Unsupervised Named-Entity Extraction from the Web: An Experimental Study

Oren Etzioni, Michael Cafarella, Doug Downey, Ana-Maria Popescu
Tal Shaked, Stephen Soderland, Daniel S. Weld, and Alexander Yates
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195-2350
etzioni@cs.washington.edu

February 28, 2005

Abstract

The KNOWITALL system aims to automate the tedious process of extracting large collections of facts (*e.g.*, names of scientists or politicians) from the Web in an unsupervised, domain-independent, and scalable manner. The paper presents an overview of KNOWITALL's novel architecture and design principles, emphasizing its distinctive ability to extract information without any hand-labeled training examples. In its first major run, KNOWITALL extracted over 50,000 class instances, but suggested a challenge: How can we improve KNOWITALL's recall and extraction rate without sacrificing precision?

This paper presents three distinct ways to address this challenge and evaluates their performance. *Pattern Learning* learns domain-specific extraction rules, which enable additional extractions. *Subclass Extraction* automatically identifies sub-classes in order to boost recall (*e.g.*, "chemist" and "biologist" are identified as sub-classes of "scientist"). *List Extraction* locates lists of class instances, learns a "wrapper" for each list, and extracts elements of each list. Since each method bootstraps from KNOWITALL's domain-independent methods, the methods also obviate hand-labeled training examples. The paper reports on experiments, focused on building lists of named entities, that measure the relative efficacy of each method and demonstrate their synergy. In concert, our methods gave KNOWITALL a 4-fold to 8-fold increase in recall at precision of 0.90, and discovered over 10,000 cities missing from the Tipster Gazetteer.

Key words: Information Extraction, Pointwise Mutual Information, Unsupervised, Question Answering.

1 Introduction and Motivation

Information Extraction is the task of automatically extracting knowledge from text. *Unsupervised* information extraction dispenses with hand-tagged training data. Because unsupervised extraction systems do not require human intervention, they can recursively discover new relations, attributes, and instances in a fully automated, scalable manner. This paper describes KNOWITALL, an unsupervised, domain-independent system that extracts information from the Web.

Collecting a large body of information by searching the Web can be a tedious, manual process. Consider, for example, compiling a comprehensive, international list of astronauts, politicians, or cities. Unless you find the "right" document or database, you are reduced to an error-prone, piecemeal search. One of KNOWITALL's goals is to address the problem of accumulating large collections of facts.

In our initial experiments with KNOWITALL, we have focused on a sub-problem of information extraction, building lists of named entities found on the Web, such as instances of the class `City` or the class `Film`. KNOWITALL is able to extract instances of relations, such as `capitalOf(City,Country)` or `starsIn(Actor,Film)`, but the focus of this paper is on extracting comprehensive lists of named entities.

KNOWITALL introduces a novel, generate-and-test architecture that extracts information in two stages. Inspired by Hearst [22], KNOWITALL utilizes a set of eight domain-independent extraction patterns to *generate* candidate facts.[1] For example, the generic pattern "NP1 such as NPList2" indicates that the head of each simple noun phrase (NP) in the list NPList2 is a member of the class named in NP1. By instantiating the pattern for the class `City`, KNOWITALL extracts three candidate cities from the sentence: "We provide tours to cities such as Paris, London, and Berlin."

Next, KNOWITALL automatically *tests* the plausibility of the candidate facts it extracts using *pointwise mutual information* (PMI) statistics computed by treating the Web as a massive corpus of text. Extending Turney's PMI-IR algorithm [42], KNOWITALL leverages existing Web search engines to compute these statistics efficiently.[2] Based on these PMI statistics, KNOWITALL associates a probability with every fact it extracts, enabling it to automatically manage the tradeoff between precision and recall. Since we cannot compute "true recall" on the Web, the paper uses the term "recall" to refer to the size of the set of facts extracted.

Etzioni [19] introduced the metaphor of an *Information Food Chain* where search engines are herbivores "grazing" on the Web and intelligent agents are *information carnivores* that consume output from various herbivores. In terms of this metaphor, KNOWITALL is an information carnivore that consumes the output of existing search engines. In its first major run, KNOWITALL extracted over 50,000 facts regarding cities, states, countries, actors, and films [20]. This initial run revealed that, while KNOWITALL is capable of autonomously extracting high-quality information from the Web, it faces several challenges. In this paper we focus on one key challenge:

> *How can we improve* KNOWITALL*'s recall and extraction rate so that it extracts substantially more members of large classes such as cities and films while maintaining high precision?*

We describe and compare three distinct methods added to KNOWITALL in order to improve its recall:

- **Pattern Learning (PL):** learns domain-specific patterns that serve both as extraction rules and as validation patterns to assess the accuracy of instances extracted by the rules.

- **Subclass Extraction (SE):** automatically identifies subclasses in order to facilitate extraction. For example, in order to identify scientists, it is helpful to determine subclasses of scientists (*e.g.*, physicists, geologists, *etc.*) and look for instances of these subclasses.

- **List Extraction (LE):** locates lists of class instances, learns a "wrapper" for each list, and uses the wrapper to extract list elements.

[1] Hearst proposed a set of generic patterns that identify a hyponym relation between two noun phrases. Examples are the pattern "NP {,} such as NP" and the pattern "NP {,} and other NP".

[2] Turney measured the similarity of two term based on how often the terms appear in proximity to each other in Web search-engine indices.

Each of the methods dispenses with hand-labeled training examples by bootstrapping from the information extracted by KNOWITALL's domain-independent patterns. We evaluate each method experimentally, demonstrate their synergy, and compare with the baseline KNOWITALL system described in [20]. Our main contributions are:

1. We demonstrate that it is feasible to carry out unsupervised, domain-independent information extraction from the Web with high precision. Much of the previous work on information extraction focused on small document collections and required hand-labeled examples.

2. We present the first comprehensive overview of KNOWITALL, our novel information extraction system. We describe KNOWITALL's key design decisions and the experimental justification for them.

3. We show that Web-based mutual information statistics can be effective in validating the output of an information extraction system.

4. We describe and evaluate three methods for improving the recall and extraction rate of a Web information extraction system. While our implementation is embedded in KNOWITALL, the lessons learned are quite general. For example, we show that LE typically finds five to ten times more extractions than other methods, and that its extraction rate is forty times faster.

5. We demonstrate that our methods, when used in concert, can increase KNOWITALL's recall by 4-fold to 8-fold over the baseline KNOWITALL system.

The remainder of this paper is organized as follows. The paper begins with a comprehensive overview of KNOWITALL, its central design decisions, and their experimental justification. Sections 3 to 5 describe our three methods for enhancing KNOWITALL's recall, and Section 6 reports on our experimental comparison between the methods. We discuss related work in Section 7, directions for future work in Section 8, and conclude in Section 9.

2 Overview of KNOWITALL

The only domain-specific input to KNOWITALL is a set of predicates that specify KNOWITALL's focus (*e.g.*, Figure 6). While our experiments to date have focused on unary predicates, which encode class membership, KNOWITALL can also handle n-ary relations as explained below. KNOWITALL's *Bootstrapping* step uses a set of *domain-independent* extraction patterns (*e.g.*, Figure 1) to create its set of extraction rules and "discriminator" phrases (described below) for each predicate in its focus. The Bootstrapping is fully automatic, in contrast to other bootstrapping methods that require a set of manually created training seeds. A system flowchart is shown in Figure 2 and pseudocode in Figure 3 for the baseline KNOWITALL system.

The two main KNOWITALL modules are the *Extractor* and the *Assessor*. The Extractor creates a query from keywords in each rule, sends the query to a Web search engine, and applies the rule to extract information from the resulting Web pages. The Assessor computes a probability that each extraction is correct before adding the extraction to KNOWITALL's knowledge base. The Assessor bases its probability computation on search engine hit counts used to compute the mutual information between the extracted instance of a class and a set of automatically generated discriminator phrases associated with that class.[3] This assessment process is an extension of Turney's PMI-IR algorithm [42].

[3]We refer to discriminator phrases as "discriminators" throughout.

Predicate:	Class1
Pattern:	NP1 “such as” NPList2
Constraints:	head(NP1)= plural(label(Class1)) & properNoun(head(each(NPList2)))
Bindings:	Class1(head(each(NPList2)))

Figure 1: This generic extraction pattern can be instantiated automatically with the pluralized class label to create a domain-specific extraction rule. For example, if `Class1` is set to “City” then the rule looks for the words “cities such as” and extracts the heads of the proper nouns following that phrase as potential cities.

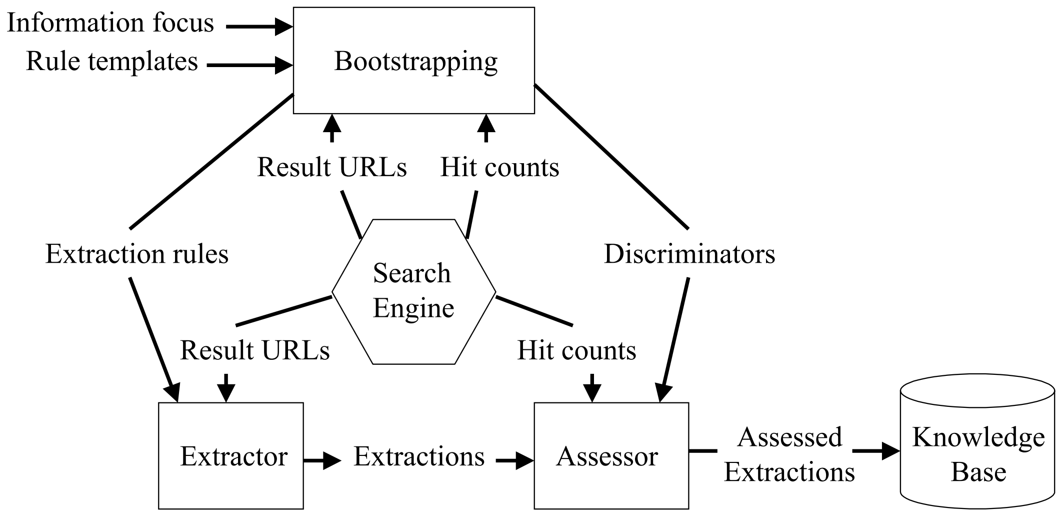


Figure 2: Flowchart of the main components in KnowItAll. Bootstrapping creates extractions rules and “discriminators” automatically with no hand-tagged training. Extractor fetches Web pages and applies extraction rules, then Assessor computes the probability of correctness before inserting in the Knowledgebase.

```
KNOWITALL(information focus $I$, rule templates $T$)
    Bootstrap($I$, $T$) sets rules $R$, queries $Q$, and discriminators $D$
    Do until queries in $Q$ are exhausted (or other termination criterion)
        Extractor($R$, $Q$) writes extractions list $E$
        Assessor($E$, $D$) adds extractions to the knowledgebase

Extractor(rules $R$, queries $Q$)
    Select queries from Q, set the number of downloads for each query
    Send selected queries to search engines
    For each webpage $w$ whose URL was returned by a search engine
        Extract fact $e$ from $w$ using the rule associated with the query
        Write $e$ to extractions list $E$

Assessor(extraction list $E$, discriminators $D$)
    For each extraction $e$ in $E$
        Assign a probability $p$ to $e$ using a Bayesian classifier based on $D$
        Add $e$,$p$ to the knowledgebase
```

Figure 3: High-level pseudocode for KNOWITALL. (See Figure 10 for pseudocode of Bootstrap(I,T).)

A Bootstrapping step creates extraction rules and discriminators for each predicate in the focus. KNOWITALL creates a list of search engine queries associated with the extraction rules, then executes the main loop. At the start of each loop, KNOWITALL selects queries, favoring predicates and rules that have been most productive in previous iterations of the main loop. The Extractor sends the selected queries to a search engine and extracts information from the resulting Web pages. The Assessor computes the probability that each extraction is correct and adds it to the knowledge base. This loop is repeated until all queries are exhausted or deemed too unproductive. KNOWITALL's running time increases linearly with the size and number of web pages it examines.

We now elaborate on KNOWITALL's Extraction Rules and Discriminators, and the Bootstrapping, Extraction, and Assessor modules.

2.1 Extraction Rules and Discriminators

KNOWITALL automatically creates a set of extraction rules for each predicate, as described in Section 2.2. Each rule consists of a predicate, an extraction pattern, constraints, bindings, and keywords. The *predicate* gives the relation name and class name of each predicate argument. In the rule shown in Figure 4, the unary predicate is "City". The *extraction pattern* is applied to a sentence and has a sequence of alternating context strings and *slots*, where each slot represents a string from the sentence. The rule may set constraints on a slot, and may bind it to one of the predicate arguments as a phrase to be extracted. In the example rule, the extraction pattern consists of three elements: a slot named NP1, a context string "such as", and a slot named NPList2. There is an implicit constraint on slots with name NP<digit>. They must match simple noun phrases and those with name NPList<digit> match a list of simple noun phrases. Slot names of P<digit> can match arbitrary phrases.

The Extractor uses regular expressions based on part-of-speech tags from the Brill tagger [5] to identify simple noun phrases and NPLists. The head of a noun phrase is generally the last word of the phrase. If the last word is capitalized, the Extractor searches left for the start of the proper noun, based on orthographic clues. Take for example, the sentence "The tour includes major cities such as New York, central Los Angeles, and Dallas". The head of the NP "major cities" is just "cities", whereas the head of "New York" is "New York" and the head of "central Los Angeles" is "Los Angeles". This simple syntactic analysis was chosen for processing efficiency, and because our domain-independent architecture avoids more knowledge intensive analysis.

Predicate:	City
Pattern:	NP1 "such as" NPList2
Constraints:	head(NP1)= "cities"
	properNoun(head(each(NPList2)))
Bindings:	City(head(each(NPList2)))
Keywords:	"cities such as"

Figure 4: An extraction rule generated by substituting the class name City and the plural of the class label "city" into a generic rule template. The rule looks for Web pages containing the phrase "cities such as" and extracts the proper nouns following that phrase as instances of the unary predicate `City`.

The *constraints* of a rule can specify the entire phrase that matches the slot, the head of the phrase, or the head of each simple NP in an NPList slot. One type of constraint is an exact string constraint, such as the constraint head(NP1) = "cities" in the rule shown in Figure 4. Other constraints can specify that a phrase or its head must follow the orthographic pattern of a proper noun, or of a common noun. The rule *bindings* specify which slots or slot heads are extracted for each argument of the predicate. If the bindings have an NPList slot, a separate extraction is created for each simple NP in the list that satisfies all constraints. In the example rule, an extraction is created with the `City` argument bound to each simple NP in NPList2 that passes the proper noun constraint.

A final part of the rule is a list of *keywords* that is created from the context strings and any slots that have an exact word constraint. In our example rule, there is a single keyword phrase "cities such as" that is derived from slot NP1 and the immediately following context. A rule may have multiple keyword phrases if context or slots with exact string constraints are not immediately adjacent.

