2008.

[13] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, San Mateo, CA, 1988.

[14] M. Poesio and A. Almuhareb. Identifying concept attributes using a classifier. In *Proceedings of the ACL Workshop on Deep Lexical Semantics*, Ann Arbor, Michigan, June 2005.

[15] J. Pustejovsky. *The Generative Lexicon*. MIT Press, Cambridge, MA, 1995.

[16] Y. Shinyama and S. Sekine. Preemptive information extraction using unrestricted relation discovery. In *Proceedings of the Human Language Technology Conference of the North American Chapter of the ACL (HLT-NAACL-06)*, pages 304–311, New York City, NY, June 2006.

[17] W. A. Woods. What's in a link: Foundations for semantic networks. In D. G. Bobrow and A. M. Collins, editors, *Representation and Understanding: Studies in Cognitive Science*, pages 35–82. Academic Press, New York, 1975.

[18] S. Zhao and J. Betz. Corroborate and learn facts from the web. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 995–1003, San Jose, CA, August 2007.