KnowItAll uses the keywords as search engine queries, then applies the rule to the Web page that is retrieved, after locating sentences on that page that contain the keywords. More details of how rules are applied is given in Section 2.3. A BNF description of the rule language is given in Figure 8. The example given here is a rule for a unary predicate, `City`. The rule language also covers n-ary predicates with arbitrary relation name and multiple predicate arguments, such as the rule for `CeoOf(Person,Company)` shown in Figure 9.

KnowItAll's Extractor module uses extraction rules that apply to single Web pages and carry out shallow syntactic analysis. In contrast, the Assessor module uses discriminators that apply to search engine indices. These discriminators are analogous to simple extraction rules that ignore syntax, punctuation, capitalization, and even sentence breaks, limitations that are imposed by use of commercial search engine queries. On the other hand, discriminators are equivalent to applying an extraction pattern simultaneously to the entire set of Web pages indexed by the search engine.

A discriminator consists of an extraction pattern with alternating context strings and slots. There are no explicit or implicit constraints on the slots, and the pattern matches Web pages where the context strings and slots are immediately adjacent, ignoring punctuation, whitespace, or HTML tags. The discriminator for a unary predicate has a single slot, which we represent as an X here, for clarity of exposition. Discriminators for binary predicates have two slots, here represented as X and Y, for arguments 1 and 2 of the predicate, and so forth.

When a discriminator is used to validate a particular extraction, the extracted phrases are substituted into the slots of the discriminator to form a search query. This is described in more detail in Section 2.4. Figure 5 shows one of several possible discriminators that can be used for the predicate `City` and for the binary predicate `CeoOf(Person,Company)`.

Discriminator for: City
  "city X"

Discriminator for: CeoOf(Person,Company)
  "X CEO of Y"

Figure 5: When the discriminator for `City` is used to validate the extraction "Paris", the Assessor finds hit counts for the search query phrase "city Paris". Similarly, the discriminator for `CeoOf` validates Jeff Bezos as CEO of Amazon with the search query, "Jeff Bezos CEO of Amazon".

We now describe how KnowItAll automatically creates a set of extraction rules and discriminator phrases for a predicate.

2.2 Bootstrapping

KnowItAll's input is a set of *predicates* that represent classes or relationships of interest. The predicates supply symbolic names for each class (*e.g.* "MovieActor"), and also give one or more labels for each class (*e.g.* "actor" and "movie star"). These labels are the surface form in which a class may appear in an actual sentence. Bootstrapping uses the labels to instantiate extraction rules for the predicate from generic rule templates.

Figure 6 shows some examples of predicates for a geography domain and for a movies domain. Some of these are "unary" predicates, used to find instances of a class such as `City` and `Country`; some are "n-ary" predicates, such as the `capitalOf` relationship between `City` and `Country` and the `starsIn` relationship between `MovieActor` and `Film`. In this paper, we concentrate primarily on unary predicates and how KnowItAll uses them to extract instances of classes from the Web. Preliminary experiments show that the same methods work well on n-ary predicates.

The first step of Bootstrapping uses a set of domain-independent generic extraction patterns (*e.g.* Figure 1). The pattern in Figure 1 can be summarized informally as <`class1`> `''such as'' NPList` That is, given a sentence that contains the class label followed by "such as", followed by a list of simple noun phrases, KnowItAll extracts the head of each noun phrase as a candidate member of the class, after testing that it is a proper noun.

Combining this template with the predicate `City` produces two instantiated rules, one for the class label "city" (shown in Figure 4 in Section 2.1) and a similar rule for the label "town". The class-specific extraction patterns are:

"cities such as " NPList
"towns such as " NPList

Predicate: City labels: “city”, “town”	Predicate: Film labels: “film”, “movie”
Predicate: Country labels: “country”, “nation”	Predicate: MovieActor labels: “actor”, “movie star”

Predicate: capitalOf(City,Country)
relation labels: “capital of”
class-1 labels: “city”, “town”
class-2 labels: “country”, “nation”
	Predicate: starsIn(MovieActor,Film)
	relation labels: “stars in”, “star of”
	class-1 labels: “actor”, “movie star”
	class-2 labels: “film”, “movie”

Figure 6: Example predicates for a geography domain and for a movies domain. The class labels and relation labels are used in creating extraction rules for the class from generic rule templates.

Each instantiated extraction rule has a list of keywords that are sent as phrasal query terms to a search engine.

A sample of the syntactic patterns that underlie KNOWITALL’s rule templates is shown in Figure 7. Some of our rule templates are adapted from Marti Hearst’s hyponym patterns [22] and others were developed independently. The first eight patterns shown are for unary predicates whose pluralized English name (or “label”) matches <class1>. To instantiate the rules, the pluralized class label is automatically substituted for <class1>, producing patterns like “cities such as” NPList.

We have also experimented with rule templates for binary predicates, such as the last two examples. These are for the generic predicate, `relation(Class1,Class2)`. The first produces the pattern <city> “is the capital of” <country> for the predicate `capitalOf(City,Country)`, and the pattern <person> “is the CEO of” <company> for the predicate `CeoOf(Person,Company)`.

Bootstrapping also initializes the Assessor for each predicate in a fully automated manner. It first generates a set of discriminator phrases for the predicate based on class labels and on keywords in the extraction rules for that predicate. Bootstrapping then uses the extraction rules to find a set of seed instances to train the discriminators for each predicate, as described in Section 2.5.

2.3 Extractor

To see how KNOWITALL’s extraction rules operate, suppose that <class1> in the pattern

<class1> “such as” NPList

is bound to the name of a class in the ontology. Then each simple noun phrase in NPList is likely to be an instance of that class. When this pattern is used for the class `Country` it would match a sentence that includes the phrase “countries such as X, Y, and Z” where X, Y, and Z are names of countries. The same pattern is used to generate rules to find instances of the class `Actor`, where the rule looks for “actors such as X, Y, and Z”.

In using these patterns as the basis for extraction rule templates, we add syntactic constraints that look for simple noun phrases (a nominal preceded by zero or more modifiers). NP must be a simple noun phrase; NPList must be a list of simple NPs; and what is denoted by <class1> is a simple noun phrase with the class name as its head. Rules that look for proper names also include an orthographic constraint that tests capitalization. To see why noun phrase analysis is essential, compare these two sentences.

A) “China is a country in Asia.”

```
NP “and other” <class1>
NP “or other” <class1>
<class1> “especially” NPList
<class1> “including” NPList
<class1> “such as” NPList
“such” <class1> “as” NPList
NP “is a” <class1>
NP “is the” <class1>

<class1> “is the” <relation> <class2>
<class1> “,” <relation> <class2>
```

Figure 7: The eight generic extraction patterns used for unary extraction rules, plus two examples of binary extraction patterns. The first five patterns also have an alternate form with a comma, *e.g.* NP “, and other” <class1>. (If a rule pattern includes punctuation, a search engine will return some Web pages that do not match the rule. Nothing is extracted from such pages.) The terms <class1> and <class2> stand for an NP in the rule pattern with a constraint binding the head of the phrase to a label of predicate argument 1 or 2. Similarly, <relation> stands for a phrase in the rule pattern with a constraint binding it to a relation label of a binary predicate.

B) “Garth Brooks is a country singer.”

In sentence A the word “country” is the head of a simple noun phrase, and China is indeed an instance of the class `Country`. In sentence B, noun phrase analysis can detect that “country” is not the head of a noun phrase, so Garth Brooks won’t be extracted as the name of a country.

Let’s consider a rule template (Figure 1) and see how it is instantiated for a particular class. The Bootstrapping module generates a rule for `City` from this rule template by substituting “City” for “Class1”, plugging in the plural “cities” as a constraint on the head of NP1. This produces the rule shown in Figure 4. Bootstrapping also creates a similar rule with “towns” as the constraint on NP1, if the predicate specifies “town” as well as “city” as surface forms associated with the class name. Bootstrapping then takes the literals of the rule and forms a set of keywords that the Extractor sends to a search engine as a query. In this case, the search query is the phrase “cities such as”.

The Extractor matches the rule in Figure 4 to sentences in Web pages returned for the query. NP1 matches a simple noun phrase; it must be immediately followed by the string “such as”; following that must be a list of simple NPs. If the match is successful, the Extractor applies constraints from the rule. The head of NP1 must match the string “cities”. The Extractor checks that the head of each NP in the list NPList2 has the capitalization pattern of a proper noun. Any NPs that do not pass this test are ignored. If all constraints are met, the Extractor creates one or more extractions: an instance of the class `City` for each proper noun in NPList2. The BNF for KNOWITALL’s extraction rules appears in Figure 8.

The rule in Figure 4 would extract three instances of `City` from the sentence “We service corporate and business clients in all major European cities such as London, Paris, and Berlin.” If all the tests for proper nouns fail, nothing is extracted, as in the sentence “Detailed maps and information for several cities such as airport maps, city and downtown maps”.

The Extractor can also utilize rules for binary or n-ary relations. Figure 9 shows a rule that finds in-

```
<rule>          |=   <predicate> <pattern> <constraints> <bindings> <keywords>
<predicate>     |=   ‘Predicate: ’ ( <predName> |
                        <predName> ‘(’ <class> ( ‘,’ <class> )+ ‘)’ )
<pattern>       |=   ‘Pattern: ’ <context> ( <slot> <context> )+
<context>       |=   ( ‘ ” ’ string ‘ ” ’ | <null> )
<slot>          |=   ( ‘NP’<d> | ‘NPList’<d> | ‘P’<d> )
<d>             |=   digit
<constraints>   |=   ‘Constraints: ’ ( <constr> )*
<constr>        |=   <phrase> ‘= ” ’ string ‘ ” ’ | ‘properNoun(’ <phrase> ‘)’
<phrase>        |=   ( ‘NP’<d> | ‘P’<d> | ‘head(NP’<d> ‘)’ |
                        ‘each(NPList’ <d> ‘)’ | ‘head(each(NPList’ <d> ‘))’ )
<bindings>      |=   ‘Bindings: ’ <predName> ‘(’ <phrase> (‘,’ <phrase>)* ‘)’
<predName>      |=   string
<class>         |=   string
<keywords>      |=   ‘Keywords: ’ ( ‘ ” ’ string ‘ ” ’ )+
```

Figure 8: BNF description of the extraction rule language. An extraction pattern alternates context (exact string match) with slots that can be a simple noun phrase (NP), a list of NPs, or an arbitrary phrase (P). Constraints may require a phrase or its head to match an exact string or to be a proper noun. The “each” operator applies a constraint to each simple NP of an NPList. Rule bindings specify how extracted phrases are bound to predicate arguments. Keywords are formed from literals in the rule, and are sent as queries to search engines.

stances of the relation `CeoOf(Person,Company)` where the predicate specifies one or more labels for the relation, such as “CEO of” that are substituted into the generic pattern in the rule template

<class1> “,” <relation> <class2>

This particular rule has the second argument bound to an instance of Company, “Amazon”, which KNOWITALL has previously added to its knowledgebase.

KNOWITALL automatically formulates queries based on its extraction rules. Each rule has an associated search query composed of the rule’s keywords. For example, if the pattern in Figure 4 was instantiated for the class `City`, it would lead KNOWITALL to 1) issue the search-engine query “cities such as”, 2) download in parallel all pages named in the engine’s results, and 3) apply the Extractor to sentences on each downloaded page. For robustness and scalability KNOWITALL queries multiple different search engines.

2.4 Assessor

KNOWITALL uses statistics computed by querying search engines to assess the likelihood that the Extractor’s conjectures are correct. Specifically, the Assessor uses a form of *pointwise mutual information* (PMI) between words and phrases that is estimated from Web search engine hit counts in a manner similar to Turney’s PMI-IR algorithm [42]. The Assessor computes the PMI between each extracted instance and multiple, *automatically generated discriminator phrases* associated with the class (such as “X is a city” for the class `City`).[4] For example, in order to estimate the likelihood that “Liege” is the name of a city, the Assessor might check to see if there is a high PMI between “Liege” and phrases such as “Liege is a city”.

[4]We use class names and the keywords of extraction rules to automatically generate these discriminator phrases; they can also be derived from rules learned using PL techniques (Section 3).

Predicate:	CeoOf(Person,Company)
Pattern:	NP1 “,” P2 NP3
Constraints:	properNoun(NP1)
	P2 = “CEO of”
	NP3 = ”Amazon”
Bindings:	CeoOf(NP1,NP3)
Keywords:	“CEO of Amazon”

Figure 9: An example of an extraction rule for a binary predicate that finds the CEO of a company. In this case, the second argument is bound to a known instance of company from the knowledgebase, Amazon.

More formally, let I be an instance and D be a discriminator phrase. We compute the PMI score as follows:

$$\text{PMI}(I, D) = \frac{|\text{Hits}(D + I)|}{|\text{Hits}(I)|} \tag{1}$$

The PMI score is the number of hits for a query that combines the discriminator and instance, divided by the hits for the instance alone. The raw PMI score for an instance and a given discriminator phrase is typically a tiny fraction, perhaps as low as 1 in 100,000 even for positive instances of the class. This does not give the probability that the instance is a member of the class, only the probability of seeing the discriminator on Web pages containing the instance.

These mutual information statistics are treated as features that are input to a *Naive Bayes Classifier* (NBC) using the formula given in Equation 2. This is the probability that fact ϕ is correct, given features $f_1, f_2, \ldots f_n$, with an assumption of independence between the features.

$$P(\phi|f_1, f_2, \ldots f_n) = \frac{P(\phi)\prod_i P(f_i|\phi)}{P(\phi)\prod_i P(f_i|\phi) + P(\neg\phi)\prod_i P(f_i|\neg\phi)} \tag{2}$$

Our method to turn a PMI score into the conditional probabilities needed for Equation 2 is straightforward. The Assessor takes a set of k positive and k negative seeds for each class and finds a threshold on PMI scores that splits the positive and negative seeds. It then uses a tuning set of another k positive and k negative seeds to estimate $P(PMI > thresh|class)$, $P(PMI > thresh|\neg class)$, $P(PMI \leq thresh|class)$, and $P(PMI \leq thresh|\neg class)$, by counting the positive and negative seeds (plus a smoothing term) that are above or below the threshold. We used k = 10 and a smoothing term of 1 in the experiments reported here.

In a standard NBC, if a candidate fact is more likely to be true than false, it is classified as true. However, since we wish to be able to trade precision against recall, we record the crude probability estimates computed by the NBC for each extracted fact. By raising the probability threshold required for a fact to be deemed true, we increase precision and decrease recall; lowering the threshold has the opposite effect. We found that, despite its limitations, NBC gave better probability estimates than the logistic regression and Gaussian models we tried.

Several open questions remain about the use of PMI for information extraction. Even with the entire Web as a text corpus, the problem of sparse data remains. The most precise discriminators tend to have low PMI scores for numerous positive instances, often as low as 10^{-5} or 10^{-6}. This is not a problem for prominent instances that have several million hits on the Web. If an instance is found on only a few thousand

Web pages, the expected number of hits for a positive instance will be less than 1 for such a discriminator. This leads to false negatives for the more obscure positive instances.

A different problem with using PMI is homonyms — words that have the same spelling, but different meanings. For example, Georgia refers to both a state and country, Normal refers to a city in Illinois and a socially acceptable condition, and Amazon is both a rain forest and an on-line shopping destination. When a homonym is used more frequently in a sense distinct from the one we are interested in, then the PMI scores may be low and may fall below threshold. This is because PMI scores measure whether membership in the class is the *most common* meaning of a noun denoting an instance, not whether membership in the class is a *legitimate but less frequent* usage of that noun.

Another issue is in the choice of a Naive Bayes Classifier. Since the Naive Bayes Classifier is notorious for producing polarized probability estimates that are close to zero or to one, the estimated probabilities are often inaccurate. However, as [15] points out, the classifier is surprisingly effective because it only needs to make an ordinal judgment (which class is more likely) to classify instances correctly. Similarly, our formula produces a reasonable *ordering* on the likelihood of extracted facts for a given class. This ordering is sufficient for KNOWITALL to implement the desired precision/recall tradeoff.

2.5 Training Discriminators

In order to estimate the probabilities $P(f_i|\phi)$ and $P(f_i|\neg\phi)$ needed in Equation 2, KNOWITALL needs a training set of positive and negative instances of the target class. We want our method to scale readily to new classes, however, which requires that we eliminate human intervention. To achieve this goal we rely on a bootstrapping technique that induces seeds from generic extraction patterns and automatically-generated discriminators.

Bootstrapping begins by instantiating a set of extraction rules and queries for each predicate from generic rule templates, and also generates a set of discriminator phrases from keyword phrases of the rules and from the class names. This gives a set of a few dozen possible discriminator phrases such as “country X”, “X country”, “countries such as X”, “X is a country”. We found it best to supply the system with two names for each class, such as “country” and “nation” for the class *Country*. This compensates for inherent ambiguity in a single name: “country” might be a music genre or refer to countryside; instances with high mutual information with both “country” and “nation” are more likely to have the desired semantic class.

Bootstrapping is able to find its own set of seeds to train the discriminators, without requiring any hand-chosen examples. It does this by using the queries and extraction rules to find a set of candidate seeds for each predicate. Each of these candidate seeds must have a minimum number of hit counts for the instance itself; otherwise the PMI scores from this seed will be unreliable.

After assembling the set of candidate seeds, Bootstrapping computes PMI(c,u) for each candidate seed c, and each untrained discriminator phrase u. The candidate seeds are ranked by average PMI score and the best m become the first set of bootstrapped seeds. Thus we can use untrained discriminator phrases to generate our first set of seeds, which we use to train the discriminators. Half of the seeds are used to find PMI thresholds for each discriminator, and the remaining seeds used to estimate conditional probabilities. An equal number of negative seeds is taken from among the positive seeds for other classes. Bootstrapping selects the best k discriminators to use for the Assessor, favoring those with the best split of positive and negative instances. Now that it has a set of trained discriminators, KNOWITALL does two more bootstrapping cycles: first, it uses the discriminators to re-rank the candidate seeds by probability; next, it selects a new set of seeds and re-trains the discriminators.

In the experiments reported in this paper, we used 100 candidate seeds, each with a hit count of at least 1,000, and picked the best 20 ($m = 20$). Finally, we set the number of discriminators k to 5. These settings

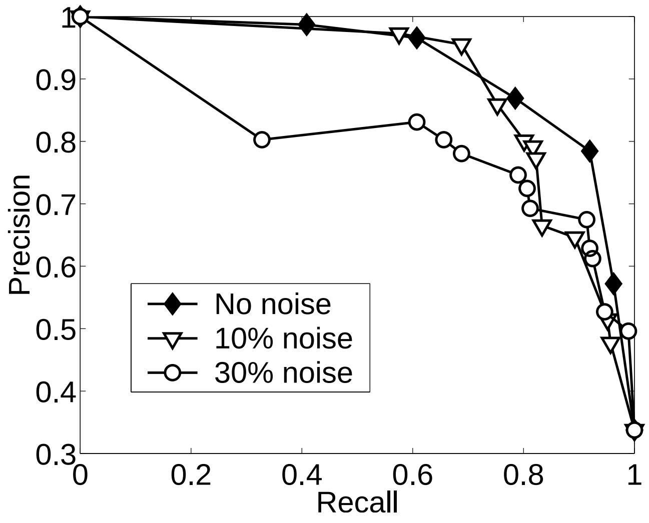
```
**BOOTSTRAP**(information focus I, rule templates T)
    R = generate rules from T for each predicate in I
    Q = generate queries associated with each rule in R
    U = generate untrained discriminators from rules in R, class names in I
    Use Q to find at least n candidate seeds for each predicate in I
        with hit counts > h
    First Iteration:
        S = select m candidate seeds for each predicate in I
            with highest average PMI over U
        D = train U on S, select best k discriminators for each predicate in I
    Subsequent Iterations:
        S = select m candidate seeds for each predicate in I
            with highest probability from D
        D = train U on S, select best k discriminators for each predicate in I
```

Figure 10: Pseudocode for Bootstrapping.

have been sufficient to produce correct seeds for all the classes we have experimented with thus far.

2.6 Bootstrapping and Noise Tolerance

An important issue with bootstrap training is robustness and noise tolerance: what is the effect on performance of the Assessor if the automatically selected training seeds include errors? Experiment 1 compares performance for `Country` trained on three different sets of seeds: correct seeds, seeds with 10% noise (2 errors out of 20 seeds), and seeds with 30% noise. The noisy seeds were actual candidate extractions that were not chosen by the full bootstrap process ("EU", "Middle East Countries", "Iroquois", and other instances semantically related to nation or country). There is some degradation of performance from 10% noise, and a sharp drop in performance from 30% noise.

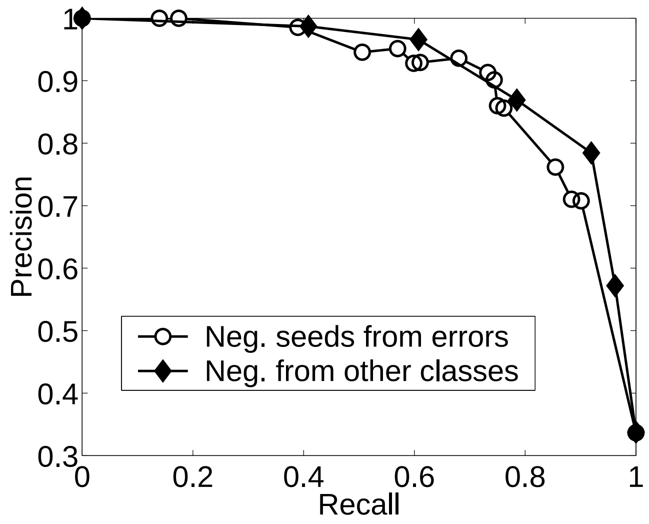


Experiment 1: The Assessor can tolerate 10% noise in bootstrapped training seeds up to recall 0.75, but performance degrades sharply after that.

Another question that troubled us is the source of negative seeds. Our solution was to train the Assessor on multiple classes at once; KNOWITALL finds negative seeds for a class by sampling positive seeds from other classes, as in [26]. We take care that each class has at least one semantically related class to provide near misses. In these experiments, `Country` gets negative seeds from `City`, `USState`, `Actor`, and `Film`, and so forth.

We tried the following alternative method of finding negative seeds. KNOWITALL runs its Extractor module to produce a set of unverified instances, then takes a random sample of those instances, which are hand-tagged as seeds. This training set has the added advantage of a representative proportion of positive and negative instances. Experiment 2 shows an experiment where a random sample of 40 extractions were hand-tagged as seeds. These seeds were then removed from the test set for that run. Surprisingly, the recall-precision curve is somewhat worse than selecting negative seeds from the other classes.

A key point in training the discriminators is to provide useful "near misses" as negative training. Using random words as negative training would nearly always give PMI scores of zero, and not produce accurate PMI thresholds or conditional probabilities. It turns out that actual extraction errors will often have zero PMI as well. Much better near misses come from using instances of classes that have a semantic relation to the target class. Instances of `City` and `USState` tend to co-occur with discriminator phrases for `Country`, and help the Assessor learn higher PMI thresholds and more conservative estimates of conditional probability.



Experiment 2: Using negative seeds that are taken from seeds of other classes works better than tagging actual extraction errors as negative seeds.

2.7 Resource Allocation

Our preliminary experiments demonstrated that KNOWITALL needs a policy that dictates when to stop looking for more instances of a predicate. For example, suppose that KNOWITALL is looking for instances of the predicate `Country`: there are only around 300 valid country names to find, but the Extractor could continue examining up to 3 million Web pages that match the query "countries including", "or other countries", and so forth. The valid country names would be found repeatedly, along with a large set of extraction errors. This would reduce efficiency – if KNOWITALL wastes queries on predicates that are already exhausted, this diverts system resources from the productive classes. Finding thousands of spurious instances can also overwhelm the Assessor and degrade KnowItAll's precision.

We use a *Signal to Noise* ratio (STN) to determine the utility of searching for further instances of a predicate. While the Extractor continues to find correct extractions at a fairly steady rate, the proportion of *new* extractions (those not already in the knowledge base) that are correct gradually becomes smaller over time. If nearly all the correct instances of a predicate are already in the knowledge base, new extractions will be mostly errors. Thus, the ratio of good extractions to noise of new extractions is a good indicator of whether KnowItAll has exhausted the predicate.

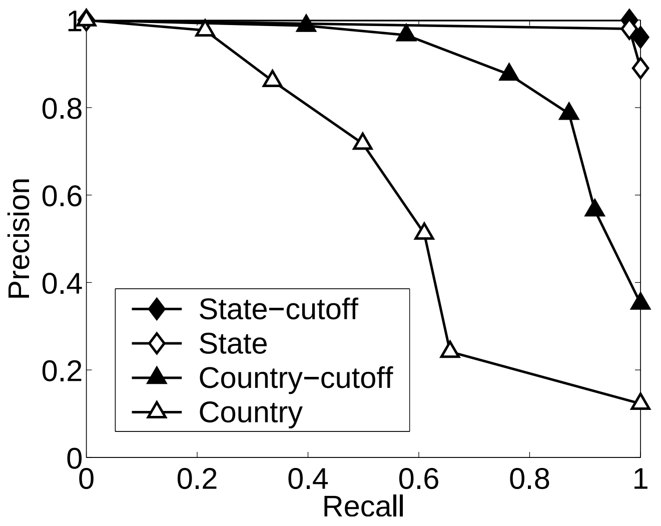
KnowItAll computes the STN ratio by dividing the number of high probability new extractions by the number of low probability ones over the most recent n Web pages examined for that predicate ($n = 5000$). A small smoothing term is added to numerator and denominator to avoid division by zero. When the STN ratio drops below a cutoff point, the Extractor is finding mostly noise, and halts search for that predicate. A cutoff of 0.10 means that there is ten times as much noise as good extractions.

The STN metric is a reflective, unsupervised computation, since KnowItAll has no outside source of information to tell it which instances are correct and which are noise. Instead, KnowItAll uses the probability estimates assigned by the Assessor, and defines "high probability" and "low probability" in terms of thresholds on these probabilities. In the experiments reported here, we used a STN cutoff of 0.10 and defined high probability as probabilities above 0.90 and low probabilities as those below 0.0001. The same settings were used for all predicates and all methods that included PMI probability assessment. The setting of 0.0001 for low probability is due to the Nave Bayes probability updates tendency to polarize the probability estimates. Relying on probability assignments by the Assessor is a limitation of the STN metric: We typically run the List Extractor without using PMI assessment.[5] LE uses an alternate Assessor method that assigns higher probability to instances that are found on a larger number of lists. This method is not suitable for a STN cutoff that is computed over new extractions, since all new extractions are necessarily on only a single list so far, thus all new extractions have "low probability".

We used an additional cutoff metric, the *Query Yield Ratio* (QYR), and halt search for new instances when either STN or QYR falls below 0.10. QYR is defined as the ratio of query yield over the most recent n Web pages examined, divided by the initial query yield over the first n Web pages, where query yield is the number of new extractions divided by the number of Web pages examined (adding a small smoothing term to avoid division by zero). If this ratio falls below a cutoff point, the Extractor has reached a point of diminishing returns where it is hardly finding any new extractions and halts the search for that predicate. The ratio of recent query yield to initial query yield is a better indicator that a predicate is nearly exhausted than using a cutoff on the query yield itself. The query yield varies greatly depending on the predicate and the extraction method used: the query yield for learned rules tends to be lower than for rules from generic patterns; the List Extractor method, where one query can produces a hundred extractions or more, has much higher query yield than the other KnowItAll extraction methods.

Experiment 3 shows the impact of the cutoff metrics. The top curve is for `USState` where KnowItAll automatically stopped looking for further instances after the STN fell below 0.10 after finding 371 proposed state names. The curve just below that is for `USState` when KnowItAll kept searching and found 3,927 proposed state names. In fact, none of the states found after the first few hours were correct, but enough of the errors fooled the Assessor to reduce precision from 1.0 to 0.98 at the highest probability. The next two curves show `Country` with and without cutoff metrics. KnowItAll found 194 correct and 357 incorrect Country names with the cutoff metrics; it found 387 correct Countries, but also 2,777 incorrect extractions

[5]A metric that does not rely on the Assessor is also useful for predicates with discriminators that provide only weak evidence for probability assignment.



Experiment 3: A comparison of `USState` and `Country` with and without metrics to cut off the search for more instances of exhausted predicates. Our cutoff metrics not only aid efficiency, but improve precision.

without cutoff metrics. The data point at precision 0.88 and recall 0.76 with cutoff metrics represents 148 correct instances; without cutoff metrics, the point at precision 0.86 and recall 0.34 represents 130 correct instances. So continuing the search actually produced fewer correct instances at a given precision level.

2.8 Extended Example

To better understand how KNOWITALL operates, we present a detailed example of learning facts about geography. A user has given KNOWITALL a set of predicates including `City`, and KNOWITALL has used domain-independent rule templates to generate extraction rules and untrained discriminator phrases for `City` as described in Section 2.2.

Bootstrapping automatically selected seeds to train discriminators for `City` that include prominent cities like London and Rome, and the obscure cities Dagupan and Shakhrisabz. Negative training comes from seeds for other classes trained at the same time, including names of countries and U.S. states. After training all discriminator phrases with these seeds, Bootstrapping has selected the five best discriminators shown in Figure 11. The thresholds are from one training set of 10 positive and 10 negative seeds; the conditional probabilities come from another training set, with a smoothing factor of 1 added to the count of positive or negative above and below the threshold.

Once Bootstrapping has generated the set of extraction rules and trained a set of discriminators for each predicate, KNOWITALL begins its main extraction cycle. Each cycle, KNOWITALL selects a set of queries, sends them to a search engine, and uses the associated extraction rules to analyze the Web pages that it downloads.

Suppose that the query is “and other cities”, from a rule with extraction pattern: NP “and other cities”. Figure 12 shows two sentences that might be found by the query for this rule. The extraction rule correctly extracts “Fes” as a city from the first sentence, but is fooled by the second sentence, and extracts “East Coast” as a city.

To compute the probability of `City(Fes)`, the Assessor sends six queries to the Web, and finds the following hit counts. “Fes” has 446,000 hits; “Fes is a city” has 14 hits, giving a PMI score of 0.000031 for this discriminator, which is over the threshold for this discriminator. A PMI score over threshold for this

Discriminator: <I> is a city
Learned Threshold T: 0.000016
P(PMI $>$ T $|$ class) = 0.83
P(PMI $>$ T $|$ $\neg$class)= 0.08

Discriminator: <I> and other towns
Learned Threshold T: 0.00000075
P(PMI $>$ T $|$ class) = 0.83
P(PMI $>$ T $|$ $\neg$class)= 0.08

Discriminator: cities <I>
Learned Threshold T: 0.00044
P(PMI $>$ T $|$ class) = 0.91
P(PMI $>$ T $|$ $\neg$class)= 0.25

Discriminator: cities such as <I>
Learned Threshold T: 0.0000053
P(PMI $>$ T $|$ class) = 0.75
P(PMI $>$ T $|$ $\neg$class)= 0.08

Discriminator: cities including <I>
Learned Threshold T: 0.0000047
P(PMI $>$ T $|$ class) = 0.75
P(PMI $>$ T $|$ $\neg$class)= 0.08

Figure 11: Trained discriminators for the class `City`. Bootstrapping has learned a threshold on PMI scores that splits positive from negative training seeds, and has estimated conditional probabilities that the PMI score is above that threshold, given that the extraction is of the class or not of the class.

"Short flights connect Casablanca with Fes and other cities."

"Since 1984, the ensemble has performed concerts throughout the East Coast and other cities."

Figure 12: Two sentences that may be found by queries "and other cities". The Assessor needs to distinguish between a correct extraction of Fes from the first sentence and an extraction error, East Coast, from the second.

discriminator is 10 times more likely for a correct instance than for an incorrect one, raising the probability that Fes is a city. Fes is also above threshold for “cities Fes” (201 hits); “cities such as Fes” (10 hits); and “cities including Fes” (4 hits). It is below threshold on only one discriminator, with 0 hits for “Fes and other towns”. The final probability is 0.99815.

In contrast, the Assessor finds that `City(East Coast)` is below threshold for all discriminators. Even though there are 141 hits for “cities East Coast”, 1 hit for “cities such as East Coast”, and 3 hits for “cities including East Coast”, the PMI scores are below threshold when divided by 1.7 million hits for “East Coast”. The final probability is 0.00027.

2.9 Experiments with Baseline KnowItAll

We ran an experiment to evaluate the performance of KNOWITALL as thus far described. We were particularly interested in quantifying the impact of the Assessor on the precision and recall of the system. The Assessor assigns probabilities to each extraction. These probabilities are the system’s confidence in each extraction and can be thought of as analogous to a ranking function in information retrieval: the goal is for the set of extractions with high probability to have high precision, and for the precision to decline gracefully as the probability threshold is lowered. This is, indeed, what we found.

We ran the system with an information focus consisting of five classes: `City`, `USState`, `Country`, `Actor`, and `Film`. The first three had been used in system development and the last two, `Actor` and `Film`, were new classes. The Assessor used PMI score thresholds as Boolean features to assign a probability to each extraction, with the system selecting the best five discriminator phrases as described in Section 2.4.

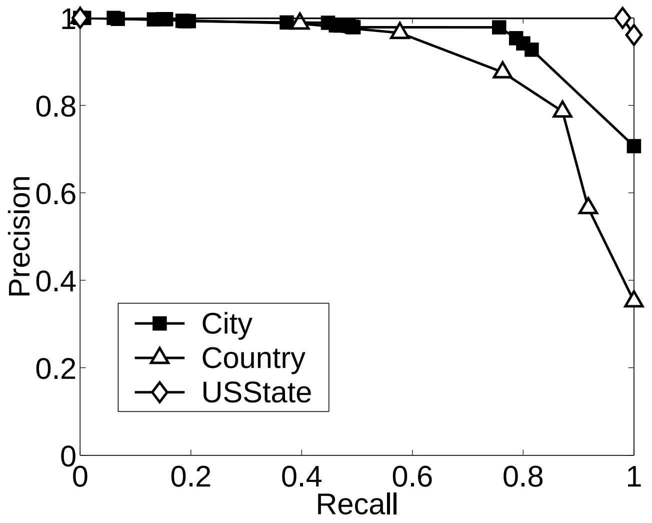
We use the standard metrics of *precision* and *recall* to measure KNOWITALL’s performance. At each probability *p* assigned by the Assessor, we count the number of correct extractions at or above probability *p*. This is done by first comparing the extracted instances automatically with an external knowledge base, the Tipster Gazetteer for locations and the Internet Movie Database (IMDB) for actors and films. We manually checked any instances not found in the Gazetteer or the IMDB to ensure that they were indeed errors.

Precision at *p* is the number of correct extractions divided by the total extractions at or above *p*. Recall at *p* is defined as the number of correct extractions at or above *p* divided by the total number of correct extractions at all probabilities. Note that this is recall with respect to sentences that the system has actually seen, and the extraction rules it utilizes, rather than a hypothetical, but unknown, number of correct extractions possible with an arbitrary set of extraction rules applied to the entire Web.

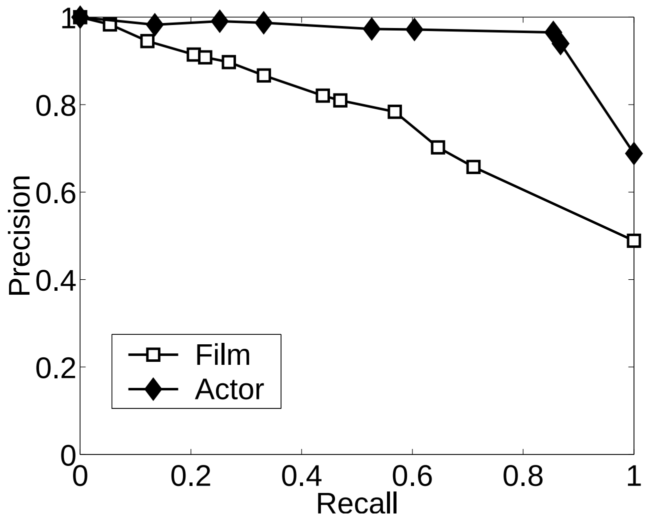
Experiments 4 and 5 show precision and recall at the end of running KNOWITALL for four days. Each point on the curves shows the precision and recall for extractions with probability at or above a given level. The curve for `City` has precision 0.98 at recall 0.76, then drops to precision 0.71 at recall 1.0. The curve for `USState` has precision 1.0 at recall 0.98; `Country` has precision 0.97 at recall 0.58, and precision 0.79 at recall 0.87.

Performance on the two new classes (`Actor` and `Film`) is on par with the geography domain we used for system development. The class `Actor` has precision 0.96 at recall 0.85. KNOWITALL had more difficulty with the class `Film`, where the precision-recall curve is fairly flat, with precision 0.90 at recall 0.27, and precision 0.78 at recall 0.57.

Our precision/recall curves also enable us to precisely quantify the impact of the Assessor on KNOWITALL’s performance. If the Assessor is turned off, then KNOWITALL’s output corresponds to the point on the curve where the recall is 1.00. The precision, with the Assessor off, varies between classes: for `City` 0.71, `USState` 0.96, `Country` 0.35, `Film` 0.49, and `Actor` 0.69. Turning the Assessor on enables KNOWITALL to achieve substantially higher precision. For example, the Assessor raised precision for `Country` from 0.35 to 0.79 at recall 0.87.



Experiment 4: Precision and recall at the end of four days at varying probability thresholds for the classes `City`, `USState`, and `Country`. KNOWITALL maintains high precision up to recall .80 for these classes.



Experiment 5: Precision and recall at the end of four days for two new classes: `Actor` and `Film`. KNOWITALL maintains high precision for actors, but has less success with film titles.

The Assessor is able to do a good job of assigning high probabilities to correct instances with only a few false positives. Most of the extraction errors are of instances that are semantically close to the target class. The incorrect extractions for `Country` with probability > 0.80 are nearly all names of collections of countries: "NAFTA", "North America", and so forth. Some of the errors at lower probability are American Indian tribes, which are often referred to as "nations". Common errors for the class `Film` are names of directors, or partial names of films (a film named "Dalmatians" instead of "101 Dalmatians").

The Assessor has more trouble with false negatives than with false positives. Even though a majority of the instances at the lowest probabilities are incorrect extractions, many are actually correct. An instance that has a relatively low number of hit counts will often fall below the PMI threshold for discriminator phrases, even if it is a valid instance of the class. An instance receives a low probability if it fails more than half of the discriminator thresholds, even if it is only slightly below the threshold each time.

3 Extending KnowItAll with Pattern Learning

While generic extraction patterns perform well in the baseline KNOWITALL system, many of the best extraction rules for a domain do not match a generic pattern. For example, "the film <*film*> starring" and "headquartered in <*city*>" are rules with high precision and high coverage for the classes `Film` and `City`. Arming KNOWITALL with a set of such domain-specific rules can significantly increase the number of sentences from which it can extract facts. This section describes our method for learning domain-specific rules. As shown in Figure 13, we introduce the insight that Pattern Learning (PL) can be used to increase both coverage (by learning extractors) and accuracy (by learning discriminators). We quantify the efficacy of this approach via experiments on multiple classes, and describe design decisions that enhance the performance of Pattern Learning over the Web.

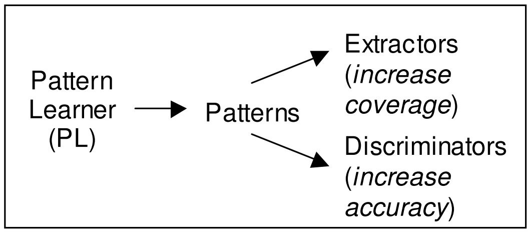

Pattern Learner (PL)
Patterns
Extractors (*increase coverage*)
Discriminators (*increase accuracy*)

Figure 13: The patterns that PL produces can be used as both extractors and discriminators.

3.1 Learning Patterns

Our Pattern Learning algorithm proceeds as follows:

1. Start with a set I of seed instances generated by domain-independent extractors.

2. For each seed instance i in I: Issue a query to a Web search engine for i, and for each occurrence of i in the returned documents record a context string comprised of the w words before i, a placeholder for the class instance (denoted by "<class-name>"), and the w words after i. (Here, we use w = 4).[6]

[6]Limited-length context strings form a rather impoverished hypothesis space for PL, but the space was adequate in our experiments. The other advantage of the strings, compared with more expressive languages for expressing PL patterns, is that the strings can be used directly as search engine queries when the patterns are employed to generate and assess candidate instances.

3. Output the best *patterns* according to some metric—a pattern is defined as any substring of a context string that includes the instance placeholder and at least one other word.

The goal of PL is to find high-quality patterns. A pattern's quality is given by its *recall* (the fraction of instances of the target class that can be found on the Web surrounded by the given pattern text) and its *precision* (the fraction of strings found surrounded by the pattern text that are of the target class). The Web contains a large number of candidate patterns (for example, PL found over 300,000 patterns for the class City), most of which are of poor quality. Thus, estimating the precision and recall of patterns efficiently (i.e. without searching the Web for each candidate pattern) is important. Estimating precision for patterns is especially difficult because we have no labeled negative examples, only positive seeds. Instead, in a manner similar to [26] we exploit the fact that PL learns patterns for multiple classes at once, and take the positive examples of one class to be negative examples for all other classes. Given that a pattern p is found for $c(p)$ distinct seeds from the target class and $n(p)$ distinct seeds from other classes, we define:

$$EstimatedPrecision = \frac{c(p)+k}{c(p)+n(p)+m} \tag{3}$$

$$EstimatedRecall = \frac{c(p)}{S} \tag{4}$$

where S is the total number of seeds in the target class, and k/m is a constant prior estimate of precision, used to perform a Laplace correction in (3). The prior estimate was chosen based on testing extractions from a sample of the learned patterns using PMI Assessment.

3.2 Learned Patterns as Extractors

The patterns PL produces can be used as extractors to search the Web for new candidate facts. For example, given the learned pattern "headquartered in <*city*>," we search the Web for pages containing the phrase "headquartered in". Any proper noun phrase occurring directly after "headquartered in" in the returned documents becomes a new candidate extraction for the class `City`.

Of the many patterns PL finds for a given class, we choose as extractors those patterns most able to efficiently generate new extractions with high precision. The patterns we select must have high precision, and extractor *efficiency* (the number of unique instances produced per search engine query) is also important.

For a given class, we first select the top patterns according to the following heuristics:

H1: As in [6], we prefer patterns that appear for multiple distinct seeds. By banning all patterns found for just a single seed (i.e. requiring that $EstimatedRecall > 1/S$ in Equation 4), 96% of the potential rules are eliminated. In experiments with the class City, H1 was found to improve the average efficiency of the resulting patterns by a factor of five.

H2: We sort the remaining patterns according to their $EstimatedPrecision$ (Equation 3). On experiments with the class City, ranking by H2 was found to further increase average efficiency (by 64% over H1) and significantly improve average precision (from 0.32 to 0.58).

Of all the patterns PL generates for a given class, we take the 200 patterns that satisfy H1 and are ranked most highly by H2 and subject them to further analysis, applying each to 100 Web pages and testing precision using PMI assessment.

3.2.1 Experimental Results

We performed experiments testing our Baseline system (KNOWITALL with only domain independent patterns) against an enhanced version, Baseline+PL (KNOWITALL including extractors generated by Pattern Learning). In both configurations, we perform PMI assessment to assign a probability to each extraction (using only domain independent discriminators). We estimated the coverage (number of unique instances extracted) for both configurations by manually tagging a representative sample of the extracted instances, grouped by probability. In the case of `City`, we also automatically marked instances as correct if they appeared in the Tipster Gazetteer. To ensure a fair comparison, we compare coverage at the same level of overall precision, computed as the proportion of correct instances at or above a given probability. We used the Google search engine in all experiments.

The results shown in Experiments 10 and 11 in Section 6 show that using learned patterns as extractors improves KNOWITALL's coverage substantially. Examples of the most productive extractors for each class are shown in Table 1.

Rule	*Correct Extractions*	*Precision*
the cities of <*city*>	5215	0.80
headquartered in <*city*>	4837	0.79
for the city of <*city*>	3138	0.79
in the movie <*film*>	1841	0.61
<*film*> the movie starring	957	0.64
movie review of <*film*>	860	0.64
and physicist <*scientist*>	89	0.61
physicist <*scientist*>,	87	0.59
<*scientist*>, a British scientist	77	0.65

Table 1: Three of the most productive rules for each class, along with the number of correct extractions produced by each rule, and the rule's overall precision (before assessment).

3.3 Learned Patterns as Discriminators

Learned patterns can also be used as discriminators to perform PMI assessment. As described above, the PMI scores for a given extraction are used as features in a Naive Bayes classifier. In the experiments below, we show that learned discriminators provide stronger features than domain independent discriminators for the classifier, improving the *classification accuracy* (the percentage of extractions classified correctly) of the PMI assessment.

Once we have a large set of learned discriminators, determining which discriminators are the "best" in terms of their impact on classification accuracy becomes especially important, as we have limited access to Web search engines. In the baseline KNOWITALL system, the same five discriminators are executed on every extraction. However, it may be the case that a discriminator will perform better on some extractions than it does on others. For example, the discriminator "cities such as <*city*>" has high precision, but appears only rarely on the Web. While a PMI score of $1/100,000$ on "cities such as <*city*>" may give strong evidence that an extraction is indeed a city, if the city itself appears only a few thousand times on the Web, the probability of the discriminator returning a false zero is high. For these rare extractions, choosing a more prevalent discriminator (albeit one with lower precision) like "<*city*> hotels" might offer better performance. Lastly, executing five discriminators on every extraction is not always the best choice. For example, if the first few discriminators executed on an extraction have high precision and return true, the system's resources would be better spent assessing other extractions, the truth of which is less certain.

In [18] formalizes the problem of choosing which discriminators to execute on which extractions as an optimization problem, and describes a heuristic method that includes the enhancements mentioned above. The paper shows that the heuristic has provably optimal behavior in important special cases, and then verifies experimentally that the heuristic improves accuracy.

3.4 Related Work

PL is similar to existing approaches to pattern learning, the primary distinction being that we use learned patterns to perform PMI-IR [42] assessment as well as extraction. PL also differs from other pattern learning algorithms in some details. Riloff and Jones [37] use bootstrapped learning on a small corpus to alternately learn instances of large semantic classes and4 patterns that can generate more instances; similar bootstrapping approaches that use larger corpora include Snowball [3] and DIPRE [6]. Our work is similar to these approaches, but differs in that PL does not use bootstrapping (it learns its patterns once from an initial set of seeds) and uses somewhat different heuristics for pattern quality. Like our work, Ravichandran and Hovy [36] use Web search engines to find patterns surrounding seed values. However, their goal is to support *question answering*, for which a training set of question and answer pairs is known. Unlike PL, they can measure a pattern's precision on seed questions by checking the correspondence between the extracted answers and the answers given by the seed. As in work by Riloff [41] and others, PL uses the fact that it learns patterns for multiple classes at once to improve precision. The particular way we use multiple classes to estimate a pattern's precision (Equation 3) is similar to that of Lin *et al.* [26]. A unique feature of our approach is that our heuristic is computed solely by searching the Web for seed values, instead of searching the corpus for each discovered pattern.

A variety of work in information extraction has been performed using more sophisticated structures than the simple patterns that PL produces. Wrapper induction algorithms [24, 30] attempt to learn wrappers that exploit the structure of HTML to extract information from Web sites. Also, a variety of rule-learning schemes [40, 7, 8] have been designed for extracting information from semi-structured and free text. Similarly, richer language models have been used to learn lexico-syntactic patterns that identify examples of the hyponym relation [39]. In this paper, we restrict our attention to simple text patterns, as they are the most natural fit for our approach of leveraging Web search engines for both extraction and PMI assessment. For extraction, it may be possible to use a richer set of patterns with Web search engines given the proper query generation strategy [2]; this is an item of future work.

4 Subclass Extraction

Another method to extend KNOWITALL's recall is Subclass Extraction (SE), which automatically identifies subclasses. For example, not all scientists are found in sentences that identify them as "scientist" – some are referred to only as chemists, some only as physicist, some only as biologists, and so forth. If SE learns these and other subclasses of scientist, then KNOWITALL can create extraction patterns to find a larger set of scientists.

As it turns out, subclass extraction can be achieved elegantly by a recursive application of KNOWITALL's main loop (with some extensions). In the following, we describe the basic subclass extraction method (SE_{base}), discuss two variations (SE_{self} and SE_{iter}) aimed at increasing SE's recall, and present encouraging results for a number of different classes.

4.1 Extracting Candidate Subclasses

In general, the SE_{base} extraction module has the same design as the original KNOWITALL extraction module. Its input consists of domain-independent extraction rules for generating candidate terms, for which matches are found on the Web. The generic rules that extract instances of a class will also extract subclasses, with some modifications. To begin with, the rules need to distinguish between instances and subclasses of a class. The rules for extracting instances in Section 2.1 contain a proper noun test (using a part-of-speech tagger and a capitalization test). Rules for extracting subclasses instead check that the extracted noun is a common noun (i.e., not capitalized). While these tests are heuristic, they work reasonably well in practice, and KNOWITALL also falls back on its Assessor module to weed out erroneous extractions. The patterns for our subclass extraction rules appear in Table 2. Most of our patterns are simple variations of well-known ones in the information-extraction literature [22]. C_1 and C_2 denote known classes and "CN" denotes a common noun or common noun phrase. Note that the last two rules can only be used once some subclasses of the class have already been found.

Pattern	Extraction
C_1 {","} "such as" CN	$isA(CN, C_1)$
"such" C_1 "as" CN	$isA(CN, C_1)$
CN {","} "and other" C_1	$isA(CN, C_1)$
CN {","} "or other" C_1	$isA(CN, C_1)$
C_1 {","} "including" CN	$isA(CN, C_1)$
C_1 {","} "especially" CN	$isA(CN, C_1)$
C_1 "and" CN	$isA(CN, class(C_1))$
C_1 {","} C_2 {","} "and" CN	$isA(CN, class(C_1))$

Table 2: Rules for Subclass Extraction, where CN is a common noun identified by these patterns as a subclass of the class C_1. In the last two rules CN is a sibling class of classes C_1 and C_2. The {","} indicates an optional comma in the pattern.

4.2 Assessing Candidate Subclasses

SE uses a generate-and-test technique for extracting subclasses, much as the main KNOWITALL algorithm does for extracting instances. The SE_{base} Assessor uses a combination of methods to decide which of the candidate subclasses from the SE_{base} Extractor are correct. First, the Assessor checks the morphology of the candidate term, since some subclass names are formed by attaching a prefix to the name of the class (e.g., "microbiologist" is a subclass of "biologist"). Then the Assessor checks whether a subclass is a hyponym of the class in WordNet and if so, it assigns it a very high probability. The rest of the extractions are evaluated in a manner similar to the instance assessment in KNOWITALL (with some modifications). The Assessor computes co-occurrence statistics of candidate terms with a set of class discriminators. Such statistics represent features that are combined in a naive Bayesian probability update. The SE_{base} Assessor uses a bootstrap training method similar to that described in Section 2.5.

Initially, we had hoped to use instance information as part of the assessment process. For instance, if a proposed subclass had extracted instances that are also instances of the target class, this would have boosted the probability of it being a true subclass. However, our instance sampling procedure revealed that reliable instances for a number of correct proposed subclasses could not be extracted (with generic rules) as instances of the target superclass. Apparently some classes, like Scientist, are very general and naturally decomposable, and so people tend to use more specific subclasses of the class when writing. Classes like

Physicist or City, on the other hand, are used more frequently together with instances, and they have far fewer useful subclasses.

4.3 Context-independent and Context-dependent Subclasses

Before presenting our experimental results, we need to introduce two key distinctions. We distinguish between finding subclasses in a *context-independent* manner versus finding subclasses in a *context-dependent* manner. The term *context* refers to a set of keywords provided by the user that suggest a knowledge domain of interest (*e.g.*, the pharmaceutical domain, the political domain, etc.). In the absence of a domain description, KNOWITALL finds subclasses in a context-independent manner and they can differ from context-dependent subclasses. For instance, if we are looking for any subclasses of Person (or People), Priest would be a good candidate. However, if we are looking for subclasses of Person (or People) in a Pharmaceutical context, Priest is probably not a good candidate, whereas Pharmacist is.

We also distinguish between *named subclasses* and *derived subclasses*. *Named subclasses* are represented by novel terms, whereas *derived subclasses* are phrases whose head noun is the same as the name of the superclass. For instance, *Capital* is a named subclass of *City*, whereas *European City* is a derived subclass of *City*. While derived subclasses are interesting in themselves, we focus on the extraction of named subclasses, as they are more useful in increasing KNOWITALL's instance recall. The reason is that extraction rules that use derived subclasses tend to extract a lot of the same instances as the rules using the name of the superclass.

We now turn to our experimental results. We have evaluated our basic subclass extraction method in two different settings.

a) **Context-independent SE** First, we chose three classes, Scientist, City and Film and looked for context-independent subclasses using the SE_{base} approach described above. SE_{base} found only one named subclass for City, "capital", which is also the only one listed in the WordNet hyponym hierarchy for this class. SE_{base} found 8 correct subclasses for Film and 11 for Scientist—this confirmed our intuition that subclass extraction would be most successful on general classes, such as Scientist and least successful on specific classes such as City. As shown in Experiment 7, we have evaluated the output of SE_{base} along four metrics: precision, recall, total number of correct subclasses and proportion of (correct) subclasses found that do not appear in WordNet. As we can see, SE_{base} has high-precision but relatively low recall, reflecting the low recall of our domain-independent patterns.

b) **Context-dependent SE** A second evaluation of SE_{base} (Experiment 8) was done for a context-dependent subclass extraction task, using as input three categories that were shown to be productive in previous semantic lexicon acquisition work [35]: People, Products and Organizations in the Pharmaceutical domain.[7] SE_{base} exhibits the same high-precision/low-recall behavior we noticed in the context-independent case. We also notice that most of the subclasses of People and Organizations are in fact in WordNet, whereas none of the found subclasses for Products in the Pharmaceutical domain appears in WordNet.

Next, we investigate two methods for increasing the recall of the subclass extraction module.

4.4 Improving Subclass Extraction Recall

Generic extraction rules have low recall and do not generate all of the subclasses we would expect. In order to improve our subclass recall, we add another extraction-and-verification step. After a set of subclasses for the given class is obtained in the manner of SE_{base}, the last two enumeration rules in Table 2 are seeded with

[7]For context-dependent subclass extraction, the search engine queries contain a relevant keyword together with the instantiated extraction rule (for instance, "pharmaceutical" in the case of the Pharmaceutical domain).

known subclasses and extract additional subclass candidates. For instance, given the sentence "Biologists, physicists and chemists have convened at this inter-disciplinary conference.", such rules identify "chemists" as a possible sibling of "biologists" and "physicists". We experiment with two methods, SE_{self} and SE_{iter} in order to assess the extractions obtained at this step.

a) SE_{self} is a simple assessment method based on the empirical observation that an extraction matching a large number of different enumeration rules is likely to be a good subclass candidate. We have tried to use the enumeration rules directly as features for a Naive Bayes classifier, but the very nature of the enumeration rule instantiations ensures that positive examples don't have to occur in any specific instantiation, as long they occur frequently enough. We simply convert the number of different enumeration rules matched by each example and the average number of times an example matches its corresponding rules into Boolean features (using a learned threshold). Since we have a large quantity of unlabeled data at our disposal, we estimate the thresholds and train a simple Naive-Bayes classifier using the *self-training* paradigm [31], chosen as it has been shown to outperform EM in a variety of situations. At each iteration, we label the unlabeled data and retain the example labeled with highest confidence as part of the training set. The procedure is repeated until all the unlabeled data is exhausted. The extractions whose probabilities are greater than 0.8 represent the final set of subclasses (since subclasses are generally used by KNOWITALL for instance extraction, bad subclasses translate into time wasted by the system and as such, we retain only candidate subclasses whose probability is relatively high).

b) SE_{iter} is a heuristic assessment method that seeks to adjust the probabilities assigned to the extractions based on *confidence scores* assigned to the enumeration rules in a recursive fashion. The *confidence score* of a rule is given by the average probability of extractions matched by that rule. After rule confidence scores have been determined, the extraction matching the most rules is assigned a probability $p = \frac{c(R1)+c(R2)}{2}$, where $R1$ and $R2$ are the two matching rules with highest confidence scores. The rule confidence scores are then re-evaluated and the process ends when all extractions have been assigned a probability. This scheme has the effect of clustering the extractions based on the rules they match and it works to the advantage of good subclasses that match a small set of good extraction rules. However, as we will later see, this method is sensitive to noise. As in the case of SE_{self}, we only retain the extractions whose probability is greater than 0.8.

4.5 Experimental Results

We evaluated the methods introduced above on two of the three context-independent classes (Scientist and Film) in Experiment 7.[8] We also evaluated the methods on all three Pharmaceutical domain classes (People, Product, Organization) in Experiment 8. We found that both SE_{self} and SE_{iter} significantly improved upon the recall of the baseline method; for both, this increase in recall is traded for a loss in precision. SE_{iter} has the highest recall, at the price of an average 2.3% precision loss with respect to SE_{base}. In the future, we will perform additional experiments to assess which one of the two methods is less sensitive to noise, but based upon inspection of the test set and the behavior of both methods, SE_{self} appears more robust to noise than SE_{iter}.

Another potential benefit of subclass extraction is an increase in the number of class instances that KNOWITALL is able to extract from the Web. In the case of the Scientist class, for example, the number of scientists extracted by KNOWITALL at precision 0.9 increased by a factor of 5. SE_{iter} was used to extract subclasses and add them to the ontology. We do not see this benefit for classes such as City, where most of

[8]We didn't have enough subclasses to instantiate enumeration patterns for City as SE_{base} only identified one named City subclass.

Method	**Scientist**				**Film**			
	Precision	Recall	NW	Total	Precision	Recall	NW	Total
SE_{base}	0.91	0.28	0.08	11	1.0	0.36	0.5	8
SE_{self}	0.87	0.69	0.15	27	0.94	0.77	0.82	**17**
SE_{iter}	0.84	0.74	0.17	**29**	0.93	0.68	0.8	16

Experiment 6: Results of the 3 Subclass Extraction methods (SE_{base}, SE_{self} and SE_{iter}) for the `Scientist` and `Film` classes. For each method, we report Precision, Recall, NW, and Total. Recall is defined in terms of the union of correct subclasses from all methods. Total is the number of correct subclasses found. NW is the proportion of correct subclasses missing from WordNet. The baseline system has high precision, but low recall. Both extensions to SE increased recall dramatically with only a small drop in precision.

Method	**People**				**Organization**				**Product**			
	Precision	Recall	NW	Total	Precision	Recall	NW	Total	Precision	Recall	NW	Total
SE_{base}	1.0	0.28	0.07	14	0.92	0.20	0.09	11	0.88	0.44	1.0	31
SE_{self}	1.0	0.86	0.02	42	0.87	0.84	0.36	47	0.86	0.74	1.0	51
SE_{iter}	0.95	0. 94	0.02	**46**	0.89	0.95	0.22	**52**	0.84	0.88	1.0	**62**

Experiment 7: Results for the Pharmaceutical domain of the 3 Subclass Extraction methods (SE_{base}, SE_{self} and SE_{iter}). The extensions to SE give a large increase in recall with only a small drop in precision, as they do with domain-independent experiments.

the extracted subclasses are derived subclasses (e.g., "European City"). The reason is that extraction rules that use derived subclasses tend to extract a lot of the same instances as the rules using the name of the superclass (see Table 2).

4.6 Discussion

It is somewhat surprising that simple features such as the number of rules matching a given extraction are such good predictors of a candidate representing a subclass. We attribute this to the redundancy of Web data (we were able to find matches for a large number of our instantiated candidate rules) and to the semantics of the enumeration patterns. The subclass sets from SE_{self} and SE_{iter} contain many of the same candidates, although SE_{iter} typically picks up a few more.

Another interesting observation is that the different sets of extracted subclasses have widely varying degrees of overlap with the hyponym information available in WordNet. In fact, all but one of the subclasses identified for People are in WordNet, whereas none of those Products appear there (*e.g.*, Antibiotics, Antihistamines, Compounds, etc.). In the case of Organizations, there is a partial overlap with WordNet and it is interesting that terms that can refer both to a Person and an Organization ("Supplier", "Exporter" etc.) tend to appear only as subclasses of Person in WordNet, although they are usually found as subclasses of Organizations by KNOWITALL's subclass extraction methods.

5 List Extractor

We now present the third method for increasing KNOWITALL's recall, the List Extractor (LE). Where the methods described earlier extract information from unstructured text on Web pages, LE uses regular page structure to support extraction. LE locates lists of items on Web pages, learns a wrapper on the fly for each list, automatically extracts items from these lists, then sorts the items by the number of lists in which they appear.

LE locates lists by querying search engines with sets of items extracted by the baseline KNOWITALL (*e.g.*, LE might query Google with “London” “Paris” “New York” “Rome”). LE leverage the fact that many informational pages are generated from databases and therefore have a distinct, but regular and easy-to-learn structure. We combine ideas from previous work done on wrapper induction in our implementation of LE to learn wrappers quickly (in under a second of CPU time per document) and autonomously (unlike much of the work on wrapper induction, LE is unsupervised).

5.1 Background and Related Work

One of the first applications of wrapper learning appeared in [16], which describes an agent that queried online stores with known product names and looked for regularities in the resulting pages in order to build e-commerce wrappers. In [24], Kushmerick generalized how to automatically learn wrappers for information extraction, and presented wrappers as regular expressions with some kind of structure or constraints. The idea is that given a fully labeled training set of sample extractions from documents, one can learn a wrapper or patterns of words that precede and follow the extracted terms. In addition to the prefixes and suffixes, there is also a notion of heads and tails, which are points that delimit the context to which the extraction pattern applies.

The base algorithm for wrapper induction is fairly straightforward. Given fully labeled texts (or oracles) in which negative examples are those parts without labels, iterate over all possible patterns to find the best heads, tails, prefixes, and suffixes, that match all the training data, and use these for extraction. The complexity and accuracy depends on the expressiveness of the expressions (i.e. wild cards, semantic/synonym matches, etc.), the amount of data to learn from, and the level of structure in the documents.

Cohen in [11] extended the notion of wrapper induction by generalizing how to automatically learn rules to include linear regular expressions as well as hierarchical paths (DOM parse) in an HTML document. Cohen also explored how to use these wrappers to automatically extract arbitrary lists of related items from Web pages for other purposes [10]. We borrow both of these ideas in our implementation, but differ in how our wrapper is trained, used, and measured experimentally.

Perhaps the work that most resembles LE is Google Sets, which is an interface provided by Google that functionally appears almost identical to LE. The input to Google Sets is several words, and the output is a list of up to 100 tokens that are found in lists on the Web. Since we do not know how Google Sets is implemented and cannot get unlimited results from their interface, we are unable to compare the two systems.

5.2 Problem Definition and Characteristics

The inputs to LE include the name of a class and a set of positive seeds. The output is a set of candidate tokens for the given class that are found on Web pages containing lists of instances, where the list includes a subset of the positive seeds. We take advantage of the repetition of information on the Web by being highly selective on which documents we choose to extract from. In particular, we want documents that contain many known positive examples and that exhibit a high amount of structure from which we can infer new examples. It is reasonable to assume that this structure exists for many classes, since many professional Web sites are automatically generated from databases.

We do not have negative examples, so any learning procedure we use will have to rely on positive examples only. This means that as we carve out a space that we believe separates the positive instances

```
LISTEXTRACTOR(seedExamples)
    documents = searchForDocuments(seedExamples)
    For each document in documents
        parseTree = ParseHTML(document)
        For each subtree in parseTree
            keyWords = findAllSeedsInTree(subtree)
            prefix = findBestPrefix(keyWords, subtree)
            suffix = findBestSuffix(keyWords, subtree)
            Add to wrapperTree from createWrapper(prefix, suffix))
        For each goodWrapper in wrapperTree
            Find extractions using goodWrapper
    Return list of extractions
```

Figure 14: High-level pseudocode for List Extractor

from the negative ones, we need to make some assumptions or apply some domain specific heuristics to create a precise information extractor. This is done by analyzing the HTML structure of a document. In particular, we localize our learning to specific blocks of HTML, and strongly favor complex hypotheses over less restrictive ones. It is better to under-generalize than to over-generalize. The intuition is that under-generalizing may result in false negatives for a given document, but that the missed opportunities on one document are likely to appear again on other documents.

5.3 Algorithm

Now we will discuss the online wrapper induction algorithm outlined in Figure 14. The input to this algorithm is a set of positive examples (seedExamples at line 1). The output is a list of tokens (extractions).

The first step is to use the seed examples to obtain a set of documents as shown in line 2. This is currently done by selecting some number of random positive seeds to combine in a query to a search engine such as Google. One can imagine more sophisticated ways of selecting seeds such as grouping popular or rare instances together (assuming like-popularity instances are found together), or grouping seeds alphabetically since lists are often alphabetical on the Web.

We apply the learning and extraction to each document individually. Within a document we further partition the space based on the HTML tags. This is done by creating a subtree (or single HTML block from the whole document) for every set of composite tags (such as `<table>`, `<select>`, `<td>`, etc.) that have a start and end tag and more text and tags in between. Once we have selected an HTML block or subtree of the parsed HTML, we must first identify all the positive seeds within that block that are the words used in the search. We may add a threshold to skip and continue with the next block if not enough seeds are found. At this point we apply the learning to induce a wrapper.

A prefix is some pattern that precedes a token (the seeds in our example). In order to learn the best prefix pattern for a given block, we consider all the keywords in that block, and find some pattern that maximally matches all of them. Generally we consider 3 - 10 keywords in a block to learn from (more discussion of this later). One option is to build a prefix that matches as many exact characters as possible for each keyword starting from the token and going outwards to the left. A more flexible option is to increase expressiveness and have wildcards, Boolean characteristics, or semantic/synonym options in the matching, similar to Perl regular expressions. The former option is too specific to generalize well in almost any context, and the latter is complicated and requires many training examples (probably best for free text with many labeled examples). We chose a compromise that we believe will work well in the Web domain. First we require that all characters match up until the first HTML tag. For example, `<center>hot Tucson</center>` and `<td>hot Phoenix</td>` would have a prefix “hot ”. If the text matches up to a tag, then we check if the tags match. In this case we do not require that the whole tag match - we just require that the tag type be the same, even though the attributes may differ. This means that for an `<a...>` tag, two keywords might have a different “href=...” but still match. The only exception is when we match a text block (or text between tags). Then these must match among all keywords in order to be included in the prefix. Some sample wrappers look like (`<td><a>TOKEN motels</a></td>`) and (`// &nbsp; TOKEN &nbsp; //`). The best prefix is generally considered to be the longest matching prefix. To learn a suffix, we apply the same idea outwards to the right of the token.

Once a wrapper is learned, we add it to a wrapper tree. The wrapper tree is a hierarchical structure that resembles the HTML structure. Each wrapper in the wrapper tree corresponds to blocks that subsume or contain other wrappers and their blocks. This can be useful for later analysis and comparison of wrappers for a given document in order to choose which wrappers to apply. One heuristic would be to only apply wrappers that are at the leaves (i.e. smallest HTML block with several keywords). Another heuristic would be to apply a wrapper only if it did not generalize any further than its children. After all the wrappers have been constructed and added to the tree, we select the best ones according to such a measure (initialized with defaults or learned in some way) and apply them to get extractions. Applying a wrapper simply means to find other sequences in the block that match the pattern completely, and then to extract the specified token.

5.4 Example and Parameters

```
Keywords: Italy, Japan, Spain, Brazil

1    <html>
2      <body>
3        My favorite countries:
4        <table>
5          <tr><td><a>Italy</a></td><td><a>Japan</a></td><td><a>France</a></td></tr>
6          <tr><td><a>Israel</a></td><td><a>Spain</a></td><td><a>Brazil</a></td></tr>
7        </table>
8        My favorite pets:
9        <table>
10         <tr><td><a>Dog</a></td><td><a>Cat</a></td><td><a>Alligator</a></td></tr>
11      </table>
12     </body>
13   </html>

Wrappers (at least 2 keywords match):
   w1 (1 - 13):  <td><a>TOKEN</a></td>
   w2 (2 - 12):  <td><a>TOKEN</a></td>
   w3 (4 - 7):   <td><a>TOKEN</a></td>
   w4 (5 - 5):   <td><a>TOKEN</a></td><td><a>
   W5 (6 - 6):   </a></td><td><a>TOKEN</a></td>
```

Figure 15: Example HTML with learned wrappers. LE selects wrapper w3 that covers the table from lines 4 to 7 and extracts all the country names without errors. Other wrappers either over-generalize or under-generalize.

We consider a relatively simple example in Figure 15 in order to see how the algorithm works, and to illustrate the effects of different parameters on precision, recall, overfitting, and generalization. On top we have the 4 seeds used to search and retrieve the HTML document, and below we have the 5 wrappers learned from at least 2 keywords and their bounding lines in the HTML.

The first wrapper, w1, is learned for the whole HTML document, and matches all 4 keywords; w2 is for the body, and is identical to w1, except for the context; w3 has the same wrapper pattern as w1 and w2, contains all keywords, but has a noticeably different and smaller context (just the single table block); w4 is interesting because here we see an example of overfitting. The suffix is too long and will not extract France. We see a similar problem in w5 where the prefix is too long and will not extract Israel.

It is easy to see that the best wrapper is w3; w4 and w5 are too specific; while w2 and w1 are too general. There are a few heuristics one can apply to prefer wrappers such as w3 over the others. One is to force most or all keywords to match (in our case, forcing 3 or 4 words to match rather than 2 would not have allowed w4 or w5). Another is to only consider leaf wrappers. In the case of having at least 2 words match for a wrapper, this would not help since we would select w4 and w5. However, if we combine selecting leaf wrappers with matching many key words, we would eliminate w4 and w5 and be left with w3, which is optimal. The intuition is that generally as we go up the wrapper tree, we generalize our wrappers to a larger part of the document which is more prone to errors. If we do not force many keywords to match, we get smaller leaves and may be more precise lower in the tree, but miss out on some of the structure and get less extractions. Below is a list of some parameters to consider when using this algorithm:

1. Number of keywords to match in a block
2. Selection of wrappers from the wrapper tree (leaves, all, other)
3. Length/complexity of prefix/suffix/both
4. Number of search words to use for retrieving documents
5. Selection of keywords for searching (random, alphabetical, popular/rare together/apart)

5.5 Results

We measured LE on three classes running it for varying number of seeds and queries. We left all parameters at their default values (meaning the wrappers were fairly selective) and searched for documents using 4 randomly drawn seeds at a time. A sample of the results are shown in Experiment 9.

Class	Seeds	Queries	Extractions	Correct	% Correct
City	3,000	9,000	190,000	90,000	47%
Film	300	9,000	31,000	24,500	79%
Scientist	50	5,000	65,000	15,000	23%
City	5	1	6,000	4,000	66%

Experiment 8: Results for LE. *Seeds* is the number of positive examples given as input. *Queries* is the number of times 4 tokens were randomly selected from the seeds to search for documents. *Extractions* is the total number of unique extractions. LE can find large numbers of extractions from relatively few queries. *Correct* is the number of extractions in the class before using the Assessor to boost precision.

As Experiment 9 shows, LE is very efficient at finding many correct extractions in a class. In under two minutes, it took five seeds and found about 4000 correct extractions. Actually this is not very impressive since some lists were found on pages that contained over 18,000 correct city instances (so the *correct* search query can get much better documents). However, in all cases, there was also a significant amount of junk. Here are some of the reasons for this:

1. Airports, Hotels, Countries, and more junk are often listed with cities
2. Actors, Musicians, and misspellings are often listed with movies
3. Famous people, random names, and other information are often listed with scientists

Intuitively this makes sense as lists and HTML structure in general often group related things together. Scientists are particularly difficult since they fall into many more general categories.

5.6 Discussion and Future Extensions

Although the percentage correct in all categories may not look very promising, these results are actually quite good since cutting down the number of candidate tokens from the whole Web to the subsets above helps the Assessor. Also, there may be many items found in lists and other structures on the Web that are not found in free text by standard information extraction methods. For example, rare cities found on long HTML select lists will often not be found in free text.

There are quite a few extensions that can be done to make LE work better. Finding more relevant documents and lists, perhaps through better selection of seeds, will probably help, since there are clearly thousands of lists still to be found in all the classes considered here. Making the wrappers more expressive and learning the best wrapper parameters for each class could help too. For example, movies could use more flexible matching since the titles sometimes have slightly different orders of words, but are still the same.

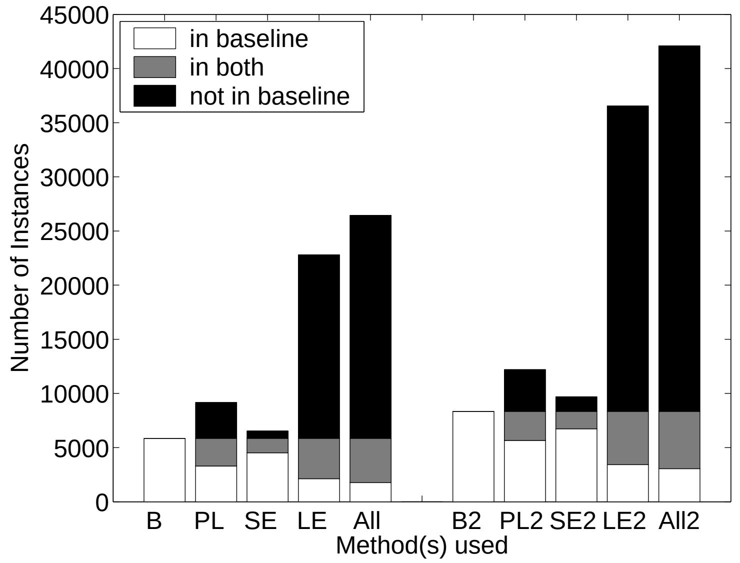
6 Experimental Comparison

We conducted a series of experiments to evaluate the effectiveness of Subclass Extraction (SE), Pattern Learning (PL), and List Extraction (LE) in increasing the recall of the baseline KnowItAll system on three classes `City`, `Scientist`, `Film`. We used the Google API as our search engine. The baseline, SE, and PL methods assigned a probability of correctness to each instance based on PMI scores; LE assigned probability based on the number of lists in which an instance was found. We estimated the number of correct instances extracted by manually tagging samples of the instances grouped by probability, and computed *precision* as the proportion of correct instances at or above a given probability. In addition, in the case of `City`, we automatically marked instances as correct when they appeared in the Tipster Gazetteer, and likewise for `Film` and the Internet Movie Database.

We were surprised to find that over half of our correct instances of `City` were not in the Tipster Gazetteer. The LE method found a total of 78,157 correct extractions for `City`, of which 44,611 or 57% were not in the Tipster Gazetteer. Even if we consider only the high probability extractions, there are still a large number of cities found by KnowItAll that are missing from the Tipster Gazetteer: we found 14,645 additional 'true' cities at precision .80 and 6,288 'true' cities at precision .90.

Experiments 10, 11, and 12 compare the number of extractions at two precision levels: at precision 0.90 for the baseline KnowItAll system (B), the baseline combined with each method (PL, SE, LE) and "All" for the union of instances extracted by B, PL, SE, and LE; and at precision .80 for the bars marked B2, PL2, SE2, LE2, and All2. In each bar, the instances extracted by the baseline exclusively (B or B2) are the white portion, and those extracted by both a new method and the baseline are shown in gray. Since each method begins by running the baseline system, the combined height of the white and gray portions is exactly that of the B bar in each Figure. Finally, instances extracted by one of this paper's methods but *not* by the baseline are in black. Thus, the black portion shows the "added value" of our new methods over the baseline system.

In the `City` class we see that each of the methods resulted in some improvement over the baseline, but the methods were dominated by LE, which resulted in more than a 4-fold improvement, and found nearly

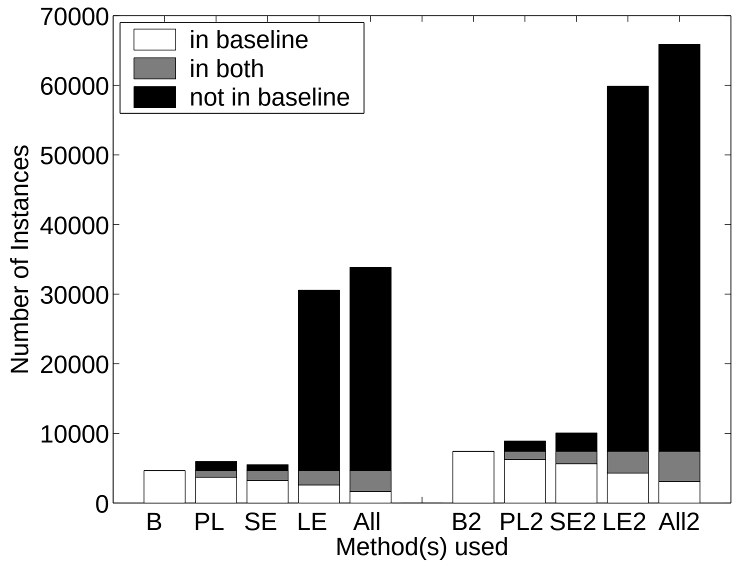


Experiment 9: The number of correct instances of `City` at precision .90 and at precision .80 for baseline KNOWITALL and extensions to the baseline system. Each extension increased recall, with List Extractor giving more than a 4-fold improvement.

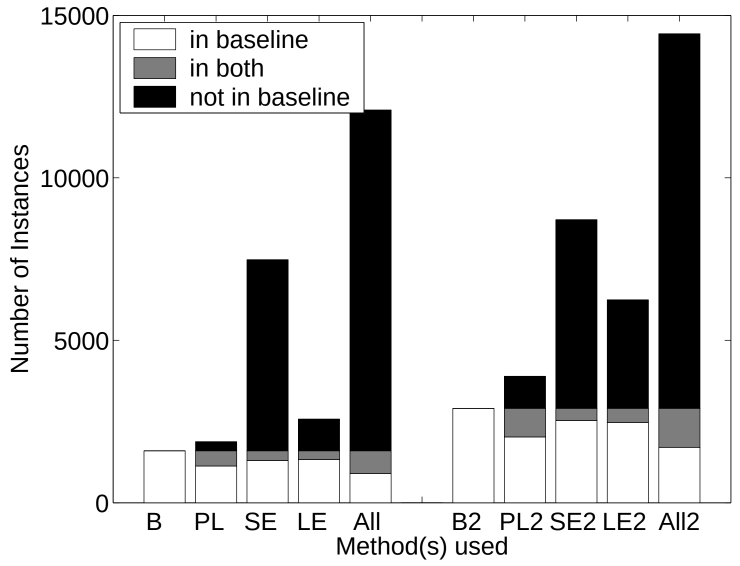
all the extractions found by other methods. We see very similar results for the class `Film` (Experiment 11), where LE gives a 7-fold improvement at precision .90 and 8-fold improvement at precision .80. We saw a different behavior for the class `Scientist` (Experiment 12), where SE's ability to extract subclasses made it the dominant method, though both PL and LE found useful extractions that SE did not. SE gave a nearly 5-fold improvement over B for `Scientist` at precision .90 and all methods combined gave a 7-fold improvement. We believe that SE is particularly powerful for general, naturally decomposable classes such as `Plant`, `Animal`, or `Machine` where text usually refers to their named subclasses (*e.g.*, `Flower, Mammal, Computer`). To use the psychological terminology of [38], we conjecture that text on the Web refers to instances as elements of "basic level" categories such as `Flower` much more frequently than as elements of superordinate ones such as `Plant`.

While our methods clearly enhance KNOWITALL's recall, what impact do they have on its extraction rate? As an information carnivore, KNOWITALL relies heavily on Web search engines for both extraction and assessment. Since it would be inappropriate for KNOWITALL to overload these search engines, we limit the number of queries per minute that KNOWITALL issues to any given search engine. Thus, search engine queries (with a "courtesy wait" between queries) are the system's main bottleneck. We measure extraction rate by the number of unique instances extracted per search engine query. We focus on *unique* extractions because each of our methods extracts "popular" instances multiple times. Table 3 shows that LE not only finds five to ten times more extractions than the other methods, but also has an extraction rate over *forty times greater* than the other methods.

Table 4 shows how the trade-off between recall and precision has major impact on KNOWITALL's performance. For each class and each method, KNOWITALL finds a total number of extractions that is larger than the number of extractions that it can reliably classify as correct. For example, LE finds a total of 151,016 extractions for `City` that include 78,157 correct cities, for an overall precision of 0.52 before applying the Assessor. A perfect Assessor would give high probability to all of the correct extractions, and low probability to all the errors; instead, the set of extractions with precision .80 has only 33,136 correct



Experiment 10: Number of correct instances of `Film` at precision .90 and .80. List Extractor gives a 7-fold increase at precision .90 and an 8-fold increase at precision .80.



Experiment 11: Correct instances of `Scientist` at precision .90 and .80. For this class, Subclass Extraction gives the greatest improvement, with 5-fold increase over the baseline system at precision .90. All methods combined give a 7-fold increase.

Method	Extractions	Queries	Extraction Rate
B	51,614	391,434	0.132
PL	31,163	273,978	0.114
SE	28,672	255,082	0.112
LE	**245,783**	**45,250**	**5.432**
All	304,557	846,674	0.360

Table 3: The total number of unique extractions by each method, along with the number of queries issued and the extraction rate (extractions per query). List Extractor not only finds 5 to 10 times as many extractions as other methods, but has an extraction rate more than 40 times greater.

cities. KNOWITALL has trouble distinguishing many of the correct extractions from the errors.

Class	Method	Extractions	Correct	Precision	Corr. at Precision .90	Corr. at Precision .80
City	B	10,094	8,342	0.83	5,852	8,342
City	PL	11,338	7,442	0.66	5,883	6,548
City	SE	5,045	3,514	0.70	2,023	2,965
City	LE	151,016	78,157	0.52	20,678	33,136
Film	B	36,739	21,859	0.59	4,645	7,436
Film	PL	15,306	9,755	0.64	2,286	2,648
Film	SE	16,820	9,840	0.57	2,286	4,424
Film	LE	78,859	61,418	0.72	27,973	55,575
Scientist	B	4,781	3,690	0.77	1,599	2,905
Scientist	PL	4,519	2,119	0.47	751	1,869
Scientist	SE	6,807	6,168	0.91	6,168	6,168
Scientist	LE	15,907	10,147	0.64	1,245	3,773

Table 4: The total number of extractions, total number correct, and overall precision for each class and method. The total number of correct extractions greatly exceeds the number of correct extractions at precision .80, which suggests that our current Assessor achieves high precision at the cost of a large number of false negatives.

We were pleasantly surprised that the alternate *list frequency* Assessor method used by LE has performance comparable to the PMI method. The PMI probability computation requires a set of search engine queries to get hit counts for each discriminator for each new extraction, which accounts for most of the queries in Table 3. LE is more efficient, because it does not use hit counts, but uses a probability computation that increases monotonically with the number of lists in which an extraction is found. The list frequency method outperformed the PMI method for the class `Film`, finding 70% of the correct films at precision .80 as compared to 34% of correct films at precision .80 for the Baseline system. On the other hand, the PMI method performed better than the list frequency method for the classes `City`, and `Scientist`. This raises an interesting question of whether a frequency-based probability computation can be devised that is effective in maintaining high precision, while avoiding a hit count bottleneck.

The variation in overall precision in Table 4 corresponds to variation in effectiveness of the Assessor in distinguishing correct extractions from noise. The baseline system halted its search for cities while the overall precision was fairly high, 0.83, because the Assessor was assigning low probability to obscure, but correct cities and the signal-to-noise ratio fell below 0.10. This was even more pronounced for SE, which cut off search for more scientists, at an overall precision of 0.91.

While each of the methods tested have numerous parameters that influence their performance, we ran our experiments using the best parameter settings we could find for each method. While the exact results will vary with different settings, or classes, we are confident that our main observations — the large increase in recall due to our methods in concert, and an impressive increase in extraction rate due to LE — will be borne out by additional studies.

7 Related Work

One of KNOWITALL's main contributions is adapting Turney's PMI-IR algorithm [42, 43, 44] to serve as validation for information extraction. PMI-IR uses search engine hit counts to compute pointwise mutual information that measures the degree of correlation between a pair of words. Turney used PMI from hit counts to select among candidate synonyms of a word, and to detect the semantic orientation of a phrase by comparing its PMI with positive words (*e.g.* "excellent") and with negative words (*e.g.* "poor"). Other researchers have also made use of PMI from hit counts. Magnini *et al.* [27] validate proposed question-answer pairs for a QA system by learning "validation patterns" that look for the contexts in which the proposed question and answer occur in proximity. Uryupina [45] classifies proposed instances of geographical classes by embedding the instance in discriminator phrases much like KNOWITALL's, which are then given as features to the Ripper classifier.

KNOWITALL is distinguished from many Information Extraction (IE) systems by its novel approach to bootstrap learning, which obviates hand-labeled training examples. Unlike IE systems that use supervised learning techniques such as *hidden Markov models* (HMMs) [21], rule learning [40, 7, 8], maximum entropy [32], or Conditional Random Fields [29], KNOWITALL does not require any manually-tagged training data.

Bootstrap learning is an iterative approach that alternates between learning rules from a set of instances, and finding instances from a set of rules. This is closely related to co-training [4], which alternately learns using two orthogonal view of the data. Jones *et al.* [23] gives a good overview of methods used in bootstrap learning. IE systems that use bootstrapping include [37, 1, 6, 33, 12, 9]. These systems begin with a set of hand-tagged seed instances, then alternately learn rules from seeds, and further seeds from rules. KNOWITALL is unique in not requiring hand-tagged seeds, but instead begins with a domain-independent set of *generic extraction patterns* from which it induces a set of seed instances. KNOWITALL's use of PMI validation helps overcomes the problem of maintaining high precision, which has plagued previous bootstrap IE systems.

KNOWITALL is able to use weaker input than previous IE systems because it relies on the scale and redundancy of the Web for an ample supply of simple sentences. This notion of *redundancy-based extraction* was introduced in Mulder [25] and further articulated in AskMSR [28]. Of course, many previous IE systems have extracted more complex relational information than KNOWITALL. KNOWITALL is effective in extracting n-ary relations from the Web, but we have yet to demonstrate this experimentally.

KNOWITALL's List Extractor (LE) module uses wrapper induction to look for lists of relevant facts on Web pages. This uses wrapper techniques developed by Kushmerick *et al.* [24], and extended by Cohen *et al.* [11, 10] to learn hierarchical paths (DOM parse) in an HTML document. Perhaps the work that most resembles LE is Google Sets: the input is several words, and the output is a list of up to 100 tokens that are found in lists on the Web. Since we do not know how Google Sets is implemented, we are unable to compare the two systems' algorithms. However, LE achieves far greater recall than Google Sets, at comparable levels of precision.

Several previous projects have automated the collection of information from the Web with some success. Information extraction systems such as Google's Froogle, Whizbang's Flipdog, and Elion, collected large bodies of facts but only in carefully circumscribed domains (*e.g.*, job postings), and only after extensive domain-specific hand tuning. KNOWITALL is both highly automated and domain independent. In fairness, though, KNOWITALL's redundancy-based extraction task is easier than Froogle and Flipdog's task of extracting "rare" facts each of which only appears on a single Web page. Semantic tagging systems, notably SemTag [14], perform a task that is complementary to that of KNOWITALL. SemTag starts with the TAP knowledge base and computes semantic tags for a large number of Web pages. KNOWITALL's task is to automatically extract the knowledge that SemTag takes as input.

KNOWITALL was inspired, in part, by the WebKB project [13]. However, the two projects rely on very different architectures and learning techniques. For example, WebKB relies on supervised learning methods that take as input hand-labeled hypertext regions to classify Web pages, whereas KNOWITALL employs unsupervised learning methods that extract facts by using search engines to home in on easy-to-understand sentences scattered throughout the Web.

8 Future Work

There are numerous directions for future work if KNOWITALL is to achieve its ambitious goals. First, while KNOWITALL can extract n-ary predicates (see, for example, the extraction rule in Figure 9), this ability has not been tested at scale. In addition, we need to generalize KNOWITALL's bootstrapping and assessment modules as well as its recall-enhancing methods to handle n-ary predicates. Second, we need to address tricky extraction problems including the word sense disambiguation (*e.g.*, Amazon is both a river and a bookstore), the extraction of temporally changing facts (*e.g.*, the identity of the president of the United States is a function of time), the distinction between facts, opinions, and misinformation on the Web (*e.g.*, Mulder [25], KNOWITALL's ancestor, was misled by a page entitled "popular Misconceptions in Astronomy"), and more. Fourth, we plan to investigate EM and related co-training techniques [4, 34] to improve the assessment of extracted instances. Finally, several authors have identified the challenges of moving from today's Web to the Semantic Web. We plan to investigate whether KNOWITALL's extractions could be used as a source of semantic annotations to Web pages, which would help to make the Semantic Web real.

The main bottleneck to KNOWITALL's scalability is the rate at which it can issue search-engine queries; While KNOWITALL issues over 100,000 queries to Web search engines daily, it inevitably exhausts the number of queries it is allowed to issue to any search engine in any given day, which forces it to "rest" until the next day. In order to overcome this bottleneck, we are incorporating an instance of the Nutch open-source search engine into KNOWITALL. Our Nutch instance has indexed 60,000,000 Web pages. However, since our the Nutch index is still one to two orders of magnitude smaller than the indices of commercial engines, KNOWITALL will continue to depend on external search engines for some queries. Using the information food chain terminology, incorporating the Nutch instance into KNOWITALL will transform it from an information carnivore to an *information omnivore*.

We have shown that KNOWITALL's PMI-based Assessor is effective at sorting extracted instances by their likelihood of being correct in order to achieve a reasonable precision/recall tradeoff. However, this Assessor suffers from two limitations. First, computing PMI necessities several search-engine queries ($d+1$ queries for d discriminators) for each instance assessed. Second, because PMI scores are combined using a Naive Bayes Classifier—the probabilities assigned to instances tend to be inaccurate. We are developing a new Assessor that addresses both problems by computing accurate probability estimates for instances based on the number of times they repeat in the extraction data, obviating any additional queries. See [17] a formal treatment of the new Assessor and early experimental results showing that its probability estimates are far more accurate than those of the PMI-based Assessor.

Finally, we have also considered creating a multi-lingual version of KNOWITALL. While its generic extraction patterns are specific to English, KNOWITALL could bootstrap its way into other languages by using the patterns to learn instances of a class (*e.g.*, cities in France) and then use its pattern learning module to learn extraction rules and discriminators in French, which may be particularly effective at extracting the names of French cities. In fact, we could restrict underlying search engines such as Google to return only pages in French. KNOWITALL's architecture applies directly to multi-lingual extraction — the main elements that would need to be generalized are the class labels, which are currently in English, and "plug in" modules such as its part of speech tagger.

9 Conclusions

The bulk of previous work on Information Extraction has been carried out on small corpora using hand-labeled training examples. The use of hand-labeled training examples has enabled mechanisms such Hidden Markov Models or Conditional Random Fields to extract information from complex sentences. In contrast, KNOWITALL's focus is on *unsupervised* information extraction from the Web. KNOWITALL takes as input a set of predicate names, but no hand-labeled training examples of any kind, and bootstraps its extraction process from a small set of generic extraction patterns. To achieve high precision, KNOWITALL utilizes a novel generate-and-test architecture, which relies on mutual-information statistics computed over the Web corpus.

The paper reports on several experiments that shaped KNOWITALL's design. The experiments suggest general lessons for the designers of unsupervised extraction systems. Experiment 1 showed that KNOWITALL can tolerate up to 10% noise in its bootstrapped training seeds. This noise tolerance is essential to unsupervised extraction. Experiment 2 showed that negative training seeds for one class can be garnered from the positive training seeds of related classes (*cf.* [26]). Finally, Experiment 3 demonstrated the importance of a well-designed search cutoff metric for both extraction efficiency and precision.

Our *pattern learning* (PL), *subclass extraction* (SE), and *list extraction* (LE) methods greatly improve on the recall of the baseline KNOWITALL system described in [20], while maintaining precision and improving extraction rate. Experiments 4 through 9 suggest design lessons specific to each method. Experiments 10 through 12 report on the relative performance of the different methods on the classes `City`, `Film`, and `Scientist`. Overall, LE gave the greatest improvement, but SE extracted the most new `Scientists`. Remarkably, we found that LE's extraction rate was over *forty times greater* than that of the other methods.

Although KNOWITALL is still "young", it suggests futuristic possibilities for systems that scale up information extraction, new kinds of search engines based on massive Web-based information extraction, and the automatic accumulation of large collections of facts to support knowledge-based AI systems.

Acknowledgments

This research was supported in part by NSF grants IIS-0312988 and IIS-0307906, DARPA contract NBCHD030010, ONR grants N00014-02-1-0324 and N00014-02-1-0932, and a gift from Google. Google generously allowed us to issue a large number of queries to their XML API to facilitate our experiments. We thank Jeff Bigham, and Nick Kushmerick for comments on previous drafts, and Bob Doorenbos, Mike Perkowitz, and Ellen Riloff for helpful discussions.

References

[1] E. Agichtein and L. Gravano. Snowball: Extracting Relations from Large Plain-Text Collections. In *Proceedings of the 5th ACM International Conference on Digital Libraries*, pages 85–94, San Antonio, Texas, 2000.

[2] E. Agichtein and L. Gravano. Querying Text Databases for Efficient Information Extraction. In *Proceedings of the 19th IEEE International Conference on Data Engineering (ICDE 2003)*, pages 113–124, Bangalore, India, 2003.

[3] E. Agichtein, L. Gravano, J. Pavel, V. Sokolova, and A. Voskoboynik. Snowball: A Prototype System for Extracting Relations from Large Text Collections. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, Santa Barbara, California, 2001.

[4] A. Blum and T. Mitchell. Combining Labeled and Unlabeled Data with Co-Training. In *Proceedings of the 11th Annual Conference on Computational Learning Theory*, pages 92–100, Madison, Wisconsin, 1998.

[5] E. Brill. Some Advances in Rule-Based Part of Speech Tagging. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 722–727, Seattle, Washington, 1994.

[6] S. Brin. Extracting Patterns and Relations from the World Wide Web. In *WebDB Workshop at 6th International Conference on Extending Database Technology, EDBT'98*, pages 172–183, Valencia, Spain, 1998.

[7] M.E. Califf and R.J. Mooney. Relational Learning of Pattern-Match Rules for Information Extraction. In *Working Notes of AAAI Spring Symposium on Applying Machine Learning to Discourse Processing*, pages 6–11, Menlo Park, CA, 1998. AAAI Press.

[8] F. Ciravegna. Adaptive Information Extraction from Text by Rule Induction and Generalisation. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI 2001)*, pages 1251–1256, Seattle, Washington, 2001.

[9] F. Ciravegna, A. Dingli, D. Guthrie, and Y. Wilks. Integrating Information to Bootstrap Information Extraction from Web Sites. In *Proceedings of the IIWeb Workshop at the 19th International Joint Conference on Artificial Intelligence (IJCAI 2003)*, pages 9–14, Acapulco, Mexico, 2003.

[10] W. Cohen and W. Fan. Web-Collaborative Filtering: Recommending Music by Crawling the Web. *Computer Networks (Amsterdam, Netherlands: 1999)*, 33(1–6):685–698, 2000.

[11] 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*, pages 323–241, Honolulu, Hawaii, 2002.

[12] M. Collins and Y. Singer. Unsupervised Models for Named Entity Classification. In *Proceedings of the Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora*, pages 100–111, Maryland, USA, 1999.

[13] M. Craven, D. DiPasquo, D. Freitag, A. McCallum, T. Mitchell, K. Nigam, and S. Slattery. Learning to Construct Knowledge Bases from the World Wide Web. *Artificial Intelligence 118(1-2)*, pages 69–113, 2000.

[14] S. Dill, N. Eiron, D. Gibson, D. Gruhl, R. Guha, A. Jhingran, T. Kanungo, S. Rajagopalan, A. Tomkins, J. Tomlin, and J. Zien. SemTag and Seeker: Bootstrapping the Semantic Web via Automated Semantic Annotation. In *Proceedings of the 12th International Conference on World Wide Web*, pages 178–186, Budapest, Hungary, 2003.

[15] P. Domingos and M. Pazzani. On the Optimality of the Simple Bayesian Classifier under Zero-One Loss. *Machine Learning*, 29:103–130, 1997.

[16] R. Doorenbos, O. Etzioni, and D. Weld. A scalable comparison-shopping agent for the World-Wide Web. In *Proceedings of the First International Conference on Autonomous Agents*, pages 39–48, Marina del Rey, California, 1997.

[17] D. Downey, O. Etzioni, and S. Soderland. A Probabilistic Model of Redundancy in Information Extraction. Submitted for publication.

[18] D. Downey, O. Etzioni, S. Soderland, and D.S. Weld. Learning Text Patterns for Web Information Extraction and Assessment. In *AAAI-04 Workshop on Adaptive Text Extraction and Mining*, pages 50–55, 2004.

[19] O. Etzioni. Moving Up the Information Food Chain: Softbots as Information Carnivores. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, 1996. Revised version reprinted in AI Magazine special issue, Summer '97.

[20] O. Etzioni, M. Cafarella, D. Downey, S. Kok, A. Popescu, T. Shaked, S. Soderland, D. Weld, and A. Yates. Web-Scale Information Extraction in KnowItAll. In *Proceedings of the 13th International World Wide Web Conference (WWW-04)*, pages 100–110, New York City, New York, 2004.

[21] D. Freitag and A. McCallum. Information Extraction with HMMs and Shrinkage. In *Proceedings of the AAAI-99 Workshop on Machine Learning for Information Extraction*, Orlando, Florida, 1999.

[22] M. Hearst. Automatic Acquisition of Hyponyms from Large Text Corpora. In *Proceedings of the 14th International Conference on Computational Linguistics*, pages 539–545, Nantes, France, 1992.

[23] R. Jones, R. Ghani, T. Mitchell, and E. Riloff. Active Learning for Information Extraction with Multiple View Feature Sets. In *Proceedings of the ECML/PKDD-03 Workshop on Adaptive Text Extraction and Mining*, Catvat–Dubrovnik, Croatia, 2003.

[24] N. Kushmerick, D. Weld, and R. Doorenbos. Wrapper Induction for Information Extraction. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 729–737. San Francisco, CA: Morgan Kaufmann, 1997.

[25] C. T. Kwok, O. Etzioni, and D. Weld. Scaling Question Answering to the Web. *ACM Transactions on Information Systems (TOIS)*, 19(3):242–262, 2001.

[26] W. Lin, R. Yangarber, and R. Grishman. Bootstrapped Learning of Semantic Classes from Positive and Negative Examples. In *Proceedings of ICML-2003 Workshop on The Continuum from Labeled to Unlabeled Data*, pages 103–111, Washington, D.C, 2003.

[27] B. Magnini, M. Negri, and H. Tanev. Is It the Right Answer? Exploiting Web Redundancy for Answer Validation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 425–432, 2002.

[28] M.Banko, E.Brill, S.Dumais, and J.Lin. AskMSR: Question Answering Using the Worldwide Web. In *Proceedings of 2002 AAAI Spring Symposium on Mining Answers from Texts and Knowledge Bases*, pages 7–9, Palo Alto, California, 2002.

[29] A. McCallum. Efficiently Inducing Features of Conditional Random Fields. In *Proceedings of the Nineteenth Conference on Uncertainty in Artificial Intelligence*, pages 403–410, Acapulco, Mexico, 2003.

[30] I. Muslea, S. Minton, and C. Knoblock. Hierarchical Wrapper Induction for Semistructured Information Sources. *Autonomous Agents and Multi-Agent Systems*, 4(1/2):93–114, 2001.

[31] K. Nigam and R. Ghani. Understanding the Behavior of Co-training. In *Proceedings of the KDD-2000 Workshop on Text Mining*, pages 105–107, Boston, Massachussetts, 2000.

[32] K. Nigam, J. Lafferty, and A. McCallum. Using Maximum Entropy for Text Classification. In *Proceedings of IJCAI-99 Workshop on Machine Learning for Information Filtering*, pages 61–67, Stockholm, Sweden, 1999.

[33] K. Nigam, A. McCallum, S. Thrun, and T. Mitchell. Learning to Classify Text from Labeled and Unlabeled Documents. In *Proceedings of the 15th Conference of the American Association for Artificial Intelligence (AAAI-98)*, pages 792–799, Madison, Wisconsin, 1998.

[34] K. Nigam, A. McCallum, S. Thrun, and T. Mitchell. Text Classification from Labeled and Unlabeled Documents using EM. *Machine Learning*, 39(2/3):103–134, 2000.

[35] W. Phillips and E. Riloff. Exploiting Strong Syntactic Heuristics and Co-Training to Learn Semantic Lexicons. In *Proceedings of the 2002 Conference on Empirical Methods in Natural Language Processing*, pages 125–132, Philadelphia, Pennsylvania, 2002.

[36] D. Ravichandran and D. Hovy. Learning Surface Text Patterns for a Question Answering System. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 41–47, Philadelphia, Pennsylvania, 2002.

[37] E. Riloff and R. Jones. Learning Dictionaries for Information Extraction by Multi-level Bootstrapping. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pages 474–479, 1999.

[38] E. Rosch, C. B. Mervis, W. Gray, D. Johnson, and P. Boyes-Bream. Basic objects in natural categories. *Cognitive Psychology*, 3:382–439, 1976.

[39] R. Snow, D. Jurafsky, and A.Y. Ng. Learning Syntactic Patterns for Automatic Hypernym Discovery. In Lawrence K. Saul, Yair Weiss, and Léon Bottou, editors, *Advances in Neural Information Processing Systems 17*. MIT Press, Cambridge, MA, 2005.

[40] S. Soderland. Learning Information Extraction Rules for Semi-structured and Free Text. *Machine Learning*, 34(1–3):233–272, 1999.

[41] M. Thelen and E. Riloff. A Bootstrapping Method for Learning Semantic Lexicons using Extraction Pattern Contexts. In *Proceedings of the 2002 Conference on Empirical Methods in NLP*, pages 214–221, Philadelphia, Pennsylvania, 2002.

[42] P. D. Turney. Mining the Web for Synonyms: PMI-IR versus LSA on TOEFL. In *Proceedings of the Twelfth European Conference on Machine Learning*, pages 491–502, Freiburg, Germany, 2001.

[43] P.D. Turney. Thumbs Up or Thumbs Down? Semantic Orientation Applied to Unsupervised Classification of Reviews. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 129–159, Philadelphia, Pennsylvania, 2002.

[44] P.D. Turney and M. Littman. Measuring Praise and Criticism: Inference of Semantic Orientation from Association. *ACM Transactions on Information Systems (TOIS)*, 21(4):315–346, 2003.

[45] O. Uryupina. Semi-Supervised Learning of Geographical References within Text. In *Proceedings of the NAACL-03 Workshop on the Analysis of Geographic References*, pages 21–29, Edmonton, Canada, 2003.